

UI Toolkit Reference Manual

Version 10.3.3

Printed: May 2016

The information contained in this document is subject to change without notice and should not be construed as a commitment by Synergex. Synergex assumes no responsibility for any errors that may appear in this document.

The software described in this document is the proprietary property of Synergex and is protected by copyright and trade secret. It is furnished only under license. This manual and the described software may be used only in accordance with the terms and conditions of said license. Use of the described software without proper licensing is illegal and subject to prosecution.

© Copyright 1997–1999, 2001–2016 by Synergex

Synergex, Synergy, Synergy/DE, and all Synergy/DE product names are trademarks or registered trademarks of Synergex.

ActiveX and Windows are registered trademarks of Microsoft Corporation. FarPoint Spread is a trademark of GrapeCity, Inc. All other product and company names mentioned in this document are trademarks of their respective holders.

DCN: UI-00-10.3_04

Synergex
2330 Gold Meadow Way
Gold River, CA 95670 USA

www.synergex.com

phone 916.635.7300

fax 916.635.6549

Contents

Preface

About this manual	xix
Manual conventions	xix
Other resources	xx
Product support information	xx
Synergex Professional Services Group	xxi
Comments and suggestions	xxi

1 Welcome to UI Toolkit

Before You Begin Using UI Toolkit	1-2
What Is UI Toolkit?	1-3
UI Toolkit components	1-4
UI Toolkit Concepts and Terminology	1-7
Display screen layout	1-7
Rendition schemes	1-7
Menus	1-8
Input windows	1-9
Selection windows	1-10
Lists	1-10
Tab set processing	1-10
Text processing	1-11
Composite window processing	1-11
Toolbar processing	1-11
Environments	1-11
Black-box processing	1-12
Keyboard independence	1-13
Getting Started with UI Toolkit	1-14

Requirements and Considerations for Synergy Programs	1-16
The tools.def file	1-16
Reserved global data section and common field names	1-16
Channel usage	1-17
Event-driven programming	1-17
Error processing	1-18
Bounds checking	1-19
Callbacks	1-20
Resizing the application	1-20
Using UI Toolkit with the .NET Framework	1-21
Using UI Toolkit code with Synergy .NET on Windows	1-21
Moving Your Application to Other Systems	1-23
Moving customized files to other systems	1-23

2 Script

Introduction to Script Files	2-3
Types of window definitions	2-3
Using the Script Commands	2-7
.BLANK – Add a blank row	2-8
.BORDER – Define border renditions	2-9
.BOX – Draw a box	2-11
.BUTTON – Include Windows-style button in a window	2-13
.BUTTON_SET – Define the placement of a set of buttons	2-17
.COLUMN – Begin a menu column definition	2-20
.DATA – Load data into a user data set field	2-22
.DATE_GENERATED – Define the date the script file was generated	2-23
.DISPLAY_AREA – Define the display area	2-24
.END – End a window definition	2-26
.ENTRY – Define a column entry	2-27
.FIELD – Define an input window field	2-31
.FILL – Define the fill character	2-73
.FONT – Specify the font for a window	2-75
.INPUT – Begin an input window definition	2-76
.ITEM – Define a selection window entry	2-77
.LINE – Draw a line	2-79
.LIST – Create a set of menu entries	2-81
.LISTCLASS – Begin a list class definition	2-82

- .LOCAL – Define subsequent fields as local 2-87
- .PAINT – Define the paint character 2-88
- .PLACEMENT – Define the placement of a window 2-89
- .REPOSITORY_STRUCTURE – Establish the current repository structure 2-91
- .SCRIPT – Define a version 6 or higher script file 2-94
- .SELECT – Begin a selection window definition 2-95
- .SET – Define an input set 2-97
- .STRUCTURE – Define the structure of a data area 2-99
- .TEXT – Add text to a window or menu column 2-101
- .TITLE – Define a window title 2-108
- .TRIM – Trim extra blanks from end of window 2-111
- .USER_DEFINE – Define a user data set 2-112
- .VERSION – Define the UI Toolkit version 2-113
- .WFIELD – Define a general window field 2-114
- .WINDOW – Begin a general window definition 2-116

Script Examples 2-117

Compiling Scripts 2-118

- Running Script interactively 2-118

- Running Script using the command-line syntax 2-119

- Accessing the Script compiler using the external routine interface 2-120

- %SCR_CLOSELIBRARY – Close the currently open window library 2-121

- %SCR_ERRORCOUNT – Return script compilation error information 2-122

- %SCR_OPENLIBRARY – Open a window library 2-123

- %SCR_PROCESS – Compile a script file into the open window library 2-124

Using the Script-to-Repository Conversion Utility 2-125

- Running scridl 2-125

- Updating the repository 2-127

3 Proto

Using Proto 3-2

- The General menu 3-2

- The Main menu 3-4

- Customizing Proto 3-7

4 ActiveX Routines

Options for ActiveX 4-3

Requirements 4-3

Methods for adding ActiveX controls 4-3

Containers for ActiveX controls 4-4

Using a single control with a Toolkit container 4-4

Using multiple controls with a single container 4-5

Creating a container for the Spread control 4-6

Using an ActiveX Toolkit list 4-6

The basic process 4-7

Error handling for ActiveX control exceptions 4-8

Registering an ActiveX control 4-9

Distributing applications that use ActiveX controls 4-10

Routines for ActiveX Controls in Toolkit Containers 4-11

AX_INPUT – Perform input to an ActiveX control 4-12

AX_REGISTER – Register an ActiveX control 4-15

AX_TIMEOUT – Set the time-out value for an ActiveX control 4-16

%AX_TKCALL – Call a method for an ActiveX control 4-17

%AX_TKCID – Get the ID for an ActiveX container 4-20

%AX_TKCTLID – Get an ActiveX control ID 4-21

AX_TKGET – Get a property for an ActiveX control 4-22

%AX_TKGETINT – Get an ActiveX control property as an integer 4-24

AX_TKSET – Set a property for an ActiveX control 4-26

%AX_TKSINGLE – Create a Toolkit container and load a single ActiveX control 4-28

%AX_TKWIN – Create a Toolkit container window 4-31

AX_WANTSKEY – Register keys to an ActiveX control 4-33

Routine for the Spread Control 4-36

%GRID_CREATE – Create a Toolkit container for the Spread control 4-36

Supplied Methods 4-39

%AX_TKEVENT_CLOSE – Call extension for close event 4-40

%AX_TKEVENT_LEFT_CLICK – Call extension for left click event 4-41

%AX_TKEVENT_LEFT_DBLCLK – Call extension for left double-click event 4-43

%AX_TKEVENT_MAXIMIZE – Call extension for maximize event 4-45

%AX_TKEVENT_MIDDLE_CLICK – Call extension for middle click event 4-46

%AX_TKEVENT_MIDDLE_DBLCLK – Call extension for middle double-click event 4-48

%AX_TKEVENT_MINIMIZE – Call extension routine for minimize event 4-50

%AX_TKEVENT_MOVE – Call extension routine for move event 4-51

%AX_TKEVENT_RESTORE – Call extension routine for restore event 4-53

%AX_TKEVENT_RIGHT_CLICK – Call extension routine for right click event 4-54

%AX_TKEVENT_RIGHT_DBLCLK – Call extension for right double-click event 4-56

%AX_TKEVENT_SCROLL – Call extension routine for scroll event 4-58

%AX_TKEVENT_SIZE – Size the Toolkit container window 4-60

5 Button Routines

Using the Button Subroutines 5-2

Creating buttons 5-2

Defining button placement and the default button 5-3

Enabling and disabling buttons 5-3

Deleting buttons 5-3

Modifying buttons 5-3

Getting information on buttons 5-3

B_BUTTON – Add, delete, or modify a button on a window or tab set 5-4

B_BUTTONSET – Define the placement of a set of buttons on a window or tab set 5-14

%B_DISABLE – Disable a button in a window or tab set 5-16

%B_ENABLE – Enable a button in a window or tab set 5-17

%B_INFO – Return information about the buttons in a window 5-19

6 Composite Window Routines

Creating and Processing Composite Windows 6-2

Placing, removing, and deleting windows 6-3

Processing composite windows 6-3

Controlling tabbing, focus, and input context 6-4

Buttons in composite windows 6-6

Positioning composite windows, child windows, and lists 6-8

Getting composite window information 6-8

Event methods 6-9

%C_CONTAINER – Perform composite window operations 6-10

C_PROCESS – Process a composite window 6-42

The Composite Window Processing Methods 6-46

- C_METHAX – Process a child ActiveX window 6-47
- C_METHCTR – Process a child composite window 6-49
- C_METHINP – Process a child input window 6-51
- C_METHLST – Process a child list 6-54
- C_METHNET – Process a child .NET form window 6-57
- C_METHNOP – Prevent a child from gaining focus 6-59
- C_METHSEL – Process a child selection window 6-61
- C_METHTS – Process a child tabbed dialog 6-63
- C_METHTXT – Process a child text window 6-65

7 Environment Routines

Using Environments 7-2

- E_ENTER – Enter a new environment 7-3
- E_EXIT – Exit to the previous environment 7-5
- %E_FONT – Modify the font for the header, footer, or information line 7-6
- %E_INFO – Return information about environment processing 7-8
- E_METHOD – Register an environment method 7-35
- E_SECT – Place text into a section of the display screen 7-38
- E_STATE – Modify the state of the current environment 7-41

The Environment Methods 7-45

- %EAPPMOVE_METHOD – Perform application move processing 7-46
- %EAPPSIZE_METHOD – Perform application resize processing 7-47
- %EAPPSTATE_METHOD – Perform application state processing 7-49
- %ECENTURY_METHOD – Perform century processing 7-50
- ECHKFLD_METHOD – Perform user field validation 7-53
- ECLOSE_METHOD – Perform application close processing 7-59
- EDSPFLD_METHOD – Perform user field display 7-62
- EEDTDSP_METHOD – Perform user field editing 7-67
- EENTRST_METHOD – Perform input override processing 7-70
- EFKEY_METHOD – Perform key mapping 7-72
- EHELP_METHOD – Perform help processing 7-73
- ESCRIPTERR_METHOD – Perform script compilation error processing 7-80
- EUTILS_METHOD – Perform utilities processing 7-82

8 Input Routines

Using Input Windows 8-3

Input fields 8-3

Building Input Windows at Runtime 8-6

I_ACTIVATE – Specify characters for input termination 8-8

I_CTRCONTEXT – Establish input context for child input window 8-9

I_DISABLE – Disable input fields 8-11

I_DISPLAY – Display data to input fields 8-13

I_DSPAREA – Set the display area for an input window 8-15

I_DSPFLD – Display data to one input field 8-17

I_ENABLE – Enable input fields 8-19

%I_FLDDIM – Retrieve an input field's dimension or number of elements 8-21

I_FLDINF – Return input field information 8-22

I_FLDMOD – Modify an input field's attributes at runtime 8-25

%I_FLDPREC – Retrieve an input field's precision 8-36

I_FLDSEL – Assign a new selection window to an input field 8-37

%I_FLDSize – Retrieve an input field's size 8-39

%I_FLDSTRPOS – Retrieve an input field's structure position 8-40

I_FORCE – Force data to or from an input field 8-41

I_FRAMES – Display frames on input fields 8-43

I_GETFLD – Get data from an input field's data area 8-44

%I_GETSTRING – Retrieve a string referenced by an input window data pointer 8-46

%I_INFO – Return general information about an input window 8-48

I_INIT – Initialize input fields 8-49

I_INPFLD – Process an input field 8-51

I_INPUT – Process an input set 8-55

I_LDINP – Load an input window from a library 8-62

I_NEXT – Specify the next input field or button to be processed 8-65

I_PROMPT – Set a variable prompt 8-69

I_PUTFLD – Put data into an input field 8-70

I_READONLY – Set read-only state for input fields 8-72

I_SETDEL – Delete fields from an input set 8-77

I_SETINF – Retrieve information about an input set 8-80

I_SNAPSHOT – Save/restore an input window's controls 8-82

I_TXTPOS – Get or set the input position for a multi-line text field 8-84

I_USER – Retrieve/load the user-defined string for an input field 8-87

IB_BUTTON – Include a Windows-style button 8-88

- IB_BUTTONSET – Define the placement of a set of buttons 8-90
- IB_DSP_AREA – Set the display area for a new input window 8-92
- IB_END – Define the end of a new input window 8-93
- IB_FIELD – Add an input field to a new input window 8-95
- IB_INPUT – Create a new input window 8-103
- IB_LOCAL – Set the current input window structure to local 8-105
- IB_PAINT – Set the paint character for a new input window 8-106
- IB_RPS_STRUCTURE – Specify the repository structure for a new input window 8-107
- IB_SET – Add a set definition to a new input window 8-109
- IB_STRUCTURE – Define a local structure within a new input window 8-111
- The Input Methods 8-113
 - IARRIVE_METHOD – Perform input field arrive processing 8-114
 - %ICHANGE_METHOD – Perform input field change processing 8-118
 - %IDISPLAY_METHOD – Perform input field display processing 8-122
 - IDRILL_METHOD – Perform drilldown button processing 8-125
 - %IEDITFMT_METHOD – Perform input field “edit format” processing 8-127
 - IHYPER_METHOD – Perform hyperlink prompt processing 8-130
 - ILEAVE_METHOD – Perform input field leave processing 8-132

9 List Routines

- Using Lists in Toolkit Applications 9-4
- Defining Lists 9-5
 - Specifying and determining the list type 9-5
 - The list class: specifying general list characteristics 9-5
 - The input window: specifying the list title and the appearance of individual list items 9-6
 - Using non-window data 9-7
 - Defining behavior for lists 9-8
- Processing and Maintaining Lists 9-10
 - Performing list processing 9-10
 - Enabling users to search a list 9-13
 - IDs for list windows 9-14
- Notes and Tips for ActiveX Toolkit Lists 9-15
- Published Properties for ActiveX Toolkit Lists 9-17
 - Examples 9-21

Published Methods for ActiveX Toolkit Lists 9-22

- GetCellColor – Retrieve the color of an individual cell 9-23
- GetCellFont – Retrieve the font of an individual cell 9-25
- GetColumnWidth – Retrieve the width of a column 9-27
- GetFooterCell – Retrieve the text in an individual cell of a footer 9-28
- GetFooterCellAlignment – Retrieve the alignment of a footer cell 9-29
- GetFooterCellFont – Retrieve the font of an individual footer row 9-30
- GetHeaderCell – Retrieve the text in an individual cell of a header 9-32
- GetHeaderCellAlignment – Retrieve the alignment of a header cell 9-33
- GetHeaderCellFont – Retrieve the font of an individual header row 9-34
- GridRepaint – Force a full repaint 9-36
- SetCellColor – Change the color of an individual cell 9-37
- SetCellFont – Change the font of an individual cell 9-39
- SetColumnWidth – Change the width of a column 9-41
- SetFooterCell – Change the text in an individual cell of a footer 9-42
- SetFooterCellAlignment – Change the alignment of a footer cell 9-43
- SetFooterCellFont – Change the font of an individual footer row 9-44
- SetHeaderCell – Change the text in an individual cell of a header 9-46
- SetHeaderCellAlignment – Change the alignment of a header cell 9-47
- SetHeaderCellFont – Change the font of an individual header row 9-48

Published Event for ActiveX Toolkit Lists 9-50

- OnOriginChange – The origin (upper left cell) has changed 9-50

The List Subroutines and Functions 9-52

- L_BORDER – Add or remove the border and drag bar on a list 9-54
- L_BUTTON – Add, delete, or modify a button on a list 9-55
- L_BUTTONSET – Define the placement of a set of buttons on a list 9-62
- %L_BUTTONSTATE – Enable or disable a button on a list 9-64
- L_CHR – Perform single-keystroke input to a list 9-66
- L_CLASS – Create a list class at runtime 9-68
- %L_CLASSINFO – Return information about a list class 9-72
- L_CREATE – Create a list 9-75
- L_CTRCONTEXT – Establish focus within a child list window 9-80
- L_DATA – Manipulate the data in a list 9-82
- L_DELETE – Delete a list 9-85
- L_DISABLE – Disable a list 9-86
- L_ENABLE – Enable a list 9-87
- L_FINDSPEC – Specify parameters for a find operation 9-88

- %L_FINDWND – Construct a find window for L_SELECT 9-92
- L_GLOBAL – Promote a list to global status 9-95
- %L_ICON – Set a list icon 9-96
- L_INPFLD – Perform single-field input within a list 9-98
- L_INPUT – Perform input to a set within a list 9-101
- L_METHOD – Register a list method 9-105
- L_NEXT – Set selected item by number 9-107
- L_PLACE – Place a list 9-109
- L_PROCESS – Process a list 9-111
- L_QUEUE – Queue commands for a list 9-118
- L_REMOVE – Remove a list 9-120
- L_RESIZE – Resize a list 9-121
- L_RESTART – Restart a list 9-122
- L_RETURNKEY – Set the return state of privatized keys 9-123
- L_SECT – Place text in a list header, footer, or title 9-125
- L_SECTDRAW – Draw a line in a list header or footer 9-127
- L_SELECT – Do list selection processing 9-128
- %L_SELSTYLE – Retrieve/change the selection style for a list 9-136
- L_STATE – Modify a list's state 9-138
- L_STATUS – Retrieve the status of a list 9-139
- %L_USER – Set/retrieve user data associated with a list 9-144
- L_VIEW – Shift the horizontal display area of a list 9-146

The List Methods 9-147

- LARRIVE_METHOD/LLEAVE_METHOD – Perform list item arrive or leave processing 9-148
- LDBLCLICK_METHOD – Perform list item double-click processing 9-151
- LLOAD_METHOD – Perform list item “load” processing 9-154

10 Menu Routines

Introduction to Menus 10-2

- General appearance of menus 10-2
- Making a menu selection 10-4
- Processing menus 10-8
- Runtime generation of menu columns 10-9
- Customizing the way environments place and remove columns 10-9
- M_COLUMN – Place, remove, and delete menu columns 10-10
- M_DEFCOL – Set default drop-down menu column or entry 10-12
- M_DISABLE – Disable an item 10-13

M_ENABLE – Enable an item 10-15
 %M_INFO – Get menu system information 10-17
 M_KEYTXT – Get shortcut text 10-36
 M_LDCOL – Load a menu column 10-37
 %M_POPUP – Process a pop-up menu 10-39
 M_PROCESS – Process a menu 10-43
 M_RESET – Reset to a disabled menu bar 10-47
 M_RPLENT – Replace menu entry names with their shortcuts 10-48
 M_SCINFO – Get menu and shortcut entry information 10-50
 %M_SIGNAL – Signal a menu entry 10-52
 %M_TEXT – Modify and retrieve menu entry text 10-56
 M_USER – Retrieve a user text string for a menu entry 10-58
 MB_BLANK – Add a blank row to a new column 10-59
 MB_COLUMN – Create a new column 10-61
 MB_END – Define the end of a new column 10-65
 MB_ENTRY – Add a menu entry to a new column 10-66
 MB_LINE – Add a line to a new column 10-68
 MB_LIST – Create a list entry in a new column 10-69
 MB_TEXT – Add a row of text to a new column 10-70

11 .NET Routines

Adding and Using .NET Forms, .NET Controls, and WPF Elements 11-2
 DOTNET_TKADDCTL – Add a form, control, or element to a previously-embedded form 11-6
 DOTNET_TKINPUT – Process an embedded form 11-8
 DOTNET_TKFORM – Retrieve form object from a Toolkit container 11-11
 %DOTNET_TKWIN – Create a container window and embed a form, control, or element 11-12

12 Selection Routines

Introduction to Selection Processing 12-2
 Display renditions 12-3
 Making a selection 12-3
 Creating a check box selection window 12-3
 S_SELBLD – Create a selection window 12-5
 S_SELDFLT – Set or return default entry for a selection window 12-8
 S_SELECT – Process a selection window 12-10
 S_SELECTCB – Process a check box selection window 12-13

- S_SELINF – Retrieve selection window information 12-15
- S_SELLD – Load new selections into a selection window 12-17
- S_UPDATECB – Update a check box selection window 12-18

13 Tab Set Routines

- Understanding Tab Set Processing 13-2
 - %TS_PROCESS – Process a tabbed dialog 13-4
 - %TS_TABSET – Create and manipulate a tabbed dialog 13-7

14 Text Routines

- Using Text Subroutines 14-2
 - T_EDIT – Edit the text in a text window 14-3
 - T_GETLIN – Get lines of text from a text window 14-7
 - T_PUTTXT – Put text into a text window 14-8
 - T_SETUP – Specify a text window's characteristics 14-11
 - T_TXTRD – Read text from a window library 14-15
 - T_TXTWRT – Store text into a window library 14-17
 - T_VIEW – Control view through a text window 14-19
 - T_WNDRD – Load a text window from a window library 14-22
 - T_WNDWRT – Store a text window's contents in a window library 14-23

15 Toolbar Routines

- Understanding Toolbar Processing 15-2
 - %TB_BUTTON – Load and manipulate toolbar buttons 15-3
 - %TB_INFO – Return toolbar information 15-15
 - %TB_TKCREATE – Create and log a toolbar with the environment 15-32
 - TB_TKDEL – Delete a toolbar that's logged with the environment 15-34
 - TB_TKGLOBAL – Promote a toolbar to global 15-35
 - TB_TKLOG – Log a toolbar with the environment 15-36
 - %TB_TOOLBAR – Create and manipulate a toolbar 15-37

16 Utility Routines

- Introduction to the Utility Routines 16-4
 - U_ABORT – Process an abort 16-5
 - U_ABOUT – Display an "About" help window 16-8
 - U_BAR – Place or remove the menu bar or information line 16-10
 - U_BEEP – Ring the terminal bell 16-11
 - U_CHARSB – Copy or load scroll bar characters 16-12

U_CHECKVERSION – Check for minimum version of any product	16-14
U_CHNINF – Return information about a channel	16-15
U_CHR – Accept one character from the terminal	16-16
U_CLOSE – Close or purge a channel	16-18
%U_CMDLINOPT – Search for a command line option	16-19
U_CNVDAT – Convert lunar and Julian dates	16-21
U_CREATESB – Create a scroll bar window	16-22
U_CTRCONTEXT – Establish context within a child window	16-24
U_DCDDAT – Decode an alpha date string into its numeric components	16-26
U_DCDTIM – Decode an alpha time string into its numeric components	16-29
U_DEBUG – Invoke the UI Toolkit window debugger	16-31
U_DIALOG – Generate and process a dialog box	16-32
U_EDIT – Edit text and control line breaks in a window	16-34
U_EDITKEYS – Edit a key map	16-40
U_EDITREND – Create and modify rendition schemes	16-42
%U_ENUMWNDS – Return a list of windows in a library	16-45
U_FINISH – Terminate the UI Toolkit environment	16-46
U_FLASH – Flash a message window	16-47
U_FLD – Accept one field from the terminal	16-48
U_FMTDAT – Format a date value into an alpha date string	16-51
U_FMTTIM – Format a time value into an alpha time string	16-53
U_FNDFIL – Search for a file	16-55
U_GBLCHN – Give channel global status	16-57
U_GBLWND – Give window global status	16-58
U_GETCHN – Obtain a channel	16-59
%U_GETFILENAME – Prompt user for filename	16-61
U_GETTXT – Get text from the text library	16-67
%U_GETWNDERR – Get a window system error	16-69
U_HELPREND – Use help window renditions	16-70
%U_HTMLHELP – Invoke HTML Help	16-72
%U_ICON – Set an icon or get an icon count	16-87
U_INITREND – Reset Toolkit renditions	16-89
U_LDREND – Load a rendition scheme	16-90
U_LDWND – Load a window from a library	16-92
U_LOGWND – Log a window as global or local	16-94
U_MESSAGE – Display a message	16-95
U_MINVERSION – Check for minimum Synergy/DE product versions	16-98
U_MODREND – Modify the current rendition scheme	16-99

U_MONTHS – Assign alpha characters to represent months	16-101
%U_MSGBOX – Pop up a Windows-style message box	16-102
U_OPEN – Open a channel/file	16-106
U_PARSEVERSION – Parse a version number	16-108
U_POPUP – Pop up a window	16-109
%U_PRINTQUERY – Retrieve printer characteristics	16-111
%U_PRINTSETUP – Display a Print Setup dialog box	16-113
U_REDRAW – Clear screen and redraw it	16-115
U_REND – Process display renditions	16-116
U_RESIZE – Resize the screen	16-122
U_RESIZESB – Resize a scroll bar window	16-123
U_RLSCHN – Release a channel	16-124
U_RSTREND – Restore renditions	16-125
U_SAVE – Save a window to a window library	16-127
%U_SAVELOG – Save an initialization setting to synergy.ini or synuser.ini	16-128
%U_SAVESETTINGS – Save application window settings to synergy.ini or synuser.ini	16-130
U_SAVREND – Save a rendition scheme to a file	16-132
%U_SELPALETTE – Select a color palette entry	16-134
U_START – Initialize the UI Toolkit environment	16-136
U_STOREND – Store a rendition scheme	16-139
U_TEXT – Load and modify text library entries	16-141
U_UPDATE – Update screen	16-143
U_UPDATESB – Update a scroll bar window	16-145
U_VERSION – Return Toolkit version number	16-147
U_WAIT – Wait for a user response	16-148
%U_WINCELLS – Convert pixels to fractional rows or columns	16-150
%U_WINCOLOR – Perform Windows-specific color operations	16-152
U_WINDOW – Place/move/remove/delete windows	16-159
%U_WINHELP – Invoke WinHelp Help	16-162
%U_WININFO – Return Windows-specific information	16-166
%U_WINMETRICS – Return information about a window	16-170
%U_WNDEVENTS – Associate methods with mouse events	16-203
%U_WNDFONT – Set/retrieve font information on Windows	16-213
%U_WNDSTYLE – Change the style of a list or window	16-232
%U_WNDTYPE – Return a window's type	16-233

The Utility Methods 16-235

 USTART_METHOD – Perform application start-up processing 16-236

 %UWNDEVENTS_METHOD – Perform window event processing 16-237

17 Debugger

What is the Window Debugger? 17-2

Using the Debugger 17-3

 CLEAR or CLS 17-4

 DISPLAY 17-4

 EXAMINE or SHOW 17-4

 QUIT 17-6

 REDRAW 17-6

 WINDOW 17-6

 LIST 17-7

 FILESTACK 17-7

18 Customizing UI Toolkit

Customizing Configuration Fields 18-2

 tkctl.def 18-2

 tools.def 18-7

Overloading UI Toolkit Routines 18-8

Customizing Key Mapping for Menu Shortcuts 18-11

 Files and environment variables used for key mapping 18-12

 Customizing with the key mapping utility (UNIX and OpenVMS) 18-15

 Modifying shortcut identifiers for menu entries defined in a script 18-19

 Redefining identifiers for shortcuts specified with MB_ENTRY 18-20

 Using keys.dbl to test shortcuts (UNIX and OpenVMS) 18-21

 Example 18-21

 Customizing reserved keys 18-22

Customizing the Look of Your Application 18-24

 Synergy colors and the color palette 18-24

 Renditions 18-25

Using Fonts on Windows 18-40

 Defining font palette entries 18-40

 Setting the font for an application 18-42

 Setting the font for an object 18-45

How fonts are applied 18-48
Using fixed fonts 18-49
Using proportional fonts 18-49

Appendix A: Maximums

Appendix B: Reserved Menu Entries

Appendix C: Keyboard Navigation for Lists on Windows

Appendix D: Methods

Methods: Asynchronous vs. Synchronous D-2
Notes on Input Methods D-3
Signaling a Menu Entry from a Method D-10

Appendix E: File-Stack Processing

Introduction to File-Stack Processing E-2
The File-Stack Subroutines E-4
 FS_APPEND – Append a record to the current logical file E-5
 FS_CLOSE – Close one or more logical files E-6
 FS_INIT – Initialize a file-stack system E-7
 FS_OPEN – Add a logical file to the top of the stack E-8
 FS_PARAM – Get a logical file's parameters E-9
 FS_QUIT – Terminate a file-stack system E-11
 FS_READ – Read an existing record from a logical file E-12
 FS_WRITE – Write a record to a logical file E-13

Appendix F: Linked-List Processing

Using Linked Lists F-2
 LL_CLOSE – Close a linked list on the file stack F-3
 LL_OPEN – Open a linked list on the file stack F-4
 %LL_PROCESS – Process a linked list F-5
 LL_SYNC – Synchronize duplicate linked-list control structures F-8

Glossary

Index

Preface

About this manual

This document describes the general philosophy, concepts, and usage of the Synergy/DE® UI Toolkit. UI Toolkit is designed to help you provide a windows interface for your application. Before you get started, we recommend that you read [chapter 1, “Welcome to UI Toolkit,”](#) carefully. This chapter describes what Toolkit is and where it fits with the windowing system as a whole and the structure of your Synergy™ applications. It also provides hints on getting started with Toolkit. The remaining chapters discuss each Toolkit component in detail and contain examples to help you code your own windows interfaces.

Manual conventions

Throughout this manual, we use the following conventions:

- ▶ In code syntax, text that you type is in `Courier` typeface. Variables that either represent or should be replaced with specific data are in *italic* type.
- ▶ Optional arguments are enclosed in *[italic square brackets]*. If an argument is omitted and the comma is outside the brackets, a comma must be used as a placeholder, unless the omitted argument is the last argument in a subroutine. If an argument is omitted and the comma is inside the brackets, the comma may also be omitted.
- ▶ Arguments that can be repeated one or more times are followed by an ellipsis...
- ▶ A vertical bar (|) in syntax means to choose between the arguments on each side of the bar.
- ▶ Data types are **boldface**. The data type in parentheses at the end of an argument description (for example, (n)) documents how the argument will be treated within the routine. An **a** represents alpha, a **d** represents decimal or implied-decimal, an **i** represents integer, an **n** represents numeric (which means the type can be **d** or **i**), and **any** represents any data type, including object types.
- ▶ To “enter” data means to type it (or highlight it, in the case of a selection window entry) and then press ENTER. (“ENTER” refers to either the ENTER key or the RETURN key, depending on your keyboard.)

- ▶ This grid indicates on which platforms and in which environments a routine, statement, etc., is supported: in traditional Synergy on Windows (WT), in Synergy .NET on Windows (WN), on UNIX (U), or on OpenVMS (V). By “supported” we mean that the item performs a useful function on that platform or environment. (For example, an unsupported routine may cause a compiler error or it may just not do anything.) Note, however, that WT indicates that a routine is available but not supported in Synergy.NET on Windows.

WT	WN	U	V

WIN

- ▶ Items or discussions that pertain only to a specific operating system or environment are called out with the name of the operating system.
-

Other resources

- ▶ The UI Toolkit Release Notes (**REL_DTK.TXT**)
- ▶ *Getting Started with Synergy/DE*
- ▶ *Repository User's Guide*
- ▶ *Synergy DBL Language Reference Manual*
- ▶ *Synergy Tools*
- ▶ *Environment Variables & System Options*
- ▶ *Professional Series Portability Guide*
- ▶ Composer Online Help

Product support information

If you cannot find the information you need in this manual or in the resources listed above, you can reach the Synergy/DE Developer Support department at the following numbers:

800.366.3472 (in the U.S. and Canada)

916.635.7300 (in all other locations)

To learn about your Developer Support options, contact your Synergy/DE account manager at one of the above numbers.

Before you contact us, make sure you have the following information:

- ▶ The version of the Synergy/DE product(s) you are running.
- ▶ The name and version of the operating system you are running.
- ▶ The hardware platform you are using.
- ▶ The error mnemonic and any associated error text (if you need help with a Synergy/DE error).

- ▶ The statement at which the error occurred.
- ▶ The exact steps that preceded the problem.
- ▶ What changed (for example, code, data, hardware) before this problem occurred.
- ▶ Whether the problem happens every time and whether it is reproducible in a small test program.
- ▶ Whether your program terminates with a traceback, or whether you are trapping and interpreting the error.

Synergex Professional Services Group

If you would like assistance implementing new technology or would like to bring in additional experienced resources to complete a project or customize a solution, Synergex® Professional Services Group (PSG) can help. PSG provides comprehensive technical training and consulting services to help you take advantage of Synergex's current and emerging technologies. For information and pricing, contact your Synergy/DE account manager at 800.366.3472 (in the U.S. and Canada) or 916.635.7300.

Comments and suggestions

We welcome your comments and suggestions for improving this manual. Send your comments, suggestions, and queries, as well as any errors or omissions you've discovered, to doc@synergex.com.

1

Welcome to UI Toolkit

Before You Begin Using UI Toolkit 1-2

Suggests the best way to get “up and running” with UI Toolkit. This section includes information for both new and experienced users.

What Is UI Toolkit? 1-3

Explains what UI Toolkit does, describes its components in general terms, and then describes each “tool” in more detail.

UI Toolkit Concepts and Terminology 1-7

Explains basic concepts and terminology of a windowing system. Describes how we designed the windowing components to work together.

Getting Started with UI Toolkit 1-14

Outlines the steps for creating a windowing application using UI Toolkit.

Requirements and Considerations for Synergy Programs 1-16

Discusses some requirements and considerations to keep in mind when writing your Synergy application.

Using UI Toolkit with the .NET Framework 1-21

Introduces methods for using Toolkit code with Microsoft .NET Framework.

Moving Your Application to Other Systems 1-23

Discusses how to deploy your application to other platforms.

Before You Begin Using UI Toolkit

If you are a new user of UI Toolkit, we suggest you do the following:

- ▶ Read all of this chapter. This chapter explains how Toolkit works.
- ▶ Scan the [Glossary](#) at the back of this manual to get a better understanding of Toolkit terminology.
- ▶ Read the introductory sections of [chapter 2, “Script,”](#) to get a better understanding of what scripts are and why you use them. Also look at the individual script commands in this chapter, paying special attention to the following commands:

.COLUMN
.END
.ENTRY
.FIELD
.INPUT
.SELECT
.WINDOW

These are the script commands you will use most often.

- ▶ Make some sample scripts and process them using the Script utility, which is explained in [“Compiling Scripts” on page 2-118](#). If you want, you can use Proto to look at the columns and windows you have just created. (See [chapter 3, “Proto.”](#))
- ▶ Learn the following basic subroutines. You can’t create a Toolkit application without them:

U_START
U_FINISH
M_LDCOL
M_PROCESS
I_LDINP
I_INPUT
U_OPEN
U_CLOSE

- ▶ Examine the example programs and scripts in this manual. It is our experience that a careful examination of examples will help you as much as the discussions for the various Toolkit subroutines.

These suggestions are intended to be a guide so you can begin using UI Toolkit as quickly as possible.

What Is UI Toolkit?

UI Toolkit is a set of subroutines and utility programs that gives you the tools you need to create user interfaces for your applications. Toolkit helps you provide a user interface that is easy to use and includes the following:

- ▶ Color
- ▶ Drop-down menus
- ▶ Cascading menus
- ▶ Pop-up help
- ▶ Ability to run in Windows environments
- ▶ ActiveX controls
- ▶ Tabbed dialogs
- ▶ Composite windows (parent/child windows)

The UI Toolkit environment establishes and maintains input, prompts, messages, menus, help, and other user interface details—so you don't have to write such code yourself. Toolkit utilities handle the user-interface details for you and ensure consistency throughout your application.

UI Toolkit is structured to support large applications. Its design assumes that the major functions of your application will be supported by individual subroutines, while the main routine will consist primarily of a small decision apparatus that makes the appropriate calls to these subroutines. These subroutines and programs interact with window library files that UI Toolkit creates when it processes the script files you have written.

To use UI Toolkit, you first create a user interface window script file with Composer or a text editor. Next, you compile the script file with one of Toolkit's utility programs: Composer, Proto, or Script. These utility programs build the windows and store their definitions in a window library file. (Proto also allows you to preview your window script.) In the final step, you write a Synergy program that calls Toolkit subroutines. [Figure 1-1](#) illustrates this basic procedure.

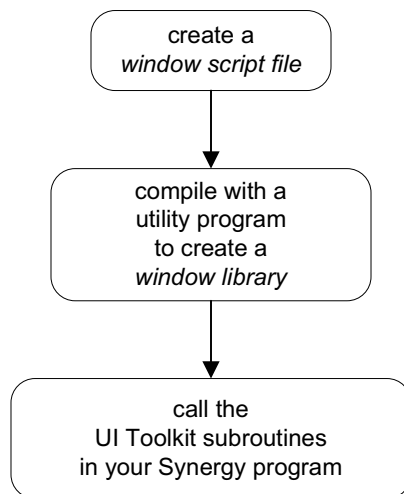


Figure 1-1. Basic procedure for using UI Toolkit.

UI Toolkit components

Window scripts

At the heart of the UI Toolkit windowing system are window script files, which enable you to quickly create general windows, input windows, menu columns, selection windows, and list classes from your own editable text files.

You provide the “script” to build a window, using script commands and arguments. The commands and arguments control the overall window characteristics and the items the window will contain.

You can create your script automatically with Composer, which enables you to interactively design windows, fields, buttons, and so on, in a graphical UI. Once your design looks exactly like you want it, you select the “Save script” option and Composer will translate your design into a script file.

Once your script file is created, you merely run a script processing utility or program (Script or Proto), or select “Compile Script” in Composer to build the window definitions. This processing also saves these definitions in a window library file.

Using UI Toolkit subroutines, your Synergy program can restore windows from the window library when they are needed. (The subroutines and script files do not interact directly. Rather, at runtime, the subroutines open and use the window library that was created when the script file was processed.)

Using window scripts has a distinct advantage. Each window is in a window library file, instead of being hard-coded into your programs. Because these libraries can be maintained outside your programs, you can modify window parameters and text without changing or recompiling your Synergy routines.

The UI Toolkit utility programs

UI Toolkit contains three utility programs: Composer, Script, and Proto.

Composer (Windows)

Composer creates and modifies window script files with a graphical user interface, enabling you to do the following:

- ▶ Interactively design windows and input windows
- ▶ Save your design to a script file
- ▶ Read and modify existing script files
- ▶ Compile window scripts, storing the definitions into a window library

Composer is available on Windows only. For more information, see the “[Designing Your User Interface](#)” chapter of *Getting Started with Synergy/DE*.

Script

The Script program processes all forms of window script files, batch-style, to generate windows. In one step, it does the following for you:

- ▶ Reads and interprets the script
- ▶ Converts window and column scripts into window and column definitions
- ▶ Stores the definitions into a window library

See [chapter 2, “Script,”](#) for more information.

Proto

Proto performs all the actions that Script does, except that it also enables you to interactively process window scripts. Proto lets you display windows, too. Using it, you can do the following:

- ▶ Load (from a window library file) and view menu columns and windows
- ▶ Test menu columns and windows
- ▶ Create and update window libraries
- ▶ Prototype display renditions
- ▶ Edit key map definitions

See [chapter 3, “Proto,”](#) for more information.

[Figure 1-2](#) illustrates how these utilities work together.

Welcome to UI Toolkit

What Is UI Toolkit?

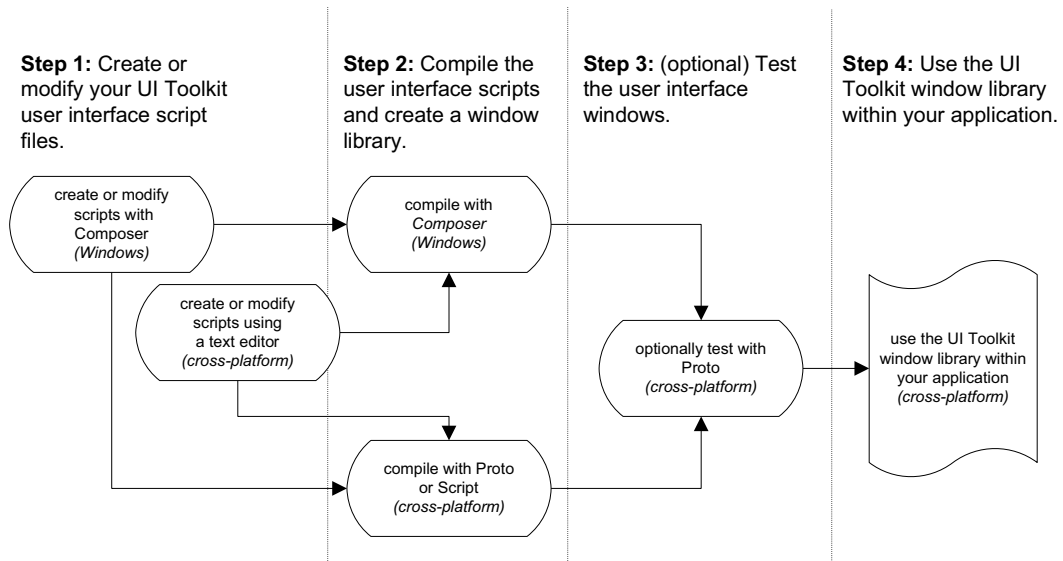


Figure 1-2. Building user interfaces with UI Toolkit utilities.

The UI Toolkit subroutines

The main component of UI Toolkit is a set of subroutines. These subroutines link your application and the windowing system. They handle menu processing, input processing, text processing, selection processing, list processing, toolbar processing, tab set processing, environment maintenance, and more. These subroutines are based on a set of low-level Synergy windowing subroutines (the “W_” routines) that are part of the Synergy runtime.

The specific attributes of menus, input windows, and other window elements are defined in window script files and stored in window libraries, external to your application. You can also set display renditions, and define control and function key sequences externally.

Window debugger

The UI Toolkit window debugger gives you information and the current status of the windowing environment. Using the debugger you can view the following:

- ▶ Any loaded window
- ▶ Any environment level
- ▶ General information about loaded windows and open channels
- ▶ List and file-stack information
- ▶ Detailed information about a specific window

See to [chapter 17, “Debugger,”](#) for more information.

UI Toolkit Concepts and Terminology

This section explains basic concepts, terminology, and how the UI Toolkit utilities and subroutines work and interact.

Display screen layout

The UI Toolkit subroutines and programs use a display screen divided into the following five horizontal sections as pictured in [figure 1-3](#):

- ▶ Header
- ▶ Menu bar
- ▶ Body
- ▶ Information line
- ▶ Footer

The *header*, at the top of the screen, can have up to four lines. Under the header is a one-line *menu bar*, which contains the titles of various drop-down columns. At the bottom of the screen is the *footer* section which, like the header, can have a maximum of four lines. Immediately above the footer is a single *information line*, which holds information for the user. The header and footer can also be used to display information to your users. The *body* of the screen is the area that remains, between the menu bar and the information line, and is where your windows will be displayed.

These five screen sections are defined by the U_START subroutine, the subroutine that sets up the UI Toolkit environment. With U_START, you can change the basic look of your screen display. Since the header and footer are optional, you can specify in U_START whether you want them, and if you do, how big you want each of them to be. You also can remove and place the information line and menu bar by calling the U_BAR subroutine.

Rendition schemes

A *rendition* is the attribute and color combination you select for each item in the UI Toolkit screen display. You can control the rendition of over 40 display items, such as the fields, the window background, the information line, and the footer. A *rendition scheme* is a collective set of chosen renditions.

You define rendition schemes by calling a Toolkit subroutine from your application, or you can use the “Renditions” function in Proto. (Proto calls the same subroutine you would call from your program.) The rendition schemes are stored as records in the rendition file. The default name of the rendition file is **dtkrnd.ism**.

See [“Customizing the Look of Your Application” on page 18-24](#) for more information.

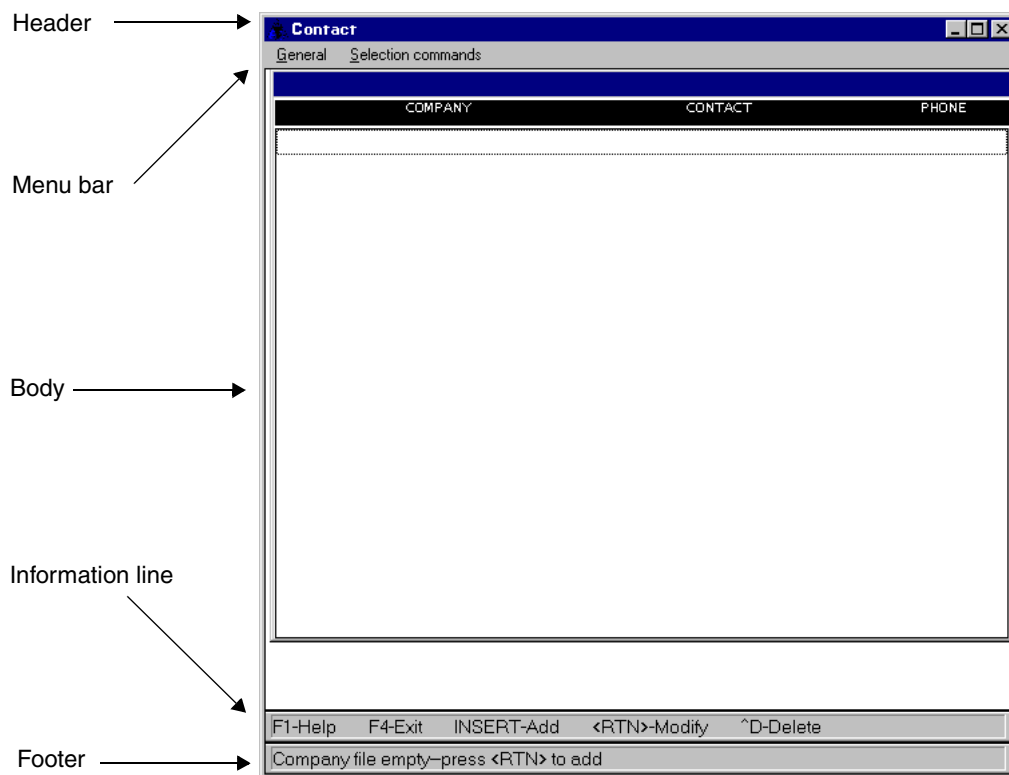


Figure 1-3. Parts of a display screen.

Menus

Toolkit supports drop-down menus using primary menu and submenu columns. A *menu column* is a special window that is generated from a menu column (window) script. Each menu column contains one or more entries. All special keys need to be defined on the menu bar to be recognized by Toolkit (on UNIX and OpenVMS). Each menu entry has the following:

- ▶ An internal entry name (used within your program)
- ▶ The entry text that will be displayed on the menu
- ▶ An optional shortcut (key sequence that automatically selects that entry)
- ▶ A quick-select character that enables users to select the menu entry without using arrow keys

UI Toolkit provides menu processing subroutines that enable you to load, remove, delete, enable, and disable menu columns; enable and disable menu entries and sets of entries; and process a menu. Also included are routines that enable you to create menu columns at runtime. See [chapter 10](#), “Menu Routines,” for more information.

Input windows

One of the easiest ways to confuse end users and make them dissatisfied with your application is to process input inconsistently. Your users want to know that the same rules and constraints apply, regardless of where they enter input. For this reason, one of the major components of UI Toolkit is input processing. Because Toolkit handles all such processing, you can ensure that input will be processed consistently throughout your entire application.

We address the need to process terminal input through *input windows*. An input window is a window that can contain text, input fields, and buttons. Each input field is assigned a position and length within the window, as well as a set of *qualifiers* that define how the field should be processed (a name for the field, the type and length of the data, any redisplay format, a selection window to use, any valid range or set of values, etc.). Field qualifiers may be specified in window scripts or in S/DE Repository. [Figure 1-4](#) below shows an example of an input window.

Toolkit enables you to load and process input windows, modify attributes of input windows and fields at runtime, access input field data, and specify the next field to be processed. You can also create input windows at runtime. See [chapter 8, “Input Routines,”](#) for information about processing input windows and for information on input-related subroutines.

The image shows a graphical user interface window titled "Contact". It contains several input fields for user data entry. The fields are arranged in a form-like layout with labels to the left of the input boxes. The labels and their corresponding input fields are: "Company" (a long single-line field), "Last name" (a single-line field), "First name" (a single-line field), "Title" (a single-line field), "Address" (a two-line field), "City" (a single-line field), "State" (a small single-line field), "Zip" (a small single-line field), "Phone" (a single-line field), "Fax" (a single-line field), "First contact date" (a single-line field), and "Notes" (a large multi-line text area at the bottom).

Figure 1-4. A sample input window.

Selection windows

Most applications include fields where the user can choose from a group of predefined selections included in a *selection window*. The UI Toolkit selection processing routines enable you to process a selection window, change the default selection, replace the entries in a selection window, and create a selection window “on-the-fly.”

See [chapter 12, “Selection Routines,”](#) for more information about building and processing selection windows and for the syntax and a discussion of each of the selection window-related subroutines.

Lists

One of the most powerful tools you can use in your application is *list processing*. The Toolkit list subroutines provide a mechanism whereby static or dynamic lists of information can be easily displayed and traversed.

A list is characterized by its row and column position, number of items visible at one time, optional header and footer lines, optional scroll bars, the rules by which it abides when processed, and whether it is a standard Toolkit list or an ActiveX Toolkit list. The loading of items into the list occurs only as needed (or “on demand”), rather than all at once at the beginning. For this reason Toolkit lists are sometimes referred to as *demand-loaded lists*.

Lists can be placed, removed, enabled, disabled, processed, and deleted. Individual items within the list can be inserted, updated, deleted, and moved.

UI Toolkit handles all the list management functions such as maintaining the data in the list, redisplaying items, scrolling up and down through the list, and tracking which item is being processed. Your program only needs to be concerned with the current item.

Refer to [chapter 9, “List Routines,”](#) for more information about processing lists and for the syntax and a discussion of each of the list-related subroutines.

Tab set processing

Complex applications developed with Synergy/DE often contain input requirements that won’t fit on a single-screen input window. Historically, this problem has been solved by either several windows, which are navigated using a Page Up/Down equivalent, or a single large window with a screen-sized movable display area.

Among software development platforms for Windows environments, a standard approach to this problem has emerged: tabbed dialogs, where windows are rendered as tabbed file-folders.

The tab set processing functions in UI Toolkit enable you to associate input windows or lists with the “tabs” of a tabbed dialog. Specific input window or list processing is handled using a method that you register with Toolkit.

See [chapter 13, “Tab Set Routines,”](#) for more information about processing a tabbed dialog and for the syntax and a discussion of each of the tab set-related functions.

Text processing

A central feature of text processing is the *text window*. A text window is a window that is configured with the text set-up subroutine. It defines formatting characteristics such as margins, indentation, the number of lines between paragraphs, and the size and position of the display area. If a text window is larger than the screen, you can define the visible portion of the window, called the *view* and then move that view throughout the window as desired (by scrolling).

A *text entry* is a special entry in a window library that contains lines of text in a compressed form. Text entries give you a convenient, efficient way to store variable-length text.

The text processing subroutines in UI Toolkit enable you to configure a window as a text window, load and retrieve text from a text window, edit text, move a view through a text window, and read and write text entries.

Refer to [chapter 14, “Text Routines,”](#) for more information about processing text windows and for the syntax and a discussion of each of the text-related subroutines.

Composite window processing

In addition to using tab sets to combine input windows and lists into a single dialog, Toolkit enables you to create composite windows. A composite window is a window that consists of a parent window and child windows and lists, but functions and appears to the user as a single window. For information, see [chapter 6, “Composite Window Routines.”](#)

Toolbar processing

A key component of most Windows applications is a toolbar. The UI Toolkit’s toolbar functions enable you to create and manipulate an application window toolbar. Selecting a toolbar button can trigger the simulation of one or more characters pressed on the keyboard, or invoke an entire subroutine of application code.

See [chapter 15, “Toolbar Routines,”](#) for more information about processing the toolbar and for the syntax and a discussion of each of the toolbar-related functions.

Environments

UI Toolkit maintains a *current environment* and a *current display* within your program.

The environment consists of the following:

- ▶ Windows
- ▶ Menu columns
- ▶ Lists
- ▶ Header
- ▶ Information line

- ▶ Footer
- ▶ Channels
- ▶ State flags
- ▶ Toolbars

The display consists of the following:

- ▶ Menu bar
- ▶ Toolbar placement
- ▶ Body of the display screen
- ▶ Menu column placement
- ▶ Window placement
- ▶ List placement

When you enter a new environment, UI Toolkit saves the current state (which includes both the display and environment) and initializes a new state. When you exit an environment, Toolkit closes its current state and resets it to the preceding state.



Toolbars are under environment control only when created with %TB_TKCREATE or logged with %TB_TKLOG.

Environments make it easier to create and maintain modular programs. If a routine enters a new environment and exits it when it returns to the calling program, the routine is completely modular. Such a self-contained routine is a powerful tool because, while it can be called and used by other routines, it does not depend on the other routines to perform. This also means that the routine looks and works the same way, no matter where your application calls it.

For example, consider an inventory maintenance function in a retail management application as an XCALLED module. Your application can call this routine from the main menu for the user who only wants to review the inventory. It can also call the same function from the inventory receiving routine so that the user automatically updates the inventory when new products come in.

Refer to [chapter 7, “Environment Routines,”](#) for more information about program environments and for the syntax and a discussion of each of the environment-related subroutines.

Black-box processing

Black-box processing is a common term used to describe the type of processing UI Toolkit enables you to do. This means that, from the standpoint of your application, no specific processing code is required for handling menus, input windows, text windows, selection windows, and lists. You simply call the appropriate processing routine (black box), and your program receives a return status that says “this action occurred” or “this option was selected.” Your program doesn’t have to track how the user makes a menu selection or enters data into input fields—it simply takes the

appropriate action. This programming style is known as *event-driven programming*. (Event-driven programming is discussed in more detail later in this chapter.) In object-oriented terminology, this is also called *encapsulation*.

Additionally, menus are “automatically” processed within black-box processing. Each black box has certain reserved menu entries that it recognizes. If any other menu entry is selected, the routine will return to the caller with the selected menu entry.

Menu entry shortcuts are also processed by the black-box routines. If the user presses an assigned shortcut, the black-box routine immediately returns the associated option’s code to your program without going through full menu processing.

Because menu processing is handled automatically by the black-box routines, and because most of your application’s time is spent in some sort of input processing, you rarely need to process the menu yourself.

The various types of black-box processing routines are those for input windows, text windows, selection windows, and lists. Before calling these routines, you can place one or more menu columns on the menu bar to define the available functions. The menu bar will be processed automatically. Any menu entries that the processing routine cannot handle will be returned to your application for processing.

Input window processing is terminated either when the user completes input of all fields, when a menu entry the input processor does not recognize is selected, or when the user processes a field that has been flagged as one that should return control to the caller.

Text window processing is terminated when a menu entry the text processor does not recognize is selected.

Selection window processing is terminated when the user makes a selection (by pressing ENTER), or when a menu entry the selection processor does not recognize is selected.

List processing is terminated when you delete the list or exit the environment in which the list was created.

For information on how to create a Toolkit event loop, see [“The UI Toolkit event loop” on page 1-17](#). This loop takes advantage of UI Toolkit’s black-box processing using event-driven programming.

Keyboard independence

UI Toolkit supports keyboard or terminal independence, which enables a variety of terminals (each with its own key map) to execute the same application on a single system.

See [“Customizing Key Mapping for Menu Shortcuts” on page 18-11](#) for more information about this feature.

Getting Started with UI Toolkit

The initial steps you should take to develop applications using UI Toolkit for traditional Synergy are as follows.

1. Define your windows and menu columns in a window script file. See [chapter 2, “Script,”](#) for more information about writing script files. We suggest you use Composer to create this script interactively. See the [“Designing Your User Interface”](#) chapter of *Getting Started with Synergy/DE* for information on using Composer.
2. Compile the script files with Proto or Script to build the windows and columns and to store them in a window library. (You can also compile the script files from within Composer.)
3. Test and view the windows and columns with Proto.
4. Write a Synergy program that calls the UI Toolkit subroutines. You will need to include the file **WND:tools.def** in your program (with the Synergy DBL `.INCLUDE` statement). Read [“Requirements and Considerations for Synergy Programs”](#) on [page 1-16](#) before writing your program.
5. Compile the source code. (See [“Creating and Running Traditional Synergy Applications”](#) in the *“Building and Running Traditional Synergy Applications”* chapter of *Synergy Tools*.)
6. Link your program with **WND:tklib.elb** (**TKLIB_SH.EXE** on OpenVMS). See [“Linking Object Modules”](#) in the *“Building and Running Traditional Synergy Applications”* chapter of *Synergy Tools* for information on commands for linking.

On Windows, you can link against **tklib_qcheck.elb**, a version of the UI Toolkit library that’s built with more thorough bounds checking (the **-qcheck** compiler option). But note that all code linked against this version of the Toolkit library must be compiled with **-qcheck**. See [“Bounds checking”](#) on [page 1-19](#) and [“Compiler options”](#) in the *“Building and Running Traditional Synergy Applications”* chapter of *Synergy Tools*.

On OpenVMS, you must replace **SYNRTL.OPT** with **SYSS\$SHARE:SYNTKLIB.OPT** to link your program. (Using **SYSS\$SHARE:SYNTKLIB.OPT** will link in the Toolkit shared image.) Additionally, you must use **TKLIB_SH.EXE** to link into the Toolkit libraries instead of **tklib.elb** (which is for Windows and UNIX only).

7. Run your application. It will use the windows and menu columns stored in the window library.

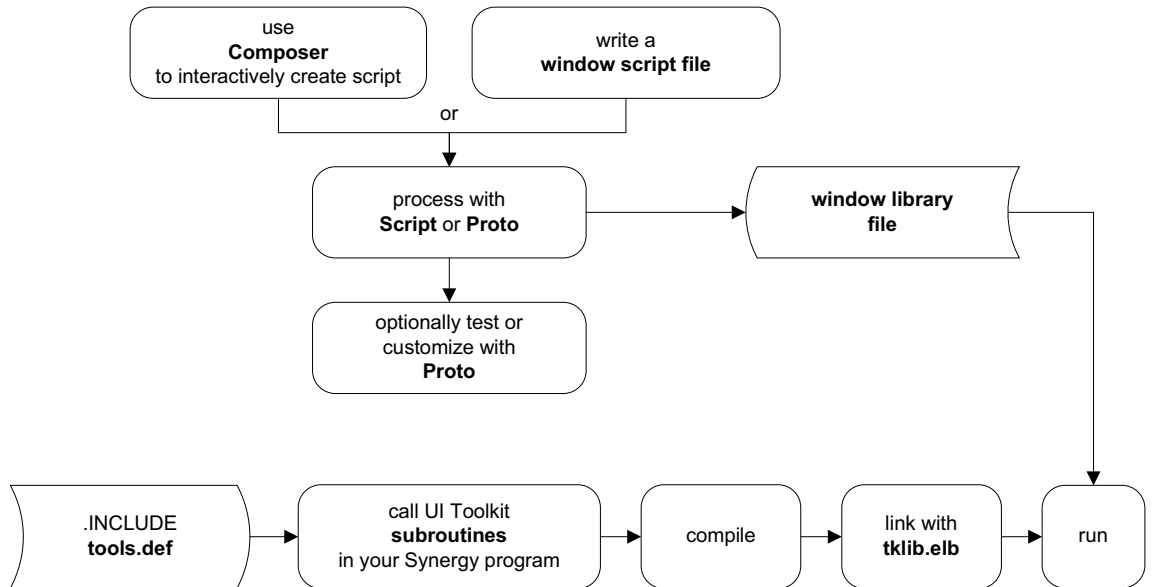


Figure 1-5. Creating a Toolkit application with traditional Synergy.

Requirements and Considerations for Synergy Programs

When writing your Synergy program, remember

- ▶ to include the **tools.def** file.
- ▶ to avoid using global data section names reserved by UI Toolkit.
- ▶ to use variables for channels.

We also recommend that you use event-style programming when using UI Toolkit. This section discusses these and other issues you should keep in mind when creating UI Toolkit applications.

The tools.def file

When using UI Toolkit, the only file you must include in your application is the **tools.def** file. It contains definitions of the flags and value-control arguments required by Toolkit. Use the Synergy DBL `.INCLUDE` statement to include this file in your program.

If you want to include only part of the **tools.def** file in your application, you can use `D_DEFINES_ONLY` and `D_NO_GLOBAL_DATA`. Defining `D_DEFINES_ONLY` before including **tools.def** will include only the Toolkit definitions from **tools.def**. Defining `D_NO_GLOBAL_DATA` will include the Toolkit definitions and the function declarations, but not the global data.

For example, you could define `D_DEFINES_ONLY` and include **tools.def** before the first routine in the file, and then you would need to include **tools.def** only in routines that need access to the global data or the function declarations.

`D_NO_GLOBAL_DATA` can be especially helpful when modularizing your code to be used with `xfServerPlus`. For example, the following code suppresses the inclusion of global data.

```
.define D_NO_GLOBAL_DATA
.include "WND:tools.def"
```

See [“Customizing Configuration Fields” on page 18-2](#) for information on customizing UI Toolkit with **tools.def**.

Reserved global data section and common field names

All Toolkit global data section names begin with “DTK_”. UI Toolkit also uses names that begin with “GLST_” for common field names. Therefore, to avoid potential conflicts, avoid using names beginning with “DTK_” or “GLST_” for common fields or global data sections.

Channel usage

The UI Toolkit subroutines use variables for all channels. You may not be used to this; most programs in the past “hard-coded” their channel numbers by using decimal literals (1, 15, 32, and so forth). However, hard-coding can get you into trouble. For example, you might inadvertently use a channel that is already in use. Enabling the development environment to manage the I/O channels eliminates this problem since Toolkit assigns the channels and maintains a database so channels don’t get re-used inappropriately.

UI Toolkit uses three types of channels: system channels, global channels, and local channels.

A system channel is reserved from use. That is, you may not manipulate it with `U_OPEN`, `U_CLOSE`, `U_GBLCHN`, and so forth. The terminal channel opened by `U_START` is a system channel, and so are all channels that have been excluded from Toolkit use, such as those before or after the *first_channel* through *last_channel* range passed to `U_START`. These channels may be accessed through non-Toolkit calls, but we do not recommend it.

A global channel is a channel that you can open or close. You can either open it as a global channel or promote it to global later. You cannot automatically close a global channel with an `E_EXIT` from the environment in which it was opened; however, you can close it using a `U_CLOSE` at any time.

You can also open a local channel, but it will be closed automatically (with purge) when the environment in which it was opened is exited. Unlike local windows, a local channel may also be closed using `U_CLOSE` at any environment level.

For information about opening and closing channels, see [U_OPEN on page 16-106](#) and [U_CLOSE on page 16-18](#).

Event-driven programming

Programming with UI Toolkit is most effective if you use event-driven (or event-style) programming. Toolkit uses black-box processing, and event-driven programming takes full advantage of this feature. For more information, see [“Black-box processing” on page 1-12](#).

In event-driven programming, control of your program is turned over to UI Toolkit by calling a Toolkit subroutine. When the UI Toolkit subroutine completes its processing, it returns control to your program. Your program tests an exit flag for the “event” that caused the routine to exit. Based on the result of this test, your program performs the appropriate function.

The UI Toolkit event loop

UI Toolkit is designed so this event-driven technique works best in a loop (`REPEAT`, `DO UNTIL`, and so forth). We call this the “Toolkit event loop.” It works for any of the UI Toolkit functions: input processing, menu processing, text editing or viewing, or list processing.

At the top of your loop, you call a Toolkit subroutine. For example, you may call a subroutine to do input processing. UI Toolkit will start input processing, and any input from the user will be processed by Toolkit.

When Toolkit determines that the user has input something that meets its criteria for termination of input processing, a flag (or flags) will be set, the input routine will exit, and control will be returned to your program.

Immediately following the call to the input processing subroutine, your program will perform a test, or series of tests, to determine the reason input processing exited. Using the test results, you can have your program call another subroutine, look up a record in a file, validate data, exit the input processing loop, or anything else that can be done with Synergy DBL.

When your program has done what it needs to do, it returns to the top of the loop and calls the input processing subroutine to get more input.

The complete steps for using event-driven programming are as follows:

1. Remove any unnecessary menu columns from the menu bar.
2. Place necessary menu columns on the menu bar.
3. Set up the following loop:
 - ▶ Get user input with direct menu bar selection, text windows, input windows, selection windows, or lists.
 - ▶ Find out what event occurred. Was a menu entry selected? Is the user done with a text or input window? Do you need to process an input window's break field? Did the user make a selection?
 - ▶ Process the event.
 - ▶ Unless the event is one that terminates processing, return to the first bulleted item under step 3.

Throughout this manual, you will be exposed to the UI Toolkit event loop. We encourage you to study this sample code, and better still, try writing some small programs using this technique. Once you become comfortable with this programming style, you will be well on your way to becoming a productive Toolkit programmer.

Error processing

Synergy DBL runtime errors (with the exception of most window errors) can be trapped with calls to the FATAL subroutine, with ONERROR statements, and with TRY-CATCH blocks.

When a Toolkit error or a window error is generated, the error is automatically sent to U_ABORT, unless the error is returned in an argument for a Toolkit routine, such as the *error* argument for L_CREATE. (If this happens, the error number is returned in the argument, and you can use %U_GETWNDERR to get the error text.) If an error is sent to U_ABORT, it will either perform a STOP and display a fatal error message (the default), or it will throw an error that can be trapped by an ONERROR statement. For more information, see [U_ABORT on page 16-5](#) and [%U_GETWNDERR on page 16-69](#).

For information on Synergy DBL runtime errors, including window errors, see “[Runtime errors](#)” in the “Error Messages” chapter of *Synergy Tools*.

Bounds checking

Bounds checking detects memory access errors (buffer overruns), where a reference exceeds the calling routine's data space, including such failures as data being overwritten and segmentation violations. There are several ways to use bounds checking with Toolkit applications:

- ▶ Set the **g_dtkbounds** global variable or the DTK_BOUNDS environment variable to 1 or 2. (DTK_BOUNDS determines the initial setting of **g_dtkbounds**.) With these settings, bounds checking is limited to a few input and list routines. Although bounds checking is built into Toolkit libraries on Windows, as discussed below, **g_dtkbounds**/DTK_BOUNDS has checks that aren't available with these other bounds checking options. For example, it tells you explicitly when a data area passed to a routine is too small (rather than issuing a generic range error), and on Windows it checks to make sure W_PROC(WP_RESIZE) is not used to resize one of the Toolkit's reserved windows (header, footer, or information line) or an input window associated with a list. For more information, see [DTK_BOUNDS](#) in the "Environment Variables" chapter of *Environment Variables & System Options* and "[g_dtkbounds](#)" on [page 18-3](#).

Note that if **g_dtkbounds** is set to 2, you can use DTK_BOUNDS_LOG to log non-fatal errors to a file. See [DTK_BOUNDS_LOG](#) in the "Environment Variables" chapter of *Environment Variables & System Options*.

- ▶ Use the **-qstrict** or **-qcheck** Synergy compiler option when you compile your application. These options, which are available in traditional Synergy on Windows and on UNIX and OpenVMS, provide different levels of bounds checking for all routines in your application. See "[Compiler options](#)" in the "Building and Running Traditional Synergy Applications" chapter of *Synergy Tools* for more information.

Note that on Windows in standard Synergy, the default Toolkit library (**tklib.elb**) is built with one of the Synergy compiler options for bounds checking (**-qstrict**) by default. And if you want even more thorough checking, you can use **tklib_qcheck.elb**, which is a version of the Toolkit library built with the **-qcheck** Synergy compiler option. To use this library, link against it (**WND:tklib_qcheck.elb**) and build your entire application using **-qcheck**. Note, however, that **-qcheck** is not generally recommended for production builds.

In Synergy .NET, all applications use bounds checking that is automatically provided by the Synergy .NET compiler. The **-qstrict** and **-qcheck** compiler options don't apply to Synergy .NET (these are for the standard Synergy compiler), but you can use **g_dtkbounds**/DTK_BOUNDS and DTK_BOUNDS_LOG. Compiler and runtime checks will generally preclude **g_dtkbounds**/DTK_BOUNDS checking, but in some cases this checking may detect additional issues.

Callbacks

A callback is a call to a Toolkit program from a routine that is not part of the Toolkit program—i.e., a callback routine. On UNIX and OpenVMS, all callbacks are synchronous. On Windows, a callback can be synchronous (e.g., a call to EUTILS_METHOD) or asynchronous (e.g., a call to click method). Most are asynchronous. Note the following:

- ▶ In method subroutines invoked by asynchronous callbacks, code should be kept to a minimum, and operations that could conflict with concurrent processing should be avoided. If you want to invoke a menu entry from a method subroutine, we recommend using the D_SIGNAL operation for %M_SIGNAL.
- ▶ Windows 7 and higher and all 64-bit Windows platforms prevent exceptions from being thrown across the kernel (**user32.dll**) boundary. This means that performing a STOP chain in an asynchronous callback or throwing an exception that could be caught in a routine on the other side of an asynchronous callback will cause random hangs and crashes. The Synergy runtime generates an \$ERR_NOCHAIN error if a callback attempts to perform a STOP chain, but it does not prevent a callback from throwing an exception that could be caught in a routine on the other side.

Resizing the application

You can resize a UI Toolkit application by using U_RESIZE to specify the number of rows and columns for the application. See [U_RESIZE on page 16-122](#).

With Traditional Synergy on Windows, you can also scale an application. That is, you can increase or decrease the size of the display screen, the windows for the application, and the text in those windows.

- ▶ To set the initial scaling percentage for an application, set the APP_SCALE environment variable. The default is 100 (no scaling). See [APP_SCALE](#) in the “Environment Variables” chapter of *Environment Variables & System Options*.
- ▶ To scale an application programmatically, use the D_SETSCALE subfunction for %U_WNDFONT. See [D_SETSCALE on page 16-230](#).
- ▶ To scale the application when a user resizes, maximizes, or restores the application, set the SYN_RESIZE_SCALE environment variable. See [SYN_RESIZE_SCALE](#) in the “Environment Variables” chapter of *Environment Variables & System Options*.

You can also instruct Toolkit to perform special processing when an application is resized. See [%EAPPSTATE_METHOD on page 7-49](#).

Using UI Toolkit with the .NET Framework

There are three ways you can use UI Toolkit code with the Microsoft .NET Framework:

- ▶ Use the Toolkit .NET routines to embed .NET forms in Toolkit windows. With this method, you compile the application with the traditional Synergy compiler and then run the application with the traditional Synergy runtime. See [chapter 11, “*.NET Routines*,”](#) for information.
- ▶ Include Toolkit code in Synergy routines called from *xfNetLink* .NET. For information, see the “[Creating Synergy .NET Assemblies](#)” chapter and “[UI Toolkit Routines](#)” in the “[Preparing Your Synergy Server Code](#)” chapter of the *xfNetLink & xfServerPlus User’s Guide*.
- ▶ Build and run it as a Synergy .NET application with a cell-based appearance. This is an unsupported option. See “[Using UI Toolkit code with Synergy .NET on Windows](#)” below.

Using UI Toolkit code with Synergy .NET on Windows

UI Toolkit includes the **Synergex.SynergyDE.tklib.dll** library for building and running Toolkit code as a .NET application. Although this is available on Windows, it is not supported. (It is not available for Mono development or portable class libraries.) On Windows, most Toolkit routines and Script commands are available, though they result in applications with cell-based rather than Windows-like appearance and functionality. For example, the mouse won’t work in most instances. Note the following:

- ▶ Toolkit routines generally function as they do on UNIX (e.g., `%U_SAVESETTINGS` always returns false). But there are exceptions. Some of these are listed below; others are documented with the routines they apply to.
- ▶ Code with calls to the following routines won’t compile: `%AX_TKCALL`, `AX_TKGET`, `%AX_GETINT`, `AX_TKSET`, `%TB_BUTTON`, `%TB_INFO`, `%TB_TOOLBAR`, and the Script compiler routines (`%SCR_` routines).
- ▶ Native .NET status bars are used for the information line and footer.
- ▶ Native .NET controls are used for menus, but note that automatic menu drop down is not supported, the input string passed to `M_PROCESS` is not used, and the only alignment settings supported for `%M_POPUP` are `D_POPUP_LEFT` and `D_POPUP_RIGHT`.
- ▶ Arguments with the “any” type cannot take objects. For example, you cannot pass an object as one of the method data arguments for `C_PROCESS`.
- ▶ Bounds checking is automatically provided by the Synergy .NET compiler, but for more complete bounds checking, you can use `g_dtkbounds` (or `DTK_BOUNDS`) and `DTK_BOUNDS_LOG`. See “[Bounds checking](#)” on page 1-19.
- ▶ If you want the font to be scaled when the application is resized, maximized, or restored, set the `SYN_RESIZE_SCALE` environment variable to 1. See [SYN_RESIZE_SCALE](#) in the “[Environment Variables](#)” chapter of *Environment Variables & System Options*.

- ▶ By default, Toolkit uses cell-based-style message boxes for `U_MESSAGE`, `%U_MSGBOX`, and `U_WAIT`. However, if `g_netnativemsgbox` is set to true, Toolkit uses native Windows message boxes for these routines. See [“g_netnativemsgbox” on page 18-7](#).
- ▶ There are some restrictions on the ways you can use global fields in **tkctl.def** and **tools.def**. See [“Restrictions for global fields”](#) below.

The following is a summary of the process you’ll follow to use Toolkit code with the .NET Framework on Windows. See the [“Developing with Synergy .NET”](#) chapter of *Getting Started with Synergy/DE* for more information.

1. Create a Synergy project in Visual Studio.
2. Add a reference to **Synergex.SynergyDE.tklib.dll** in your project. This is installed in a CLR directory under Program Files\Common Files\Synergex.
3. Use supported Synergy DBL and Toolkit routines to create the program. You can also define your Toolkit windows in windows script files and then compile the script files with Proto or Script. (See [step 1](#) through [step 3](#) of [“Getting Started with UI Toolkit” on page 1-14](#).) Script and Proto run only in traditional Synergy, but the libraries they create are compatible with Toolkit applications for .NET.
4. Build your application in Visual Studio. This creates an **.exe** or a **.dll** assembly.

Restrictions for global fields

When using Toolkit code with .NET, there are some limitations to the ways you can use global fields defined in **tools.def** and **tkctl.def**. The following are *not* supported:

- ▶ Ranging when assigning a value—for example, `“g_entnam(7:4) = number”`. Instead, you can assign the value of the global field to a temporary field, and then use ranging with that field.
- ▶ Assigning a value from a different type (implicit conversion). Instead, use explicit conversion.
- ▶ Assigning a value and then using the result of the assignment in the same statement—for example, `“if (g_select = value)”`. Instead, separate the assignment and the use of the assignment into two statements:

```
g_select=1
if (g_select) ...
```

- ▶ Modifying through a cast—for example, `“clear ^i(g_entnam)”`. Instead, use a temporary field:

```
clear ^i(temp_var)
g_entnam=temp_var
```
- ▶ Accessing groups or record names.
- ▶ Passing global fields as modifiable parameters to subroutines, functions, or methods. Instead, you can pass a temporary field and then assign its value to a global field.

Moving Your Application to Other Systems

You can deploy your application on any of the platforms supported by Synergy/DE while maintaining a single set of source code for your entire application. It does not matter on which platform your application was developed or to which platform you want to deploy.

Refer to the *Professional Series Portability Guide* for information about the functions, requirements, and processes that differ on the various platforms.

Moving customized files to other systems

Most of the following provide two ways to move files, depending on whether or not the destination system has the same endian type as the original system. If you're unsure of your system's endian type, refer to [“Big-endian and little-endian”](#) in the “UNIX Development” chapter of the *Professional Series Portability Guide* for a complete list of systems and their endian types.

Moving key mapping files

Follow the rules in [“Moving Database Files to Other Systems”](#) in the “Synergy DBMS” chapter of *Synergy Tools*.

Moving text message files

To move Synergy/DE text message files to another system, do one of the following:

- ▶ Copy the files as you would copy any other text file.
- ▶ Use the Synergy UI Toolkit Control Panel to unload and then load the files on the new system. (For more detail, see [“Making major changes: Unloading messages to an ASCII file”](#) in the “General Utilities” chapter of *Synergy Tools*.)

Moving window library files

To move your window library files to another system, do one of the following:

- ▶ If the systems have the same endian type and the same data file type (RMS or ISAM), copy the window library files as you would copy any other file.
- ▶ If the systems have different endian types, you'll need to copy the customized window script file (which has the extension **.wsc**) to the new system and rescript it there. (UI Toolkit must also be installed on the destination system.)

2

Script

Introduction to Script Files 2-3

Explains what a script file is and gives an overview of its use.

Using the Script Commands 2-7

.BLANK – Add a blank row	2-8
.BORDER – Define border renditions	2-9
.BOX – Draw a box	2-11
.BUTTON – Include Windows-style button in a window	2-13
.BUTTON_SET – Define the placement of a set of buttons	2-17
.COLUMN – Begin a menu column definition	2-20
.DATA – Load data into a user data set field	2-22
.DATE_GENERATED – Define the date the script file was generated	2-23
.DISPLAY_AREA – Define the display area	2-24
.END – End a window definition	2-26
.ENTRY – Define a column entry	2-27
.FIELD – Define an input window field	2-31
.FILL – Define the fill character	2-73
.FONT – Specify the font for a window	2-75
.INPUT – Begin an input window definition	2-76
.ITEM – Define a selection window entry	2-77
.LINE – Draw a line	2-79
.LIST – Create a set of menu entries	2-81
.LISTCLASS – Begin a list class definition	2-82
.LOCAL – Define subsequent fields as local	2-87
.PAINT – Define the paint character	2-88
.PLACEMENT – Define the placement of a window	2-89
.REPOSITORY_STRUCTURE – Establish the current repository structure	2-91
.SCRIPT – Define a version 6 or higher script file	2-94

.SELECT – Begin a selection window definition	2-95
.SET – Define an input set	2-97
.STRUCTURE – Define the structure of a data area	2-99
.TEXT – Add text to a window or menu column.....	2-101
.TITLE – Define a window title	2-108
.TRIM – Trim extra blanks from end of window	2-111
.USER_DEFINE – Define a user data set.....	2-112
.VERSION – Define the UI Toolkit version.....	2-113
.WFIELD – Define a general window field.....	2-114
.WINDOW – Begin a general window definition.....	2-116

Script Examples 2-117

Contains examples of window scripts, such as a menu column, an input window, and a selection window.

Compiling Scripts 2-118

Describes the utility that compiles script files and builds windows for your application from script commands.

%SCR_CLOSELIBRARY – Close the currently open window library.....	2-121
%SCR_ERRORCOUNT – Return script compilation error information	2-122
%SCR_OPENLIBRARY – Open a window library.....	2-123
%SCR_PROCESS – Compile a script file into the open window library	2-124

Using the Script-to-Repository Conversion Utility 2-125

Describes **scridl**, a utility that converts field information from a script file into a format readable by S/DE Repository.

Introduction to Script Files

A script file is a standard text file that contains special window building commands. It must begin with the `.SCRIPT` command. The recommended extension for a script file is `.wsc`.

Storing all the window building information in a script file enables you to maintain your windows external to your program. You can do things like redefine border attributes, change shortcuts, and redefine field characteristics without ever touching your code, which means you don't have to recompile or relink.

You can create your script file with any text editor, using the commands described in “[Using the Script Commands](#)” on page 2-7. You can also use Composer to create and modify script files interactively. See the “[Designing Your User Interface](#)” chapter of *Getting Started with Synergy/DE* for more information about Composer.

Types of window definitions

You can define these five types of windows in a script file:

- ▶ General window
- ▶ Menu column
- ▶ Input window
- ▶ Selection window
- ▶ List class

General window definitions

A general window definition defines a general purpose window. It must begin with the `.WINDOW` command.

Menu column definitions

Menu column definitions define primary menu and submenu column windows, and also column entries, quick-select characters, and shortcuts. A menu column definition must begin with the `.COLUMN` command.

Menu entries on a menu column are defined with the `.ENTRY` command. Menu entries are added to the menu in the order of their definitions in the script file. The first `.ENTRY` command defines the first entry, the second `.ENTRY` command defines the second entry, and so on. The total number of menu entries in the column is equal to the total number of entry specifications.

Input window definitions

Input window definitions define windows that can contain text, input fields, and buttons. An input window script must begin with the **.INPUT** command.

Input fields are defined with the **.FIELD** command and are created according to the order of their definitions in the script file. The total number of fields in the window is equal to the total number of field specifications.

Selection window definitions

Selection window definitions define windows that contain one or more selection items. A selection window definition must begin with the **.SELECT** command.

Selection items are defined with the **.ITEM** command and are created according to the order of their definitions in the script file. The total number of selection items in the window is equal to the total number of item specifications.

List class definitions

List class definitions define a set of specific list characteristics with the **.LISTCLASS** command.

List characteristics that are likely to be consistent across multiple lists, such as the number of lines for headers and footers, can be defined in a list class.

Which window definitions use which commands?

The following table shows which window building commands you can use in which kind of window definition. The script commands **.LOG**, **.DATE_GENERATED**, **.SCRIPT**, and **.VERSION** are specified outside of a window definition and are therefore not included in the table.

Commands for Defining Windows					
Command	General window	Menu column	Input window	Selection window	List class
.BLANK		✓		✓	
.BORDER	✓		✓	✓	
.BOX	✓		✓		
.BUTTON			✓		
.BUTTON_SET			✓		
.COLUMN		✓			
.DATA	✓				
.DISPLAY_AREA	✓		✓		

Commands for Defining Windows (Continued)					
Command	General window	Menu column	Input window	Selection window	List class
.END	✓	✓	✓	✓	✓
.ENTRY		✓			
.FIELD			✓		
.FILL	✓		✓		
.FONT	✓		✓	✓	
.INPUT			✓		
.ITEM				✓	
.LINE	✓	✓	✓	✓	
.LIST		✓			
.LISTCLASS					✓
.LOCAL			✓		
.PAINT			✓		
.PLACEMENT	✓		✓	✓	✓
.REPOSITORY_STRUCTURE			✓		
.SELECT				✓	
.SET			✓		
.STRUCTURE			✓		
.TEXT	✓	✓	✓	✓	
.TITLE	✓		✓	✓	
.TRIM	✓		✓		
.USER_DEFINE	✓				
.WFIELD	✓				
.WINDOW	✓				
Data control commands	✓		✓		

Output logging

To send window script output and error messages to the standard output stream, add the `..LOG` command (yes, there are two periods before `LOG`) to your script. Output from all lines that follow the `..LOG` command are sent to the standard output stream.

You can add the `..LOG` command at any point in a script, but it must be on a line by itself. Like all other script commands, `..LOG` is case insensitive.

Using the Script Commands

Script commands always begin with a period (.). Commands are not case sensitive and can be abbreviated to uniqueness. For example, “.C” is valid for .COLUMN, but “.ENT” is required to distinguish .ENTRY from .END.

Window building commands

Window building commands are script commands that define overall window characteristics, including name, size, screen position, border renditions, and title.

Window definitions within a script file begin with one of the following commands (depending on what kind of window you are building): [.WINDOW](#), [.COLUMN](#), [.INPUT](#), [.SELECT](#), [.LISTCLASS](#). The [.END](#) command ends a window definition. You can have more than one window definition (window script) in a single script file.

Arguments

Script commands may have one or more arguments. The first argument must be separated from the command by at least one blank space or tab, and any additional arguments must be separated by commas. There may be one or more blanks or tabs before or after the commas. If you omit optional trailing arguments, you do not need to specify separating commas.

Like commands, arguments can be abbreviated to uniqueness.

Continuation lines

Each script command is a single logical line in the script file. The maximum length of a physical line is 255 characters. You can continue many script commands, such as [.BORDER](#), [.FIELD](#), and [.LISTCLASS](#), on more than one physical line by placing a dash continuation mark (–) at the end of each line that will be continued. The dash can either precede or follow the comma.

Comments and blank lines

A comment line is any line whose first non-blank, non-tab character is a semicolon (;). Comment lines can be placed outside or within a window definition. See [“Special characters” on page 2-107](#) for more information about comment lines within the text of a window.

You may include blank lines in your script files to aid in readability. They are ignored when the script is compiled.

.BLANK – Add a blank row

WT	WN	U	V

.BLANK

Discussion

The .BLANK command adds a blank row to a menu or selection window column. This blank row is not a blank menu entry or selection item; it is just a row without text.

If you have a selection window defined by the .SELECT command that has more than one column but does not have corresponding text entries in the same row of each column, left and right movement within the window won't be consistent. We therefore recommend that you make sure each column in a multi-column selection window contains its text entries in the same row.

On Windows, using .BLANK within a selection window script (.SELECT) has no meaning. And while blank rows in a menu column on Windows are not *selectable*, they may still be highlighted.

.BORDER – Define border renditions



```
.BORDER [border_on_off][, rendition, ...][, DRAGBAR[ (dragbar_on_off) ]]
```

Arguments

border_on_off

(optional) Determines whether the window displays a border. Valid options are as follows:

ON	Window displays a border. (default)
OFF	Window does not display a border.
NOCELL	Window displays a border only if it does not require a full character cell for display.

rendition

(optional) One or more of the following display renditions:

BLINK ITALIC	Blinking (on UNIX and OpenVMS) or italic (on Windows).
NOBLINK NOITALIC	Not blinking (on UNIX and OpenVMS) or not italic (on Windows). (default)
BOLD	Highlighted (or boldfaced).
NOBOLD	Not highlighted (or boldfaced). (default)
COLOR (<i>palette#</i>)	Assign specified <i>palette#</i> , where <i>palette#</i> is a palette number in the range 1 to 16. The default is 1.
HIGHLIGHT	Highlighted (or boldfaced).
NOHIGHLIGHT	Not highlighted (or boldfaced). (default)
REVERSE	Reverse video.
NOREVERSE	Normal video. (default)
UNDERLINE	Underlined.
NOUNDERLINE	Not underlined. (default)

DRAGBAR

(optional) The specification that the window will or will not have a drag bar. (Windows only)

dragbar_on_off

(optional) The specification that turns on or off a drag bar on Windows; this option has no effect on UNIX and OpenVMS. Valid options are as follows:

ON Window has a drag bar if in a Windows environment. (default)

OFF Window doesn't have a drag bar if in a Windows environment.

Discussion

The **.BORDER** command defines the renditions of the border around the window. *Renditions* can be listed in any order.

WIN

In Windows environments, window borders may not take up an entire “cell” as on cell-based environments. Therefore, note the following:

- ▶ If you specify **NOCELL**, the window will have a border only if the application runs in a Windows environment.
- ▶ If you are depending on one window's border to fully occlude a certain area of another window when your application runs on UNIX or OpenVMS, you won't be able to depend on this same occlusion when your application runs in a Windows environment.

None of the *renditions* available for borders have any effect in a Windows environment.

You can optionally specify **DRAGBAR** and the argument *dragbar_on_off*. **DRAGBAR** only affects the window when the border is turned on. If you don't pass **DRAGBAR** and if the window is more than one row in height, the default is **ON**. If the window is one row in height, the default is **OFF**. If you specify **DRAGBAR** without specifying **ON** or **OFF**, **DRAGBAR** defaults to **ON**.

Examples

This example displays a border in reverse video with color palette number 3.

```
.border on, reverse, color(3)
```

This Windows example displays a border around a window that the user will not be able to drag.

```
.border on, dragbar(off)
```


.BOX – Draw a box



`.BOX name, height, width[, POSITION (row, column) ][, rendition, ...]`

Arguments

name

The name of the box (a maximum of 30 characters).

height

The height of the box (number of rows).

width

The width of the box (number of columns).

POSITION

(optional) The position specification for the box.

row

The row at which the upper-left corner of the box is positioned.

column

The column at which the upper-left corner of the box is positioned.

rendition

(optional) One or more of the following display renditions:

BLINK | ITALIC Blinking (on UNIX and OpenVMS) or italic (on Windows).

NOBLINK | NOITALIC Not blinking (on UNIX and OpenVMS) or not italic (on Windows). (default)

BOLD Highlighted (or boldfaced).

NOBOLD Not highlighted (or boldfaced). (default)

COLOR(*palette#*) Assign specified *palette#*, where *palette#* is a palette number in the range 1 to 16.

HIGHLIGHT Highlighted (or boldfaced).

NOHIGHLIGHT Not highlighted (or boldfaced). (default)

REVERSE Reverse video.

NOREVERSE Normal video. (default)

UNDERLINE

Underlined.

NOUNDERLINE

Not underlined. (default)

Discussion

The `.BOX` command draws a box in a general or input window.

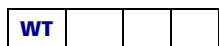
If `POSITION` is not specified, the position of the box defaults to the current screen position.

Examples

This example draws a box with a height of 4 rows, width of 10 columns, at row 3 and column 3 of a window in reverse video.

```
.box box1, 4, 10, pos(3, 3), reverse
```

.BUTTON – Include Windows-style button in a window



```
.BUTTON name [, TEXT (button_text) | BITMAP (file) ] –  
      [, METHOD (method_name[, elb_name]) ] –  
      [, SELECT (select_char) ]
```

Arguments

name

The name of the button (a maximum of 30 characters).

TEXT

(optional) Display text on the button face.

button_text

The text to display on the button face.

BITMAP

(optional) Display a graphic on the button face.

file

The filename for the graphic to display on the button face.

METHOD

(optional) Specify a method to invoke when the button is clicked.

method_name

The name of the method to invoke.

elb_name

(optional) The name of the library from which *method_name* will be drawn.

SELECT

(optional) Specify a quick-select character for the button.

select_char

The quick-select character.

Discussion

You can use the `.BUTTON` command to add Windows-style buttons to a window. The maximum number of buttons allowed in a window is 127.

When a window is being processed, its buttons are automatically enabled by default, and when a window is not being processed, its buttons are automatically disabled by default. However, note the following:

- ▶ Buttons that have been disabled (with `%B_DISABLE`) will not be automatically enabled, and buttons that have been enabled (with `%B_ENABLE`) will not be automatically disabled unless `%B_ENABLE` is used with the *input_only* argument set to true to restore the default behavior for the button. See [%B_ENABLE on page 5-17](#) for more information.
- ▶ To enable buttons on general windows, you must either use `%B_ENABLE` or process the window with `T_VIEW` or `T_EDIT`.
- ▶ For composite windows, `C_PROCESS` enables buttons for the composite container window and all enabled child windows and lists, even those that are not active. Note, however, that this is not the case when `C_PROCESS` is called for a composite window that's part of another composite window. In this case, buttons will have already been enabled by the `C_PROCESS` call for the outer composite window. Additionally, when `C_PROCESS` returns, it disables buttons for the composite window and all child windows and lists—unless the `C_PROCESS` call is for a composite window that's part of another composite window. If it is, buttons will be disabled when `C_PROCESS` returns for the outer composite window.
- ▶ For composite windows, buttons that are explicitly enabled with `%B_ENABLE` (without the *input_only* flag set true) remain enabled when composite window processing is finished.

The placement of the button is determined by the `.BUTTON_SET` command. If no `.BUTTON_SET` command is specified, the buttons are placed (in their defined order) at the bottom of the window, left-justified.

The order in which buttons are defined determines how they are added to the window tabbing order. `TAB` moves forward and `SHIFT+TAB` moves backward through the buttons as if they were fields in the input window.

Focus (display of the active item) is shown on the buttons as a thin dotted line around the text portion of the button. When focused, the button can be “pressed” using either the spacebar or the `ENTER` key. Whether focused or not, any button can be selected (“pressed”) by clicking on it with the left mouse button.

If neither `TEXT` nor `BITMAP` is specified, the button name will be displayed on the button face.

Use `BITMAP(file)` to specify a graphic to display on the button face. See [“Toolkit button graphics”](#) below for information on supported formats and creating transparent button graphics.

If *method_name* is specified, it will be invoked when the button is clicked. The arguments passed to the method are the window ID, button name, and the 20 optional method data arguments passed to `I_INPUT`.

If *method_name* is not specified, the button's name is the menu entry signaled when the button is pressed.

If *select_char* is specified, pressing ALT+*select_char* will have the effect of pressing the button. See the [B_BUTTON Discussion on page 5-4](#).

If *select_char* is not passed or is blank, the button will not have a quick-select character.

The window will expand downward or to the right to accommodate the buttons. This expansion may cause your window to extend beyond the bottom or right side of your application window. Also, the button will be a standard size. Therefore, text that exceeds the size of the button will not be visible.

Toolkit button graphics

UI Toolkit buttons support the following graphics formats:

- ▶ BMP—Windows bitmap
- ▶ GIF—Graphics Interchange Format
- ▶ JPEG—Joint Photographic Experts Group (stored in JPEG File Interchange Format, JFIF)
- ▶ Exif—Exchangeable Image File
- ▶ PNG—Portable Network Graphics
- ▶ TIFF—Tag Image File Format

If an image is too large for the button face, the image will be cropped; Toolkit won't scale the image to fit the button. If Toolkit can't find the file, or if it isn't one of the supported file types, Toolkit will leave the button face blank.

On Windows versions prior to Windows 8, you can make button images “transparent”—that is, you can specify a color to be replaced by the button face color, which can vary from user to user. To do this, set the SYN_TRANSPARENT_COLOR environment variable and, if necessary, the SYN_TRANSPARENCY_THRESHOLD environment variable.

- ▶ SYN_TRANSPARENT_COLOR. Set this environment variable to the color you want replaced. Set it to an RGB triplet using hexadecimal values prefixed with 0x. For example, if your button images are yellow figures on a gray (RGB 192, 197, 202) background, setting SYN_TRANSPARENT_COLOR=0xC0,0xC5,0xCA (the equivalent of 192, 197, 202) will cause the background of the button images to take on the color of the button face. That gray will never appear in a button image unless it happens to be the button face color. For more information, see [SYN_TRANSPARENT_COLOR](#) in the “Environment Variables” chapter of *Environment Variables & System Options*.
- ▶ SYN_TRANSPARENCY_THRESHOLD. Use this environment variable to specify how much a color can vary from the SYN_TRANSPARENT_COLOR setting and still be treated as transparent. Typically, you don't need to set this unless you're using JPEG graphics, which use a lossy compression algorithm that may alter colors in the image—or unless you don't know the exact background color. If you do set this, set it to a hexadecimal value with the prefix 0x.

Toolkit adds this value to and subtracts it from each of the RGB values specified for `SYN_TRANSPARENT_COLOR` to establish upper and lower limits for transparency. For example, if you specify the following, Toolkit will treat any color from RGB B0, B5, BA to RGB D0, D5, DA as transparent.

```
SYN_TRANSPARENT_COLOR=0xC0, 0xC5, 0xCA  
SYN_TRANSPARENCY_THRESHOLD=0x10
```

For more information, see [SYN_TRANSPARENCY_THRESHOLD](#) in the “Environment Variables” chapter of *Environment Variables & System Options*.

When a button with a graphic is disabled (grayed out), the transparent portions of the graphic remain transparent. In other words, they retain the button face color.

`.BUTTON` has no effect on applications running on UNIX or OpenVMS. Script does, however, save button information on these platforms, so if the window is loaded on a Windows platform, the button will be displayed (assuming ISAM-compatible platforms).

See also

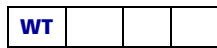
- ▶ [.BUTTON_SET](#) on page 2-17 for more information about specifying the placement of button sets.
- ▶ [IB_BUTTON](#) on page 8-88 for more information about specifying buttons for an input window built at runtime.
- ▶ [B_BUTTON](#) on page 5-4 for more information about specifying buttons in a window or tab set.
- ▶ [L_BUTTON](#) on page 9-55 for more information about specifying buttons on a list.
- ▶ [%B_DISABLE](#) on page 5-16 and [%B_ENABLE](#) on page 5-17 for more information about disabling and enabling buttons.

Examples

This example displays an OK button that signals “I_OK,” a Cancel button that signals “O_ABANDON,” and a Help button that calls `HELP_METHOD` when pressed. The OK button has the quick-select character “K.” When ALT+K is pressed, a click of the OK button is simulated.

```
.button I_OK, text("OK"), select(K)  
.button O_ABANDON, text("Cancel")  
.button help, text("Help"), method("HELP_METHOD")
```

.BUTTON_SET – Define the placement of a set of buttons



```
.BUTTON_SET [BOTTOM (buttons_per_row) | RIGHT (buttons_per_column) ] –  
            [, justification][, DEFAULT (button_name) ]
```

Arguments

BOTTOM

(optional) Places the buttons at the bottom of the window with the maximum number of buttons per row specified.

buttons_per_row

The maximum number of buttons to display per row. (default)

RIGHT

(optional) Places the buttons at the right of the window with the maximum number of buttons per column specified.

buttons_per_column

The maximum number of buttons to display per column.

justification

(optional) Specifies how the button set is aligned in a row at the bottom of the window or a column.

BEGINNING Justify to the beginning of the row or column. (default)

CENTER Justify to the center of the row or column.

END Justify to the end of the row or column.

DEFAULT

(optional) Specifies the default button.

button_name

The name of the default button.

Discussion

You can use the .BUTTON_SET command to define the placement of a set of buttons in a window. It can be specified before or after the .BUTTON commands.

The default button set placement is BOTTOM.

If *buttons_per_row* or *buttons_per_column* is not specified, the maximum number of buttons that will fit on a row or column of the window will be displayed before a new row or column is created.

If the `E_STATE` option `D_RETURNBTN` is set, the default button will be selected if you press `RETURN`, no matter what field you're on in the window. If *button_name* is not specified and `D_RETURNBTN` is set, the first button in the window is the default.

If a tab set container and its contained window both have buttons, the following rules are used to determine which button is the default:

- ▶ If only one, either the contained window or the tab set container, defines a default button, the specified button is the default button.
- ▶ If neither the contained window nor the tab set container defines a default button, the first button on the contained window is the default button.
- ▶ If both the contained object and the tab set container define a default button, the default button of the contained window is the default button.

The window will expand downward or to the right to accommodate the buttons. This expansion may cause your window to extend beyond the bottom or right side of your application window.

`.BUTTON_SET` has no effect on applications running on UNIX or OpenVMS. Script does, however, save button information on these platforms, so if the window is loaded on a Windows platform, the button will be displayed (assuming ISAM-compatible platforms).

See also

- ▶ [.BUTTON on page 2-13](#) for more information about including buttons in a window.
- ▶ [IB_BUTTONSET on page 8-90](#) for more information about specifying button placement for an input window built at runtime.
- ▶ [B_BUTTONSET on page 5-14](#) for more information about specifying button placement for a window or tab set.
- ▶ [L_BUTTONSET on page 9-62](#) for more information about specifying button placement for a list.
- ▶ [DTS_BUTTONSET on page 13-17](#) for more information about specifying button placement on a tab set. (`DTS_BUTTONSET` is a subfunction of `%TS_TABSET`.)
- ▶ [E_STATE on page 7-41](#) for more information about the `ENTER` key selecting the default button.

Examples

This example shows OK, Cancel, and Help buttons centered on the right side of an input window.

```
.button I_OK, text("OK")  
.button O_ABANDON, text("Cancel")  
.button O_HELP, text("Help")  
.button_set right, center
```

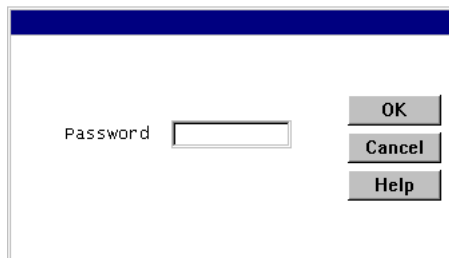


Figure 2-1. Buttons on an input window.

.COLUMN – Begin a menu column definition



```
.COLUMN name, text[, justification][, DISABLE][, SELECT (character) ]
```

Arguments

name

The name of the column (a maximum of 15 characters).

text

A quoted or alphanumeric string that contains the text of the column's heading.

justification

(optional) Designates how the column is aligned beneath the heading.

LEFT Align the left sides. (default)

RIGHT Align the right sides.

CENTERED Center the column and heading.

DISABLE

(optional) Specifies that when the column is placed on the menu bar, it will be disabled.

SELECT

(optional) Specify a quick-select character.

character

A single, printable quick-select character.

Discussion

The .COLUMN command marks the beginning of a new menu column definition. This column can be used either as a primary or a submenu column.

Name is case insensitive, and must be a unique window name within the window library.

Text is the text that will be displayed on the menu bar. Upper and lowercase will appear as entered. *Text* is required for submenu column definitions, but is currently ignored.

Justification defaults to LEFT if you do not specify LEFT, RIGHT, or CENTERED. If you specify RIGHT justification, the right sides of the column and heading will be aligned. If you specify CENTERED justification, the column will be centered beneath the heading. (See [figure 2-2](#).) When the column is being used as a submenu, the *justification* setting will be ignored. On Windows, *justification* has no effect.

Menu column placement is limited by available screen space.

If you specify `DISABLE`, you can later re-enable the column with the `M_ENABLE` subroutine.

If you do not specify `SELECT(character)`, UI Toolkit uses the first non-blank character of text. A column's quick-select character will not work if any menu column is dropped down, which sometimes happens automatically. You can use the `M_DEFCOL(0)` subroutine to disable automatic drop-down.

The width of the column's heading is two spaces larger than the length of text, with the text beginning at the second column. If you want the heading text to have more than one leading or trailing space, you can include spaces in the quoted text string.

To create a submenu column, define the column and the parent column (the menu from which the submenu will cascade) as you would any other, but in the parent menu column's definition, use the `SUB` argument for the menu entry that will cause the submenu to be displayed. And, for the same menu entry, use the submenu column's name (the *name* argument for `.COLUMN`) as the entry name (the *name* argument for `.ENTRY`). See Examples below.

See also

[.ENTRY on page 2-27](#) for more information about defining column entries.

Examples

This example generates a new menu column named **util** whose text in the menu bar is "General utilities," with an extra leading space. The column is left-aligned beneath the heading by default.

```
.column util, "General utilities"
```

The column generated by the second example is named **receive**; the text entry in the menu bar is "Receivables." The column is centered beneath the heading.

```
.column receive, Receivables, centered
```

This example creates a menu column **primary** that references a submenu column **submenu**.

```
.column primary, "Primary"
.entry submenu, "Submenu", sub
.end

.column submenu, "Submenu"
.entry subent1, "Sub entry 1"
.entry subent2, "Sub entry 2"
.end
```

.DATA – Load data into a user data set field

WT	WN	U	V

`.DATA field_number, data`

Arguments

field_number

Either the decimal value of the user data set field or one of the following options:

FIRST

LAST

NEXT

PREVIOUS

CURRENT

data

The data to be loaded into the user data set field.

Discussion

A user data set is a one-dimensional alpha array that is associated with a window (rather than your program) and is maintained within the window data area by the window system. Its parameters and contents are completely defined by the user.

The `.DATA` command loads data into a user data set field.

Use the Synergy DBL `W_FLDS` subroutine to process user data sets. See [W_FLDS](#) in the “Synergy Windowing API” chapter of the *Synergy DBL Language Reference Manual* for more information.

If *data* is longer than one word, it must be enclosed in quotation marks.

Examples

This window has a copy of its text in another language. At runtime, the program can use either the English or the Spanish version.

```
.window nofile, 3, 30
.text nofile, pos(2,1)
Cannot find file.
.user_define 1, 30
.data 1, "No puedo encontrar el archivo."
.end
```

.DATE_GENERATED – Define the date the script file was generated



```
.DATE_GENERATED "date"
```

Arguments

date

The date that the **scrcnvrt** conversion utility or Composer generated the script.

Discussion

This command is generated by UI Toolkit and appears immediately following the .VERSION command. This command is ignored by Toolkit when reading scripts; therefore, it is not required.

Examples

This script was generated by Composer on November 25, 2005.

```
.date_generated "25-NOV-2005"
```

.DISPLAY_AREA – Define the display area

WT	WN	U	V

`.DISPLAY_AREA row, column, height, width`

Arguments

row

The row position of the upper-left corner of the display area (relative to the upper-left corner of the window, not the display screen).

column

The column position of the upper-left corner of the display area (relative to the upper-left corner of the window, not the display screen).

height

The number of rows deep the display area will be.

width

The number of columns wide the display area will be.

Discussion

The `.DISPLAY_AREA` command defines the dimensions and placement of the display area for a general window (`.WINDOW`) or input window (`.INPUT`). The display area defines the view into a window and defaults to the entire window.

The view is movable if the display area is smaller than the size of the window. If the window is an input window, the input processing system adjusts the location if necessary to include the entire input field being processed.

Row 1, column 1 is the upper-left corner.

See also

- ▶ [“Input fields” on page 8-3.](#)
- ▶ [.PLACEMENT on page 2-89](#) for considerations of display area size.
- ▶ [“Notes and Tips for ActiveX Toolkit Lists” on page 9-15](#) for information on setting the display area for an input window that is associated with an ActiveX Toolkit list.

Examples

The following .DISPLAY_AREA command defines a movable view for the **realbig** window, which is larger than the screen. The upper-left corner of the display area is at window row 3, column 1, and the area is 16 rows deep by 78 columns wide.

```
.input realbig, 48, 78  
.display_area 3, 1, 16, 78
```

.END – End a window definition

WT	WN	U	V

.END

Discussion

The .END command terminates the current window definition.

All commands and text between the .WINDOW, .COLUMN, .INPUT, .SELECT, or .LISTCLASS command and the .END command define the window definition. This command must be present, following all other script commands for a window or column. When .END is processed by the Script compiler, the window or column is constructed and stored in a window library.

.ENTRY – Define a column entry



```
.ENTRY name, text [, justification] [, DISABLE] [, KEY (shortcut) | SUB] [, NORESET] –  
    [, SELECT (character) ] [, USER (string) ]
```

Arguments

name

The entry name (a maximum of 10 characters).

text

An alphanumeric or quoted string that contains the text of the column entry.

justification

(optional) Designates how the entry text is aligned within the column.

LEFT Align the left sides. (default)

RIGHT Align the right sides.

CENTERED Center the entry text.

DISABLE

(optional) Specifies that when the column is placed on the menu bar, the entry will be disabled.

KEY

(optional) Specifies that a shortcut key follows.

shortcut

The shortcut key or key sequence.

SUB

(optional) Specifies that a submenu column is associated with the entry.

NORESET

(optional) Keeps the menu from being reset when the entry is selected. (The column and menu stay dropped down.) On Windows, NORESET has no effect.

SELECT

(optional) Specify a quick-select character.

character

A single, printable quick-select character.

USER

(optional) Specifies that a user text string follows.

string

A user-defined text string to be associated with the entry (a maximum of 80 characters).

Discussion

The .ENTRY command generates a new entry within a primary menu or submenu column. You can define up to 20 menu entries per menu column.

Name is case insensitive. Don't use names that begin with "U_" unless you want such entries to access the EUTILS_METHOD subroutine. (See [chapter 7, "Environment Routines,"](#) for more information.) Also avoid entries that begin with "E_", "S_", "T_", or "I_" (other than the reserved entries); some subroutines will generate errors if they encounter unrecognized entries of these types. The name O_HELP is reserved for the EHELP_METHOD subroutine. See ["Appendix B: Reserved Menu Entries"](#) for a list of reserved menu entries.

The first character of *text* is loaded at the second character position within the entry's display if the entry is left-justified. You can define additional leading or trailing spaces by explicitly including them in the quoted string. If required, the overall width of the column is increased so that at least one trailing blank follows *text*.

Justification defaults to LEFT if you do not specify RIGHT or CENTERED. If you specify RIGHT, the right side of the column and this entry's text will be aligned. If you specify CENTERED, this entry will be centered within the column. (See [figure 2-2.](#)) On Windows, you should always use LEFT justification for menu entry text. RIGHT and CENTERED will not operate as expected because Windows uses a proportional font rather than a fixed font for menu column text.

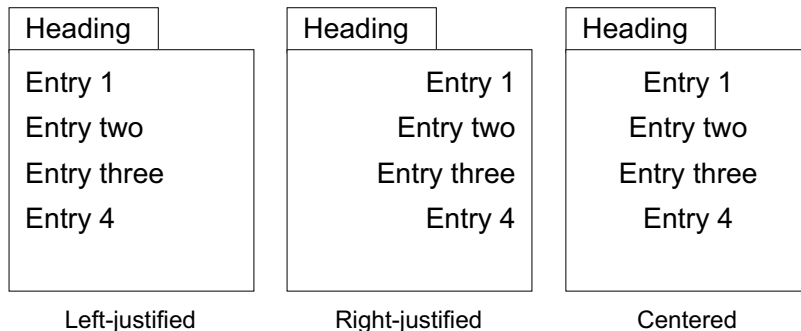


Figure 2-2. Entry justification.

If you specify DISABLE, you can re-enable the entry with the M_ENABLE subroutine later.

NORESET will keep the menu column displayed after a selection has been made. This is useful if you want to do additional processing while the menu is still dropped down. Note, however, that if NORESET is specified, it is the responsibility of the application program to reset the menu with the

M_RESET subroutine. If you do not specify NORESET, the menu will be reset to the disabled menu bar after the user makes a selection. Placement or removal of any menu columns will also automatically reset the menu bar.

KEY(*shortcut*) defines the entry's shortcut. If *shortcut* is defined in the key mapping script file (**keymap.ctl** by default), it must have the form used there. For example, if the key mapping script file has an entry f42=alt_s("Alt-S"), you can pass only KEY(alt_s) for function 42. If it isn't defined in the key mapping script file, you can use the following forms for *shortcut*:

- Fnn** where *nn* is a numeric value from 1 through 63. This form assigns the specified function key as the shortcut for the entry, and can be used only when the **keymap.ctl** file doesn't have a corresponding entry. (On Windows, every function from F11 to F63 is already defined in **keymap.ctl** as distributed.)
- ^x** where *x* is an alphabetic value. This form specifies that a control character is the shortcut for the entry (^A for CTRL+A, ^T for CTRL+T, and so forth). ^M is not valid as a shortcut because it is a carriage return. (There are other keys you should avoid using. See ["Customizing reserved keys" on page 18-22.](#))
- LEFT** Specifies the LEFT arrow key as the shortcut for the entry. Can be abbreviated to L.
- RIGHT** Specifies the RIGHT arrow key as the shortcut for the entry. Can be abbreviated to R.
- UP** Specifies the UP arrow key as the shortcut for the entry. Can be abbreviated to U.
- DOWN** Specifies the DOWN arrow key as the shortcut for the entry. Can be abbreviated to D.

The KEY qualifiers are case insensitive (^e is the same as ^E, f3 is the same as F3, and "left" is the same as "LEFT"). If KEY(*shortcut*) is not present, no shortcut is assigned.

If you use KEY(*shortcut*) for an entry in a menu column, the width of the column is a combination of the width of the longest text for a menu shortcut and the width of the longest menu entry in the column.



You can add, change, and delete definitions in **keymap.ctl** to make KEY(*shortcut*) definitions more readable. For example, if you want to use the CTRL+P key combination as a shortcut for a menu entry on Windows, you'll need to pass KEY(alt_p) if you're using the distributed version of **keymap.ctl**. (This is because, for historical reasons, the **keymap.ctl** file includes the definition ^p=alt_p.) If you want to instead pass KEY(^p) to make your code more readable, just delete the ^p=alt_p definition.

If you want to pass an identifier for *shortcut* that isn't one of the forms listed above, you can create a definition in **keymap.ctl** for this identifier. For example, if you want to pass KEY(print) to specify the CTRL+P key combination, just create the definition ^p=print in **keymap.ctl**. See ["Modifying shortcut identifiers for menu entries defined in a script" on page 18-19](#) for information.

By passing SUB, you can cause the menu entry to display a submenu column. (Note that you must also pass the submenu column's name as the entry's name. See the [.COLUMN Discussion on page 2-20](#) for more information.) If you specify SUB, the KEY(*shortcut*) argument is ignored.

If you do not specify SELECT(*character*), the quick-select character defaults to the first non-blank character of text.

On Windows, quick-select characters are always designated by an underscore (“_”).

Quick-select characters, shortcuts, and disabling/enabling are discussed further in the “Introduction to Menus” section of the “Menu Subroutines” chapter.

USER(*string*) enables you to specify a user-defined text string. If you pass it, you can

- ▶ use the M_USER subroutine to retrieve the user string specified by any .ENTRY on a primary menu column.
- ▶ test the **g_entuser** field to find out the last .ENTRY the user selected on a primary menu column.

UI Toolkit includes several reserved menu entries that perform scrolling and editing functions (see [“Appendix B: Reserved Menu Entries”](#)). You must specify these entries in your script file (with .ENTRY) to include them on a menu column.

See also

[M_USER on page 10-58](#) for more information on retrieving the USER(*string*) for a primary menu entry.

Examples

The first example generates a new entry named **o_help**. The text that appears in the menu column for this entry is HELP, F1 is the shortcut, and “H” is the quick-select character by default.

```
.entry o_help, Help, key(F1)
```

The new entry generated by the second example is named **gl**. The text “General Ledger” is centered beneath the heading, and when the column is placed on the menu bar, the entry will be disabled. The quick-select character for this entry is the letter “G” by default.

```
.entry gl, "General Ledger", centered, disable
```

The third example generates an entry named **ap**, whose entry text is “Accounts Payable.” The quick-select character is the letter “P”. The column stays on the screen when the **ap** entry is selected and a submenu named **ap** is displayed. (For example, the submenu might contain the choices “Maintenance,” “Processing,” and “Reports.”)

```
.entry ap, "Accounts Payable", select(p), sub
```

.FIELD – Define an input window field



```
.FIELD field_spec, [type] –
    [, ALLOW (valid [, ...]) ][, NOALLOW][, ARRIVE_METHOD (arrive_method) ] –
    [, NOARRIVE_METHOD][, AUTOMATIC][, NOAUTOMATIC] –
    [, BLANK_IFZERO][, NOBLANK_IFZERO][, BLINK][, NOBLINK] –
    [, BOLD][, NOBOLD][, BREAK ([CHANGE | ALWAYS | RETURN]) ][, NOBREAK] –
    [, CHANGE_METHOD (change_method) ][, NOCHANGE_METHOD][, CHECKBOX] –
    [, NOCHECKBOX][, COLOR (palette#) ][, NOCOLOR][, COPY] –
    [, DATE ([TODAY][, PERIOD][, SHORT]) ][, NODATE][, DECREMENT] –
    [, DEFAULT (default) ][, NODEFAULT][, DISABLED][, NODISABLED | ENABLED] –
    [, DSP_LENGTH (display_length) ][, NODSP_LENGTH] –
    [, DISPLAY_METHOD (display_method) ][, NODISPLAY_METHOD] –
    [, DRILL_METHOD (drill_method) ][, NODRILL_METHOD] –
    [, EDITFMT_METHOD (editfmt_method) ][, NOEDITFMT_METHOD] –
    [, ENUMERATED (display_length [, base][, step]) ][, NOENUMERATED] –
    [, FIELD_POSITION ([CHANGE, row, column]) ][, NOFIELD_POSITION] –
    [, FONT (palette_name) ][, NOFONT][, FORMAT (fmt_string) ][, NOFORMAT] –
    [, HELP (help_id) ][, NOHELP][, HIGHLIGHT][, NOHIGHLIGHT] –
    [, HYPERLINK_METHOD (hyperlink_method) ][, NOHYPERLINK_METHOD] –
    [, INCREMENT][, INFORMATION (info_string) ][, NOINFORMATION] –
    [, INPUT_LENGTH (input_length) ][, NOINPUT_LENGTH][, ITALIC][, NOITALIC] –
    [, JUSTIFY (LEFT | RIGHT | CENTER) ][, NOJUSTIFY] –
    [, LEAVE_METHOD (leave_method) ][, NOLEAVE_METHOD] –
    [, MATCH ([CASE | NOCASE][, EXACT | NOEXACT]) ] –
    [, NEGATIVE (ALLOWED | ONLY | ORZERO) ][, NONEGATIVE] –
    [, NODECIMAL][, DECIMAL_REQUIRED] –
    [, NOECHO ([display_character]) ][, ECHO][, NOTERM][, TERM] –
    [, PAINT (character) ][, NOPAINT][, POSITION ([CHANGE, row, column]) ] –
    [, NOPOSITION][, PROMPT (prompt) ][, NOPROMPT][, NOPROMPTFONT] –
    [, RADIO][, NORADIO][, RANGE (minimum, maximum) ][, NORANGE][, READONLY] –
    [, NOREADONLY][, REQUIRED][, NOREQUIRED][, RETAIN_POSITION] –
    [, NORETAIN_POSITION][, REVERSE][, NOREVERSE] –
    [, SELECT (srow, scolumn, [height], entry [, ...]) ][, NOSELECT] –
    [, SELWND (srow, scolumn, window) ][, NOSELWND [ , TIME ([AMPM][, NOW]) ] ] –
    [, NOTIME][, TIMESTAMP ([AMPM][, SHORT]) ][, NOTIMESTAMP] –
    [, UNDERLINE][, NOUNDERLINE][, UPPER_CASE_ONLY | UC] –
    [, NOUPPER_CASE_ONLY | NOUC | LOWER_CASE_ALLOWED] –
    [, USER ([user_text][, user_type]) ][, NOUSER][, VIEW_LENGTH (view_length) ] –
    [, NOVUEW_LENGTH][, WAIT (IMMEDIATE | FOREVER | GLOBAL | n) ]
```

Arguments

field_spec

The field specification for the input window field. See “[Field specifications](#)” on page 2-35 for information.

type

The data type and size, specified *only* for local fields, which must have one of the following formats:

A <i>size</i>	Alpha type of <i>size</i> .
<i>dimension</i> A <i>size</i>	Alpha array of <i>dimension</i> elements of <i>size</i> .
D <i>size</i>	Decimal type of <i>size</i> .
D <i>size.decimal</i>	Implied-decimal type of <i>size</i> with <i>decimal</i> places.
I <i>size</i>	Integer type of <i>size</i> .

Where *size* is the size of the input field, *dimension* is the number of dimensions for the array (only single-dimension alpha arrays are supported), and *decimal* is the number of digits after the decimal point. (See the Discussion below for information on maximums.)

QUALIFIERS

See “[.FIELD qualifiers](#)” on page 2-36.

Discussion

The .FIELD command defines a field to be included in the window’s input set. The maximum number of input fields in an input window or input set is 253.

Fields can be placed anywhere in the input window. On exit, the current window position is updated to the next horizontal character following the input field, including the display width.

Default values will be used for all options that are not specified.

A field with the form *dimension***A***size* (a single-dimension alpha array) is displayed as a multi-line text field. *Dimension* determines the number of lines in the field and *size* determines the width of the field. For example, **4a50** designates a four-line, 50-column text field, which is technically four 50-column alpha fields stacked vertically. (Note, however, that on Windows, these stacked fields are enclosed in a single frame and are a single Windows control.) Wrap mode is set automatically for input, and a text field can be edited with any editing function. Note that *dimension***A***size* is an array declaration (unlike an *index* specification, which merely references an element in an existing array defined in a repository).

On a decimal or implied-decimal field, the decimal point can either be explicitly entered or implied during input. If you specify that a field is of type **d** and has a decimal value of 2 (the number of decimal places), an input of 55 will yield “55.00”, while an input of 8.7 will yield “8.70”. If you specify the NODECIMAL qualifier, which means the user doesn’t need to type the decimal point

(see [NODECIMAL](#), [DECIMAL_REQUIRED](#) on page 2-57), the entry of 55 will yield “.55”. (The entry 8.7 will still yield “8.70”.) Note that whether a decimal field is defined in Repository, with .FIELD, or with IB_FIELD, the field can have no more than nine digits after the decimal point.

Implied-decimal fields will always display their full precision unless the field is formatted otherwise. Any explicit spacing may be overwritten if the field needs it to display the full precision. (See the FORMAT option for information about formatting the field’s display.)

Alpha fields and single-dimension alpha array fields can have a *size* of up to 65,535 characters. If you specify a *size* greater than 65,535, Toolkit will generate an error. Decimal fields (including implied decimal fields) have a maximum size of 28, and for implied-decimal fields, up to 28 digits can follow the decimal point. On any operating system, integer fields can have 1, 2, 4, or 8 digits, depending on the size of the integer field. Integer fields support all qualifiers that are valid for decimal fields without precision.

Using .FIELD with .REPOSITORY_STRUCTURE

If you specify a repository structure (using [.REPOSITORY_STRUCTURE](#)), note the following:

- ▶ You must call .REPOSITORY_STRUCTURE prior to the .FIELD commands so that fields can be drawn from that repository structure.
- ▶ All fields defined after a .REPOSITORY_STRUCTURE command must exist in that repository.
- ▶ Repository fields that have the “Excluded by Toolkit” flag set, will not be accessible by Script (or Composer or the IB_FIELD subroutine). By default, this flag is not set.
- ▶ You cannot specify *type*, *size*, *dimension*, or *decimal* places for fields drawn from a repository.
- ▶ For field qualifiers that are not explicitly specified, UI Toolkit uses the default values defined in the repository. If neither option from an option pair has been specified in the repository, the Toolkit default applies.
- ▶ If you wish to override the default values (whether they are repository defaults or Toolkit defaults), use the qualifiers. See [“.FIELD qualifiers” on page 2-36](#).
- ▶ When you include time and date fields from the repository, you must specify any desired time and date formats in your script files. UI Toolkit does not import repository formatting for time and date fields. For timestamp fields, however, there is a default display format, which you can change or override,
- ▶ If you are drawing an input field’s allow list or selection list from a repository, any quoted entries (single or double quotes) will be unquoted, unless you have enclosed the entry and the quotes in the opposite type of quotes—for example, for “Allow” use ““Allow”” (double quotes enclosing an entry with single quotes) in your repository definition, and for ‘Allow’ use “‘Allow’” (single quotes enclosing an entry with double quotes).
- ▶ Multiple repository structures can be combined together enabling access as one structure by using the implicit group feature of Repository. For more information, see [“Basic field information”](#) in the “Working with Fields” chapter of the *Repository User’s Guide*.

- ▶ For repository arrayed fields, use the *index* argument to access individual elements. See “[Field specifications](#)” below.
- ▶ For fields that are group members, you must use the group prefix unless the field name is unique within the structure. See “[Field specifications](#)” below.
- ▶ For fields that are members of a structfield, the field name must be unique within the repository structure.

Renditions

Specifying the BLINK, BOLD, HIGHLIGHT, ITALIC, REVERSE, UNDERLINE, or COLOR qualifier for a field overrides the corresponding rendition set with .FILL for the window. Specifying NOBLINK, NOBOLD, NOHIGHLIGHT, NOITALIC, NOREVERSE, NOUNDERLINE, or NOCOLOR causes the corresponding window rendition that’s been set by .FILL to be used. (See [.FILL on page 2-73](#) for information on specifying these renditions for a window.)

Renditions specified with .FIELD override user-controlled renditions for the field (renditions set with Synergy UI Toolkit Control Panel or Windows Control Panel). If a rendition is overridden for a field, the default Toolkit behavior is no longer used for the field. This means that no rendition is applied when the field is being processed, displayed, or cleared, unless done so programmatically with I_FLDMOD, %IDISPLAY_METHOD, or %IEDIT_FMTMETHOD.

Note the following:

- ▶ For all edit controls, renditions set with .FIELD affect only text within the field’s frame.
- ▶ For check box fields on UNIX or OpenVMS, only the check box area is affected. On Windows, no area of a check box field is affected.
- ▶ For combo boxes on Windows, text within the field’s frame is affected. The actual list is affected only by UNDERLINE, ITALIC, and BOLD.
- ▶ For radio buttons on Windows, the area occupied by the entries’ text is affected. The buttons themselves are not.
- ▶ Disabled fields on Windows always have gray text. Specifying a foreground color for a disabled field has no effect.
- ▶ Modifying a rendition for a field affects only that rendition. Other renditions, including color, remain the same.
- ▶ For ActiveX® Toolkit lists, field attributes and color have precedence over any specified alternating background colors for the list.
- ▶ When using Toolkit lists (non-ActiveX lists) on Windows, the field’s rendition isn’t updated until the field is focused.
- ▶ It may be best to use different renditions for different platforms or when using different fonts.

See “[Customizing the Look of Your Application](#)” on page 18-24 for more rendition information.

Field specifications

Field specifications have the following form and cannot exceed 30 characters:

```
[struct_alias.]field_name[ index, ... ]
```

where *struct_alias* is the alias for a structure (used only after the input window has been created), *field_name* is the name of the field (which may include a group prefix), and *index* is an index for a dimension of a repository array field. (*Index* specifications must be enclosed in square brackets.) *Struct_alias*, *index*, and group prefixes are valid only for fields defined in a repository.

Struct_alias is necessary only after the input window has been created and must be followed by a period. (When using .FIELD, .SET, IB_FIELD, or IB_SET, the .REPOSITORY_STRUCTURE specification or the IB_RPS_STRUCTURE call determines the structure used for the field.)

Index specifications enable you to access individual elements of a repository array field. For multi-dimensional arrays, pass the same number of *index* specifications as there are dimensions, and pass them in the same order as the corresponding dimensions. For example, for a two-dimensional array named **total**, specify the following to use the seventh field (second dimension) of the third element (in the first dimension):

```
.FIELD total[3,7]
```

The name of the resulting input field will include brackets, indices, and separating commas, but not spaces, tabs, or leading zeros. For the field definition above, for example, this would be total[3,7], so to use I_DISABLE for this field, you could pass something like the following:

```
xcall i_disable(wndid, "total[3,7] ")
```

For single-dimension alpha arrays, you can use *index* to specify a particular field in the array, or you can omit *index* and the square brackets. (If you omit *index*, you must also omit the square brackets.) Omitting *index* instructs Toolkit to display the entire array as a text field. The number of elements determines the number of lines in the text field, and the field size for the array becomes the length of the text field. Wrap mode is set automatically during input, and a text field can be edited with any editing function.

Script will generate an error in the following circumstances:

- ▶ If an *index* value is less than 1 or greater than the number of elements specified for that dimension.
- ▶ If the number of *index* specifications does not match the number of dimensions. (The exception is single-dimension alpha arrays. For these, *index* is optional.)

For fields that are group members in a repository, use the group prefix (if one has been specified for the group) to make the field name unique—for example, if the group prefix is **gr2** and the field name is **lastname**, you can use **gr2lastname**. (You can omit the group prefix only if the name is unique within the entire structure.)

For example, the following *field_spec* references the second element in **address**, a single-dimension array which is a member of a group with the prefix **gr1**:

```
.REPOSITORY_STRUCTURE customer  
.FIELD gr1address[2]  
.SET custset, customer, gr1address[2]
```

If the **customer** structure in the above example has the alias **cust**, you could use the following to disable the field after the input window had been created:

```
xcall i_disable(wndid, "cust.gr1address[2]")
```

Note that for arrayed groups, you can use a field name (with or without a group prefix) only to access a member from the first element in the array. You can use *index* specifications to access an arrayed group element as a whole, but you cannot use *index* to access individual members within an arrayed group element.

.FIELD qualifiers

All .FIELD qualifiers are optional. Separate qualifiers with a comma. For all qualifiers, except ALLOW and SELECT, you can break a line only between qualifiers, not in the middle of a qualifier and its arguments.

In most cases, text for qualifier parameters must be enclosed in quotation marks, but there are some exceptions, which are noted. For example, the following includes two qualifiers (ALLOW, and ARRIVE_METHOD) with quoted parameter strings:

```
.field myfield, a3, allow("yes", "no"), arrive_method("mymethod")
```

ALLOW, NOALLOW

```
ALLOW(valid[, ...])  
NOALLOW
```

ALLOW specifies which entries are valid for the field. You can specify up to 100 entries with *valid*.

For example, if “yes” and “no” were the only possible entries, the syntax for the .FIELD command would be as follows:

```
.field yesno, a3, allow("yes", "no")
```

If you don’t specify the MATCH qualifier and the allowable entry is alpha, the input system will only look for a match for the shortest entry, and there will be no case sensitivity. You can change these match characteristics by specifying the MATCH qualifier. (See [MATCH on page 2-56](#) for more information.)

NOALLOW is the default state if neither ALLOW nor NOALLOW is specified in the script or in the repository. If ALLOW has been specified in the repository, NOALLOW will cancel the allow list associated with the field.

ARRIVE_METHOD, NOARRIVE_METHOD

ARRIVE_METHOD (*arrive_method*)
NOARRIVE_METHOD

ARRIVE_METHOD specifies a pre-input subroutine, where *arrive_method* is the name of the subroutine to invoke. *Arrive_method* is case insensitive and is valid with or without quotation marks.

For example,

```
.field field,...,arrive_method("mymethod")
```

If you specify ARRIVE_METHOD, the *arrive_method* subroutine will be called before the field is processed by I_INPUT. Up to 20 optional data arguments can be passed to I_INPUT. These optional data arguments are then, in turn, passed to the *arrive_method* subroutine.

The routine specified in *arrive_method* must be callable as a subroutine and must reside either in the program image itself or in an open ELB. In the latter case, on some systems a call to the OPENELB routine is required. See [IARRIVE_METHOD on page 8-114](#) for more information on writing the *arrive_method* subroutine.

NOARRIVE_METHOD is the default state if neither ARRIVE_METHOD nor NOARRIVE_METHOD is specified in the script or in the repository. If ARRIVE_METHOD has been specified in the repository, specifying NOARRIVE_METHOD will override this setting and the arrive method will not be invoked.

AUTOMATIC, NOAUTOMATIC

AUTOMATIC
NOAUTOMATIC

AUTOMATIC specifies that when an empty field is processed, a return with no input is automatically assumed (i.e., the specified default action—DEFAULT, COPY, INCREMENT, or DECREMENT—will occur automatically). In the following example, the **ordnum** field automatically contains the default value (which happens to be 1).

```
.field ordnum, d4, automatic, default(1)
```

NOAUTOMATIC is the default state if neither AUTOMATIC nor NOAUTOMATIC is specified in the script or in the repository. If AUTOMATIC has been specified in the repository, NOAUTOMATIC cancels automatic entry of the field.

BLANK_IFZERO, NOBLANK_IFZERO

BLANK_IFZERO
NOBLANK_IFZERO

BLANK_IFZERO (which may also be specified BZRO or BLANK_IF_ZERO) indicates that a numeric (**d** or **i**) field will be left blank if the user enters a value of 0. For example, the **amount** field below will be left blank if the user enters a 0 value.

```
.field amount, d5, bzro
```

NOBLANK_IFZERO (which may also be specified NOBZRO or NOBLANK_IF_ZERO) is the default state if neither BLANK_IFZERO, BZRO, nor BLANK_IF_ZERO is specified in the script or in the repository. If BLANK_IFZERO has been specified in the repository, specifying NOBLANK_IFZERO will override this setting and will display a zero on zero input.

BLINK, NOBLINK

BLINK
NOBLINK

BLINK specifies italicized text on Windows and blinking text on UNIX and OpenVMS. It is synonymous with ITALIC. For detailed information on BLINK and other input field renditions, see [“Renditions” on page 2-34](#).

In the following example, the contents of the field are italicized on Windows and blinking on UNIX and OpenVMS.

```
.field date_due, d8, blink
```

NOBLINK is the default state if neither BLINK nor NOBLINK is specified in the script or repository. If BLINK has been specified in the repository, NOBLINK cancels the blinking. NOBLINK is synonymous with NOITALIC.

BOLD, NOBOLD

BOLD
NOBOLD

BOLD specifies bold text on Windows and highlighted text on UNIX and OpenVMS. It is synonymous with HIGHLIGHT. For detailed information on BOLD and other input field renditions, see [“Renditions” on page 2-34](#).

In the following example, the contents of the field are bold.

```
.field balance, d18.2, bold
```

NOBOLD is the default state if neither BOLD nor NOBOLD is specified in the script or repository. If BOLD has been specified in the repository, NOBOLD cancels this attribute. NOBOLD is synonymous with NOHIGHLIGHT.

BREAK, NOBREAK

BREAK (/CHANGE | ALWAYS | RETURN/)
NOBREAK

BREAK triggers a break in input set processing on a field. Specify the type of break with one of the following:

CHANGE—Causes break processing to occur when the field is changed. (default)

ALWAYS—Causes break processing to occur on the field every time the user leaves the field.

RETURN—Causes break processing to occur on the field only when the field is changed or RETURN (ENTER) is pressed.

Sometimes you'll want to interrupt input set processing on a field, perform external operations, and then continue with the input set. The BREAK qualifier without an argument or with the CHANGE argument specifies that a break will occur after input to this field has been processed (that is, whenever the field's data is changed). If the ALWAYS argument is present, a break will occur regardless of whether or not input has been processed (for instance, even if the user merely leaves the field with the I_NEXT or I_PREV menu entry). If the RETURN argument is present, a break will occur when RETURN (ENTER) is pressed, but not when the field has been accessed with the I_NEXT or I_PREV menu entry (assuming the field was not modified).

The global variable **g_fldnam** specified in **tools.def** will contain the name of the field with the break specification.

The BREAK qualifier might be useful in specifying an inventory item in a sales transaction. You could break on the specification field, search for the item in inventory, display any pertinent information, and then continue with the rest of the input set.

In the following example, a break will occur after input from the **custno** has been processed:

```
.field custno, d6, break
```

NOBREAK is the default state if neither BREAK nor NOBREAK is specified in the script or in the repository. If BREAK has been specified in the repository, NOBREAK will cancel break processing for the field. To change to another form of break processing (ALWAYS, RETURN, or the default, CHANGE), specify BREAK with the desired optional qualifier.

On Windows, when used in conjunction with CHECKBOX, SELECT, SELWND, or RADIO, the BREAK qualifier will not trigger a return to the calling routine until you leave that field. For example, clearing a check box will not return from I_INPUT until another field is accessed. If, however, the E_STATE (D_VALSTCHG) option has been set, then a break occurs immediately on a change of state, and does not occur when leaving the field (unless the two are simultaneous).

CHANGE_METHOD, NOCHANGE_METHOD

CHANGE_METHOD (*change_method*)
NOCHANGE_METHOD

CHANGE_METHOD specifies a post-validation routine, where *change_method* is the name of the routine to invoke. *Change_method* is case sensitive and is valid with or without quotation marks. For example,

```
.field name, ..., change_method("mymethod")
```

The specified routine will be called after the field is validated by I_INPUT, L_INPUT, I_INPFLD, or L_INPFLD. (By default, validation will not occur until focus on the field is lost. Immediate validation can be specified for check boxes, radio buttons, and combo boxes by using the D_VALSTCHG option in E_STATE.) Up to 20 optional data arguments can be passed to I_INPUT and L_INPUT. These optional data arguments are then, in turn, passed to the *change_method* routine.

The routine specified in *change_method* must be callable as a function and must reside either in the program image itself or in an open ELB. In the latter case, on some systems a call to the OPENELB routine is required. See [%ICHANGE_METHOD on page 8-118](#) for more information on writing the *change_method* routine.

NOCHANGE_METHOD is the default state if neither CHANGE_METHOD nor NOCHANGE_METHOD is specified in the script or in the repository. If CHANGE_METHOD has been specified in the repository, specifying NOCHANGE_METHOD will override this setting and the change method will not be invoked.

CHECKBOX, NOCHECKBOX

CHECKBOX
NOCHECKBOX

CHECKBOX indicates that this field is to be displayed as a check box on Windows. On UNIX and OpenVMS, check boxes are one-character fields ("X" if non-zero, or space if zero).

On Windows, a check box is a small square that is either empty or has a check in it. Every check box displays one of these states, and the state is represented in the data area of input as a 1 or 0 for checked or unchecked, respectively.

At runtime, the spacebar can be used to toggle the state.

A numeric input window field can be designated a check box field by adding the script option "checkbox" to the .FIELD command. When this is done, a check box is created at the location of the field. Prompt text becomes the text associated with the check box and is always placed adjacent and to the right of the check box, regardless of any specified prompt position. You can use IB_FIELD with the D_FLD_CHECKBOX qualifier to create a check box field when creating windows at runtime, or you can create a check box using Composer. See the ["Designing Your User Interface"](#) chapter of *Getting Started with Synergy/DE* for more information on Composer.

A check box is implicitly an enumerated field, and therefore *cannot* be defined with the ENUMERATED qualifier.

Toggleing a check box does not, by default, cause the field to be immediately validated. Until the field loses focus, no Toolkit validation tests or change method occurs. Likewise, no leave and arrive methods will be invoked until focus is lost. In order for a field to be immediately validated, the D_VALSTCHG option of E_STATE needs to be set. See [E_STATE on page 7-41](#) for more information about this subroutine and the D_VALSTCHG option.

On Windows, if a check box has no prompt, there is no visual change when the check box gains focus. Focus is denoted by a dotted rectangle around the prompt and changes to the renditions for the prompt (if needed E_STATE options are set).

NOCHECKBOX is the default state if neither CHECKBOX nor NOCHECKBOX is specified in the script or in the repository. If CHECKBOX has been specified in the repository, specifying NOCHECKBOX will override this setting and the field will not be displayed as a check box.

COLOR, NOCOLOR

COLOR (*palette#*)
NOCOLOR

COLOR specifies the color palette entry to use for foreground and background colors for the text, where *palette#* is a palette number in the range 1 to 16. For example,

```
.field amount_due, d10.2, color(5)
```

For detailed information on COLOR and other input field renditions, see [“Renditions” on page 2-34](#).

NOCOLOR is the default state if neither COLOR nor NOCOLOR is specified in the script or in the repository. If a color has been specified in the repository, NOCOLOR specifies that the window’s color be used, or color palette number 1 if not specified.

COPY

COPY

COPY copies the value from the data area that corresponds to this field (as defined by the structure) if the field is empty. For example,

```
.field cusnam, a30, copy
```

If you use COPY, you should *not* use DECREMENT, INCREMENT, or DEFAULT. These four qualifiers are mutually exclusive.

If COPY has been specified in the repository, specifying NODEFAULT will cancel it. (See [DEFAULT, NODEFAULT on page 2-44](#) for more information.)

DATE, NODATE

DATE (/TODAY//, PERIOD//, SHORT/)
NODATE

DATE enables you to process dates by specifying one or more of the following:

TODAY—Defaults the date to the system date if the user presses ENTER in a blank date field.

PERIOD—Specifies that the date is a period. (A period is a user-defined unit within the range 1 to 99, typically representing an accounting period.)

SHORT—Shortens the display of the date from the normal 11 characters. A short period date (one with both the SHORT and PERIOD arguments) will have a display length of 5 characters. All other short dates will have a display length of 8 characters.

You can use this qualifier only on **d4** through **d8** fields, or on **i2** or **i4** fields; fields of any other type or size will generate an error. The input for a date is either numeric or delimited. The size of the field defines the way the date is stored, as shown in the table below.

Date Formats for Field Sizes		
Size	Date format	Meaning
d4 field	YYPP	Two-digit year, period
d5 field	YYJJJ	Two-digit year, Julian day
d6 field	YYMMDD or YYYYPP	Two-digit year, month, day or Four-digit year, period
d7 field	YYYYJJJ	Four-digit year, Julian day
d8 field	YYYYMMDD	Four-digit year, month, day
i2 field	YYPP	Two-digit year, period
i4 field	YYYYMMDD	Four-digit year, month, day

Users don't have to type just digits—they can type a three-letter abbreviation for the month, and UI Toolkit will convert what was typed to one of the storage formats shown in the table above. (The U_MONTHS subroutine determines the abbreviation for each month. The default is the first three letters of the month in English.)

Using the DATE qualifier, any value that is not specified defaults to the current date. If the user specifies */day/year*, the month will default to the current month, and if the user enters a single value, it will be used as the day. For example, if the user specifies **5**, the date will default to the fifth day of

the current month and year. If the user enters a single value followed by the date separator, it will be used as the month. For example, the user can specify **10/**, and the date will default to October and the current day and year.

Several global fields in **tkctl.def** affect date processing. The **g_date_order** global field affects date input and display as follows:

If g_date_order is...	Input and display will have this order...
0	month day year or period year (if PERIOD is specified)
1	day month year or period year (if PERIOD is specified)
2	year month day or year period (if PERIOD is specified)

Additionally, the **g_date_sep** global field specifies the character that will separate parts of the date on redisplay. This is initialized to a slash (/). And the **g_periods** global field defines the maximum number of periods that are allowed on input. This is initialized to **12** if you don't specify a value.

The following example defaults the date in the **invdat** field to today's date if the user presses ENTER without entering anything into the field. The date is stored in the format **YYMMDD**.

```
.field invdat, d6, date(today)
```

In the next example, if the user enters **Dec 7, 12071994, 1207, 12/7, or 12/7/94** (assuming the current year is 1994), the date will be displayed as "12/07/94" and stored as 19941207.

```
.field birthdat, d8, date(short)
```

NODATE is the default state if neither DATE nor NODATE is specified in the script or in the repository. If DATE has been specified in the repository, NODATE will reset the field to a normal decimal field. To override any of the other date arguments, specify DATE without one or more of them.

DECREMENT

DECREMENT

DECREMENT designates that if the user does not enter a value the first time a numeric (**d** or **i**) field is processed, the last value in the field minus one will be used. For example, the following script will decrement the last value in the **quantity** field by one (if the user doesn't enter a value).

```
.field quantity, d4, decrement
```

If you use DECREMENT, you should *not* use COPY, INCREMENT, or DEFAULT. These four qualifiers are mutually exclusive.

If DECREMENT has been specified in the repository, NODEFAULT will cancel it.

DEFAULT, NODEFAULT

DEFAULT (*default*)

NODEFAULT

DEFAULT defines a default value for the field, where *default* is a value up to 80 characters long. The default value is automatically displayed in the field if the user has not previously entered anything in the field. The user can accept the default by pressing ENTER, edit the default, or type a new value. In the following example, the default value is 1.

```
.field quantity, d4, default(1)
```

If a selection window is associated with this field, the entry in the selection window that matches the default value will be the first highlighted entry, if found. Note that a default value specified here takes precedence over any default value specified in the .ITEM selection window building command or any call to the S_SELDFLT subroutine.

For date and time fields, the default value must be specified in field storage form, rather than field input or display form. For example, the default value for a six-digit period date should be specified in the form "199813." If the field is a date field, the default value must match the date field's storage format. For example, the default value for a six-digit period date should be specified as "199813."

If you use DEFAULT, you should *not* use the COPY, INCREMENT, or DECREMENT qualifiers. These four qualifiers are mutually exclusive.

If you use DEFAULT with ENUMERATED, DEFAULT must be specified after ENUMERATED.

NODEFAULT is the default state if there is no specification for this qualifier in the script or in the repository. If DEFAULT has been specified in the repository, NODEFAULT cancels any default value specified. It also cancels any COPY, INCREMENT, or DECREMENT qualifier that has been specified for the field.

DISABLED, NODISABLED

DISABLED
NODISABLED

DISABLED disables the field. A disabled field is unable to receive keyboard focus either by mouse click or by pressing one of the following: TAB, ARROW, or ENTER. For example,

```
.field state, a2, disabled
```

On Windows, except within ActiveX Toolkit lists, disabled fields will be grayed out. (Disabled fields within an ActiveX Toolkit list are rendered the same as enabled fields.)

On UNIX and OpenVMS, disabled fields will use the “disabled field display” rendition. This rendition can be modified using U_REND but cannot be modified using U_MODREND. See [U_REND on page 16-116](#) for more information on the U_REND subroutine and processing display renditions.

NODISABLED (which may also be specified ENABLED) is the default state if there is no specification for this qualifier in the script or in the repository. If DISABLED has been specified in the repository, NODISABLED cancels it.

DSP_LENGTH, NODSP_LENGTH

DSP_LENGTH (*display_length*)
NODSP_LENGTH

DSP_LENGTH specifies the maximum number of characters that can be displayed in a field, overriding Toolkit’s default display length. Valid values for *display_length* are 0 through 65,535. (Unless an explicit view length is set for the field, however, *display_length* cannot be set to a value that extends the field beyond the window border.) If *display_length* is set to 0, Toolkit uses the default.

This maximum includes any formatting characters and the negative sign (if a negative value is allowed for the field). Display lengths cannot be specified for check box fields, radio buttons, selection fields (including fields with selection windows), or text fields (multi-dimensional alpha fields).

If you don’t specify *display_length*, or if you set it to 0, it defaults to one of the following:

- ▶ For a field that’s an enumerated list, display length defaults to the display length set for the enumeration setting (for example, the *display_length* argument for the ENUMERATED qualifier).
- ▶ For a field with a format string, display length defaults to the length of the format string. (This does not apply to fields that are enumerated lists.)
- ▶ For a field that’s unformatted, display length defaults to the input length. (This does not apply to fields that are enumerated lists.)

For information on how display length works with other length settings for input fields, see [“Understanding display, input, and view lengths” on page 8-5](#).

NODSP_LENGTH is the default state if neither DSP_LENGTH nor NODSP_LENGTH is specified in the script or in the repository. If DSP_LENGTH has been specified in the repository, NODSP_LENGTH cancels the display length setting, causing the default to be used.

DISPLAY_METHOD, NODISPLAY_METHOD

DISPLAY_METHOD (*display_method*)
NODISPLAY_METHOD

DISPLAY_METHOD defines a display method for the field, where *display_method* is the name of the subroutine to invoke. *Display_method* is case insensitive and is valid with or without quotation marks. For example,

```
.field field,..., display_method("mymethod")
```

The subroutine specified with *display_method* will be called whenever the field is about to be displayed, after Toolkit has formatted the display according to its rules. Up to 20 optional data arguments can be passed to I_INPUT. These optional data arguments are then, in turn, passed to the *display_method* subroutine.

Any field with a user-defined data type will not invoke the *display_method* subroutine. Instead, use the EDSPFLD_METHOD routine.



Display methods are not allowed for check box, selection list, selection window, or radio button fields.

NODISPLAY_METHOD is the default state if neither DISPLAY_METHOD nor NODISPLAY_METHOD is specified in the script or in the repository. If DISPLAY_METHOD has been specified in the repository, specifying NODISPLAY_METHOD will override this setting and the display method will not be invoked.

DRILL_METHOD, NODRILL_METHOD

DRILL_METHOD (*drill_method*)
NODRILL_METHOD

DRILL_METHOD defines a drill method for the field, where *drill_method* is the name of the method to invoke. *Drill_method* is case insensitive and is valid with or without quotation marks. For example,

```
.field field,..., drill_method("mymethod")
```

A drill method is generally used for some form of lookup or additional information. On Windows, if an input field has a drill method and is a member of an input set being processed by I_INPUT, a drilldown button (a small button with “...” on the face) is placed immediately to the right of the field. If you click this button, the drill method is invoked. On UNIX and OpenVMS, there is no

drilldown button, but instead an I_DRILL menu entry will trigger UI Toolkit to invoke the specified drill method. Up to 20 optional data arguments can be passed to I_INPUT. These optional data arguments are then, in turn, passed to the *drill_method* subroutine.

NODRILL_METHOD is the default state if neither DRILL_METHOD nor NODRILL_METHOD is specified in the script or in the repository. If DRILL_METHOD has been specified in the repository, specifying NODRILL_METHOD will override this setting and the drill method will not be invoked.

EDITFMT_METHOD, NOEDITFMT_METHOD

EDITFMT_METHOD (*editfmt_method*)
NOEDITFMT_METHOD

EDITFMT_METHOD defines an edit format method for the field, where *editfmt_method* is the name of the subroutine to invoke. *Editfmt_method* is case insensitive and valid with or without quotation marks. For example,

```
.field field, ..., editfmt_method("mymethod")
```

The *editfmt_method* subroutine will be called whenever data in a field is about to be edited, after Toolkit has formatted the display according to its rules. Up to 20 optional data arguments can be passed to I_INPUT. These optional data arguments are then, in turn, passed to the *editfmt_method* subroutine.

For fields with a user-defined data type, any EEDTDSP_METHOD subroutine is called prior to the *editfmt_method* subroutine.



Edit format methods are not allowed for check box, selection list, selection window, or radio button fields.

NOEDITFMT_METHOD is the default state if neither EDITFMT_METHOD nor NOEDITFMT_METHOD is specified in the script or in the repository. If EDITFMT_METHOD has been specified in the repository, specifying NOEDITFMT_METHOD will override this setting and the edit format method will not be invoked.

ENUMERATED, NOENUMERATED

ENUMERATED (*display_length*[, *base*][, *step*])

NOENUMERATED

ENUMERATED specifies an enumerated data field, which is a field that returns a decimal or integer value for the corresponding displayed text entry. This field type corresponds to the Repository “enumerated” field attribute used in conjunction with an allow list, selection list, or selection window. It does not correspond to the Synergy DBL or Repository enumeration data type.

ENUMERATED takes the following arguments:

display_length

The number of characters (from 0 to 65535) that can be displayed for each select or allow list item.

base

(optional) The return value of the first entry. If you don’t specify a base value, the default is 1.

step

(optional) The value added to each successive entry. If you don’t specify a step value, the default is 1.

The ENUMERATED qualifier must be used in conjunction with the SELECT, ALLOW, SELWND qualifiers, or with a selection window built “on-the-fly” using S_SELBLD. If used with SELECT or ALLOW, it must precede them. An enumerated data field must have a decimal (**d**) or integer (**i**) data type, and its size must be able to handle the largest number expected to be returned.

This example defines two enumerated input fields:

```
.input inp, 20, 70
.field fld1, d2, enum(6, 1, 1), sel(1, 2, 3, red, orange, yellow)
.field fld2, d2, enum(6, 3, 2), sel(1, 2, 3, green, blue, violet)
```

The first input field, **fld1**, is two digits in size, and has three selection items: **red**, **orange**, and **yellow**. This field has a display length of **6** characters, a base value of **1**, and a step value of **1**. It will return the value **1** for **red**, **2** for **orange**, and **3** for **yellow**.

The second input field, **fld2**, is also two digits in size and has three selection items: **green**, **blue**, and **violet**. The display length is 6 characters, but the base value is **3** and the step value is **2**. This means it returns the value **3** for **green**, **5** for **blue**, and **7** for **violet**.

When UI Toolkit processes an enumerated data field at runtime, the data area contains the enumerated field calculated as follows:

$$(entry\_number - 1) * step + base$$

where *entry_number* is the number of the entry specified by the SELECT, ALLOW, or SELWND qualifier for each field.

If there's a field-level display length (set with the `DSP_LENGTH` qualifier, for example), it overrides the `display_length` setting for `ENUMERATED`.

For more information on display length, including default settings, and for information on how display length, input length, and view length interact, see [“Understanding display, input, and view lengths” on page 8-5](#).



This setting may be used as the view length if no view length or field-level display length (specified with `DSP_LENGTH`, for example) is specified for the field. In this case, `display_length` is used to calculate the width of the field and must not be set to a value that causes the field to extend beyond the window's borders. (See [VIEW_LENGTH](#), [NOVIEW_LENGTH](#) on page 2-71 for more information on the field width calculation.)

If you use `ENUMERATED` with `DEFAULT`, `ENUMERATED` must precede `DEFAULT`.

`NOENUMERATED` is the default state if this category is not specified in the script or in the repository. If `ENUMERATED` has been specified in the repository, `NOENUMERATED` overrides this setting and resets the field to a non-enumerated field. If you only wish to override the field parameters for `ENUMERATED`, use `ENUMERATED` with the new parameters.



A check box is implicitly an enumerated field and therefore cannot be defined with the enumerated qualifier.

FIELD_POSITION, NOFIELD_POSITION

`FIELD_POSITION` (*[CHANGE,]row, column*)
`NOFIELD_POSITION`

`FIELD_POSITION` (which may be abbreviated `FPOS`) specifies the position of a field, independent of its prompt. It takes the following arguments:

CHANGE

(optional) Designates that `FIELD_POSITION` is a degree of movement rather than an actual position. This value can be negative or positive.

row

Either a row position or the number of rows that the current position will move.

column

Either a column position or the number of columns that the current position will move.

If you don't specify the `FIELD_POSITION` qualifier on a `.FIELD` statement, the position of the field defaults to the current screen position if there isn't a prompt, or immediately following the prompt if one exists. (See the [POSITION, NOPOSITION](#) on page 2-59 for more information.)

If the position of a field specified by `FIELD_POSITION` falls on or before any part of the prompt, a warning occurs but the command is still processed.

Any POSITION or PROMPT qualifier is processed first, and FIELD_POSITION will be relative to the next position after the prompt (or POSITION, if PROMPT is not specified).

If the CHANGE qualifier is absent, the specified *row* and *column* will be the absolute coordinates for the field relative to the input window.

The following example positions the “Name” prompt at the tenth row and first column (10,1) and the input field at the tenth row and tenth column (10,10). Thus, five spaces are between the end of the “Name” prompt and the beginning of the input field.

```
.field fld1, a10, position(10,1), prompt("Name"), fpos(10,10)
```

This second example has the same result as the previous example except that it specifies a change in position from the end of the prompt rather than specifying an absolute position for the input field.

```
.field fld1, a10, position(10,1), prompt("Name"), fpos(change, 0, 5)
```

NOFIELD_POSITION (which may be abbreviated NOFPOS) is the default state if this category has not been specified in the script or in the repository. If FIELD_POSITION has been specified in the repository, NOFIELD_POSITION resets the position of the input field to default to the next character after the prompt rather than the position specified in the repository.

FONT, NOFONT

FONT (*palette_name*)
NOFONT

FONT specifies the font for the field, where *palette_name* is the name of a font palette entry. *Palette_name* must be known at runtime or a default font will be used. See [“Using Fonts on Windows” on page 18-40](#) for more information on font palettes.



Do not enclose *palette_name* in quotation marks; it will cause an error.

Although FONT has no visual effect on UNIX and OpenVMS, it is preserved with the window definition so that the same window library can be used for both non-Windows and Windows systems.

Specifying a field font with the FONT qualifier overrides any default font that would have been used for the field. NOFONT indicates that the window’s font should be used. See also [.FONT on page 2-75](#) for more information about specifying a window’s font, and see [PROMPTFONT, NOPROMPTFONT on page 2-60](#) for more information about specifying a prompt’s font.

The following example specifies *my_field_font* for the field **inpfld**.

```
.field inpfld, a30, font(my_field_font)
```

NOFONT is the default state if neither FONT nor NOFONT is specified in the script or in the repository. If FONT has been specified in the repository, NOFONT cancels the field’s font and uses the window’s font instead.

FORMAT, NOFORMAT

FORMAT (*fmt_string*)

NOFORMAT

FORMAT (which may be abbreviated FMT) designates a redisplay format for the field. *Fmt_string* is a quoted format string that determines the format in which the field input will be displayed.

On numeric fields, *format_string* follows the rules for Synergy DBL formatting. If you don't specify the FORMAT qualifier, numeric input is redisplayed, right-justified over blanks. When a redisplay format is defined and justification is not specified, the resulting format string is right-justified within the display field.

On alphanumeric fields, the FORMAT qualifier permits a special form, in which every "@" character is replaced by another character. This replacement is processed from left to right. For example, if the user enters **ab** and you have the format string

```
"The first two are @@"
```

you would get "The first two are ab".

The system only looks at the leftmost characters and substitutes blanks. For example, if your field has a data type of **a3**, the fourth @ will be blank.

On both numeric and alpha fields, the size of the display field is the width of the input field or the maximum number of characters the format can generate, whichever is greater.

In the following example, the contents of the field will be displayed in the form \$XX,XXX.

```
.field amount, d5, fmt("$XX,XXX")
```

Thus, if the user enters **1234**, the resulting display would be "\$01,234".

If you specify any justification for the field, UI Toolkit will justify only within the display length. See also [JUSTIFY, NOJUSTIFY on page 2-55](#).

UI Toolkit supports the Repository format justification specification. Format justification defines how a format is truncated before being applied to a field. See "[Defining Field Formats](#)" in the "Working with Fields" chapter of the *Repository User's Guide* for more information.

NOFORMAT (which may be abbreviated NOFMT) is the default state if neither FORMAT nor NOFORMAT is specified in the script or in the repository. If FORMAT has been specified in the repository, NOFORMAT cancels the field's format string. If you only wish to specify a different format string, use FORMAT with the new string.

HELP, NOHELP

HELP (*help_id*)
NOHELP

HELP specifies a help identifier. *Help_id* is used as an argument to the EHELP_METHOD subroutine. If no EHELP_METHOD subroutine is registered, *help_id* will be passed to the UI Toolkit default help subroutine. (See [EHELP_METHOD on page 7-73](#) for more information on help processing.)

In the following example, **h_name** is passed as an argument to the EHELP_METHOD subroutine, if registered; otherwise, it's passed to UI Toolkit's default help routine.

```
.field name, a30, help(h_name)
```

NOHELP is the default state if neither of these qualifiers is specified in the script or in the repository. If HELP has been specified in the repository, NOHELP cancels the help identifier. If you only wish to specify a different help identifier, use HELP with the new identifier.

HIGHLIGHT, NOHIGHLIGHT

HIGHLIGHT
NOHIGHLIGHT

HIGHLIGHT specifies highlighted text on UNIX and OpenVMS and bolded text on Windows. It is synonymous with BOLD. For detailed information on HIGHLIGHT and other input field renditions, see [“Renditions” on page 2-34](#).

In the following example, the contents of the field are highlighted/bolded.

```
.field balance, d18.2, highlight
```

NOHIGHLIGHT is the default state if neither HIGHLIGHT not NOHIGHLIGHT is specified in the script or the repository. IF HIGHLIGHT has been specified in the repository, NOHIGHLIGHT cancels the highlighting. NOHIGHLIGHT is synonymous with NOBOLD.

HYPERLINK_METHOD, NOHYPERLINK_METHOD

HYPERLINK_METHOD (*hyperlink_method*)
NOHYPERLINK_METHOD

HYPERLINK_METHOD specifies a hyperlink method for the field, where *hyperlink_method* is the name of the subroutine to invoke.

On Windows, if an input field has a hyperlink method, when it is a member of an input set being processed by I_INPUT, any prompt text associated with the field is highlighted. If you click the mouse button with the cursor on the highlighted text, the hyperlink prompt text method is invoked.

On UNIX and OpenVMS, there is no hyperlink text, but instead an I_HYPER menu entry triggers UI Toolkit to invoke the specified *hyperlink_method*. For example,

```
.field field,...,hyperlink_method("mymethod")
```

NOHYPERLINK_METHOD is the default state if neither HYPERLINK_METHOD nor NOHYPERLINK_METHOD is specified in the script or in the repository. If HYPERLINK_METHOD has been specified in the repository, specifying NOHYPERLINK_METHOD will override this setting and the hyperlink method will not be invoked.

INCREMENT

INCREMENT

INCREMENT designates that if the user does not enter a value the first time a numeric (**d** or **i**) field is processed, the last value in the field plus one will be used. For example,

```
.field ordnum, d5, increment
```

If you use INCREMENT, you should *not* use COPY, DECREMENT, or DEFAULT. These four qualifiers are mutually exclusive.

If INCREMENT has been specified in the repository, specifying NODEFAULT will cancel it. (See [DEFAULT, NODEFAULT on page 2-44](#) for more information.)

INFORMATION, NOINFORMATION

INFORMATION (*info_string*)

NOINFORMATION

INFORMATION specifies a text string (*info_string*) that is displayed on the information line when input is being processed for that field. The information line will revert to the text it contained previously once input to the field is terminated. If *info_string* is absent, the information line will not be modified.

In the following example, “Please enter your name” is displayed on the information line while the name field is processed.

```
.field name, a30, information("Please enter your name")
```

NOINFORMATION is the default state if neither INFORMATION nor NOINFORMATION is specified in the script or in the repository. If INFORMATION has been specified in the repository, NOINFORMATION cancels the information string. If you only wish to specify a different information string, use INFORMATION with the new string.

INPUT_LENGTH, NOINPUT_LENGTH

INPUT_LENGTH (*input_length*)
NOINPUT_LENGTH

INPUT_LENGTH specifies the maximum number of characters the user can enter in an input field, overriding Toolkit's default input length. Valid values for *input_length* are 0 to 65,535. If you set this to 0, the default input length is used.

If there is a change method for the field, the input length determines how many characters will be passed to the change method. If the input length is longer than the storage size, the additional characters may be ignored if there is no change method.

Input length cannot be specified for check box fields, radio buttons, selection fields (including fields with selection windows), or text fields (multi-dimensional alpha fields).

If you don't specify *input_length*, it defaults to one of the following:

- ▶ For an alphanumeric or user-defined field, the default input length is the storage size of the field.
- ▶ For a decimal field, the default input length is the storage size plus 1 if a decimal point is allowed. If a negative value is allowed, another character is added.
- ▶ For an integer field, the default input length is 3 for an **i1**, 5 for an **i2**, 10 for an **i4**, or 20 for an **i8**. If a negative value is allowed, the length is increased by 1.
- ▶ For a date field, the default input length is determined by the date format. Period dates are 8. Short period dates are 5. Calendar (non-period) dates are 11. Short calendar dates are 8.
- ▶ For a time field, the default input length is determined by the time format. HHMM is 5. HHMM AMIPM is 8. HHMMSS is 8. HHMMSS AMIPM is 11.

NOINPUT_LENGTH is the default state if neither INPUT_LENGTH nor NOINPUT_LENGTH is specified in the script or in the repository. If INPUT_LENGTH has been specified in the repository, NOINPUT_LENGTH cancels the input length setting, which causes the default input length to be used.

For information on how input length works with other length settings for input fields, see [“Understanding display, input, and view lengths” on page 8-5](#).

ITALIC, NOITALIC

ITALIC
NOITALIC

ITALIC specifies italicized text on Windows and blinking text on UNIX and OpenVMS. It is synonymous with BLINK. For example,

```
.field date_due, d8, italic
```

For detailed information on ITALIC and other input field renditions, see “Renditions” on [page 2-34](#).

NOITALIC is the default state if neither ITALIC nor NOITALIC is specified in the script or repository. If ITALIC has been specified in the repository, NOITALIC cancels the italics. NOITALIC is synonymous with NOBLINK.

JUSTIFY, NOJUSTIFY

JUSTIFY (LEFT | RIGHT | CENTER)
NOJUSTIFY

JUSTIFY determines whether the text in the field will be displayed left-justified, right-justified, or centered. The field defined in the following example will display left-justified text.

```
.field custno, d5, justify(left)
```

Restrictions and defaults depend on the field type:

- ▶ Numeric fields accept only RIGHT (default) or LEFT alignment.
- ▶ Text fields accept only LEFT alignment.
- ▶ Alpha fields accept all three types. LEFT is the default. (Note that leading spaces are not stripped from the data.)

If you have formatted the display of the field (using the FORMAT qualifier), the justification is performed only within the length of the formatted display.

NOJUSTIFY is the default state if this category has not been specified in the script or in the repository. If JUSTIFY has been specified in the repository, NOJUSTIFY resets the justification to the field type default. If you wish to change the justification alignment, use JUSTIFY with the new alignment.

LEAVE_METHOD, NOLEAVE_METHOD

LEAVE_METHOD (*leave_method*)
NOLEAVE_METHOD

LEAVE_METHOD specifies a post-input subroutine for the field. *Leave_method* is the name of the subroutine to invoke. A leave method provides a much more reusable and encapsulated mechanism than break processing, which must be at each usage of a field. For example,

```
.field field_spec, field_qualifiers, leave_method("mymethod")
```

NOLEAVE_METHOD is the default state if neither LEAVE_METHOD nor NOLEAVE_METHOD is specified in the script or in the repository. If LEAVE_METHOD has been specified in the repository, specifying NOLEAVE_METHOD will override this setting and the LEAVE_METHOD will not be invoked.

MATCH

`MATCH (/CASE | NOCASE / [, EXACT | NOEXACT /)`

MATCH specifies whether the alpha field input must match the case of a specified allowable entry and/or match all characters in the entry. Specify one or more of these options:

CASE—Match case as well as characters.

NOCASE—Do not match case as well as characters. (default)

EXACT—Match the characters in the entry exactly for the number of characters specified.

NOEXACT—Do not match the characters exactly. (default)

If **CASE** is set, the match will be case sensitive. The entry **no** will match with “no”, but will not match with “NO” or “No”. If the **UPPER_CASE_ONLY** or **UC** qualifier is specified, **MATCH(CASE)** is ignored.

If **EXACT** is not set, UI Toolkit will look for a match based on the shortest string. For example, if you are looking for a match with the entry **Ann**, a match will be acknowledged with “A”, “An”, and “Ann”. However, if the entry is longer than **Ann** (for example, “Anne”), it will *not* be recognized as a match. **EXACT** mode means that input must match the allowed qualifier exactly for the full length of the input. If **MATCH(EXACT)** mode is set, **Ann** will only match with “Ann”.

CASE/NOCASE and **EXACT/NOEXACT** are not mutually exclusive; you can use these four options together in any combination. For example, if **MATCH(CASE, EXACT)** mode is set, the entry **no** will only match with “no”, but if **MATCH(CASE, NOEXACT)** mode is set, “NO”, “n”, and “no way” will all match the entry **no**.

NOCASE is the default if neither **CASE** nor **NOCASE** is specified in the script or repository. If **CASE** has been specified in the repository, **NOCASE** cancels case sensitive matching for the field.

NOEXACT is the default state if neither **EXACT** nor **NOEXACT** is specified in the script or in the repository. If **EXACT** has been specified in the repository, **NOEXACT** cancels full-length matching for the field.

NEGATIVE, NONEGATIVE

`NEGATIVE (ALLOWED | ONLY | ORZERO)`
`NONEGATIVE`

NEGATIVE allows negative values on numeric type (**d** or **i**) fields. Specify one of the following:

ALLOWED—Allows negative entries. (default)

ONLY—Allows negative entries only.

ORZERO—Allows negative or zero entries only.

If you specify the **NEGATIVE** qualifier, a negative or minus sign (–) can be placed either before or after the numeric value. This qualifier causes the size of the input field to be one character larger than the size of the data field. For example, if negative were specified on a **d8** field, nine input characters would be allowed.

In the following example, the **credit** field can contain a negative value and allows a total of eight input characters (seven digits and a minus sign):

```
.field credit, d7, negative
```

NONEGATIVE is the default state if neither **NEGATIVE** nor **NONEGATIVE** is specified in the script or in the repository. If **NEGATIVE** has been specified in the repository, **NONEGATIVE** will override it and enable the input of non-negative values only.

NODECIMAL, DECIMAL_REQUIRED

```
NODECIMAL  
DECIMAL_REQUIRED
```

NODECIMAL (which may also be specified as **NODECIMAL_REQUIRED**) indicates that the user does not need to type a decimal point when placing input in a **d** type field. The input is stored right-justified, with the number of decimal places inferred from the storage format. In the following example, an entry of **55** will be interpreted as “.55”:

```
.field amount, d7.2, nodecimal
```

By way of contrast, an entry of **55** in the following example (which does not specify the **NODECIMAL** qualifier) will be interpreted as “55.00”:

```
.field amount, d7.2
```

An explicit decimal point overrides the **NODECIMAL** qualifier. For example, an entry of **12.34** always yields a result of “12.34”, even if the **NODECIMAL** qualifier is present.

DECIMAL_REQUIRED is the default state if neither **DECIMAL_REQUIRED** nor **NODECIMAL** is specified in the script or in the repository. If **NODECIMAL** has been specified in the repository, specifying **DECIMAL_REQUIRED** cancels it and aligns the numeric value based on an entered decimal point only.

NOECHO, ECHO

NOECHO (*[display_character]*)

ECHO

NOECHO prevents the text entered by the user from being displayed in the field. If specified, *display_character* will display for each character typed. This feature is commonly used for password fields. After the user presses ENTER, the *display_character* fills in any remaining spaces at the end of the input so others can't see how many characters were entered.

If no *display_character* is specified, the field simply remains blank and the cursor remains stationary as the user types.

NOECHO has the following restrictions:

- ▶ It can only be used for alpha fields.
- ▶ You cannot use “E_” entries (field editing menu entries) on fields that have NOECHO.
- ▶ REQUIRED is the only validation allowed. (See [REQUIRED](#), [NOREQUIRED](#) on page 2-63.)

The following example defines a password field in which an asterisk (*) is displayed for every character the user types. The typed characters themselves are not echoed in the input field.

```
.field password, a10, noecho(*), pos(1,1), prompt("Password:")
```

ECHO is the default state if neither NOECHO nor ECHO has been specified in the script or in the repository. If NOECHO has been specified in the repository, ECHO will cancel it and leave echo on for field input. To change or cancel the character specified with NOECHO, use the NOECHO qualifier with another character or no character specified.

NOTERM, TERM

NOTERM

TERM

NOTERM is *not* supported on Windows.

NOTERM automatically terminates the field when the field is full so the user doesn't have to press ENTER. For example, if you type **555-1079** into the **acctnum** field defined below, it would be equivalent to typing **555-1079** and pressing ENTER without NOTERM. (You can still terminate a NOTERM field by pressing ENTER before it is full.)

```
.field acctnum, a8, noterm
```

TERM requires the user to press ENTER to terminate field input. If a field is full and the user tries to enter more characters, the terminal bell is rung for each extra character typed.

TERM is the default state if neither one of these qualifiers is specified in the script or in the repository. If NOTERM is specified in the repository, specifying TERM will override it, requiring the user to press ENTER to end input.

PAINT, NOPAINT

PAINT (*character*)
NOPAINT

PAINT specifies a character that “paints” the empty field with *character* to indicate where the user should type input. This character overrides any paint character designated by the .PAINT command.

In the following example, when the field is empty, it will be painted with the “#” character to indicate where the user should type input. (When input processing begins, the field is blanked.)

```
.field amount, d7, paint(#)
```

NOPAINT is the default state if neither of these qualifiers is specified in the script or in the repository. If PAINT has been specified in the repository, NOPAINT resets the field to the default paint character (which is a space).

This qualifier is ignored on Windows.

POSITION, NOPOSITION

POSITION ([CHANGE,]*row, column*)
NOPOSITION

POSITION specifies either a specific position or the number of rows and columns that the current position will change. It supports the following arguments:

CHANGE

(optional) Designates that POSITION is a degree of movement rather than an actual position. This value can be positive or negative.

row

Either a row position or the number of rows that the current position will move.

column

Either a column position or the number of columns that the current position will move.

If explicit positioning is not specified, UI Toolkit will automatically update the position to follow any field or text that was specified immediately before this command.

If there is no prompt defined (see the PROMPT qualifier section), the input field begins at the specified position. If a prompt is defined, the prompt begins at the specified position.

In the following example, the prompt begins at row 12, column 10, and the **name** field immediately follows the prompt (at row 12, column 15).

```
.field name, a30, position(12, 10), prompt("Name ")
```

In the next example, the **name** field will move 2 rows down from its current position but will stay in its current column.

```
.field name, a30, position(change, 2, 0)
```

NOPOSITION is the default state if neither POSITION nor NOPOSITION is specified in the script or in the repository. If POSITION has been specified in the repository, NOPOSITION resets the position of the prompt to default to the next character available rather than the position specified in the repository.

PROMPT, NOPROMPT

PROMPT (*prompt*)

NOPROMPT

PROMPT provides a fixed (static) or variable (dynamic) prompt to the user for input:

- ▶ With a fixed prompt, *prompt* is a quoted string that must include any desired spacing between the prompt and the input. Fixed prompts may be modified at runtime using I_PROMPT.
- ▶ With a variable prompt, *prompt* is a number that specifies the length of the prompt. The I_PROMPT subroutine uses this length to set the variable prompt.

If you do not use the FIELD_POSITION qualifier, field input will immediately follow the prompt. Your prompt must include any desired spacing between the prompt and the input.

The following example defines a fixed prompt that prompts the user for a name:

```
.field name, a30, prompt("Name ")
```

The following example defines a variable prompt that can be up to 20 characters long:

```
.field name, a30, prompt(20)
```

NOPROMPT is the default state if this category is not specified in the script or in the repository. If PROMPT has been specified in the repository, NOPROMPT will cancel it. If you wish to specify a new input prompt, use PROMPT with the desired parameter.

PROMPTFONT, NOPROMPTFONT

PROMPTFONT (*palette_name*)

NOPROMPTFONT

PROMPTFONT specifies a font for a prompt, where *palette_name* is the name of a font palette entry. *Palette_name* must be known at runtime or a default font will be used. See [“Using Fonts on Windows” on page 18-40](#) for more information on font palettes.

Although the PROMPTFONT qualifier has no visual effect on UNIX and OpenVMS, it is preserved with the window definition so that the same window library can be used for both non-Windows and Windows systems.

Specifying a font with PROMPTFONT overrides any default font that would have been used for the prompt. Specifying NOPROMPTFONT indicates that the default font for the prompt is to be used.

If a prompt is not specified for the field, PROMPTFONT will be ignored.

See also [.FONT on page 2-75](#) for more information about specifying a window's font, and [FONT, NOFONT on page 2-50](#) for more information about specifying a prompt's font.

The following example specifies "my_prompt_font" for field **inpfld**.

```
.field inpfld, a30, prompt("Name  "), promptfont(my_prompt_font)
```

NOPROMPTFONT is the default state if neither PROMPTFONT nor NOPROMPTFONT is specified in the script or in the repository. If PROMPTFONT has been specified in the repository, NOPROMPTFONT cancels the field's font and the uses window's font instead.

RADIO, NORADIO

RADIO
NORADIO

The RADIO qualifier indicates that this field is to be displayed as a radio button group on Windows. (Specifying RADIO on UNIX or OpenVMS has no effect.) It must be used in conjunction with either SELWND or SELECT. Radio buttons are a form of selection window, where each entry becomes a radio button, and the text for each entry is displayed to the right of the radio button. Note the following:

- ▶ All radio buttons for a field receive focus as a single field. Pressing TAB, SHIFT+TAB, or RETURN, or clicking on another field advances focus to the next field.
- ▶ If you use the SELECT qualifier with RADIO, radio buttons are placed vertically in a single column unless you pass the *height* argument for SELECT. If you pass *height*, this argument determines how many radio buttons a column can hold. For example, if you pass *height* as 3 and there are eight radio buttons, there will be three columns: the first and second columns will have three radio buttons, and the third column will have two radio buttons.
- ▶ If you use the SELWND qualifier with RADIO, the height of the associated selection window determines how many radio buttons can fit in a column. For example, if the selection window is only one row high, the buttons will be placed horizontally; there will be only one button per column. If the selection window is three rows high and there are five radio buttons, the first column will have three radio buttons, and the second will have two.
- ▶ If a font is not specified for the field, the default prompt font will be used for the radio button's selection entry text.
- ▶ Make sure radio buttons do not overlap other controls or extend beyond the limits of the containing window. Toolkit does not limit radio button placement.

For information on creating an input window with radio buttons at runtime, see the [IB_FIELD Discussion on page 8-98](#). For information on how radio button fields are validated, see [SELECT, NOSELECT on page 2-64](#).



Composer can also be used to create radio buttons. Refer to the “[Designing Your User Interface](#)” chapter of *Getting Started with Synergy/DE* for more information on Composer.

You can use the selection subroutines with radio buttons to load new selections, get or set a default, and so forth. For example, to make a change to a radio button set (add a button, remove a button, etc.), use `S_SELLD` to reload the associated selection window with a different set of selections. See [S_SELLD on page 12-17](#) for information.

You can retrieve the selection window ID with `I_FLDINF`. See [I_FLDINF on page 8-22](#).

`NORADIO` is the default state if neither `RADIO` nor `NORADIO` is specified in the script or in the repository. If `RADIO` has been specified in the repository, specifying `NORADIO` will override this setting and the field will not be displayed as a group of radio buttons.

RANGE, NORANGE

`RANGE` (*minimum, maximum*)

`NORANGE`

`RANGE` defines the minimum and maximum input values for a numeric (**d** or **i**) field.

For date and time fields, the minimum and maximum values must be specified in field storage format, rather than the field input or display format. Also note that the minimum and maximum values will not be validated for date and time fields; they will only be compared with the resulting value from the user’s input.

The following example defines a minimum value of 1 and a maximum value of 100.

```
.field amount, d3, range(1, 100)
```

`NORANGE` is the default state if neither of these qualifiers is specified in the script or in the repository. If `RANGE` has been specified in the repository, `NORANGE` cancels range checking for the field. If you only wish to specify new range parameters, use `RANGE` with the new parameters.

READONLY, NOREADONLY

READONLY
NOREADONLY

READONLY renders the field read-only. Only edit controls can be rendered read-only. Selection window, radio button, and check box fields that specify the READONLY qualifier will generate an error when compiled by script or when read into Composer.

Like disabled fields, the contents of a read-only field cannot be modified. Unlike a disabled field, a read-only field is able to receive focus. On Windows, the field's contents can be highlighted for the purpose of copying to the clipboard. On Windows, the field will use the "3D face" color for its background and "window text" color for the text. Typically, this is black on dark gray. This rendition can be modified by using U_REND but cannot be modified using U_MODREND. See [U_REND on page 16-116](#) for more information on processing display renditions.

Read-only fields within an ActiveX Toolkit list are rendered the same as disabled fields.

```
.field description, 3a50, readonly
```

NOREADONLY cancels the field from being read-only.

REQUIRED, NOREQUIRED

REQUIRED
NOREQUIRED

REQUIRED specifies that a non-blank alpha or non-zero numeric entry is required in the field. For example, the **custno** field doesn't allow an entry of 0.

```
.field custno, d6, required
```

You can enable the user to temporarily exit from a required field. If the first keystroke in the field is a menu entry that causes an exit from the field, no validation will be done by the input processor. However, the user will still be required to enter data into all required fields before completing the input set.

NOREQUIRED is the default state if neither REQUIRED nor NOREQUIRED is specified in the script or in the repository. If REQUIRED has been specified in the repository, specifying NOREQUIRED overrides it and enables the user to skip this field or enter a zero.

RETAIN_POSITION, NORETAIN_POSITION

RETAIN_POSITION
NORETAIN_POSITION

RETAIN_POSITION enables the user to leave and then return to the text field, retaining the current position until the field is re-initialized or redisplayed. It is allowed only for text fields.

NORETAIN_POSITION is the default state if neither RETAIN_POSITION nor NORETAIN_POSITION is specified in the script or in the repository. If RETAIN_POSITION has been specified in the repository, specifying NORETAIN_POSITION overrides it.

REVERSE, NOREVERSE

REVERSE
NOREVERSE

REVERSE specifies reverse video text. For detailed information on REVERSE and other input field renditions, see [“Renditions” on page 2-34](#). In the following example, the contents of the field are in reverse video.

```
.field description, 4a60, reverse
```

NOREVERSE is the default state if neither REVERSE nor NOREVERSE is specified in the script or in the repository. If REVERSE has been specified in the repository, NOREVERSE cancels the reversing.

SELECT, NOSELECT

SELECT (*srow*, *scolumn*, [*height*], *entry*[, ...])
NOSELECT

SELECT defines a selection window by specifying the position where the window is placed on the screen and designating one or more text entries. Up to 100 text entries may be specified. SELECT takes the following arguments;

srow

The row at which the upper-left corner of the selection window will be placed, relative to the beginning of the field.

scolumn

The column at which the upper-left corner of the selection window will be placed, relative to the beginning of the field.

height

(optional) The maximum number of rows in the selection window (or the maximum number of radio buttons placed in a column if the RADIO qualifier is specified). A *height* of 0 is the same as omitting *height*.

entry

A text entry in the selection window.

Data on UNIX and OpenVMS and radio buttons on Windows are organized first by column, top to bottom, and then left to right. For example, if *height* is **3** and there are eight entries or radio buttons, the window will contain three columns with three entries or radio buttons in the first, three in the second, and two in the third. If *height* is not specified, the height of the window is the number of entries or radio buttons in a column.

WIN

Srow and *scolumn* have no meaning on Windows and are ignored.

On Windows, fields associated with selection windows use native Windows combo boxes (unless you specify the **RADIO** qualifier). Combo boxes combine the field edit control with an arrow box. Clicking on the arrow or anywhere within the field will drop down the combo box. The combo box arrow is only displayed when the input set containing the field is active. If the *height* of the selection window is less than the number of entries it contains, the drop-down list will display *height* number of entries and a scroll bar.

If you specify the **RADIO** qualifier with **SELECT**, the field will be a group of radio buttons. These buttons are placed vertically in a single column unless you pass the *height* argument for **SELECT**. If you pass *height*, this argument determines how many radio buttons can fit in a single column. For example, if there are three radio buttons and *height* is passed as 1, there will be three columns with one radio button in each. (In other words, the buttons are placed horizontally.) If, however, there are eight radio buttons and *height* is passed as 3, there will be three radio buttons in the first column, three in the second, and two in the third. For more information on radio buttons, see [RADIO, NORADIO on page 2-61](#).

Highlighting an entry in a combo box or selecting a radio button does not, by default, cause the field to be immediately validated. Until the field loses focus, no Toolkit validation tests or change method occurs. Likewise, no leave or arrive methods will be invoked until focus is lost. In order for a field to be immediately validated, the **D_VALSTCHG** option of **E_STATE** needs to be set. See [E_STATE on page 7-41](#) for more information on this subroutine and the **D_VALSTCHG** option.

If the user selects an entry by dropping down the combo box's list, focus moves to the next field by default. If the user selects an entry without dropping down the combo box's list, focus will remain on the combo box. To always maintain focus on the combo box set after a selection is made, turn off the **D_CBADVANCE** option of **E_STATE**.

The following example defines a selection window with a maximum height of two rows and the text entries "yes" and "no." The upper-left corner of the selection window will be placed at row 1 and column 2, relative to the beginning of the field.

```
.field yesno, a3, select(1, 2, 2, yes, no)
```

NOSELECT is the default state if neither **SELECT** nor **NOSELECT** has been specified in the script or in the repository. If **SELECT** has been specified in the repository, **NOSELECT** cancels the selection window associated with the field. If you only wish to specify different parameters for the selection window, use **SELECT** with the new parameters.

You can use the selection subroutines to load new selections, get or set a default, and so forth. See [chapter 12, "Selection Routines,"](#) for details.

SELWND, NOSELWND

SELWND (*srow*, *scolumn*, *window*)

NOSELWND

SELWND specifies a selection window to be placed on the screen. It is similar to SELECT, except the selection window already exists. SELWND takes the following arguments:

srow

The screen row at which the upper-left corner of the selection window will be placed, relative to the beginning of the field.

scolumn

The screen column at which the upper-left corner of the selection window will be placed, relative to the beginning of the field.

window

The name of a selection window. The window must either reside in the same window library or be created during runtime (using the S_SELBLD routine) before the input window is loaded.

WIN

If you use the SELWND qualifier on Windows, the field will be either a native Windows combo box or, if you also specify RADIO, a group of radio buttons. If you use SELWND and RADIO, the height of the selection window determines the maximum number of radio buttons that can fit in a column. For example, if the window is two rows high and there are five radio buttons, the first column will have two radio buttons, the second column will have two, and the third column will have one. You can control spacing between radio button columns by changing the column width for the selection window. This spacing is calculated by adding the width of the radio button to the width of a column in the selection window (which is equal to the number of characters times the character width for the font).

See the Windows-specific information in [SELECT, NOSELECT on page 2-64](#) for more information on native Windows combo boxes and radio buttons.

The following example specifies that the **yesno** selection window's upper-left corner will be placed at row 1, column 2, relative to the beginning of the field.

```
.field yesno, a3, selwnd(1, 2, yesno)
```

NOSELWND is the default state if neither SELWND nor NOSELWND is specified in the script or in the repository. If SELWND has been specified in the repository, NOSELWND cancels the association between the pre-built selection window and the field. If you only want to specify different parameters for this category, use SELWND with the new parameters.

TIME, NOTIME

TIME (/AMPM[/], NOW/)
NOTIME

TIME enables you to process times.

AMPM—Specifies that the display form will use a 12-hour clock with an AM/PM designation. AMPM is not case sensitive and may be abbreviated. These designators are set by DTK_AM and DTK_PM in **syntxt.ism**. If AMPM is not specified, a 24-hour clock is used.

NOW—Defaults the time to the current system time if the user presses ENTER without entering a time.

You can use this qualifier only on **d4**, **d6** (or greater), **i2**, or **i4** fields; fields of any other size will generate an error. The input for a time is either numeric or delimited.

The size of the field defines the way the time is stored as the table below indicates. The size used for the hour will be extended if a field larger than **d6** is used.

Time Formats for Field Sizes		
Size	Time format	Meaning
d4 field	HHMM	Hour and minute
d6 field	HHMMSS	Hour, minute, second
i2 field	HHMM	Hour and minute
i4 field	HHMMSS	Hour, minute, second

For numeric input

- ▶ if one or two digits are entered, it is considered to be the hour.
- ▶ if three or four digits are entered, it is considered to be the hour and minute. The two rightmost digits will be the minute, and the leftmost one or two digits will be the hour.
- ▶ if five or more digits are entered, it is considered to be the hour, minute, and second. The two rightmost digits will be the second, the next two digits are the minute, and the leftmost one or two digits are the hour.

When the input string is delimited, if any component of the time is missing that part will be assumed to be zero. If one of the non-numeric components is the post-meridian indicator, twelve hours will be added to the time specified.

The global field **g_time_sep** in **tkctl.def** specifies the character that will separate parts of the time on redisplay. It is initialized to a colon (:) by U_START.

In the following example, if the user enters **1545**, the time will be displayed as “3:45 PM”, and stored as **154500**.

```
.field shiptime, d6, time(ampm)
```

In this example, the time in the **ordtime** field defaults to the current system time if the user presses ENTER without entering anything into the field. The time is stored in the form *HHMMSS*.

```
.field ordtime, d6, time(now)
```

NOTIME is the default state if neither TIME nor NOTIME is specified in the script or in the repository. If TIME has been specified in the repository, NOTIME resets the repository field to a normal decimal field. To override either the AMPM or NOW argument, specify TIME without the one you want to override.

TIMESTAMP, NOTIMESTAMP

TIMESTAMP ([/AMPM]/[, SHORT/])

NOTIMESTAMP

TIMESTAMP creates a read-only 8-byte integer field that corresponds to a timestamp key in an ISAM file. (See “[Keys in ISAM files](#)” in the “Synergy DBMS” chapter of *Synergy Tools*.) It enables you to create a timestamp field *without* drawing it from a repository autotime or ctimestamp field.

AMPM—Sets the display format to use a 12-hour clock with an AM/PM designation preceded by a space. AMPM is not case sensitive and may be abbreviated. These designators are set by DTK_AM and DTK_PM in **syntxt.ism**. If AMPM is not specified, a 24-hour clock is used.

SHORT—Sets the display format to a short form that omits the following: the century portion of the year, the seconds value and its separator, and the milliseconds value and the period (.) that precedes it.

By default, fields drawn from repository autotime and ctimestamp fields become timestamp fields in the window library. If you do not want this behavior, you can use NOTIMESTAMP to create a standard integer field from a repository autotime or ctimestamp field. You can alter the default behavior by using AMPM and SHORT with TIMESTAMP to change the display format for a timestamp field drawn from a repository autotime or ctimestamp field.

An error will occur if you use TIMESTAMP with a field that is set to anything but a read-only 8-byte integer, or if the field is set to be a selection field, a radio button set, or a check box.

The default display format for a timestamp field consists of a date and a time separated by a space. For example:

```
12/31/2013 23:59:59.999999
```

The date portion consists of two-digit numbers for month and day and a four-digit number for year. (Two digits are used for the year if SHORT is specified.) The **g_date_order** setting defines the order of date components (month, day, year), and **g_date_sep** defines the separator for date components (see “[tkctl.def](#)” on page 18-2).

The time consists of an hour value (1 to 24, unless AMPM is specified), a minute value, a seconds value, and a six-digit microseconds value. The separator for hour, minutes, and seconds is defined by `g_time_sep` (see “`tkctl.def`” on page 18-2), and a period (.) precedes the microseconds value.

Qualifier(s) in effect	Default display length and view length ^a	Example
none	26	12/31/2013 23:59:59.999999
AMPM	29	12/31/2013 11:59:59.999999 PM
SHORT	14	12/31/13 23:59
AMPM and SHORT	17	12/31/13 11:59 PM

a. Lengths are based on two-character AM and PM designators.

Timestamp fields are right-justified by default, though justification has an effect only if the month or hour requires one digit, or if the display length has been changed from the default.

You can change the display format by setting the view length, display length, or justification for the field, and you can use an `IDISPLAY_METHOD` method to set up a different display format.

UNDERLINE, NOUNDERLINE

UNDERLINE
NOUNDERLINE

UNDERLINE specifies underlined text. For detailed information on UNDERLINE and other input field renditions, see “[Renditions](#)” on page 2-34.

In the following example, the contents of the field are underlined.

```
.field contact, a30, underline
```

NOUNDERLINE is the default state if neither UNDERLINE nor NOUNDERLINE is specified in the script or in the repository. If UNDERLINE has been specified in the repository, NOUNDERLINE cancels the underlining.

UPPER_CASE_ONLY, NOUPPER_CASE_ONLY

UPPER_CASE_ONLY
NOUPPER_CASE_ONLY

UPPER_CASE_ONLY (which can be abbreviated UC) converts lowercase input characters to uppercase on alpha fields. All input will be displayed in uppercase. In the following example, all entries in the **state** field will be displayed in uppercase.

```
.field state, a2, uc
```

Note that only *input* is uppercased. If your program loads data directly into this field, you must uppercase that data.

NOUPPER_CASE_ONLY (which can be abbreviated NOUC or LOWER_CASE_ALLOWED) is the default state if neither of these qualifiers is specified in the script or in the repository. If UPPER_CASE_ONLY has been specified in the repository, NOUPPER_CASE_ONLY will cancel it and enable lowercase input.

USER, NOUSER

```
USER ([user_text][, user_type])  
NOUSER
```

USER enables you to associate a user-defined text string with input fields and/or validate a data type that you've defined. *User_text* is the text to be stored, while *user_type* is a string that represents the user-defined data type. Either *user_text* or *user_type* (or both) must be specified.

The size of *user_text* is fixed by the size of the string used during the window's compilation; writing data out to the string won't change the size of the string. The maximum size of *user_text* is 80 characters. You can access *user_text* at runtime with the I_USER subroutine.

The *user_type* argument is available only on alpha fields. It flags the runtime input processor to call the ECHKFLD_METHOD (for extra input validation), EDSPFLD_METHOD (for display), and EEDTDSP_METHOD (for editing) subroutines. *User_type* is passed to EDSPFLD_METHOD, ECHKFLD_METHOD, and EEDTDSP_METHOD to be used as a control code. The maximum size of *user_type* is 30 characters.

For example, you might want to define a zip code data type and a phone number data type. If the zip code type is "**ZIP_TYPE**", then a *user_type* specified as "**ZIP_TYPE**" will indicate that you want to check the zip code. To check your user-defined data type, you must write the ECHKFLD_METHOD and EDSPFLD_METHOD subroutines. (If you choose to use the EEDTDSP_METHOD or %IEDITFMT_METHOD routines, you will need to write them as well.) If ECHKFLD_METHOD is not specified, your code will still run but UI Toolkit will bypass user-defined data type checking. (See "[Environment Routines](#)" on page 7-1 for more information about ECHKFLD_METHOD, EDSPFLD_METHOD, and EEDTDSP_METHOD.)

In the following example, the string "This can contain whatever you want" is associated with the **name** field.

```
.field name, a30, user("This can contain whatever you want")
```

The following sample window script includes a user-defined zip code data type. It assumes that the subroutines ECHKFLD_METHOD and EDSPFLD_METHOD will control the zip code input validation and display format.

```
.input mail, 1, 17  
.placement 5, 29  
.title " Zip Test "  
.field zip, a9, pos(1,2), prompt("Zip"), user(, "ZIP_TYPE"), uc  
.end
```

NOUSER is the default state if neither USER nor NOUSER is specified in the script or in the repository. If USER has been specified in the repository, NOUSER cancels both the user-text and user-defined data type specified in the repository. To cancel only one, you must specify USER with the desired parameter.

VIEW_LENGTH, NOVIEW_LENGTH

VIEW_LENGTH (*view_length*)

NOVIEW_LENGTH

VIEW_LENGTH determines the width of the field (that is, the width of the area on the screen that will display data for the field), overriding Toolkit's default view length. Valid values for *view_length* are 0 through 9,999. This value should not be so large that it would cause the field to extend beyond the window border. If you set *view_length* to 0, the default view length is used.

View length cannot be specified for check box fields or radio buttons.

On Windows, if the view length is less than the field's display length, the field is scrollable. For text fields (multi-dimensional alpha fields), this doesn't affect the number of lines, just the width of each line. You can scroll through all displayed characters and delete any displayed character, no matter what the input length. (However, you can enter a character only when there are fewer characters than the limit imposed by the input length.)



On UNIX and OpenVMS, if the view length is less than the field's display length, the contents of a field will be truncated to the view length if you move the cursor in the field or modify the contents of the field.

If you specify an explicit view length on Windows, the width of the field is computed by multiplying the view length value by the width of the sizing character for the font used by the field. (If the font is changed, the field's width is adjusted.) On UNIX and OpenVMS, the width is set to the number of characters specified by *view_length*.

If you specify an explicit view length for a field in an ActiveX list, this setting overrides input and display lengths, but may be extended by the length of a header, a footer, or another field that occupies the same column in a multi-line item. A view length may also be overridden by dragging column borders or by a call to SetColumnWidth, a published method for ActiveX Toolkit lists.

If you don't set an explicit view length, the default is the greater of display length or input length.



If no explicit view length is in effect, a value of 0 is returned in **gs_viewlen**. This is unlike **gs_dsplen** and **gs_inplen**, which always return display length and input length settings, whether you set them or the defaults are in effect. (**Gs_viewlen**, **gs_dsplen**, and **gs_inplen** are members of the **gs_inpfld** record returned by I_FLDINF.)

For information on how view length works with other length settings for input fields, see [“Understanding display, input, and view lengths” on page 8-5](#).

NOVIEW_LENGTH is the default state if neither VIEW_LENGTH nor NOVIEW_LENGTH is specified in the script or in the repository. If VIEW_LENGTH has been specified in the repository, NOVIEW_LENGTH cancels the view length setting, which causes the longer of the display length or input length to determine the field width.

WAIT

WAIT (IMMEDIATE | FOREVER | GLOBAL | *n*)

WAIT specifies an input time-out, overriding the **g_wait_time** setting, which by default instructs Toolkit to wait until I/O processing is complete (i.e., no time-out) before returning to the calling routine. (See “[g_wait_time](#)” on page 18-6.) The wait time can be specified with one of the following:

IMMEDIATE—Time out immediately. When this is set the user will be unable to enter data in the field.

FOREVER—Wait until I/O processing is completed.

GLOBAL—Use the globally specified time-out (**g_wait_time**).

n —Wait up to *n* seconds (where *n* is a positive value) for I/O processing to be complete.

If a time-out is set and the user doesn’t complete input within the specified time, **g_select** is set to true and **g_entnam** is set to the entry name set with **g_time_entry**, which is defined in **tkctl.def**. The default value for **g_time_entry** is “O_TIMEOUT” (see “[g_time_entry](#)” on page 18-6).

Note the following platform differences:

- ▶ On UNIX and OpenVMS, the time-out is reset after the user presses the first keystroke in an input field. For instance, if you have a 60 second time-out on a field, and after 30 seconds the user presses a key, the user has an additional 60 seconds to complete input. Subsequent keystrokes, however, do not reset the time-out.
- ▶ On Windows, the time-out is not reset by keystrokes. Additionally, no time-out will occur while a menu is dropped down or when the application is otherwise suspended (as when a message window is open). When the application continues, the time-out occurs as soon as the application can process the message from the system.

.FILL – Define the fill character



`.FILL string [, rendition, ...]`

Arguments

string

A quoted string that will fill the window.

rendition

(optional) One or more of the following display renditions:

BLINK ITALIC	Blinking (UNIX and OpenVMS) or italic (Windows).
NOBLINK NOITALIC	Not blinking (UNIX and OpenVMS) or not italic (Windows). (default)
BOLD	Highlighted (or boldfaced).
NOBOLD	Not highlighted (or boldfaced). (default)
COLOR(<i>palette#</i>)	Assign specified <i>palette#</i> , where <i>palette#</i> is a palette number in the range 1 to 16. The default <i>palette#</i> is 1.
HIGHLIGHT	Highlighted (or boldfaced).
NOHIGHLIGHT	Not highlighted (or boldfaced). (default)
REVERSE	Reverse video.
NOREVERSE	Normal video. (default)
UNDERLINE	Underlined.
NOUNDERLINE	Not underlined. (default)

Discussion

The `.FILL` command defines a string to fill the entire window as well as default renditions for the window and every object on the window. Because you will normally want any input to be placed over the filled background, you should use this command at the beginning of your window definition before any other text is loaded. If you process input before filling the window, the fill string will overwrite the entire window.

On Windows, when used within an input window that contains buttons, *string* and *rendition* (with the exception of `COLOR`) will not extend into the area added onto the window to accommodate the buttons.

Script

.FILL

See also

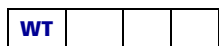
[“Renditions” on page 2-34](#) for information on how field-level renditions override window-level renditions.

Examples

The following example fills the window with blanks and sets the default attribute to reverse.

```
.fill " ", reverse
```


.FONT – Specify the font for a window



`.FONT palette_name`

Arguments

palette_name

The name of a font palette entry.

Discussion

The `.FONT` command allows a font to be specified for a general window, input window, or selection window.

Specifying a proportional font for a window causes the window to be sized according to the metrics of that font. Objects are positioned according to rows and columns based upon the metrics of that font, taking into consideration the sizing character — for example, a capital “W” represents a wider cell size than a lowercase “a” — and the existing leading calculations.

If a proportional font is specified, objects in the window that do not support proportional fonts may not display as expected. See [“Using Fonts on Windows” on page 18-40](#) for more information.

Although the `.FONT` command has no visual effect on UNIX and OpenVMS systems, it is preserved with the window definition so that the same window library can be used for both non-Windows and Windows systems.

See also

- ▶ [D_SETWNDFONT on page 16-231](#) and [D_GETWNDFONT on page 16-226](#) for more information on modifying and retrieving a window’s font at runtime. (`D_SETWNDFONT` and `D_GETWNDFONT` are subfunctions of `%U_WNDFONT`.)
- ▶ [FONT, NOFONT on page 2-50](#) and [PROMPTFONT, NOPROMPTFONT on page 2-60](#) if you need to specify a field or prompt font, respectively. (These are qualifiers for `.FIELD`.)

Examples

This example uses “my_input_font” for the window **inpwnd**.

```
.input inpwnd  
.font my_input_font
```

.INPUT – Begin an input window definition

WT	WN	U	V

`.INPUT` *name, rows, columns*

Arguments

name

The name of the input window (a maximum of 15 characters).

rows

The number of rows in the window.

columns

The number of columns in the window.

Discussion

The `.INPUT` command defines the name and size of an input window and marks the beginning of the input window definition.

Name must be a unique window name within the window library.

If you use `.PLACEMENT`, *rows* and *columns* must be set to values that enable the input window to fit in the application window, unless you first use `.DISPLAY_AREA`. (You can use `.DISPLAY_AREA` to make a display area that fits in the application window, even if the input window doesn't. See [.DISPLAY_AREA on page 2-24](#).) The `.PLACEMENT` command causes Script to logically place the input window, which in turn causes Script to determine whether the input window will fit in the application window. (See [.PLACEMENT on page 2-89](#).)

Examples

The following examples show the beginning of two input window definitions: **cusinf** and **realbig**. **Cusinf** is 12 rows by 40 columns, while **realbig** is 48 rows by 78 columns.

```
.input cusinf, 12, 40
.input realbig, 48, 78
```

.ITEM – Define a selection window entry

WT	WN	U	V

```
.ITEM text [, justification][, DEFAULT][, SELECT (character) ]
```

Arguments

text

An alphanumeric or quoted string that contains the text of the selectable entry.

justification

(optional) Designates how the entry text is aligned within the selection window column.

LEFT Align the left side. (default)

RIGHT Align the right side.

CENTERED Center the entry text.

DEFAULT

(optional) Specifies that this entry will be the default entry for the selection window.

SELECT

(optional) Specifies a quick-select character for the entry.

character

A single, printable quick-select character.

Discussion

The .ITEM command creates an entry in a selection window.

Entries are loaded into a selection window in the order specified, along with any intervening .TEXT, .BLANK, and .LINE entries.

The specified *text* is returned if the entry is selected either during input processing or through S_SELECT. UI Toolkit also adds leading and trailing blanks to the display but doesn't consider those blanks part of the entry.

You can use the S_SELECTCB and S_UPDATECB subroutines to process “check box” type selection windows. See [“Creating a check box selection window” on page 12-3](#) for information on designing check box selection windows. If you want your selection window to show a check mark to the left of the item selected, make sure the first character of *text* is a blank space. This space will be used for the check mark.

The default justification for entries is LEFT. When you select an entry that is right-justified or centered, any leading blanks are included in the returned entry.

On Windows, you should always use LEFT justification for selection items. RIGHT and CENTERED will not operate as expected because Windows uses a proportional font, rather than a fixed font, for selection window text.

You can specify the DEFAULT option for only one entry. The default entry can be changed or overridden by

- ▶ the .FIELD DEFAULT qualifier (which takes precedence over all other specifications for a default entry).
- ▶ subsequent calls to S_SELDFLT.
- ▶ the optional parameters to S_SELECT.

If you don't specify DEFAULT, UI Toolkit uses the first entry in the selection window as the default.

If you do not specify SELECT(*character*), UI Toolkit defaults to the first non-blank character in the entry.

On Windows, quick-select characters specified by .ITEM are ignored. The first non-blank character of the item's text is always used as the quick-select character. Also, quick-select characters are never highlighted and if a user presses an invalid quick-select character, no beep is generated.

Examples

This script defines the following selection window on UNIX or OpenVMS:

```
.select teams, 5, 8
.text "Team 1"
.line
.item Jim
.item Kim
.item Tim
.text "Team 2"
.line
.item Bill
.item Jill
.item Phil
.text "Team 3"
.line
.item Jan
.item Dan
.item Fran
.end
```

Team 1	Team 2	Team 3
Jim	Bill	Jan
Kim	Jill	Dan
Tim	Phil	Fran

.LINE – Draw a line



.LINE *name*, *length*[, POSITION (*row*, *column*)][, *direction*][, *rendition*, ...]

or

.LINE

Arguments

name

The name of the line (a maximum of 30 characters).

length

The length of the line.

POSITION

(optional) The position specification for the line.

row

The row at which the upper-left end of the line is positioned.

column

The column at which the upper-left end of the line is positioned.

direction

(optional) One of the following:

HORIZONTAL Draw a horizontal line. (default)

VERTICAL Draw a vertical line.

rendition

(optional) One or more of the following display renditions:

BLINK | ITALIC Blinking (UNIX and OpenVMS) or italic (Windows).

NOBLINK | NOITALIC Not blinking (UNIX and OpenVMS) or not italic (Windows). (default)

BOLD Highlighted (or boldfaced).

NOBOLD Not highlighted (or boldfaced). (default)

COLOR(*palette*#) Assign specified *palette*#, where *palette*# is a palette number in the range 1 to 16.

HIGHLIGHT Highlighted (or boldfaced).

NOHIGHLIGHT	Not highlighted (or boldfaced). (default)
REVERSE	Reverse video.
NOREVERSE	Normal video. (default)
UNDERLINE	Underlined.
NOUNDERLINE	Not underlined. (default)

Discussion

The first form of the .LINE command above is used to draw a line within a general or input window definition.

If *position* is not specified, the position of the line defaults to the current screen position.

The second form of the .LINE command adds a row with a horizontal line to a menu column or selection window definition. These horizontal lines do not contain leading or trailing blanks, and will not connect with the menu column or selection window border.

On Windows, .LINE in a selection window definition (.SELECT) has no effect.

Examples

This example draws a vertical line 8 rows long in reverse video positioned at row 2 and column 1 of a window.

```
.line line1, 8, pos(2, 1), vertical, reverse
```

This example separates the “Exit” menu entry from the “About” menu entry with a horizontal line.

```
.column c_general, "General"
.entry o_help, "Help", key(F1)
.entry o_exit, "Exit", key(F4)
.line
.entry u_about, "About"
.end
```

.LIST – Create a set of menu entries



```
.LIST list_name=entry_name[, ...]
```

Arguments

list_name

The name of the list set (a maximum of 10 characters).

entry_name

The identifier for a previously defined entry in the menu column.

Discussion

The .LIST command enables you to group specific menu entries together so that you can enable or disable them as a set.

You use the *list_name* with the M_ENABLE and M_DISABLE subroutines, which can enable or disable lists of entries. An entry must be defined with .ENTRY before you can include it in a list.

Examples

This example groups the **jobcost**, **salesman**, **smgr**, and **sched** entries into a list named **sysmgr**.

```
.list sysmgr=jobcost, salesman, smgr, sched
```

The second example groups the entries **po**, **inven**, and **bom** into a list named **retail**.

```
.list retail=po, inven, bom
```

.LISTCLASS – Begin a list class definition



```
.LISTCLASS name [, ARRIVE_METHOD (arrive_method) ] -  
    [, DOUBLE_CLICK_METHOD (double_click_method) ] -  
    [, FOOTERS ([fline_count [, STATIC]]) ] -  
    [, HEADERS ([hline_count [, STATIC]]) ] [, INPUT_SNAPSHOT]-  
    [, ITEMS (item_count) ] [, LEAVE_METHOD (leave_method) ]-  
    [, LOAD_METHOD (load_method) ] [, NONULL] [, OPTIONS (option_string) ]
```

Arguments

name

The name of the list class (a maximum of 15 characters). This name must be unique within the window library.

ARRIVE_METHOD

(optional) Specifies that you want the list processor to call an arrive method subroutine whenever an item is accessed (“arrived at”). You can overload this method at runtime with the L_METHOD subroutine. (See [L_METHOD on page 9-105](#).) You may want to use ARRIVE_METHOD when you want additional setup performed when the item is accessed. If ARRIVE_METHOD is not specified, no special action is taken when a new item is accessed.

arrive_method

The name of the (list) arrive method subroutine you have written. See [LARRIVE_METHOD/LLEAVE_METHOD on page 9-148](#) for more information.

DOUBLE_CLICK_METHOD

(optional) Specifies that you want the list processor to call a double-click method subroutine whenever the user double-clicks within the list. You can overload this method at runtime with the L_METHOD subroutine. (See [L_METHOD on page 9-105](#).) If DOUBLE_CLICK_METHOD is not specified and the user double-clicks on the current item, the list processor will assume that the item is being selected.

double_click_method

The name of the double-click method subroutine you have written.

FOOTERS

(optional) Specifies the number of footer lines for the list. If not specified, defaults to 0. If the FOOTERS qualifier is present but *fline_count* is not specified, the line count defaults to 1.

fline_count

The number of footer lines for the list.

STATIC

(optional) Indicates that the horizontal display area of the footer will not be adjusted by the L_VIEW subroutine, or by a D_LRIGHT or D_LLEFT request to L_PROCESS, or when the input window display is adjusted to encompass the current input field. The default is not static. For ActiveX Toolkit lists, text is centered in each column by default. If you specify STATIC, however, text will be left-justified in each column.

HEADERS

(optional) Specifies the number of header lines for the list. If HEADERS isn't specified, the default is 0. If the HEADERS qualifier is present but *hline_count* is not specified, the default line count is 1.

hline_count

The number of header lines for the list.

STATIC

(optional) Indicates that the horizontal display area of the header will not be adjusted by the L_VIEW subroutine, or by a D_LRIGHT or D_LLEFT request to L_PROCESS, or when the input window display is adjusted to encompass the current input field. The default is not static. For ActiveX Toolkit lists, text is centered in each column by default. If you specify STATIC, however, text will be left-justified in each column.

INPUT_SNAPSHOT

(optional) Specifies that the list will maintain input window control information for each item in the list, by using the I_SNAPSHOT subroutine. (See [I_SNAPSHOT](#) on page 8-82 for more information.) This option is useful when using input sets in a list but creates unnecessary overhead in other cases (such as selection processing).

If INPUT_SNAPSHOT is not specified, the default action is not to maintain item-specific input window control information.

ITEMS

(optional) Specifies that the number of items to be displayed at one time in the list follows. If ITEMS is not defined, the default number depends on the available screen-body space between the placement row (inclusive) and the information bar (exclusive). Display items such as borders, headers, and footers are made to fit within this area.

item_count

The number of list items to be displayed at one time. If *item_count* is too large for the screen, the default number (as defined above) will be used.

LEAVE_METHOD

(optional) Specifies that you want the list processor to call a leave method subroutine whenever the user leaves a list item. You can overload this method at runtime with the L_METHOD subroutine. (See [L_METHOD](#) on page 9-105 for more information.) You can use LEAVE_METHOD when you want the item to perform additional processing (such as

validating the input) when the user leaves it. If `LEAVE_METHOD` is not specified, no special action is taken when the user leaves the item.

leave_method

The name of the list leave method subroutine you have written. See [LARRIVE_METHOD/LLEAVE_METHOD on page 9-148](#) for more information.

LOAD_METHOD

(optional) Specifies that you want the list processor to call a load method subroutine whenever it loads existing items (on demand) into the list. You can overload this method at runtime with the `L_METHOD` subroutine. (See [L_METHOD on page 9-105](#).) If `LOAD_METHOD` is not specified, load requests are returned to the calling routine. See [L_PROCESS on page 9-111](#) for more information.

On Windows, you must write and use a *load_method* subroutine to enable scrolling of the list.

load_method

The name of the list load method subroutine you have written. See [LLOAD_METHOD on page 9-154](#) for more information.

NONULL

(optional) Specifies that the list is not allowed to remain empty. A list is null if no items are loaded into it, or if the last list item is deleted. If `NONULL` is specified, an empty list will cause the `L_PROCESS` subroutine to return a `D_ABORT` request, which asks the calling routine to immediately delete the list. The default is to enable a null list.

OPTIONS

(optional) Defines an option string to be used by `L_CREATE` when instances of this class are created.

option_string

One or more of the following creation options separated by commas:

ACTIVEX Specify that the ActiveX implementation of a Toolkit list will be used.
See [“Specifying the ACTIVEX option \(Windows\)” on page 2-86](#).

GRIDCOLOR (*[odd_color], [even_color]*)
Specify background colors for alternating ActiveX Toolkit list items.
See [“Specifying the ACTIVEX option \(Windows\)” on page 2-86](#).

NOACTIVEX Specify that a Toolkit list (non-ActiveX) will be used.

NOAUTOSIZE Suppress automatic sizing of the list when the container is resized. On Windows, the size of the list is not affected by the size of its container and can only be resized using `L_RESIZE`. This option is ignored on UNIX and OpenVMS systems. See the [DTS_LIST Discussion on page 13-14](#) for an explanation of how this feature can be used.

NOBAR Suppress the list's vertical and horizontal scroll bars.



On Windows, mouse wheel scrolling works even if you suppress scroll bars with NOBAR, HOHBAR, or NOVBAR.

NOCELLHBAR Suppress the list's horizontal scroll bar only when running in a UNIX or OpenVMS environment.

NOCELLVBAR Suppress the list's vertical scroll bar only when running in a UNIX or OpenVMS environment.

NOHBAR Suppress the list's horizontal scroll bar.

NOVBAR Suppress the list's vertical scroll bar.

Discussion

The .LISTCLASS command enables you to define a list class. A list class defines a set of list characteristics. When a list is created in a Toolkit application, its characteristics can be drawn from a given list class.

A list class defines features of lists that are likely to be common across multiple lists. When you define these features in classes, global changes in presentation style are simple. Because you define classes in scripts, global customization of distributed list classes is possible.

The placement of a list can be specified as a list class characteristic using the .PLACEMENT command. The placement coordinates can be overridden when a list is created with the L_CREATE subroutine; or the list can be created without being placed. If so, these coordinates will be retained and used when the list is placed.

If placement is not specified, the default placement is inherited from the associated input window specified in the L_CREATE subroutine.

When using the L_INPUT subroutine on Windows, be aware that the user can change the current list item without control returning to your program first. This means that the state of the input fields and the input set context can be propagated across list items. For example, if your program uses I_INIT to initialize fields in the input set, and the user moves to a different item in the list, the fields are still considered empty. This results in a different redisplay of the data, and validation occurs unexpectedly. There are two ways to solve this problem. You can use the INPUT_SNAPSHOT option to preserve input context and field states for each item, or you can use a list item arrive method to either initialize or display the data in the associated input window (depending on the content of the list item), and then use I_NEXT to establish the correct context.

Specifying the ACTIVEX option (Windows)

Specifying the ACTIVEX option for the OPTIONS qualifier creates an ActiveX implementation of a Toolkit list when displaying and processing the list in a Windows environment. (On UNIX and OpenVMS, this option has no effect; you get a standard Toolkit list. For information on the differences between Toolkit lists and ActiveX Toolkit lists, see [“Using Lists in Toolkit Applications”](#) on page 9-4.)

For ActiveX Toolkit lists, the GRIDCOLOR *odd_color* and *even_color* options are numbers between 0 and 511 that correspond to Synergy color definitions. *Odd_color* is used for the background color for odd numbered items in the list, and *even_color* for even numbered items. Both default to white. If an item spans multiple rows, all rows for that item use the same background color.

Additionally, note the following:

- ▶ *Odd_color* and *even_color* must be different colors. (If you want to set the entire background to one color, use the BackColor property. For information, see [“Published Properties for ActiveX Toolkit Lists”](#) on page 9-17.)
- ▶ Field attribute settings have precedence over GRIDCOLOR settings and the DarkItemColor and LightItemColor published properties for ActiveX lists.
- ▶ GRIDCOLOR settings are used only when a list is created. So changes to the Synergy colors used for *odd_color* and *even_color* won't affect lists that exist when the changes are made.

For information on Synergy colors, see [“Colors and the color palette”](#) in the “Synergy Windowing API” chapter of the *Synergy DBL Language Reference Manual*.

Examples

This example defines a list class named **my_list** which enables 14 items to be displayed at once. A load method subroutine called **my_load_method** has been written, which will be called by the Toolkit list processor to load entries into the list.

```
.listclass my_list, items(14), load_method("my_load_method")  
.placement 3, 1  
.end
```

This example defines a list class with one header named **my_activex_list** which enables 14 items to be displayed at once. A double click method subroutine called **my_dclick_method** has been written, which will be called by the list processor whenever the user double clicks within the list. This list will use an ActiveX Toolkit list when run on Windows.

```
.listclass my_activex_list, items(14), -  
  double_click_method("my_dclick_method"), -  
  options(activex), headers  
.placement 3, 1  
.end
```

.LOCAL – Define subsequent fields as local

WT	WN	U	V

.LOCAL

Discussion

The .LOCAL command specifies that subsequent fields in the current input window definitions are to be created, instead of drawn from the repository. This is the default state when an .INPUT command is issued. This command is used after you have specified .REPOSITORY_STRUCTURE to draw one or more fields from the repository, and you now want to create local fields.

An input window can contain local fields and fields from one or more repository structures.

Examples

This example defines an input window containing four fields. The first two fields, **empnam** and **empid**, are to be drawn from the repository. The last two fields, **start_time** and **end_time**, are local and will therefore be created.

```
.input emprec, 5, 35
.repository_structure empmas
.field empnam
.field empid
.local
.field start_time, d4, time, prompt("Enter starting time ")
.field end_time, d4, time(now), prompt("Enter quitting time ")
.structure time, start_time, end_time
.set wktime, time
.end
```

.PAINT – Define the paint character



.PAINT character

Arguments

character

A single, printable paint character.

Discussion

The .PAINT command determines which character “paints” an empty field to indicate where the user types input.

Initially the paint character is a blank. The .PAINT command resets the default paint character. Note that the PAINT qualifier of .FIELD will override the input window’s .PAINT character currently in effect for that field. Fields drawn from a repository always override .PAINT.

Paint characters specified by .FIELD PAINT or .PAINT have no effect in a Windows environment.

See also

[PAINT](#), [NOPAINT](#) on [page 2-59](#) for information about an alternative way to specify the paint character for an empty field. (PAINT is a qualifier for the .FIELD command.)

Examples

In the following example, the underscore character indicates where the user should type input.

```
.paint _
```

.PLACEMENT – Define the placement of a window



`. PLACEMENT row, column[, UNBOUNDED]`

Arguments

row

The screen row at which the upper-left corner of the window is placed (relative to the upper-left corner of the screen body, not the entire screen).

column

The screen column at which the upper-left corner of the window is placed.

UNBOUNDED

(optional) The specification that you want absolute window placement that is not bounded by UI Toolkit's screen. UNBOUNDED must be omitted for a list class.

Discussion

The .PLACEMENT command defines the location of an input, selection, or general window on the screen. You can also use it to specify the default placement for a list class. If both row and column are set to a value of 0, the window is centered in the application window.

The default position is row 1, column 1. Any border goes around the window's placement area.

When a selection window defined by the .SELECT command is processed, the position specified by the SELWND input field qualifier will override the position specified by .PLACEMENT. Therefore, with selection windows, .PLACEMENT is most useful when processing a selection window directly through U_LDWND and either S_SELECT or S_SELECTCB.

If .PLACEMENT is specified for a list class, you can override the placement when a list is created with the L_CREATE subroutine. If .PLACEMENT is not defined for a list class, the default placement is inherited from the associated input window specified in the L_CREATE subroutine.

Any window with a display area of up to 50 x 255 can be placed with the .PLACEMENT command. If you use a .PLACEMENT command for a window that Script considers to be larger than the application window, it will generate a non-fatal "Could not place window" error. Script will then continue processing. (Proto will generate the same error for windows larger than 24 x 80 that have a .PLACEMENT command.) Script determines if a window will fit in the application window

- ▶ on UNIX and OpenVMS by assuming that the application window is 132 columns by 50 rows.
- ▶ on Windows by assuming that the application window height and width are equal to the APP_HEIGHT and APP_WIDTH settings from **synergy.ini** or **synuser.ini**. (The default for APP_HEIGHT is 25, and the default for APP_WIDTH is 80.)

See also

[.DISPLAY_AREA](#) on [page 2-24](#) for more information on defining a movable display area (a view) for general windows (.WINDOW) and input windows (.INPUT).

Examples

The following .INPUT command in the example defines an input window named **cusinf** whose dimensions are 12 rows by 40 columns. The .PLACEMENT command places the upper-left corner of that window at screen row 4, column 21.

```
.input cusinf, 12, 40  
.placement 4, 21
```

The list class defined in the following example is placed at row 3, and column 1.

```
.listclass my_list, items(14), load_method("my_load_method")  
.placement 3,1  
.end
```


.REPOSITORY_STRUCTURE – Establish the current repository structure



```
.REPOSITORY_STRUCTURE structure_name[=alias_name], ["main_file"][, "text_file"]
```

Arguments

structure_name

The name of the repository structure to access.

alias_name

(optional) The name by which *structure_name* will be referenced within this input window (a maximum of 30 characters).

main_file

(optional) The name of the main repository file to use.

text_file

(optional) The name of the repository text file to use.

Discussion

The .REPOSITORY_STRUCTURE command specifies that subsequent fields in the current input window definition are to be drawn from the given repository structure when you are compiling your script. If no fields exist in the structure, an error will be generated. The specified repository structure is in effect until a .LOCAL, .END, or another .REPOSITORY_STRUCTURE.

Field qualifiers on subsequent .FIELD commands will override repository specifications.

Additional requirements for drawing field definitions from a repository are discussed in the [.FIELD Discussion on page 2-32](#).

The .REPOSITORY_STRUCTURE command automatically creates a .STRUCTURE in the window library with the specified structure name, containing all the referenced fields in the appropriate positions.

If you make changes to your repository structure after compiling your script, you will need to re-compile.

Use the *alias_name* argument in the following situations:

- ▶ When you include the same repository structure more than once within an input window.
- ▶ When you include two or more fields that have the same name although they are referenced from different repository structures.

In either of the preceding situations, you must rename the second and succeeding “same” structures using the *alias_name* argument. Note that there are no spaces between the equal sign (=) and the names.

After you have assigned an *alias_name* to a structure, any following references to the structure (for example, in a .SET) must use the alias name instead. If you have given the structure an *alias_name* and are referencing a field at runtime (for example, with I_NEXT, I_DSPFLD, or I_INPFLD), the reference to the field must have the following syntax:

```
alias_name.field_name
```

This syntax can be used only for fields that are in a renamed structure. The first 30 characters of the full reference above must be unique within the input window. Thus, we recommend that you use short alias names.

You can combine multiple repository structures, enabling their access as one structure, by using the implicit group feature of Repository. See “[Basic field information](#)” in the “Working with Fields” chapter of the *Repository User’s Guide* for more information.

If *main_file* and *text_file* are not passed or are blank, the repository main and text file specified by the environment variables RPSMFIL and RPSTFIL, respectively, will be used. If these environment variables are not set, UI Toolkit will use the default repository files (**rpmain.ism** and **rpstext.ism**) found in the directory specified by the RPSDAT environment variable.

If specified, *main_file* and *text_file* must be enclosed in quotation marks, and if you use an environment variable, it must be followed by a colon. For example:

```
.repository_structure s1, "D:\test\rpsmain.ism", "D:\test\rpstext.ism"
.repository_structure s2, "MAINFILE:", "TEXTFILE:"
```

See also

- ▶ The [.FIELD Discussion on page 2-32](#) for more information on repository fields.
- ▶ “[Basic field information](#)” in the “Working with Fields” chapter of the *Repository User’s Guide* for more information on using implicit groups to combine repository structures.

Examples

The following example defines an input window that contains four fields from the repository structure **cusmas**. A default set **cusinf** will be created that references the structure **cusmas**.

```
.input cusinf, 4, 45
.repository_structure cusmas, "RPS1:rpsmain.ism", "RPS1:rpstext.ism"
.field name
.field address
.field city
.field state
.end
```

The next example defines an input window that contains two fields from the repository structure **cusmas**, and one field from the repository structure **cushis**. The position specified for the field **prev_address** in the repository is overridden here. This example creates only one input set, **cusinf**, which references the first defined structure (**cusmas**). **Cushis** is a structure without a set here, and note that **cushis** is from the default repository (the **rpsmain.ism** and **rpstext.ism** files in the directory specified by the RPSDAT environment variable).

```
.input cusinf, 4, 45
.repository_structure cusmas, "RPS1:rpsmain.ism", "RPS1:rpstext.ism"
.field name
.field address
.repository_structure cushis
.field prev_address, pos(3,2)
.end
```

This example creates two input sets, **old** and **new**, which reference two identical structures, both of which are defined by the repository structure **cusmas**. In the program that uses this window, if you want to use **old** set, you would reference the **name** field. If you want to use **new** set, you would reference **new.name**.

```
.input cusinf, 4, 45
.repository_structure cusmas, "RPS1:rpsmain.ism", "RPS1:rpstext.ism"
.field name
.field address
.repository_structure cusmas=new
.field name
.field address
.set old, cusmas, name, address
.set new, new, name, address
.end
```

.SCRIPT – Define a version 6 or higher script file



.SCRIPT

Discussion

The .SCRIPT command defines the beginning of a version 6 or higher script file. It must be the first command in the file, and may only be preceded by comments.

.SELECT – Begin a selection window definition



```
.SELECT name, [rows], [column_width][, NOPAD]
```

Arguments

name

The name of the selection window (a maximum of 15 characters).

rows

(optional) The number of rows for the selection window.

column_width

(optional) The width of each column of the selection window.

NOPAD

(optional) The specification to suppress the blank columns that occur to the left and right of each entry in the selection window.

Discussion

The .SELECT command defines the name and size of a selection window, and marks the beginning of a selection window definition.

Name must be unique within the set of windows in the library. You can use *name* in the SELWND qualifier of the .FIELD command to attach the selection window to an input field. You can also pass *name* to the U_LDWND subroutine to load the window directly.

If *rows* is not specified or is zero (0), UI Toolkit uses the total number of entries. However, if the window contains more than 20 entries, the entries are evenly divided into as many columns as are required to use 20 rows or less.

If *column_width* is not specified, UI Toolkit uses the width of the widest entry. Note that this width does not include the leading and trailing blanks, which are always added to each entry (unless NOPAD is specified).

If either *rows* or *column_width* is not specified and you are using the NOPAD qualifier, you must use the appropriate comma placeholder for the missing qualifier.

A selection window must contain at least one item and may not contain more than 32767. (Note, however, that because Synergy cannot place windows that do not fit on the screen, the practical maximum number of entries will be much lower. For instance, if the size of the Toolkit's virtual screen is 25 rows by 80 columns, a selection window cannot be placed if it has more than 2000 entries—and 2000 would only be possible if there is no padding between the columns and each entry is a single character.)

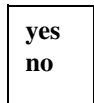
See also

[.ITEM](#) on page 2-77 for more information about defining selection window items.

Examples

This example creates a two-item selection window with a vertical format, as follows:

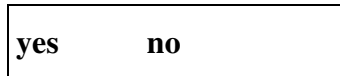
```
.select yesorno  
.item "yes"  
.item "no"  
.end
```



Changing the `.SELECT` command to

```
.select yesorno, 1, 10
```

creates the same selection window with a horizontal layout, as follows:



Each selection item in this example occupies 10 spaces and is displayed in its own column.

.SET – Define an input set



```
.SET set_name, [structure_name][, field_spec, ...]
```

Arguments

set_name

The name of the input set (a maximum of 30 characters).

structure_name

(optional) The name of the structure to associate with the set.

field_spec

(optional) The field specification for one or more fields in the set. (See “Field specifications” on page 2-35 for information.)

Discussion

The .SET command specifies a set of fields to be associated with one another and their order of reference. If you don’t issue a .SET command, a default set will be created that comprises all fields defined in the first structure in the order they are defined. The default set will have the name of the input window.

List fields in the order that you want them processed at runtime. An input set can contain up to 253 input fields. (See [L_INPUT on page 8-55](#), [L_DISPLAY on page 8-13](#), and [L_NEXT on page 8-65](#) for more information about the purpose and use of input sets.)

If no structure has been defined (with .REPOSITORY_STRUCTURE or .STRUCTURE), when a .SET command is issued, you must pass the default structure name as *structure_name* if you want the input set to have any fields. The default structure has the same name as the input window, and will include all the fields defined for the input window.

If you omit *structure_name* and *field_spec*, Toolkit creates an empty input set. (If you omit *structure_name*, you must also omit *field_spec*.) This is useful if you want a set of buttons that can be placed anywhere in a composite window. See “Buttons in composite windows” on page 6-6 for more information.

If you include *structure_name* and omit *field_spec*, all the fields within the specified structure will be used in the order of their definition within that structure. Note the following:

- ▶ All *field_specs* in the set must be members of the same structure. (See [.STRUCTURE on page 2-99](#) and [.REPOSITORY_STRUCTURE on page 2-91](#).)
- ▶ You don’t have to put all fields in sets. You can process fields individually as well.

Examples

All fields specified in the following example are members of the same structure. All sets are associated with the structure **cusinf**: **setone** consists of **cusnum** and **name**; **settwo** consists of **cusnum** and **state**; and **cusinf** consists of **cusnum**, **name**, **addr1**, **addr2**, **city**, **state**, and **zip**. When a *set_name* is used, its fields will be processed in the order they appear in the set specification.

```
.structure cusinf, cusnum, name, addr1, addr2, city, state, zip
.set setone, cusinf, cusnum, name
.set settwo, cusinf, cusnum, state
.set cusinf, cusinf, cusnum, name, addr1, addr2, city, state, zip
```

All fields in the next example are also members of the same structure, but this structure is defined in a repository (and includes the arrays **addresses**, **city**, and **state**). In this example, the first input set will enable users to modify **address[1]**, **city[1]**, and **state[1]** but not **address[2]**, **city[2]**, or **state[2]**, which belong to the second input set.

```
.repository_structure cusmas
.field name
.field address[1]
.field address[2]
.field city[1]
.field city[2]
.field state[1]
.field state[2]
.set shipping, cusmas, name, address[1], city[1], state[1]
.set billing, cusmas, name, address[2], city[2], state[2]
```


.STRUCTURE – Define the structure of a data area

WT	WN	U	V

```
.STRUCTURE data_area[, field_def | value][, ...]
```

Arguments

data_area

The name of the data area (a maximum of 30 characters).

field_def

(optional) A field definition has one of the following forms:

field

field@*character_pos*

field@*field_name*

field@*field_name*+*offset*

where *character_pos* is a base-one character position within the structure, *field_name* is a field previously specified within the structure, and *field_name*+*offset* is an offset from a field previously specified within the structure.

value

(optional) The number of pad characters, enclosed in angle brackets. You can use *value* to skip space in your data area between fields.

Discussion

The .STRUCTURE command defines field membership and position within a data area (usually a record).

.STRUCTURE creates a local structure within the current input window. A structure definition is used to verify that no input set contains fields from different structures; a field can be a member of only one structure. Note, however, that a field does not have to be a member of any structure.

At runtime, structure definitions are used to transfer data to and from the field's data position within its record buffer.

Although a given field can be a member of only one structure, you can specify up to 25 structures in an input window.

You cannot use .STRUCTURE for a repository structure (.REPOSITORY_STRUCTURE). .STRUCTURE is only needed for local fields.

If there are no repository structures and you do not include a `.STRUCTURE` statement for local fields, a default structure is defined. Its name is the name of the input window, and it comprises all fields in the order that their `.FIELD` commands are specified. (If the input window has no fields, the default structure will have no fields.)

Thus, if you need only one record containing all fields in the order defined, you don't need to define the record with `.STRUCTURE` because it will be created by default.

You can specify up to 507 *field_def* or *values*. If you do not specify either, all the fields will be used in the order they are defined. A field name can have a maximum of 30 characters. The maximum size of a structure is 9999 characters (**d4**). Text fields take up space that equals *size x dimension*.

In the first form of *field_def*, the position of *field* within the structure follows the previous field (or filler) definition. In the second form, *field* starts at position *character_pos* within the structure. The third and fourth forms position *field* relative to *field_name* within the structure. Note that if the field is specifically positioned by one of the latter three forms, the default position for the next field definition is after this field, regardless of this field's position within the structure. (Specifically, the next field will not default to the end of the structure.)

Examples

The following example defines a data area named **cusinf** that has the following fields: **cusnum**, **name**, **addr1**, **addr2**, **city**, **state**, **zip**.

```
.structure cusinf, cusnum, name, addr1, addr2, city, state, zip
```

The next example is very similar to the first, but it designates that 60 pad characters should be placed in the data area between the fields **name** and **city**.

```
.structure cusinf, cusnum, name, <60>, city, state, zip
```

The following example specifies **city** to be positioned 30 characters after the beginning of **address**. If **address** began at 60, **city** would begin at 90.

```
.structure cusinf, cusnam, name, address, city@address+30, state, zip
```

.TEXT – Add text to a window or menu column



`.TEXT name [, POSITION (row, column) ][, MARGIN (left, right) ] lines_of_text...`

or

`.TEXT text [, justification]`

Arguments

name

The name of the text (a maximum of 30 characters).

POSITION

(optional) The position specification for the text. The position doesn't have to be in the display area, but it must be within the window.

row

The row at which the upper-left corner of the text is positioned.

column

The column at which the upper-left corner of the text is positioned.

MARGIN

(optional) The margin specification for the text. Margins are the number of spaces from the left and right edge of the window.

left

The number of columns specified for the left margin. (If *left* is less than or equal to *column*, this setting applies to all but the first line, whose indent in this case is controlled by the *column* argument for POSITION.)

right

The number of columns specified for the right margin.

lines_of_text

One or more lines of text in general or input windows.

text

A text string in menu columns or selection windows.

justification

Designates how the text is aligned within the entry-text area, as opposed to the entire width of the menu or selection column. It must be one of the following:

LEFT	Left-justify the column text. (default)
RIGHT	Right-justify the column text.
CENTERED	Center the column text.

Discussion

The first form of the .TEXT command above is used to add text to a general or input window. *Lines_of_text* can contain data control commands that control the renditions of the text. See [“Data Control Commands and Text”](#) below.

When justified text is needed on the same line as an input field, make sure that the .FIELD command occurs after the .TEXT command.

The second form of the .TEXT command adds the contents of *text* to a column or selection window. This text is not a selection. The required width of this entry is two spaces longer than the length of text, and the text is loaded starting at the second character position (when left-justified). The width of the selection column is increased, if necessary, to accommodate the width of the entry.

If you don't specify LEFT, RIGHT, or CENTERED, the text will be left-justified by default.

If you have a selection window defined by .SELECT that has more than one column but does not have corresponding text entries in the same row of each column, left and right movement within the window won't be consistent. We therefore recommend that you make sure each column in a multi-column selection window contains its text entries in the same row.

.TEXT in a selection window definition (.SELECT) is ignored on Windows.

Note that .TEXT is optional (unless you use the script in Composer, which requires it). If you include a line of text that is not preceded by a .TEXT command, Script adds the text to the window, using margin and rendition settings that match the previous line of text processed by Script, and positioning the text to follow the previous line of text. If Script has not previously processed a line of text, Script uses the following settings: a position of 1,1; no margins; no attributes; and color 1.

Data Control Commands and Text

Data control commands affect the position and display of text within general and input windows.

The script compiler initially loads text into the window starting at window row 1, column 1, unless a data control command changes the position. Each time the script compiler loads text into the window, it updates the position. (The .FIELD command also will update the position.)

The maximum length of a text line in a script file is 255 characters. If there are too many text lines for the windows, all overflow lines are ignored. Tabs are converted into a space.

If a word starts with a backslash (\), it is considered a data control command. Data control commands can be anywhere on a text line, even embedded in text. A dash (–) after the backslash resets the command, but only commands that can be reset process the dash; all others ignore it. Data control commands are one character long, and any arguments must be enclosed in parentheses.

Rendition commands

You can turn five window renditions on and off in the script file:

- ▶ Blink (on UNIX and OpenVMS) or italic (on Windows)
- ▶ Color
- ▶ Highlight (bold on Windows)
- ▶ Reverse video
- ▶ Underline

The lines in the script file are processed sequentially, so any text after a rendition change is displayed with the new renditions until you change renditions again.

In each window definition, attributes are initially clear. Once set, they remain set until explicitly cleared.

The commands for setting and resetting the renditions are shown in the following table.

Rendition Commands	
Command	Meaning
\b \i	Blinking on (UNIX and OpenVMS) or italic (Windows)
\-b \-i	Blinking off (UNIX and OpenVMS) or italic off (Windows)
\c (<i>palette#</i>)	Color palette number (default value is 1)
\h	Highlighting (bolding) on
\-h	Highlighting off
\r	Reverse video on
\-r	Reverse video off
\u	Underlining on
\-u	Underlining off

For example:

```
.window renditions, 20, 45
.place 3, 18
.title "Renditions"
.border rev
.text mytext, position(2,2), margin(2,2)
\bThis blinks.\-b\n
\hThis is bold.\-h\n
\rThis is in reverse video.\-r\n
\uThis is underlined.\-u\n
This isn't.\n
.trim
.end
```

Miscellaneous commands

The remaining data control commands are described below:

\j (*option*)

Justifies the text that follows over blanks within the margins. Justification doesn't take effect until you have a new line or word wrap. If you've done a fill on the line, no justification will occur.

Option can be one of the following:

- l** Left. (default)
- r** Right.
- c** Center.
- f** Fill line (right- and left-justify text).

Lines are left-justified by default. The next new line is either the next line after a **\n** or the next line of overflow text.

When the **f** option is set, spacing between words on a line will be adjusted so the line fits exactly between the margins. Line filling is only done when an implicit new line (because of word wrap) occurs. An explicit new line (because of a **\n**) will not cause the previous line to be filled.

If the last specified renditions (such as **\b**, **\h**, **\r**) were specified on a previous line, the renditions will apply not only to the justified text, but to the entire line (except the margins). If you specify renditions on the same line as the justified text, the renditions will apply only to the justified text (blanks on either side of the text will receive any previously-specified renditions).

If you need to center text that's enclosed in a box, draw the box at the end of the window definition.

For example:

```
.window justify, 10, 45
.title "Justifying"
.border rev
```

```
.text mytext, position(1,2), margin(2,2)
\n
\j(l)This text is left-justified.\n
\n
\j(r)This text is right-justified.\n
\n
This
text
\j(c)
is
centered.
\n
\n
\j(f)This text left- and right-justified.
.end
```

\n

Moves the window display position to the next line's left margin. It also forces justification after a justify (**\j**) command.

For example:

```
.window newline, 20, 45
.place 3, 18
.title "New Lines"
.border rev
```

```
.text mytext, position(2,2), margin(2,2)
```

The **\n** command moves the text display to the next line. If a script contains two new line commands in a row (**\n\n**), it will skip two lines.

```
\nLike that.\n\n
```

This command is sometimes necessary when you want to justify text with the **\j** command. Justification occurs at the next new line, so you may need to use a **\n** command after the text you want justified.

```
.trim
.end
```

\s and \-s

The **\s** command (default) adds a space to the end of a text line in the script file so that when words wrap across lines in the window, they are properly delimited by a space. This spacing occurs only if wrap mode is active and the last word on the line in the script file is not already followed by a space.

The **\-s** command resets this feature.

For example:

```
.window spacing, 3,10
.title "Spacing"
```

```
.border rev
.text mytext
word
wrap
\s
word
wrap
.end
```

This script first uses the default settings of `\s` and `\w` (see description of `\w` later in this section), which means that the first line of the resulting window is as follows:

word wrap

When we turn off the default spacing, however, “wrap” will still be displayed on the same line as “word” because of the automatic word wrap, but the space will not be inserted automatically. The second line of text in the resulting window is therefore:

wordwrap

\w and \-w

The `\w` command (default) turns word wrap on. If a word overflows the current display line, the entire word is moved to the left margin on the next line, and trailing blanks on the previous line are trimmed. If the text overflows the last line of the window, it is ignored. If wrap is not set, any overflow at the end of a line is ignored.

The `\-w` command turns word wrap off.

For example:

```
.window wrap, 20, 25
.title "Wrap"
.border rev
.text mytext, margin(2,2)
\n\n
```

Wrap mode is the default in the script compiler, which makes it easy to load different blocks of text into the same window. You may, however, want to turn off the wrap mode with `\-w`.

```
\-w
\n\n
Now wrap mode\n
is turned off.\n
If text goes past\n
the window's edge,\n
it simply will not appear.
.end
```


Special characters

`\.`

Puts a single period in the text. This command is especially useful if you want to have a leading period in a line of text. This command is unnecessary if the period is within or at the end of text.

`\\`

Places a backslash into the text. (See the example under `\;` below.)

`\;`

Treats the text line as a comment. The end of the text line is the end of the comment. The comment line must be within a window definition and on its own line.

For example:

```
.window comments, 20, 45
.place 3, 18
.title "Comments"
.border rev
.text mytext, position(2, 2), margin(2, 2)
When you are developing with UI Toolkit, you may find it useful or
necessary to include comments within scripts that you do not want to
display. To do this, begin a comment with \\;.
\n\n
\;For example, this sentence will not be displayed.
\;To continue a comment on another line, you must begin the
\next line with another \\;.
.trim
.end
```

Comment lines can exist outside of a text definition if preceded by a semicolon (;). See [“Comments and blank lines” on page 2-7](#) for more information.

Examples

This example displays text at row 2 and column 2 of a general or input window—2 spaces in front of the left and right margins.

```
.text mytext, position (2,2), margin (2,2)
This is some text in my window.
And here is some more text.
```

The following example adds the text “System Manager Functions” to the menu column. This text is aligned with the left side of the column entries.

```
.text "System Manager Functions"
```

.TITLE – Define a window title



`.TITLE title[, ON|OFF][, rendition, ...][, border][, position]`

Arguments

title

The title for a window.

ON

(optional) Window displays a title. (default)

OFF

(optional) Window does not display a title.

rendition

(optional) One or more of the following display renditions:

BLINK | ITALIC

Blinking (UNIX and OpenVMS) or italic (Windows).

NOBLINK | NOITALIC

Not blinking (UNIX and OpenVMS) or not italic (Windows). (default)

BOLD

Highlighted (or boldfaced).

NOBOLD

Not highlighted (or boldfaced). (default)

COLOR(*palette#*)

Assign specified *palette#*, where *palette#* is a palette number in the range 1 to 16. Default is 1.

HIGHLIGHT

Highlighted (or boldfaced).

NOHIGHLIGHT

Not highlighted (or boldfaced). (default)

REVERSE

Reverse video.

NOREVERSE

Normal video. (default)

UNDERLINE

Underlined.

NOUNDERLINE

Not underlined. (default)

border

(optional) Designates which border holds the title. *Border* must be one of the following:

TOP

Top border. (default)

BOTTOM

Bottom border.

LEFT Left border.

RIGHT Right border.

position

(optional) Determines the position of the title within the border. *Position* must be one of the following:

BEGINNING Position at the beginning of the border.

CENTER Position at the center of the border. (default)

END Position at the end of the border.

VALUE An absolute column position within a border.

Discussion

The `.TITLE` command defines a window title name, its renditions, and its border position.

The title is displayed only if both the border and title are on. You can list the *renditions*, *border*, and *position* in any order, but *title* must always be first. If the title is longer than one word, it must be enclosed in quotation marks.

The window must be as wide as the title for the title to be displayed.

Any repeated options override previous ones.

On Windows, the Windows theme in use determines the location of the title; Toolkit settings have no effect.

Examples

The following example defines a highlighted border in color palette number 3, and then defines the title “My To Do List” in color palette number 3.

Your window will resemble [figure 2-3](#) (below) on UNIX and OpenVMS and [figure 2-4](#) (below) on Windows.

```
.border on, highlight, color(3)
.title " My To Do List ", color(3)
```

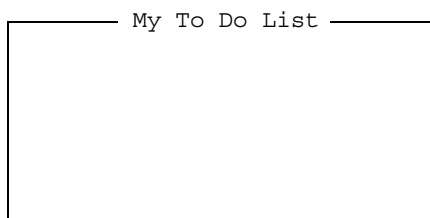


Figure 2-3. Example of a window title in a UNIX or OpenVMS environment.



Figure 2-4. Example of a window title in a Windows environment.

The second example defines the title “New Customer,” which will be placed in the center of the left border. The border must be ON and the window must be at least 12 columns wide for the title to be displayed.

```
.title "New Customer", left, center
```

.TRIM – Trim extra blanks from end of window



```
.TRIM [#lines_to_leave]
```

Arguments

#lines_to_leave

(optional) The number of blank lines to leave at the bottom of the window after it is trimmed.

Discussion

The .TRIM command tells the script compiler to trim extra blank lines from the end of the window and redefine the window height given in the .WINDOW or .INPUT command. If *#lines_to_leave* is omitted, all blank lines will be trimmed.

If you have an empty window and either omit *#lines_to_leave* or specify a value of 1, UI Toolkit will create a one-line window.

The displayed height cannot be less than one line, so .TRIM cannot reduce a window to 0 lines.

Examples

This example trims all blank lines but one from the bottom of the window.

```
.window h_general, 5, 40
.placement 1,1
.text mytext, position(1,1)
This window is trimmed to three lines, with the last line blank
.trim 1
.end
```

.USER_DEFINE – Define a user data set



`.USER_DEFINE #fields, field_size`

Arguments

#fields

The number of fields to be in the user data set.

field_size

The size of each data set field.

Discussion

The `.USER_DEFINE` command defines the number of fields and the size of each field in a user data set.

If you have previously defined a user data set for the window (by a previous `.USER_DEFINE` command), that data set and any data that has been placed in the set is replaced. The specified user data set is initialized to blanks.

Use the Synergy DBL `W_FLDS` subroutine to process user data sets. For more information, see [W_FLDS](#) in the “Synergy Windowing API” chapter of the *Synergy DBL Language Reference Manual*.

Examples

The following example defines a user data set that has 5 fields of 10 characters each.

```
.user_define 5, 10
```

This window has a copy of its text in another language. At runtime, the program can use either the English or the Spanish version.

```
.window nofile, 3, 30
.text nofile, position(2,1)
Cannot find file.
.user_define 1, 30
.data 1, "No puedo encontrar el archivo."
.end
```

.VERSION – Define the UI Toolkit version



```
.VERSION "version"
```

Arguments

version

The version of UI Toolkit from which the script was generated.

Discussion

This command is generated by UI Toolkit and appears immediately following the .SCRIPT command. This command is ignored by Toolkit when reading scripts, therefore it is not required.

Examples

This script was generated by version 8.3 of Composer.

```
.version "8.3"
```

.WFIELD – Define a general window field



`.WFIELD name, LENGTH (length) | TEXT (text) [, POSITION (row, column)] –
[, rendition, ...]`

Arguments

name

The name of the window field (a maximum of 30 characters).

LENGTH

Specifies that the length of the field follows.

length

The length of the field.

TEXT

Specifies that the text of the field follows.

text

The text in the window field.

POSITION

(optional) The position specification for the window field.

row

The row at which to position the field.

column

The column at which to position the field.

rendition

(optional) Used only with TEXT. One or more of the following display renditions:

BLINK | ITALIC

Blinking (on UNIX and OpenVMS) or italic (on Windows).

NOBLINK | NOITALIC

Not blinking (on UNIX and OpenVMS) or not italic (on Windows). (default)

BOLD

Highlighted (or boldfaced).

NOBOLD

Not highlighted (or boldfaced). (default)

COLOR(*palette#*)

Assign specified *palette#*, where *palette#* is a palette number in the range 1 to 16.

HIGHLIGHT	Highlighted (or boldfaced).
NOHIGHLIGHT	Not highlighted (or boldfaced). (default)
REVERSE	Reverse video.
NOREVERSE	Normal video. (default)
UNDERLINE	Underlined.
NOUNDERLINE	Not underlined. (default)

Discussion

Defines a window field (which is different from an input field). The *row* and *column* values set the field's beginning position within a window. The field's length is defined either by the value of *length* or by the length of the text. Fields cannot be redefined and are numbered sequentially by the script compiler. The window display position is updated to the end of the field. If you do not specify *row* or *column*, the script compiler uses the current row and column position.

Use the `W_FLDS` subroutine to process window fields, not `I_INPUT`. For more information, see [W_FLDS](#) in the “Synergy Windowing API” chapter of the *Synergy DBL Language Reference Manual*.

Examples

```
.window functions, 15, 20
.title "Function Fields"
.wfield add, position(6,5), text("Add")
.wfield sub, position(8,5), text("Subtract")
.wfield mult, position(10,5), text("Multiply")
.wfield div, position(12,5), text("Divide")
.wfield mod, position(14,5), text("Modulo")
.end
```

.WINDOW – Begin a general window definition



`.WINDOW` *name*, *height*, *width*

Arguments

name

The name of the window (a maximum of 15 characters).

height

The height of the window in rows.

width

The width of the window in columns.

Discussion

The `.WINDOW` command defines the name and size of a general window and marks the beginning of the general window definition.

The window defined by `.WINDOW` is a general or display-type window that contains only text. You might use this type of window to display such items as messages, errors, or help information.

The window *name* must be unique within the set of windows in the library.

If you use `.PLACEMENT`, *height* and *width* must be set to values that enable the general window to fit in the application window, unless you first use `.DISPLAY_AREA`. (You can use `.DISPLAY_AREA` to define a display area that fits in the application window, even if the general window you are defining doesn't. See [.DISPLAY_AREA on page 2-24](#).) The `.PLACEMENT` command causes Script to logically place the general window, which in turn causes Script to determine whether the window will fit in the application window. (See [.PLACEMENT on page 2-89](#).)

See also

- ▶ [.INPUT on page 2-76](#) for more information about defining input windows.
- ▶ [.SELECT on page 2-95](#) for more information about defining selection windows.
- ▶ [.DISPLAY_AREA on page 2-24](#) for more information about defining a view into the window.
- ▶ [“Text processing” on page 1-11](#) for more information on processing.

Examples

The following example defines a window named **win**, which is 9 rows by 12 columns.

```
.window win, 9, 12
```

Script Examples

The following are examples of window scripts. If you want, you can type these examples into a script file and then run Proto or Composer to see what the windows look like.

The following example defines a menu column with the name “Utilities” on the menu bar. The internal name is **util**. It contains three entries and defines a shortcut for each.

```
.column util, "Utilities"
.entry abort, Abort, key(^A)
.entry exit, "Exit function", key(F4)
.entry o_help, Help, key(Help)
.end
```

The next example defines an input window with the title “Address Book.” It contains seven fields, which are painted with the dot character to indicate where the user types input. The border is in reverse video. The first field, **lastnm**, is required—the user can’t skip it. Input to the **state** field is displayed in uppercase.

The **phone** field is left blank if the user enters a value of 0; input is redisplayed in the format (XXX) XXX-XXXX.

```
.input address, 14, 60
.place 2, 10
.border reverse
.title " Address Book "
.paint '.'
.field lastnm, a15, required, position(2,2), prompt("Last name ")
.field firstnm, a15, position(3,2), prompt("First name ")
.field addr, a30, position(4,2), prompt("Address ")
.field city, a20, position(5,2), prompt("City ")
.field state, a2, uc, position(5,35), prompt("State ")
.field zip, d5, position(5,45), prompt("Zip code ")
.field phone, d10, fmt("(XXX) XXX-XXXX"), position(6,2), bzro, -
    prompt("Phone ")
.end
```

The following example defines a selection window called **prod** that has four entries, whose quick-select characters are **F**, **R**, **S**, and **U**. The entries in the window are “Full System,” “Runtime(s),” “Support,” and “Update,” respectively.

```
.select prod, 5, 13
.item "Full System"
.item "Runtime(s)"
.item "Support"
.item "Update"
.end
```

Compiling Scripts

Once you have created a script file, it must be processed for use by UI Toolkit. The Script compiler converts window and column scripts into window and column definitions. These definitions are then loaded into a window library (ISAM file) created by Script. (Proto and Composer can also compile a script file.)

You can compile scripts in the following ways:

- ▶ Run the Script compiler interactively, providing the necessary information when prompted (see below)
- ▶ Run the Script compiler using the command-line syntax (see [“Running Script using the command-line syntax” on page 2-119](#))
- ▶ Access the Script compiler from your application using the external routine interface (see [“Accessing the Script compiler using the external routine interface” on page 2-120](#))

Running Script interactively

When you run Script interactively, it will prompt you for the information it needs. Files are compiled one at a time, but after compiling a file, you’ll be prompted to compile another.

1. Start Script with your system’s execute command.

On...	Do this...
Windows and UNIX	At a command prompt, enter <code>dbr WND:script</code>
OpenVMS	At a command prompt, enter <code>run WND:script</code>

Script will display this prompt:

Window Libr:

2. Enter the name of the window library that will contain the compiled window and column definitions. Script will search for an existing library and modify it by loading the specified script files. If you append `/c` (or `/C`) to the library name, Script either creates a new library file, if it doesn’t already exist, or completely overwrites the existing library.

You are then prompted with the following:

Script File:

3. Enter the name of the file that contains the window script(s) that you want to compile. The default extension is **.wsc**.
 - ▶ If Script cannot find or create the library file, it displays an error message then waits for valid input.
 - ▶ If Script cannot save because of locked window library records, it waits two seconds and tries again. Script attempts this ten times before displaying a message and prompting you again for a filename.
 - ▶ Script cannot process a line that is longer than 255 characters.
4. After Script creates the window library, it prompts for another script file. Type another filename or press ENTER at the prompt to exit. (Alternatively, you can type an end-of-file character to exit.)



When Script ends, it will exit with a status of `D_EXIT_FAILURE` if it encountered any errors (0 on OpenVMS, 1 on all other systems). Otherwise, it will exit with a status of `D_EXIT_SUCCESS` (1 on OpenVMS, 0 on all other systems). Most operating systems have commands that enable you to test for the exit status. See the documentation for your operating system for more information.

Running Script using the command-line syntax

Running Script from the command line enables you to compile multiple files at once. You can also put these commands in a batch file to build windows quickly and automatically.

On Windows and UNIX, from the command line run:

```
dbr WND:script [-c library_name | -l library_name] [-i script_file[, ...]] [-o error_file]
```

On OpenVMS, define this symbol:

```
script==$WND:script.exe
```

and then use this syntax:

```
run script [-c library_name | -l library_name] [-i script_file[, ...]] [-o error_file]
```

Arguments

-c library_name

Create a new library. The default extension is **.ism**.

-l library_name

Specify an existing library. The default extension is **.ism**.

-i script_file [, ...]

Specify one or more script files that you want to compile. The default extension is **.wsc**.

-o *error_file*

Specify an error log file. The default extension is **.err**.

Discussion

When running Script from the command line, you must specify either a library name or script file(s). The library can be one that already exists (**-l**) or a new one (**-c**)—but not both.

If no library file is specified, *library_name* will default to the same base name as the first script file specified, but with the extension **.ism**. (This works only with an existing library file; it will not create a new one.)

If no script file is specified, then a single script will be processed which has the same name as the library file specified, but with a **.wsc** extension.

You can specify up to 255 script files, which are processed in the order specified.

Specify **-o** to write error output to a file. If there are no errors, there will be no file. If **-o** is not specified, error output is directed to standard output (the terminal).

Accessing the Script compiler using the external routine interface

The following routines enable you to call the Script compiler directly from your application:

- ▶ **%SCR_CLOSELIBRARY**
- ▶ **%SCR_ERRORCOUNT**
- ▶ **%SCR_OPENLIBRARY**
- ▶ **%SCR_PROCESS**

These routines are contained in **WND:script.elb** (**script_sh.exe** on OpenVMS). To call these routines, you must either link **script.elb** with your program or open it using the **OPENELB** subroutine.



A sample program (**scrcmp.dbl**) is included in the toolkit\examples\script directory in your distribution.

%SCR_CLOSELIBRARY – Close the currently open window library

WT		U	V

```
status = %SCR_CLOSELIBRARY
```

or

```
xcall SCR_CLOSELIBRARY
```

Return value

status

True if successful; false if not. (^VAL)

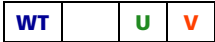
Discussion

The %SCR_CLOSELIBRARY function closes the window library that was opened by %SCR_OPENLIBRARY. If an error occurs, the error information is sent to the ESCRIPTERR_METHOD routine before this function returns.

See also

- ▶ [E_METHOD on page 7-35](#) for more information on registering environment methods.
- ▶ [EScriptErr_METHOD on page 7-80](#) more information on registering and writing a routine for handling script compilation errors.

%SCR_ERRORCOUNT – Return script compilation error information



num_error = %SCR_ERRORCOUNT ([*type*][, *scope*])

Return value

num_error

The number of script compilation errors and/or warnings. (^VAL)

Arguments

type

(optional) One of the following: (**n**)

D_ERROR	Report errors only. (default)
D_ALERT	Report warnings only.
D_ALL	Report both errors and warnings.

scope

(optional) One of the following: (**n**)

D_LIBRARY	Report the count for all scripts compiled in the window library that was opened by %SCR_OPENLIBRARY. (default)
D_SCRIPT	Report only the count for the most recently compiled script.

Discussion

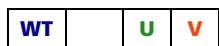
The %SCR_ERRORCOUNT function returns the number of script compilation errors and/or warnings for all scripts compiled in the currently open window library (opened by %SCR_OPENLIBRARY) or for the most recently compiled script.

The actual script compilation errors themselves are sent to the ESCRIPTERR_METHOD routine.

See also

- ▶ [E_METHOD on page 7-35](#) for more information on registering environment methods.
- ▶ [EScriptErr_METHOD on page 7-80](#) more information on registering and writing a routine for handling script compilation errors.

%SCR_OPENLIBRARY – Open a window library



```
channel = %SCR_OPENLIBRARY (library [, create])
```

or

```
xcall SCR_OPENLIBRARY (library [, create])
```

Return value

channel

The channel of the opened library if successful; otherwise 0. (^VAL)

Arguments

library

The name of the window library file to open. (a)

create

(optional) A flag that indicates whether to create the library file. (n)

D_CREATE_NEVER Do not create. (default)

D_CREATE_IFNEEDED Create if it doesn't exist.

D_CREATE_ALWAYS Create even if it exists.

Discussion

The %SCR_OPENLIBRARY function opens the specified window library file. If the library is already open, it is closed first and then reopened. If the open is successful, the channel is opened in the current environment. To promote it to global, use the U_GBLCHN subroutine.

If *library* does not specify an extension, it defaults to **.ism**.

If an error occurs, the error information is sent to the ESCRIPTERR_METHOD routine before this function returns.

A macro is provided in **tools.def** to simplify creating a library:

```
.define SCR_CREATELIBRARY(library) SCR_OPENLIBRARY(library,  
&                                    D_CREATE_ALWAYS)
```

See also

- ▶ [E_METHOD on page 7-35](#) for more information on registering environment methods.
- ▶ [ESCRPTERR_METHOD on page 7-80](#) more information on registering and writing a routine for handling script compilation errors.

%SCR_PROCESS – Compile a script file into the open window library



```
status = %SCR_PROCESS (script)
```

or

```
xcall SCR_PROCESS (script)
```

Return value

status

True if successful; false if not. (^**VAL**)

Arguments

script

The name of the window script file to compile. (**a**)

Discussion

The %SCR_PROCESS function compiles the specified script file into the window library that was opened by %SCR_OPENLIBRARY.

If *script* does not specify an extension, it defaults to **.wsc**.

If any errors occur (including no open library file, errors opening the script file, or syntax errors in the script), the error information is sent to the **ESCRPTERR_METHOD** before this routine returns. (If Script cannot save to the window library, Script waits two seconds and tries again to save. Script attempts this ten times before displaying a message and prompting you again for a filename.)

See also

- ▶ [E_METHOD on page 7-35](#) for more information on registering environment methods.
- ▶ [ESCRPTERR_METHOD on page 7-80](#) for more information on registering and writing a routine for handling script compilation errors.

Using the Script-to-Repository Conversion Utility

To help move field information from existing script files into a repository, you can use the **scriidl** conversion utility. Conversion is a two-step process:

1. Run the **scriidl** utility to generate a Synergy Data Language file.
2. Update an existing repository using Repository’s Load Repository Schema utility.

Running scriidl

Scriidl creates a Synergy Data Language file containing input field information extracted from the script file. Non-input windows, such as .WINDOW and .COLUMN, are ignored.



If your .INPUT window scripts do not contain .STRUCTURE statements, you must add them so that **scriidl** knows which structure to add/update in the repository. See [.STRUCTURE on page 2-99](#).

1. Run **scriidl** on your script file, using the syntax in the table below.

On...	Do this...
Windows and UNIX	At a command prompt, enter dbr WND:scriidl
OpenVMS	At a command prompt, enter run WND:scriidl

An input window is displayed on the screen. (See [figure 2-5](#).)

2. Specify the repository main and text files.
 - ▶ In the **Repository main file** field, enter the name of the target repository main file.
 - ▶ In the **Repository text file** field, enter the name of the target repository text file.
3. In the **Script file** field, specify the name of the existing script file.

A new script file is generated with a **.new** extension. This file references the structures in the repository above. (Repository filenames are added only if they override the default filename.)

A Synergy Data Language file is generated with an **.sdl** extension. If errors occur during the conversion, an error log file is created with an **.err** extension. (See [“Errors during conversion” on page 2-127](#).) All of these files inherit the name of the original script file but with different extensions.

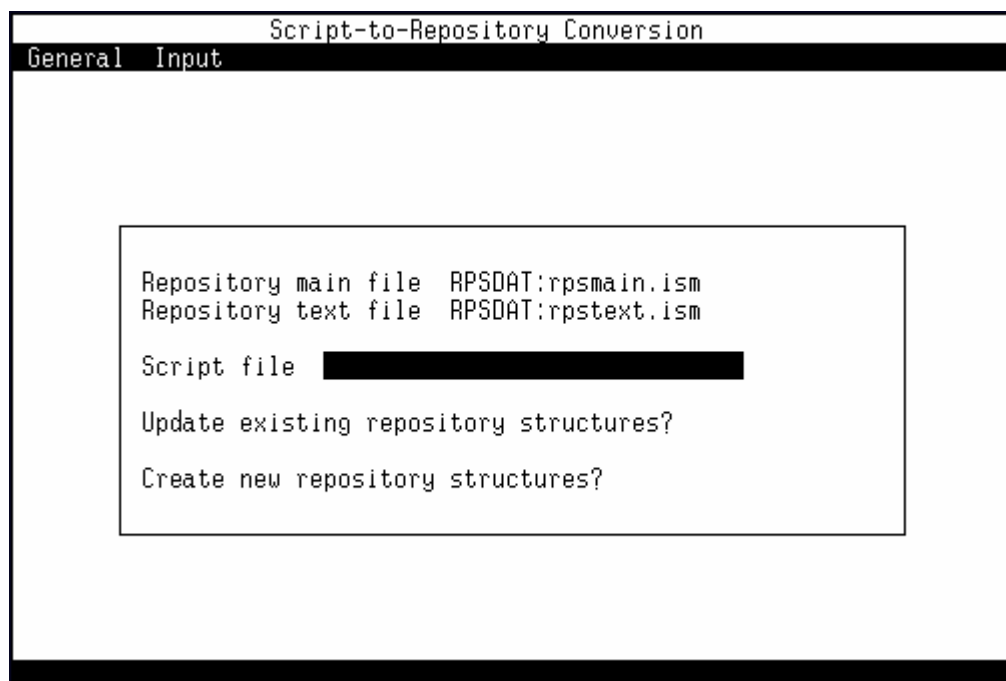


Figure 2-5. *Scridl* input window.

4. Specify whether you want to update the existing repository structures.

Select **Yes** if you want **scridl** to update any existing repository structures that have the same names as structures in your script file. (Note that **scridl** is not case sensitive.) For each matching structure, **scridl** will do the following:

- ▶ Add appropriate `.REPOSITORY_STRUCTURE` statements to the output script file. (The contents of the repository file override the defaults.)
- ▶ Remove `.STRUCTURE` statements.
- ▶ Add `.LOCAL` statements for fields not contained in repository structures.
- ▶ Generate errors if fields within an existing structure are not found in the corresponding repository structure or if the type, size, dimension, or position in the structure of those fields do not match the repository specification.

If all of these tests pass, all the other script qualifiers for this field will be copied to the Synergy Data Language repository field specification and removed from the output script specification. (Exceptions to this rule are the formats for date- and time-type fields.)

Select **No** if you want **scridl** to ignore structure names that already exist in the repository. The output script remains unchanged for these structures.

5. Specify whether you want to create new repository structures.

Select **Yes** if you want **scridl** to do the following when it encounters structure names that are not in the repository:

- ▶ Create the structure names in the Synergy Data Language output file.
- ▶ Add the appropriate `.REPOSITORY_STRUCTURE` and `.LOCAL` statements to the output script file. (Repository filenames are added only if the contents of the file override the defaults.)
- ▶ Remove the `.STRUCTURE` statements.
- ▶ Use `.FIELD` statements for fields contained in those structures to create Synergy Data Language output in order to generate those fields.
- ▶ Remove output script qualifiers (except for date- and time-type fields, which remain in the script).

Select **No** if you want **scridl** to ignore any new structure names not already in the repository. The output script remains unchanged for these structures, even if it refers to these new names.

Errors during conversion

If any errors occur during conversion, they are logged in the error log file (*script_filename.err*), and a message is displayed on the screen indicating the number of errors found.

The same message is also written to the end of the Synergy Data Language output file (*script_filename.sdl*) and the script output file (*script_filename.new*), to prevent compiling of these files from occurring without modification. We recommend that you review scripts before converting and review the output Synergy Data Language and script files before using them.

Updating the repository

Once you've generated the Synergy Data Language file (*filename.sdl*), you'll use it to update an existing repository. See [“Loading a Repository Schema”](#) in the “Utility Functions” chapter of the *Repository User's Guide* for instructions.

When the Load Repository Schema utility prompts you to merge the schema, select **Yes**. You then will be prompted for how to handle new and existing definitions found in the Synergy Data Language file. Select **Yes** at the “Add new?” prompt, and select **Overlay** at the “Change existing?” prompt.

3

Proto

Using Proto 3-2

Gives a description and overview of the Proto program. Covers the basic rules of using Proto, describes the General and Main menus, and discusses each entry associated with those menus. Also briefly covers the ways you can customize the Proto program.

Using Proto

Proto is a menu-driven, prototyping tool that does the following:

- ▶ Builds windows and columns from a script file
- ▶ Displays the windows and columns on the screen
- ▶ Places the windows and columns in a window library.

Proto provides a quick and easy way to view and test UI Toolkit screen functions. It enables you to load and place menu columns, input windows, and other windows on your screen so you can see what they look like.



On Windows, we recommend you use Composer instead.

Start Proto as follows:

On...	Do this...
UNIX	At a command prompt, enter <code>dbr WND:proto</code>
OpenVMS	At a command prompt, enter <code>run WND:proto</code>

Proto was written using Toolkit; thus it follows the same “rules” as any other Toolkit application:

- ▶ If the menu bar is disabled, press CTRL+P to enable it.
- ▶ An entry in a menu or list is always highlighted. To select an item, use the arrow keys to move the highlighted area until the item you want is highlighted and then press ENTER.
- ▶ You can also select a menu entry using the quick-select character (when the menu bar is active) or shortcut for that entry. The shortcuts (which in this case are all function keys) are listed on the information line at the bottom of your screen.

After you start Proto, you’ll see the menu bar, which contains two headings: General and Main.

The General menu

The General menu contains the following entries:

- Help
- Exit
- About
- View

Directory
Renditions
Key mapping
App keypad

The View, Directory, Renditions, Key mapping, and App keypad entries are hot-entries, meaning they are available at all times while you are running Proto.

Getting help

The Help entry displays information about the current screen. Use the Help shortcut if you are not sure what a menu entry means.

Exiting the current entry

Selecting Exit takes you out of the current menu entry and returns you to the previous entry. You can use the Exit shortcut to stop doing almost anything. If you are at the Main menu, Exit will exit Proto altogether.

Viewing a window without obstruction

The View command is only enabled when you are loading windows. View removes all prompts and selection windows to enable you to view any windows you have placed. (Note that View is meaningless when you are loading columns, because the columns are not placed on the screen and you therefore will not be able to see them.)

View is a toggle; to replace the prompt and selection window so you can continue loading windows, press the View command key again. You can also press ENTER to continue.

Getting a directory listing

When you press the hot-entry key for the Directory, a text window that lists the current contents of the window library is placed on the screen. A Scroll commands menu is added to enable you to page through the directory. The left and right arrow keys are the shortcuts for page down and page up, respectively.

If no window library is currently open, this entry is disabled and your terminal will beep if you try to access it by pressing its shortcut.

Renditions

Select Renditions from the General menu if you want to define your own rendition schemes for the various screen areas. You can edit the display renditions for such items as menu entries, input fields, and windows. (You can add this same rendition editing capability to your own programs by calling UI Toolkit's U_EDITREND subroutine, which is what Proto does.) See [“Using the Renditions window \(Windows\)” on page 18-36](#) for information.

Key mapping

Key mapping is the internal processing that associates a key on a keyboard with a Toolkit menu entry. The key mapping utility enables you to add or modify the key mapping for one or more terminal types.

You can start the key mapping utility from Proto or the Synergy UI Toolkit Control Panel. You can also start it from your own programs by calling UI Toolkit's `U_EDITKEYS` subroutine (which is what Proto and Synergy UI Toolkit Control Panel do).

See [“Customizing Key Mapping for Menu Shortcuts” on page 18-11](#) for information.

Setting application keypad mode

Selecting App keypad toggles the keypad mode on and off and generates a message stating if keypad mode is set or reset.

If you have a VT100 keyboard, you can set your keypad to application mode using an escape sequence. If your menu column uses the keypad but your terminal is not set in application keypad mode, you can use the “App keypad” hot-entry to set it. (The “App keypad” menu entry has no effect on terminals that are not VT100-compatible.)

Proto does not automatically reset the application keypad mode when it exits.

The Main menu

The Main menu contains the following entries:

- Create library
- Process script
- View/Test
- Open library

The entries in this menu are highlighted in a logical order of choices you are likely to make.

The first selection highlighted is Create library. Since Proto uses a window library file for all user-defined windows and columns (those you are prototyping), you may need to create a library into which Proto can load columns and windows when a script file is compiled. After the library is created and you are once again at the Main menu, the Process script entry is highlighted, followed by the View/Test entry.

If you are testing windows in an existing window library, you can select the “Open library” entry first. When Proto returns to the Main menu, the View/Test entry will be highlighted.

The information line contains the name of the current window library.

Creating a library

When you select Create library from the menu, an input window prompts you for the name of the library to create. Type the name of a library file. The default extension for a library file is **.ism**.

If Proto cannot create the library because a library of that name already exists, it displays an error message and waits for valid input.

Processing a script file

The “Process script” option is available only when a window library is open. When you select “Process script,” an input window prompts you for the name of the script file you want to compile.

Type the name of the script file (the default extension is **.wsc**). If Proto cannot find the script file, it displays an error message and waits for valid input. If Proto is unable to save because of locked window library records, it retries up to 10 times with two-second delays between retries.

Proto can process physical lines containing up to 255 characters.

As each window is processed, it is placed in the current window library. When processing is complete, Proto displays a directory that lists all windows or columns in the script file that was just compiled. Select “Exit” to leave this directory.

Viewing and testing your windows

When you select View/Test, the Main menu is replaced with a Functions menu that has options for loading, processing, and removing columns and windows.

Loading a column

When you select Load column from the menu, Proto displays a selection window that contains a list of the columns that can be loaded. A Selection functions menu is added to the menu bar. Use the navigation options on this menu (Next selection, Prior selection, etc.) to scroll through the selection windows.

When you select a column, Proto loads that column in memory but does not place it on the screen. You will get a message telling you the column was loaded; press ENTER to continue.

You can load up to seven columns at one time. If you want to manually enter the name of a column, press the Manual entry shortcut (CTRL+E). The list of columns is removed from the screen, and you can type the name of the column you want to load.

When you are finished loading columns, press the Exit shortcut. To place your column, select Exercise columns from the menu. See [“Exercising \(placing\) columns” on page 3-7](#).

Loading an input, selection, or general window

To load an input, selection, or general window, select Load input window, Load selection window, or Load window from the menu.

Proto displays a selection window that contains a list of the available windows. The Selection functions menu is added to the menu bar.

When you select a window, Proto loads that window and places it on the screen. You can place up to 10 windows at one time. (No more than five input windows and five selection windows are allowed.)

If you want to manually enter the name of a window, press the Manual entry shortcut (CTRL+E). The list of windows is removed from the screen, and you can type the name of the window you want to load.

The View shortcut is enabled while you are loading windows. View removes the window selection list from the screen so you can view your placed windows without obstruction. Press ENTER or the View shortcut again to continue.

When you are finished loading and placing windows, press the Exit shortcut.

Processing an input window

To process an input window, select Process input window from the menu. Proto displays a list of placed input windows on the screen.

When you select the input window you want to process, input processing begins. You are responsible for having previously loaded any columns required by the input window. When the input routine reaches a break field, it displays a message that tells you which break field it has reached.

Remember, you can only submit input to visible fields. If you try to process an occluded field, a fatal Toolkit error occurs. Proto always displays the input termination message when testing input windows. Proto always uses the default input set.

Processing a selection window

To process a selection window, select Process a selection window from the menu. Proto displays a list of placed selection windows on the screen.

When you choose the selection window you want to process, selection processing begins. You are responsible for having previously loaded any columns required by the selection window. When you select an item by pressing ENTER, Proto displays a message that tells you which selection item was chosen.

Exercising (placing) columns

When you load a menu column, it is not automatically placed on the screen. You must select Exercise columns to actually place all loaded columns on the menu bar.

The menu bar is enabled after selecting Exercise columns, and the first column that you loaded is dropped down by default. You can move through the menu column entries using quick-select characters and the up and down arrow keys, and drop down other columns using the left and right arrow keys. If you select a menu entry that Proto does not recognize, Proto generates a message that tells you which entry was selected.

The General menu is still present on the menu bar. To end the column exerciser, press the Exit shortcut.

Removing a column or window

You can either remove one column or window at a time, or you can remove all columns or all windows.

When you select “Remove column” or “Remove window” from the menu, Proto displays a selection window that contains a list of the placed columns or windows. Select the item you want to remove, and Proto removes it from the screen. Press the “Exit” shortcut when you are finished removing columns or windows.

When you select “Remove all columns” or “Remove all windows” from the menu, Proto removes all columns or windows from the screen and deletes them from memory. The columns and windows are not deleted from the window library.

Opening an existing library

When you select “Open library” from the menu, an input window prompts you for the name of the library you want to open. Type the name of the library. If Proto can’t find the library file, it displays an error message and waits for valid input. If the library filename is valid, the specified library is opened and Proto returns to the Main menu.

Once a library is opened, the “Directory” shortcut is enabled.

Customizing Proto

Your UI Toolkit distribution includes the script file (**proto.wsc**) for the menus and windows in the Proto program. If you need to translate the menu items or the help windows into another language, edit this script file and rebuild the windows. The menus and help windows are saved in a window library named **proto.ism**.

You can modify the text messages displayed in Proto by using the Synergy UI Toolkit Control Panel. These messages are stored in **syntxt.ism** under the facility PRO. See [“The Synergy UI Toolkit Control Panel”](#) in the “General Utilities” chapter of *Synergy Tools*.

4

ActiveX Routines

UI Toolkit ActiveX routines are not available in Synergy .NET.

Options for ActiveX 4-3

Lists and discusses methods you can use to add ActiveX controls to your Toolkit application.

Routines for ActiveX Controls in Toolkit Containers 4-11

Discusses and provides the syntax for routines that enable you to use ActiveX controls with Toolkit container windows.

AX_INPUT – Perform input to an ActiveX control.....	4-12
AX_REGISTER – Register an ActiveX control	4-15
AX_TIMEOUT – Set the time-out value for an ActiveX control	4-16
%AX_TKCALL – Call a method for an ActiveX control.....	4-17
%AX_TKCID – Get the ID for an ActiveX container	4-20
%AX_TKCTLID – Get an ActiveX control ID	4-21
AX_TKGET – Get a property for an ActiveX control.....	4-22
%AX_TKGETINT – Get an ActiveX control property as an integer	4-24
AX_TKSET – Set a property for an ActiveX control	4-26
%AX_TKSINGLE – Create a Toolkit container and load a single ActiveX control	4-28
%AX_TKWIN – Create a Toolkit container window	4-31
AX_WANTSKEY – Register keys to an ActiveX control.....	4-33

Routine for the Spread Control 4-36

Discusses and provides the syntax for a routine that makes it easier to create a container for and load the Spread™ control.

%GRID_CREATE – Create a Toolkit container for the Spread control.....	4-36

Supplied Methods 4-39

Discusses %UWNDEVENTS_METHOD methods that are loaded by %AX_TKSINGLE and %GRID_CREATE. These methods automatically call developer-supplied methods if you pass the *prefix* argument to %AX_TKSINGLE or %GRID_CREATE.

%AX_TKEVENT_CLOSE – Call extension for close event	4-40
%AX_TKEVENT_LEFT_CLICK – Call extension for left click event	4-41
%AX_TKEVENT_LEFT_DBLCLK – Call extension for left double-click event	4-43
%AX_TKEVENT_MAXIMIZE – Call extension for maximize event.....	4-45
%AX_TKEVENT_MIDDLE_CLICK – Call extension for middle click event	4-46
%AX_TKEVENT_MIDDLE_DBLCLK – Call extension for middle double-click event	4-48
%AX_TKEVENT_MINIMIZE – Call extension routine for minimize event	4-50
%AX_TKEVENT_MOVE – Call extension routine for move event.....	4-51
%AX_TKEVENT_RESTORE – Call extension routine for restore event.....	4-53
%AX_TKEVENT_RIGHT_CLICK – Call extension routine for right click event.....	4-54
%AX_TKEVENT_RIGHT_DBLCLK – Call extension for right double-click event	4-56
%AX_TKEVENT_SCROLL – Call extension routine for scroll event	4-58
%AX_TKEVENT_SIZE – Size the Toolkit container window	4-60

Options for ActiveX

UI Toolkit enables you to add ActiveX controls to Toolkit windows, including composite windows. By adding ActiveX components to your application, you can extend its capabilities with a wide variety of interactive and multimedia effects. Toolkit includes a routine (`%GRID_CREATE`) that is specially designed for the FarPoint™ Spread control, but the other ActiveX routines can be used for any ActiveX control. Toolkit automatically uses an ActiveX control (based on the Spread control) when you choose to include ActiveX Toolkit lists in your application.

Requirements

Before an application can use an ActiveX control, the control must be registered with the Windows operating system. You can register an ActiveX control manually, or you can use `AX_REGISTER`, `%AX_TKSINGLE`, or `%GRID_CREATE` to register it programmatically. See [“Registering an ActiveX control” on page 4-9](#) for details.

32-bit ActiveX controls can be used only with the 32-bit Runtime, and 64-bit ActiveX controls can be used only with the 64-bit Runtime.

If you use any of the routines in the Synergy ActiveX API, you must include the file **DBLDIR:activex.def** and compile your program using the **-T** and **-X** compiler options. See the [“Synergy ActiveX API”](#) chapter of the *Synergy DBL Language Reference Manual* for details. (You do not need to link against **axlib.elb**, though, because Toolkit automatically links to this library when you use the `AX_` routines.)

Methods for adding ActiveX controls

You can use the following methods to add ActiveX controls to your Toolkit application:

- ▶ *Use the ActiveX API.* The ActiveX API is a group of routines included in Synergy DBL that enable you to use an ActiveX control in a Synergy application. You don’t need UI Toolkit to use the ActiveX API, but some of these routines are useful if you do use Toolkit. For information, see the [“Synergy ActiveX API”](#) chapter of the *Synergy DBL Language Reference Manual*.
- ▶ *Use a single control with a Toolkit container.* The most common way to handle ActiveX controls is to use a single ActiveX control with a Toolkit container. UI Toolkit includes routines that simplify this scenario. For information, see [“Using a single control with a Toolkit container” on page 4-4](#).
- ▶ *Use multiple controls with a single Toolkit container.* UI Toolkit enables you to use multiple controls with a single container. For information, see [“Using multiple controls with a single container” on page 4-5](#).

- ▶ *Add the Spread control.* UI Toolkit includes a routine specifically designed to make it easy to add the Spread control, a spreadsheet encapsulated in an ActiveX control. For information, see [“Creating a container for the Spread control” on page 4-6.](#)
- ▶ *Add an ActiveX Toolkit list.* UI Toolkit includes an ActiveX implementation of the Toolkit list. This implementation enables Toolkit list routines to control an ActiveX list. For information, see [“Using an ActiveX Toolkit list” on page 4-6.](#)

Containers for ActiveX controls

To use an ActiveX control with a UI Toolkit application, you’ll need two types of containers for the control: a *Toolkit container window* and an *ActiveX container*. Toolkit includes routines that create these containers, load ActiveX controls, and retrieve the IDs for ActiveX containers.

The Toolkit container window is a Toolkit window that you can manipulate as you would any other Toolkit window (environment control, place/remove, and so forth). The ActiveX container is placed within the Toolkit window and is exactly the same size as the Toolkit window. This container is “slaved” to the Toolkit container window, so you won’t manipulate it directly. The `%AX_TKCID` enables you to get the ActiveX container ID, and you can also get this ID by passing the optional `ax_container_id` argument to `%AX_TKWIN`. You will need this ID if you want to bind to the ActiveX container’s events or if you want to manipulate the ActiveX container’s properties. (See [“Extended properties,” “Ambient properties,”](#) and [“Local properties”](#) in the “Synergy ActiveX API” chapter of the *Synergy DBL Language Reference Manual*.)

Using a single control with a Toolkit container

The most common way to use ActiveX controls with a Toolkit application is to use a single ActiveX control with a single Toolkit container window (and a single ActiveX container). To do this, use the following routines, which are designed for this scenario and are easier to use than ActiveX API counterparts. These routines are documented in [“Routines for ActiveX Controls in Toolkit Containers” on page 4-11](#) unless otherwise noted.

- ▶ To create a Toolkit container and an ActiveX container and load the ActiveX control, use `%AX_TKSINGLE`. This routine returns an ID for the Toolkit container window, registers a method set for the window, and enables you to extend the method set with your own routines.
- ▶ To retrieve the ActiveX control ID, pass the Toolkit container ID to `%AX_TKCTLID`.
- ▶ To retrieve the ActiveX container ID, pass the Toolkit container ID to `%AX_TKCID`.
- ▶ To trigger an event for the Toolkit container window, use the `D_SIGNAL` subfunction of `%U_WNDEVENTS`. (See [%U_WNDEVENTS on page 16-203](#) for information.)
- ▶ To use the Toolkit container window ID to access the properties and methods for the ActiveX control, use `%AX_TKCALL`, `AX_TKGET`, `%AX_TKGETINT`, or `AX_TKSET`.

- ▶ To use the ActiveX container ID to access the properties and methods for the ActiveX container, use `%AX_CALL`, `AX_GET`, `%AX_GETINT`, or `AX_SET`. (For information on these routines, see the “[Synergy ActiveX API](#)” chapter of the *Synergy DBL Language Reference Manual*.)
- ▶ To control how keystrokes are passed to the ActiveX control, use `AX_WANTSKEY`.
- ▶ To wait for user input for the ActiveX control, use `AX_INPUT`.

You can also use the method documented in “[Using multiple controls with a single container](#)” below, and load only one control into the container. However, we recommend using `%AX_TKSINGLE` when working with a single control because it minimizes the number of separate IDs that need to be maintained.

Using multiple controls with a single container

You can use multiple controls with a single Toolkit container window (and a single ActiveX container). Note, however, that with this method, you can’t use the Toolkit container window ID to access properties and methods for an ActiveX control (i.e., you can’t use `%AX_TKCALL`, `AX_TKGET`, `%AX_TKGETINT`, or `AX_TKSET`). Unless otherwise noted, the following routines are documented in “[Routines for ActiveX Controls in Toolkit Containers](#)” on page 4-11

To use multiple controls with a single container, use the following routines.

- ▶ To create a Toolkit container and an ActiveX container, use `%AX_TKWIN`. This routine returns an ID for the Toolkit container window.
- ▶ To load the ActiveX controls, use `%AX_LOAD`. The ActiveX control IDs are the return value of this function. (For information on this function, see [%AX_LOAD](#) in the “[Synergy ActiveX API](#)” chapter of the *Synergy DBL Language Reference Manual*.)
- ▶ To retrieve the ActiveX container ID, pass the Toolkit container ID to `%AX_TKCID` or pass the optional `ax_container_id` argument to `%AX_TKWIN`.
- ▶ To trigger an event for the Toolkit container window, use the `D_SIGNAL` subfunction of `%U_WNDEVENTS`. (See [%U_WNDEVENTS](#) on page 16-203 for information.)
- ▶ To use the ActiveX control ID or the ActiveX container ID to access the properties and methods for the ActiveX control, use `%AX_CALL`, `AX_GET`, `%AX_GETINT`, or `AX_SET`. (For information on these routines, see the “[Synergy ActiveX API](#)” chapter of the *Synergy DBL Language Reference Manual*.)
- ▶ To control how keystrokes are passed to an ActiveX control, use `AX_WANTSKEY`.
- ▶ To wait for user input for an ActiveX control, use `AX_INPUT`.

Creating a container for the Spread control

Professional Series includes the FarPoint Spread control, which is a full-featured spreadsheet encapsulated in an ActiveX control. To use the Spread control, you'll treat it as a single control in a Toolkit window as discussed in [“Using a single control with a Toolkit container” on page 4-4](#). However, instead of using %AX_TKSINGLE to create the containers and load the control, you'll use %GRID_CREATE. (Unless otherwise noted, the following routines are documented in [“Routines for ActiveX Controls in Toolkit Containers” on page 4-11](#).)

- ▶ To create a Toolkit container and an ActiveX container and load the Spread control, use %GRID_CREATE. This function returns an ID for the Toolkit container window, registers a method set for the window, and enables you to extend the method set with your own routines. For information, see [%GRID_CREATE on page 4-36](#).
- ▶ To retrieve the ActiveX control ID, pass the Toolkit container ID to %AX_TKCTLID.
- ▶ To retrieve the ActiveX container ID, pass the Toolkit container ID to %AX_TKCID.
- ▶ To trigger an event for the Toolkit container window, use the D_SIGNAL subfunction of %U_WNDEVENTS. For information, see [%U_WNDEVENTS on page 16-203](#).
- ▶ To use the Toolkit container window ID to access the properties and methods for the ActiveX control, use %AX_TKCALL, AX_TKGET, %AX_TKGETINT, or AX_TKSET.
- ▶ To use the ActiveX control ID to access the properties and methods for the ActiveX control, use %AX_CALL, AX_GET, %AX_GETINT, or AX_SET. (For information on these routines, see the [“Synergy ActiveX API”](#) chapter of the *Synergy DBL Language Reference Manual*.)
- ▶ To control how keystrokes are passed to the ActiveX control, use AX_WANTSKEY.
- ▶ To wait for user input for the ActiveX control, use %AX_INPUT.

For information on obtaining the Spread control, visit www.componentone.com. For definitions of the methods, properties, and events of the Spread control, see the Spread documentation.

Using an ActiveX Toolkit list

UI Toolkit includes an ActiveX implementation of the Toolkit list. This implementation wraps some of the functionality of the Spread control, providing an ActiveX list that you can control with Toolkit list routines. Your code creates an ActiveX Toolkit list on Windows platforms and a standard Toolkit list on UNIX and OpenVMS. For information, see [“Defining Lists” on page 9-5](#).

In addition to the Toolkit list routines, you can also access published properties and methods for the ActiveX list. For information, see [“Using Lists in Toolkit Applications” on page 9-4](#).

The basic process

To access an ActiveX control from your application, you'll need to do the following:

1. Register the file that contains the ActiveX control. See [“Registering an ActiveX control” on page 4-9](#).
2. Write a routine that loads and processes the ActiveX control and call the routine from your program. For example, you might write a hyperlink method, a drill method, or a subroutine to process a menu entry, depending on how the user will access the ActiveX control.
3. Modify your user interface to allow access to the ActiveX control. For example, you might add a hyperlink prompt, drilldown button, or menu entry.

You could combine steps 2 and 3 and make your hyperlink or drill method (for example) the routine that loads and processes the ActiveX control.

Let's say that we want users to be able to display a calendar to enter a date in a **Renewal date** field. We'll use an ActiveX calendar control called **AXCalendar**, and the user will click on a drilldown button to display it, as shown below.

To process the control, we write the routine **ax_date_drillmethod**, which includes a drill method that calls the ActiveX routine.

```
.subroutine ax_date_drillmethod

.include "DBLDIR:ActiveX.def"
.include WND:tools.def

record
    tkwid      ,i4          ;Toolkit window ID
    axwid      ,i4          ;ActiveX container ID
    axid       ,i4          ;ActiveX control ID
    method_id ,i4          ;Window event ID

group a_date ,a
    day        ,d2
    month      ,d2
    year       ,d4
endgroup
```

```
.proc
    tkwid = %ax_tkwin("tkdate",12,40, axwid)      ;Create toolkit container
    axid = %ax_load(axwid, "MSCAL.Calendar.7")    ;Load control
    xcall ax_bind(axid, "DbClick", "OnDbClick") ;Bind double-click event

;Create and assign toolkit container window close event
    method_id=%u_wndevents(D_CREATE,,D_EVENT_CLOSE,'OnClose')
    xcall u_wndevents(D_ASSIGN,method_id,tkwid)
    xcall u_window(D_PLACE, tkwid, 10, 20)        ;Place Toolkit window
    xcall ax_input(tkwid)                        ;Process control
    if (g_entnam.eq.'DONE')                      ;Check for menu entry
        begin
            a_date.day = %ax_getint(axid, "Day")
            a_date.month = %ax_getint(axid, "Month")
            a_date.year = %ax_getint(axid, "Year")
            xcall i_force(a_date)
        end
    xcall u_window(D_DELETE, tkwid)                ;Delete Toolkit container
    clear g_select
    xreturn
.end

.subroutine OnDbClick
.proc
    xcall m_signal('DONE')
    xreturn
.end

.subroutine OnClose
.proc
    xcall m_signal('CLOSE')
    xreturn
.end
```

Finally, we need to modify the script file entry for the **Renewal date** field to add the drilldown button. The entry will look something like this:

```
.field c_rd, d8, date, pos(3, 38), prompt("Renewal date"), fpos(3, 52), -
    bzro, help("h_rd"), info("DDMMYY"), drill_method("ax_date_drillmethod")
```

Error handling for ActiveX control exceptions

You can trap exceptions generated by ActiveX controls as you load them or use them in your Toolkit application. We use the EXITE mechanism to generate errors from these exceptions. These errors can be trapped by ONERROR statements if they are in effect when the ActiveX subroutine that loads or uses the control is called. When an error is trapped, control is transferred to the code for the error if the error is in the error list for the ONERROR statement (or there's a catch-all label).

If the error isn't trapped by the ONERROR statement, or if no ONERROR statement is in effect, any exception generated by the ActiveX control will cause a fatal runtime error at the line where the ActiveX subroutine was called.

- ▶ See the “[Error Messages](#)” chapter of *Synergy Tools* for a list of possible errors.
- ▶ See [ONERROR](#) in the “Synergy DBL Statements” chapter of the *Synergy DBL Language Reference Manual* for information on the ONERROR statement.
- ▶ See [EXITE](#) in the “System-Supplied Subroutines and Functions” chapter of the *Synergy DBL Language Reference Manual* for information on trapping errors generated by EXITE.

Registering an ActiveX control

Before an application can use an ActiveX control, the control must be registered with the Windows operating system. Once the file is registered, it should not be moved or renamed. You can register a control manually or programmatically:

- ▶ Use the **regsvr32.exe** utility that is distributed with Windows. On 64-bit Windows, use the version of **regsvr32** that corresponds to the bitness of the control. That is, to register a 64-bit control, use the **regsvr32** in %windir%\system32, and to register a 32-bit control, use the **regsvr32** in %windir%\syswow64.
`regsvr32 filename`
- ▶ Use the Add Control button in the ActiveX Diagnostic Utility (**axutil.exe**) that is included in your distribution. See [The ActiveX Diagnostic Utility](#)” in the “General Utilities” chapter of *Synergy Tools* for more information.
- ▶ Register the control programmatically on first use using AX_REGISTER, %AX_TKSINGLE, or %GRID_CREATE.



You can also use the Synergy DLL API to register ActiveX controls. Use the DLL API to open the **.ocx** file as a DLL, and then call the DllRegisterServer routine within it. For example,

```
hndl = %dll_open("OCXDIR:mycontrol.ocx")
xcall dll_call(hndl, "DllRegisterServer")
xcall dll_close(hndl)
```

To avoid doing this every time the control is loaded, trap an error on the load and register the control only if the load fails. This method is particularly useful for shared server installations where controls may be added to the existing application without performing an installation script on each workstation. See “[Synergy DLL API](#)” in the *Synergy DBL Language Reference Manual* for more information.

We recommend you always check the loading of your ActiveX control with the ActiveX Diagnostic Utility before accessing the control from your application.

Distributing applications that use ActiveX controls

When you distribute your application, make sure you distribute the following files:

- ▶ The file that contains your ActiveX controls (usually **.ocx**)
- ▶ Your main program (**.dbr**)
- ▶ Any files on which your control depends. For example, for controls created with Visual Basic 6, you should redistribute **MSVBVM60.DLL** (the VB6 runtime) to ensure the target system has it.



If you redistribute shared DLLs (such as **MSVBVM60.DLL**), you must ensure that your installation either puts the file only in your software's private directory or only replaces older versions of the same file.

The **.ocx** file (or whatever you named it) must be registered before it can be used. If you've written an installation script, make sure it registers the file. If not, you can do it manually as described in [“Registering an ActiveX control” on page 4-9](#).

Routines for ActiveX Controls in Toolkit Containers

In addition to the ActiveX API routines (documented in the “[Synergy ActiveX API](#)” chapter of the *Synergy DBL Language Reference Manual*), you can use the routines below to load and use ActiveX controls with Toolkit containers. See “[Options for ActiveX](#)” on page 4-3 for information about the different ways to create Toolkit containers and load ActiveX control.



You do not need to link against **axlib.elb** to use these routines. UI Toolkit automatically links to this library when you use these AX_ routines.

AX_INPUT – Perform input to an ActiveX control

AX_REGISTER – Register an ActiveX control

AX_TIMEOUT – Set the time-out value for an ActiveX control

%AX_TKCALL – Call a method for an ActiveX control

%AX_TKCID – Get the ID for an ActiveX container

%AX_TKCTLID – Get an ActiveX control ID

AX_TKGET – Get a property for an ActiveX control

%AX_TKGETINT – Get an ActiveX control property as an integer

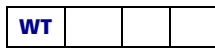
AX_TKSET – Set a property for an ActiveX control

%AX_TKSINGLE – Create a Toolkit container and load a single ActiveX control

%AX_TKWIN – Create a Toolkit container window

AX_WANTSKEY – Register keys to an ActiveX control

AX_INPUT – Perform input to an ActiveX control



```
xcall AX_INPUT (tk_container_id, [help_id], [wait_time][, menu_flag])
```

Arguments

tk_container_id

The ID of the Toolkit ActiveX container window. **(n)**

help_id

(optional) A help identifier that will be passed if the menu entry “O_HELP” is signaled or selected. **(a)**

wait_time

(optional) The time-out limit for I/O processing to be completed before returning to the calling routine. This setting overrides any AX_TIMEOUT setting. **(n)**

D_FOREVER or **-1**

Never time out.

D_GLOBAL or **-2**

Use the global value (see “[g_wait_time](#)” on page 18-6).
(default)

D_IMMEDIATE or **0**

Time out immediately.

n

Wait up to *n* seconds (where *n* is a positive value) for the user to press a key.

menu_flag

(optional) If passed and true, a flag that causes AX_INPUT to return utility menu entries (U_ and O_HELP) instead of processing them. **(n)**

Discussion

The AX_INPUT routine should be used instead of UI Toolkit’s I_INPUT routine whenever you want to do input to an ActiveX control while Toolkit is running. AX_INPUT enables you to focus the component and allow the user to perform input on it, while at the same time allowing menu entries to be selected and returned from the Toolkit. (On UNIX, this routine calls M_PROCESS. On OpenVMS, it is ignored.)

The specified Toolkit ActiveX container window is activated and used to determine which keys get passed through to ActiveX controls. Before calling AX_INPUT, the client routine should establish focus on the correct control by setting the “Ext_Focus” property of the desired control.

AX_INPUT recognizes the following reserved menu entries, which are documented in [“Appendix B: Reserved Menu Entries”](#):

```
E_CLEAR
E_COPY
E_CUT
E_MARK
E_PASTE
```

Unless *menu_flag* is passed as true, the help method for the current environment will be invoked if the “O_HELP” menu entry is signaled or selected, and if a “U_” menu entry is signaled or selected, EUTILS_METHOD for the current environment will be invoked. If *menu_flag* is passed as true, AX_INPUT returns these menu entries instead of processing them.

Note that when an ActiveX control is part of a composite window, TAB and SHIFT+TAB may not move focus to the next or previous child if the ActiveX control consumes these as accelerators. If this is the case, you can work around this by signaling “C_NEXT” or “C_PREV”. For information, see [“Controlling tabbing, focus, and input context”](#) on page 6-4.

When AX_INPUT returns, **g_select** will be true and **g_entnam** will contain the selected menu entry. Note that an ActiveX event handler routine can use %M_SIGNAL to force a break from AX_INPUT.

If the operation times out, **g_select** is set to true and **g_entnam** is set to the entry name specified in **g_time_entry**, which is defined in **tkctl.def**. See [“g_time_entry”](#) on page 18-6.

See also

- ▶ [“Error handling for ActiveX control exceptions”](#) on page 4-8.
- ▶ The [“Synergy ActiveX API”](#) chapter of the *Synergy DBL Language Reference Manual*.

Examples

The following example uses this script file:

```
.script

.column c_general, "General"
.entry o_help, "Help", key(f1)
.entry o_exit, "Exit", key(esc), select(x)
.end
```

This example performs input to an ActiveX control.

```
main ToolkitActiveX          ;Toolkit ActiveX example

.include "DBLDIR:activex.def" ;ActiveX definitions
.include "WND:tools.def"      ;Toolkit definitions
.include "WND:inpctl.def"     ;Toolkit input controls
```

```

.align
stack record
    colid      ,i4                ;Column ID
    axwin      ,i4                ;ActiveX Toolkit window ID
    axcntr     ,i4                ;ActiveX Container ID
    axctrl     ,i4                ;ActiveX Control ID

proc
    xcall u_start("ToolkitActiveX")
                                ;Start Toolkit and open our window lib
    xcall m_ldcol(colid,, "c_general") ;Load our general column
    axwin = %ax_tkwin("", 10, 40, axcntr)
                                ;Create Toolkit ActiveX window and
                                ; container
    axctrl = %ax_load(axcntr, "MSMask.MaskedTextBox.1", 15, 15, 100, 20)
                                ;Create a masked edit control at 15,15
                                ; (pixels) within the container, sized
                                ; at 100x20 (pixels)
    xcall u_window(D_PLACE, axwin, 5, 21) ;Place the window
    xcall ax_wantskey(axwin, D_KEY_ALL, D_MAP_CONTROL, D_KEY_MENU,
&                                D_MAP_MENU)
                                ;Tell the window that we want any key
                                ; that is not a menu shortcut to be
                                ; passed to the ActiveX controls
    xcall ax_set(axctrl, "Ext_Focus", TRUE)
                                ;Focus the masked edit control
                                ;Here's our input loop
    repeat
        begin
            xcall ax_input(axwin) ;Allow input to anything available
            if (g_select)          ;Menu entry or shortcut?
                begin
                    using g_entnam select ;Which one?
                        ("O_EXIT "), exitloop
                    endusing
                end
            end
            xcall u_finish          ;Close Toolkit
        endmain

```

AX_REGISTER – Register an ActiveX control



```
xcall AX_REGISTER(ocx_file)
```

Arguments

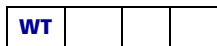
ocx_file

The name of the **.ocx** file for an ActiveX control. (**a**)

Discussion

The AX_REGISTER routine attempts to register the specified **.ocx** file as an ActiveX control. If it is unable to find or register the control, it resignals the error trapped for the load failure.

AX_TIMEOUT – Set the time-out value for an ActiveX control



```
xcall AX_TIMEOUT(wait_time)
```

Arguments

wait_time

The input time-out (or “wait”) in seconds for I/O processing to be completed before returning to the calling routine. (**n**)

D_IMMEDIATE or **0** Time out immediately.

D_FOREVER or **-1** Never time out.

D_GLOBAL or **-2** Use the global value. (default)

#seconds Wait up to *#seconds* for I/O processing to be complete.

Discussion

The AX_TIMEOUT routine changes the time-out setting for AX_INPUT. A call to this routine overrides the **g_wait_time** setting (the default time-out setting), but will be overridden by a time-out value set with AX_INPUT.

Note that Toolkit does not time out while an application is suspended. For example, Toolkit does not time out while a menu is dropped down or a message window is open on Windows.

See also

- ▶ [g_wait_time](#) on page 18-6.
- ▶ [AX_INPUT](#) on page 4-12.
- ▶ **WAIT** in the “Synergy DBL Statements” chapter of the *Synergy DBL Language Reference Manual*.

%AX_TKCALL – Call a method for an ActiveX control

WT			

value = %AX_TKCALL (*tk_container_id*, *method_name*[, *arguments*, ...])

or

xcall AX_TKCALL (*tk_container_id*, *method_name*[, *arguments*, ...])

Return value

value

The value returned by the method on Windows in traditional Synergy. If the method returns a value that isn't meaningful when expressed as an integer, *value* is returned as zero, and the value returned by the method is ignored. On UNIX, a call to this routine will cause an error. On OpenVMS, this routine always returns true. (^VAL)

Arguments

tk_container_id

The window ID for the ActiveX control's Toolkit container. (n)

method_name

The name of the method to call. (a)

arguments

(optional) Up to nine arguments to be passed to the method. (See the [Discussion](#) for information on passing more than nine.)

Discussion

This routine enables you to access an ActiveX control's methods by passing the Toolkit container ID. (If you use %AX_CALL, you must pass the control ID.) Note the following:

- ▶ You can use %AX_TKCALL only if you used %AX_TKSINGLE or %GRID_CREATE to create the container and load the control.
- ▶ If you used %AX_TKWIN to create the container, you're limited to %AX_CALL. See [%AX_CALL](#) in the "Synergy ActiveX API" chapter of the *Synergy DBL Language Reference Manual*.

Because %AX_TKCALL is a macro, you can pass more than nine arguments by grouping arguments together using angle brackets (< >). If you enclose all optional arguments in a single set of angle brackets, the macro expansion will treat the enclosed text as a single unit.

%AX_TKCALL will attempt to convert all optional arguments into the type of data that the method expects. You must ensure that the type of data passed has meaning within the context of the targeted conversion. Failure to do so will yield undefined results. For more information, see [“ActiveX data types”](#) in the “Synergy ActiveX API” chapter of the *Synergy DBL Language Reference Manual*.



If you make multiple method calls to the same control, use %AX_TKCTLID to get the ActiveX control ID, and then use AX_CALL to call the methods. %AX_TKCALL is convenient (because you don't need the control ID), but it calls AX_CALL, so it's not as efficient.

See also

- ▶ [“Error handling for ActiveX control exceptions”](#) on page 4-8.
- ▶ The [“Synergy ActiveX API”](#) chapter of the *Synergy DBL Language Reference Manual*.

Examples

The following example uses %AX_TKCALL to display the About window for the control when the O_ABOUT menu entry is selected. It also displays two messages when the O_DBLCLK menu entry is signaled.

```
proc
  xcall u_start

  xcall mb_column(cntrl, "Col", "General")
  xcall mb_entry(cntrl, "o_about", "About")
  xcall mb_line(cntrl)
  xcall mb_entry(cntrl, "o_exit", "Exit", F4_KEY)
  xcall mb_end(cntrl, col_id)

  wndid = %grid_create(10, 40, D_SPREAD_LICENSE, "tgrid_")
  xcall u_window(D_PLACE, wndid, 10, 21)
  xcall ax_tkset(wndid, "VirtualMode", TRUE)
  xcall ax_tkset(wndid, "VirtualRows", 100)
  xcall ax_tkset(wndid, "MaxCols", 5)
  repeat
    begin
      xcall ax_input(wndid)
      if (g_select)
        begin
          using g_entnam select
("O_DBLCLK "),      xcall u_message("Double-clicked row " +
&                  %string(%ax_tkgetint(wndid, "ActiveRow"))+
&                  ", column " +
&                  %string(%ax_tkgetint(wndid, "ActiveCol"))+
&                  "\n\n    using %ax_tkgetint call")
```



```
("O_ABOUT "),      xcall ax_tkcall(wndid, "AboutBox")
("O_EXIT "),       exitloop
(),                exitloop
    endusing
    end
end
```

%AX_TKCID – Get the ID for an ActiveX container

WT			

container_id = %AX_TKCID(*tk_container_id*)

Return value

container_id

On Windows, %AX_TKCID returns the associated ActiveX container ID. On UNIX, this routine always returns false. On OpenVMS, this routine always returns true. (^VAL)

Arguments

tk_container_id

The ID of the Toolkit container window. (n)

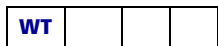
Discussion

Each UI Toolkit container window has an ActiveX container associated with it. This function enables you to obtain the ID for the ActiveX container by passing the ID for the Toolkit container window. (%AX_TKWIN, %AX_TKSINGLE, and %GRID_CREATE return the ID for the Toolkit container window.)

See also

- ▶ [“Error handling for ActiveX control exceptions” on page 4-8.](#)
- ▶ The [“Synergy ActiveX API”](#) chapter of the *Synergy DBL Language Reference Manual*.

%AX_TKCTLID – Get an ActiveX control ID



```
control_id = %AX_TKCTLID(tk_container_id)
```

or

```
xcall AX_TKCTLID(tk_container_id)
```

Return value

control_id

The ID of the ActiveX control on Windows. On UNIX, a call to this routine will cause an error. On OpenVMS, this routine always returns true. (^**VAL**)

Arguments

tk_container_id

The window ID for the ActiveX control's Toolkit container. (**n**)

Discussion

This routine enables you to retrieve the ID for an ActiveX control associated with a given Toolkit container window. You can use this routine only if you used %AX_TKSINGLE or %GRID_CREATE to create the containers and load the control.

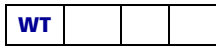
See also

- ▶ [“Error handling for ActiveX control exceptions” on page 4-8.](#)
- ▶ The [“Synergy ActiveX API”](#) chapter of the *Synergy DBL Language Reference Manual*.

Examples

See [Examples on page 4-38.](#)

AX_TKGET – Get a property for an ActiveX control



```
xcall AX_TKGET (tk_container_id, property, current_value[, parameters])
```

Arguments

tk_container_id

The window ID for the ActiveX control's Toolkit container. **(n)**

property

The name of the property to access. For information on supported properties, see “[Published Properties for ActiveX Toolkit Lists](#)” on page 9-17 and “[Properties](#)” in the “Synergy ActiveX API” chapter of the *Synergy DBL Language Reference Manual*. **(a)**

current_value

A field that will be loaded with the current value for the property. **(a)**

parameters

(optional) Up to eight additional parameters required by a property. **(a or n)**

Discussion

This routine enables you to retrieve an ActiveX control's properties. However, you can use AX_TKGET only if you used %AX_TKSINGLE or %GRID_CREATE to create the container and load the control. Note the following:

- ▶ If you used %AX_TKWIN to create the container, you must use %AX_GET. See [%AX_GET](#) in the “Synergy ActiveX API” chapter of the *Synergy DBL Language Reference Manual*.
- ▶ To retrieve a property for an ActiveX control as an integer, use %AX_TKGETINT. See [%AX_TKGETINT](#) on page 4-24.



Rather than making multiple calls to AX_TKGET to get properties for the same control, use %AX_TKCTLID to get the ActiveX control ID and then use AX_GET to get the properties. AX_TKGET is convenient (because you don't need the control ID), but it calls AX_GET, so it's not as efficient.

Make sure the data type of *current_value* has meaning within the context of the specified property. If it does not, the results are undefined.

The *parameters* argument can pass only integer values. It cannot be used to pass additional parameters for a property that expects alpha parameters.

It is the caller's responsibility to ensure that the data type of *current_value* has meaning within the context of the specified property. If it does not, the results are undefined.

Some controls support indexed properties, which are stored as arrays in a control and have one or more indices. If the property is an indexed property, *parameters* must be the index values of the array element to reference. The user documentation for a given control should specify when a property is indexed.

On UNIX, a call to this routine will cause an error. On OpenVMS, this routine is ignored.

See also

- ▶ [“Error handling for ActiveX control exceptions” on page 4-8.](#)
- ▶ The [“Synergy ActiveX API”](#) chapter of the *Synergy DBL Language Reference Manual*.

Examples

See [Examples on page 4-18.](#)

%AX_TKGETINT – Get an ActiveX control property as an integer

WT			

value = %AX_TKGETINT (*tk_container_id*, *property* [, *parameters*])

Return value

value

On Windows, %AX_TKGETINT returns the current value for the property. On UNIX a call to this routine will cause an error. On OpenVMS this routine always returns true. (^VAL)

Arguments

tk_container_id

The window ID for the ActiveX control's Toolkit container. (n)

property

The name of the property to access. For information on supported properties, see “[Published Properties for ActiveX Toolkit Lists](#)” on page 9-17 and “[Properties](#)” in the “Synergy ActiveX API” chapter of the *Synergy DBL Language Reference Manual*. (a)

parameters

(optional) Up to eight additional parameters required by a property. (n)

Discussion

This routine enables you to retrieve an integer value for an ActiveX control. You can use %AX_TKGETINT only if you used %AX_TKSINGLE or %GRID_CREATE to create the container and load the control. Note the following:

- ▶ If you used %AX_TKWIN to create the container, you're limited to %AX_GETINT. See [%AX_GETINT](#) in the “Synergy ActiveX API” chapter of the *Synergy DBL Language Reference Manual*.
- ▶ To retrieve an ActiveX control property with a non-integer value, use AX_TKGET. See [AX_TKGET](#) on page 4-22.

If *property* does not reference a property whose value has an integer representation, *value* will be undefined.

Some controls support indexed properties, which are stored as arrays in a control and have one or more indices. If the property is an indexed property, *parameters* must be the index values of the array element to reference. The user documentation for a given control should specify when a property is indexed.

The *parameters* argument can pass only integer parameters. It cannot be used to pass additional parameters for a property that expects alpha parameters.



Rather than making multiple calls to AX_TKGETINT to get properties for the same control, use %AX_TKCTLID to get the ActiveX control ID, and then use AX_GETINT to get the properties. AX_TKGETINT is convenient (because you don't need the control ID), but it calls AX_GETINT, so it's not as efficient.

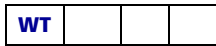
See also

- ▶ [“Error handling for ActiveX control exceptions” on page 4-8.](#)
- ▶ The [“Synergy ActiveX API”](#) chapter of the *Synergy DBL Language Reference Manual*.

Examples

See [Examples on page 4-30](#).

AX_TKSET – Set a property for an ActiveX control



```
xcall AX_TKSET (tk_container_id, property, new_value)
```

Arguments

tk_container_id

The window ID for the ActiveX control's Toolkit container. (**n**)

property

The name of the property to access. For supported properties, see “[Published Properties for ActiveX Toolkit Lists](#)” on page 9-17 and “[Properties](#)” in the “Synergy ActiveX API” chapter of the *Synergy DBL Language Reference Manual*. (**a**)

new_value

The new value for the property. (**a** or **n**)

Discussion

This routine enables you to set an ActiveX control's properties by passing the Toolkit container ID. You can use AX_TKSET only if you used %AX_TKSINGLE or %GRID_CREATE to create the container and load the control. (If you used %AX_TKWIN to create the container, you must use %AX_SET. See %AX_SET in the “Synergy ActiveX API” chapter of the *Synergy DBL Language Reference Manual*.)

Make sure the data type of *new_value* has meaning within the context of the specified property. If it doesn't (for example, if you specify an alpha string for a numeric property), the results are undefined.

We recommend that you use XCALL AX_TKSET instead of accessing it as a function.



Rather than making multiple calls to AX_TKSET to set properties for the same control, use %AX_TKCTLID to get the ActiveX control ID, and then use AX_GET to get the properties. AX_TKSET is convenient (because you don't need the control ID), but it calls AX_SET, so it's not as efficient.

On UNIX, a call to this routine will cause an error. On OpenVMS, this routine is ignored.

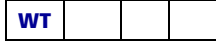
See also

- ▶ [“Error handling for ActiveX control exceptions”](#) on page 4-8.
- ▶ The [“Synergy ActiveX API”](#) chapter of the *Synergy DBL Language Reference Manual*.

Examples

See [Examples](#) on page 4-18.

%AX_TKSINGLE – Create a Toolkit container and load a single ActiveX control



```

window_id = %AX_TKSINGLE (tk_container_name, rows, columns, control_name, [ocx_file],
&                               [license_string][, prefix])

```

Return value

window_id

On Windows, %AX_TKSINGLE returns the ID of the created Toolkit container window. On UNIX, this routine always returns false. On OpenVMS, this routine always returns true. (^VAL)

Arguments

tk_container_name

The name of the Toolkit container window to create. (a)

rows

The number of rows to initially size the window. (n)

columns

The number of columns to initially size the window. (n)

control_name

The name of the ActiveX control you want to load. (a)

ocx_file

(optional) The name of the OCX file that contains the control. (a)

license_string

(optional) The optional license string for the control. (a)

prefix

(optional) The optional prefix for binding extension routines to the automatically registered %UWNDEVENTS_METHOD set and for auto-binding control events. (See the documentation for the *autobind_prefix* argument for [AX_LOAD](#) in the “Synergy ActiveX API” chapter of the *Synergy DBL Language Reference Manual*.) If you pass *prefix* to %AX_TKSINGLE, it is used in two places: for the %UWNDEVENTS_METHOD set that is registered for the Toolkit container window and for binding (AX_BIND) routines to each of the control events. (a)

Discussion

The %AX_TKSINGLE routine does the following:

- ▶ Creates a Toolkit container window.
- ▶ Creates an ActiveX container window.
- ▶ Loads the ActiveX control.
- ▶ Registers a %UWNDEVENTS_METHOD set. (This includes a routine for each event.)
- ▶ Optionally saves the addresses of extension routines so they can be used later within the registered method set. (Use the *prefix* argument to do this.)

%AX_TKSINGLE loads only one ActiveX control per Toolkit container. (To use more than one ActiveX control in a single Toolkit container, you must use %AX_TKWIN.) You maintain only one ID: the window ID returned by %AX_TKSINGLE. The resulting Toolkit container window is part of the current environment, but can be promoted to global.

%AX_TKSINGLE automatically registers a default method set that includes an entry for each event. For example, for D_EVENT_LEFT_CLICK, there's an %AX_TKEVENT_LEFT_CLICK entry. (The names for these routines all begin with AX_TKEVENT_.) This method set also includes %AX_TKEVENT_SIZE (for the D_EVENT_SIZE event), which automatically resizes the ActiveX control when the user or the program resizes the Toolkit container window.

If you want to create your own routines for these events, there are three ways to replace these methods: by creating extensions, by creating your own routines with the same name and linking them in at runtime, or by registering your own method set. We recommend extensions, which are routines you write whose names have a common prefix, but otherwise are modeled after the event names for the AX_TKEVENT_* supplied methods (documented in [“Supplied Methods” on page 4-39](#)). For example, if the common prefix is GRID_ and you create an extension for the left click event, your extension would need to be named GRID_TKEVENT_LEFT_CLICK. You then pass the prefix (“GRID_”, in this case) as the *prefix* argument to %AX_TKSINGLE.

Note the following:

- ▶ Only %AX_TKEVENT_SIZE contains code that affects the ActiveX control. The other automatically registered methods do nothing unless you pass *prefix* or you have created your own routines with the same names as the automatically registered set.
- ▶ Only those routines available at runtime with the correct naming convention are registered as extensions.
- ▶ You don't have to supply an extension for every event.

If you pass a null string (“”) for *tk_container_name*, Toolkit assigns a unique window name in the form _W_nnn, where *nnn* is the window ID.

If the ActiveX control cannot be loaded and the *ocx_file* argument

- ▶ is passed and non-blank, %AX_TKSINGLE attempts to automatically register the OCX file specified by *ocx_file* and then attempts to load the control a second time. If either fails, %AX_TKSINGLE resignals the error trapped on the load failure.
- ▶ is *not* passed or is blank, %AX_TKSINGLE resignals the error trapped on the load failure.

See also

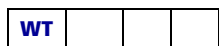
- ▶ [“Error handling for ActiveX control exceptions” on page 4-8.](#)
- ▶ The [“Synergy ActiveX API”](#) chapter of the *Synergy DBL Language Reference Manual*.

Examples

The following example creates a Toolkit container window, creates an ActiveX container, and loads the **MSCAL.Calendar** ActiveX control. When the CAL_SELECT menu entry is signaled, a message box is displayed indicating the currently selected date in the calendar control.

```
proc
  xcall u_start
  wndid = %ax_tksingle("", 10, 40, "MSCAL.Calendar", "mscal.ocx", ,
    & "calendar_")
  xcall b_button(wndid, "CAL_SELECT", DSB_TEXT, "OK")
  xcall b_button(wndid, "O_EXIT", DSB_TEXT, "Cancel")
  xcall u_window(D_PLACE, wndid, 5, 21)
  repeat
    begin
      xcall ax_input(wndid)
      if (g_select)
        using g_entnam select
          ("CAL_SELECT"), begin
            xcall u_message("Selected date (MM/DD/YYYY): " +
              & %string(%ax_tkgetint(wndid, "Month")) + "/" +
              & %string(%ax_tkgetint(wndid, "Day"), "XX") + "/" +
              & %string(%ax_tkgetint(wndid, "Year"), "XXXX"))
          end
        end
      exitloop
    endusing
  end
  xcall u_finish
.end
```

%AX_TKWIN – Create a Toolkit container window



```
window_id = %AX_TKWIN(tk_container_name, rows, columns[, ax_container_id][, font_id])
```

Return value

window_id

On Windows, %AX_TKWIN returns the Toolkit window ID. On UNIX and OpenVMS, this routine returns false. (^VAL)

Arguments

tk_container_name

A string that contains the name of the Toolkit container window. This can be up to 15 characters. (a)

rows

The height of the window in character rows. (n)

columns

The width of the window in character columns. (n)

ax_container_id

(optional) Returned with the ActiveX container ID. (n)

font_id

(optional) The name of a font palette entry, a font handle, or one of the following: (a or n)

DF_CURRENT Current font.

DF_DEFAULT Global font.

DF_ALTERNATE Alternate font.

DF_DEBUGGER Debugger font.

Discussion

%AX_TKWIN creates two windows: a standard UI Toolkit window and an ActiveX container that is “slaved” to it. The ActiveX container is placed within the Toolkit window with an initial size that’s exactly the same size as the Toolkit window.

%AX_TKWIN also sets up the necessary control structures to use the window with AX_INPUT and then to control the mapping of keys as either accelerators or characters destined for the control or the Synergy application (via AX_WANTSKEY).

Because the outer container is a Toolkit window, you can perform all the Toolkit's standard window-related functions (place, remove, delete, resize, and so forth, including border and title manipulation) by referencing the Toolkit window ID returned by %AX_TKWIN. Similarly, you can perform all the ActiveX-related functions (load a control, get a property value, set a property value, and so on) by referencing *ax_container_id*.

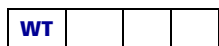
If you pass a null string ("") for *tk_container_name*, Toolkit assigns a unique window name in the form *_W_nnn*, where *nnn* is the window ID.

Font_id enables you to specify a font for the container window as you create it. Changing the font later, after %AX_TKWIN has returned, may change the size of the container window. Therefore the size of the ActiveX control may need to be changed as well.

See also

- ▶ [“Error handling for ActiveX control exceptions” on page 4-8.](#)
- ▶ The [“Synergy ActiveX API”](#) chapter of the *Synergy DBL Language Reference Manual*.

AX_WANTSKEY – Register keys to an ActiveX control



```
xcall AX_WANTSKEY (tk_container_id, key_code, mapping_flag[, key_code,
&                               mapping_flag, ...])
```

Arguments

tk_container_id

The ID of the UI Toolkit ActiveX container window. (n)

key_code

The Toolkit key code for the desired key (defined in **inpctl.def**) or one of the following: (n)

D_KEY_ALL All key codes (CTRL_A_KEY through CTL_SHFTTAB)

D_KEY_MENU All keys that are currently mapped to shortcuts

mapping_flag

One of the following values (defined in **tools.def**): (n)

D_MAP_DEFAULT Map as a control accelerator, then menu accelerator, then keystroke. (default)

D_MAP_KEY Map only as a keystroke.

D_MAP_CONTROL Map as a control accelerator, then keystroke.

D_MAP_MENU Map only as a menu accelerator (shortcut).

Discussion

This routine allows function and control keys to be passed to the ActiveX control instead of being interpreted by Toolkit. (On UNIX and OpenVMS, this routine is ignored.)

The special *key_code* values enable you to allow only certain keys to be processed as shortcuts.

When a function or control key is pressed while in AX_INPUT, the default processing of that keystroke is as follows:

- ▶ If the key is mapped as an accelerator by the focused control or any of its containers, the key will be mapped accordingly.
- ▶ Otherwise, if the key is available to Toolkit as a shortcut, it will be mapped accordingly, whether or not that shortcut is currently active.
- ▶ Finally, if the key is neither an accelerator for the control nor an available Toolkit shortcut key, it is translated as a keystroke and passed to the focused control.

AX_WANTSKEY allows this behavior to be overridden in one of three ways for any control or function key when processing a specific ActiveX container window. (The mapping flag D_MAP_DEFAULT specifies the default processing and is only needed to reset a previously specified mapping flag for that key.)

- ▶ If a key is designated with a D_MAP_KEY mapping flag, neither the control's accelerators nor UI Toolkit's shortcuts will be mapped from this key. Rather, the key will immediately be interpreted as a keystroke and passed to the focused control. This option is useful with Delphi controls that typically respond to keystrokes using an OnKeyDown or OnKeyPress event.
- ▶ If a key is designated with a D_MAP_CONTROL mapping flag, the processing is the same as D_MAP_DEFAULT except that UI Toolkit's shortcut mapping is skipped. The key will only be used by the control as either an accelerator (if the control defines it as such) or as a keystroke. This option is useful when you know that the control needs to see the key, but you do not know whether it defines it as an accelerator or a keystroke. Visual Basic controls often fall into this category.
- ▶ If a key is designated with a D_MAP_MENU mapping flag, the control will not see the keystroke at all, and the key will be mapped as a UI Toolkit shortcut. This option is useful if you want to override a key that is used by the control to signal a menu entry instead.

The following table illustrates the types of key mapping that may occur for each option. Where more than one type of mapping is shown, the leftmost marked column has priority.

Option	Control accelerator	UI Toolkit shortcut	Keystroke to control
D_MAP_DEFAULT	✓	✓	
D_MAP_KEY			✓
D_MAP_CONTROL	✓		✓
D_MAP_MENU		✓	

Because all key codes that can be passed to AX_WANTSKEY are available to Toolkit as shortcuts, a key will never be seen as a keystroke to the control when shortcut mapping is enabled for the key.



The application used to create the ActiveX control can affect whether or not you need to use AX_WANTSKEY. For instance, controls written using Visual Basic rarely need to use AX_WANTSKEY, because keystrokes like TAB, SHIFT+TAB, and ESC are already mapped as accelerators. However, controls written using Delphi usually require you to use AX_WANTSKEY, because Delphi doesn't automatically translate keystrokes. Unless you developed the control, you won't know if it is handling the keystrokes or not. Therefore, we recommend that at first you try not using AX_WANTSKEY. If a keystroke doesn't get through to the control, you can add the key using AX_WANTSKEY.

See also

- ▶ [“Error handling for ActiveX control exceptions”](#) on page 4-8.
- ▶ The [“Synergy ActiveX API”](#) chapter of the *Synergy DBL Language Reference Manual*.

Examples

The default mapping works for most controls, but a more robust setting can be established by calling the following statement after you have placed and enabled all desired menu shortcuts:

```
xcall ax_wantskey(tk_container_id, D_KEY_ALL, D_MAP_CONTROL, D_KEY_MENU,
&                                D_MAP_MENU)
```

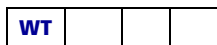
This statement first designates that all keys belong to the control (either as an accelerator or as a keystroke), and then it maps all enabled menu shortcuts to Toolkit. Therefore, if the key corresponds to an enabled menu shortcut it will be interpreted as such. Otherwise, it is left to the control to interpret.

Also see the [Examples](#) on page 4-13.

Routine for the Spread Control

To use the full functionality of the FarPoint Spread control, use %GRID_CREATE to create the Toolkit container window, create the ActiveX container, and load the Spread control. See [“Creating a container for the Spread control” on page 4-6](#) for information on other routines you can use.

%GRID_CREATE – Create a Toolkit container for the Spread control



```
window_id = %GRID_CREATE (rows, columns, license_string, [prefix], [control_name]
&                        [, ocx_file])
```

Return value

window_id

On Windows, %GRID_CREATE returns the ID of the created Toolkit container window. On UNIX, this routine always returns false. On OpenVMS, this routine always returns true. (^VAL)

Arguments

rows

The number of rows for the initial container window size. (n)

columns

The number of columns for the initial container window size. (n)

license_string

Your developer license string for the Spread control. (a)

prefix

(optional) The optional prefix for binding extension routines to the automatically registered %UWNDEVENTS_METHOD set and for auto-binding control events. (See the documentation for the *autobind_prefix* argument for [AX_LOAD](#) in the “Synergy ActiveX API” chapter of the *Synergy DBL Language Reference Manual*.) If you pass *prefix* to %GRID_CREATE, it is used in two places: for the %UWNDEVENTS_METHOD set that is registered for the Toolkit container window and for binding (AX_BIND) routines to each of the control events. (a)

control_name

(optional) The name of the control you want to load. (a)

ocx_file

(optional) The filename of the OCX file that contains the control. (a)

Discussion

%GRID_CREATE makes it easy to add the FarPoint Spread control to a Toolkit application. You maintain only one ID: the window ID returned by %GRID_CREATE. The %GRID_CREATE routine

- ▶ creates a Toolkit container window.
- ▶ creates an ActiveX container window.
- ▶ loads the Spread control.
- ▶ registers a %UWNDEVENTS_METHOD set. (This includes a routine for each element.)
- ▶ optionally saves the address of extension routines so they can be used later within the registered method set. (Use the *prefix* argument to do this.)

The resulting Toolkit container window is part of the current environment, but can be promoted to global.

To obtain your developer license string for the *license_string* argument, use the ActiveX Diagnostic Utility (**axutil.exe**). You must have installed a licensed copy of the Spread control (purchased from GrapeCity, Inc.) to get this string.

For information on the *prefix* argument and on causing the routines in the automatically registered method set to call routines (extensions) you've created, see the %AX_TKSINGLE [Discussion on page 4-29.2](#)

By default %GRID_CREATE loads the Spread control distributed with Toolkit. If you want to use a different version of the Spread control, you can use the *control_name* argument to pass a different control name, and you can use the *ocx_file* argument to pass the name (and path, if necessary) of a different OCX file and automatically register it.

If the Spread control cannot be loaded and the *ocx_file* argument

- ▶ is passed and non-blank, %GRID_CREATE attempts to automatically register the OCX file specified by *ocx_file* and then attempts to load the control a second time. If either of these attempts fails, %GRID_CREATE resignals the error trapped on the load failure.
- ▶ is not passed or is blank, %GRID_CREATE resignals the error trapped on the load failure.

The UI Toolkit distribution includes **grid.def**, a file that provides definitions for constants used by the Spread control. See Examples below.

See the Spread control documentation for information on the methods, properties, and events supported by the Spread control.

See also

- ▶ [“Error handling for ActiveX control exceptions” on page 4-8.](#)
- ▶ The [“Synergy ActiveX API”](#) chapter of the *Synergy DBL Language Reference Manual*.

Examples

The following example creates a Toolkit container, creates an ActiveX container, and loads the Spread control.

```
.include "WND:tools.def"
.include "DBLDIR:activex.def"
.include "WND:grid.def"
.define  D_SPREAD_LICENSE, 1234567

proc
    xcall u_start
    wndid = %grid_create(10, 40, D_SPREAD_LICENSE, "tgrid_")
    xcall u_window(D_PLACE, wndid, 10, 21)
    ctlid = %ax_tkctlid(wndid)
    xcall ax_set(ctlid, "Col", 1)
    xcall ax_set(ctlid, "Col2", 5)
    xcall ax_set(ctlid, "Row", 10)
    xcall ax_set(ctlid, "Row2", 10)
    xcall ax_set(ctlid, "BlockMode", TRUE)
    xcall ax_set(ctlid, "BackColor", RGB_VALUE(0,0,255))
    xcall ax_set(ctlid, "BlockMode", FALSE)
    xcall ax_set(ctlid, "Row", 2)
    xcall ax_set(ctlid, "Col", 3)
    xcall ax_set(ctlid, "ForeColor", RGB_VALUE(255,0,0))
    xcall ax_set(ctlid, "Text", "Hello world!")
    xcall ax_set(ctlid, "Action", SS_ACTION_ACTIVE_CELL)
    repeat
        begin
            xcall ax_input(wndid)
            if (g_select)
                exitloop
            end
        xcall u_finish
    .end
```

Supplied Methods

The `AX_TKEVENT_*` supplied methods are `%UWNDEVENTS_METHOD` methods that are loaded by `%AX_TKSINGLE` and `%GRID_CREATE`. These methods automatically call developer-supplied methods if you pass the *prefix* argument to `%AX_TKSINGLE` or `%GRID_CREATE`. For more information, see the `%AX_TKSINGLE` [Discussion on page 4-29](#).

`%AX_TKEVENT_CLOSE` – Call extension for close event

`%AX_TKEVENT_LEFT_CLICK` – Call extension for left click event

`%AX_TKEVENT_LEFT_DBLCLK` – Call extension for left double-click event

`%AX_TKEVENT_MAXIMIZE` – Call extension for maximize event

`%AX_TKEVENT_MIDDLE_CLICK` – Call extension for middle click event

`%AX_TKEVENT_MIDDLE_DBLCLK` – Call extension for middle double-click event

`%AX_TKEVENT_MINIMIZE` – Call extension routine for minimize event

`%AX_TKEVENT_MOVE` – Call extension routine for move event

`%AX_TKEVENT_RESTORE` – Call extension routine for restore event

`%AX_TKEVENT_RIGHT_CLICK` – Call extension routine for right click event

`%AX_TKEVENT_RIGHT_DBLCLK` – Call extension for right double-click event

`%AX_TKEVENT_SCROLL` – Call extension routine for scroll event

`%AX_TKEVENT_SIZE` – Size the Toolkit container window

%AX_TKEVENT_CLOSE – Call extension for close event

WT			

value = %AX_TKEVENT_CLOSE (*resv1*, *resv2*, ^VAL(*container_id*), ^VAL(*event*))

Return value

value

The return value for the method. On Windows, this function always returns true unless the extension returns false. On UNIX, this function will cause an error if invoked. On OpenVMS, it returns true. (^VAL)

Arguments

resv1

resv2

Reserved for internal use. (a)

container_id

The ID of the Toolkit container window for the ActiveX control. This must be passed by value: ^VAL(*container_id*). (i)

event

The event code for the event that invokes the extension. This must be D_EVENT_CLOSE, and it must be passed by value: ^VAL(D_EVENT_CLOSE). (i)

Discussion

If you passed the *prefix* argument to %AX_TKSINGLE or %GRID_CREATE, and a routine with the name *prefix_TKEVENT_CLOSE* was available when the container was created, %AX_TKEVENT_CLOSE calls your extension routine for the D_EVENT_CLOSE event. If you didn't pass the *prefix* argument, %AX_TKEVENT_CLOSE does nothing.

This routine is part of the method set that's registered when you use %AX_TKSINGLE or %GRID_CREATE.

See also

- ▶ [%UWNDEVENTS_METHOD](#) on page 16-237.
- ▶ [%AX_TKSINGLE](#) on page 4-28.
- ▶ [%GRID_CREATE](#) on page 4-36.
- ▶ [“Appendix D: Methods”](#).

%AX_TKEVENT_LEFT_CLICK – Call extension for left click event

WT			

```
value = %AX_TKEVENT_LEFT_CLICK(resv1, resv2, ^VAL(container_id), ^VAL(event),
&      ^VAL(col_cell), ^VAL(row_cell), ^VAL(col_pix), ^VAL(row_pix),
&      ^VAL(add_info) )
```

Return value

value

The return value for the method. On Windows, this function always returns true unless the extension returns false. On UNIX, this function will cause an error if invoked. On OpenVMS, it returns true. (**^VAL**)

Arguments

resv1

resv2

Reserved for internal use. (**a**)

container_id

The ID of the Toolkit container window for the ActiveX control. This must be passed by value: ^VAL(*container_id*). (**i**)

event

The event code for the event that invokes the extension. This must be D_EVENT_LEFT_CLICK, and it must be passed by value: ^VAL(D_EVENT_LEFT_CLICK). (**i**)

col_cell

The horizontal position (x axis) as a cell column number. This must be passed by value: ^VAL(*col_cell*). (**i**)

row_cell

The vertical position (y axis) as a cell row number. This must be passed by value: ^VAL(*row_cell*). (**i**)

col_pix

The horizontal position (x axis) as a pixel column number. This must be passed by value: ^VAL(*col_pix*). (**i**)

row_pix

The vertical position (y axis) as a pixel row number. This must be passed by value: ^VAL(*row_pix*). (**i**)

add_info

Additional mouse click information. This must be passed by value: ^VAL(*add_info*). (i)

D_AREA_CLIENT	Click was in client area.
D_AREA_NONCLIENT	Click was in non-client area.
D_BUTTON_LEFT	Left mouse button was down during click.
D_BUTTON_MIDDLE	Middle mouse button was down during click.
D_BUTTON_RIGHT	Right mouse button was down during click.
D_KEY_CONTROL	Control key was down during click.
D_KEY_SHIFT	Shift key was down during click.
D_LSTCLK_FOOTER	Click was in footer section.
D_LSTCLK_HEADER	Click was in header section.
D_LSTCLK_ONLINE	Click was on a line.

Discussion

If you passed the *prefix* argument to %AX_TKSINGLE or %GRID_CREATE, and a routine with the name *prefix_TKEVENT_LEFT_CLICK* was available when the container was created, %AX_TKEVENT_LEFT_CLICK calls your extension routine for the D_EVENT_LEFT_CLICK event. If you didn't pass the *prefix* argument, %AX_TKEVENT_LEFT_CLICK does nothing.

This routine is part of the method set that's registered when you use %AX_TKSINGLE or %GRID_CREATE.

See also

- ▶ [%UWNDEVENTS_METHOD](#) on page 16-237.
- ▶ [%AX_TKSINGLE](#) on page 4-28.
- ▶ [%GRID_CREATE](#) on page 4-36.
- ▶ “Appendix D: Methods”.

%AX_TKEVENT_LEFT_DBLCLK – Call extension for left double-click event

WT			

```
value = %AX_TKEVENT_LEFT_DBLCLK(resv1, resv2, ^VAL(container_id), ^VAL(event),
&      ^VAL(col_cell), ^VAL(row_cell), ^VAL(col_pix), ^VAL(row_pix),
&      ^VAL(add_info) )
```

Return value

value

The return value for the method. On Windows, this function always returns true unless the extension returns false. On UNIX, this function will cause an error if invoked. On OpenVMS, it returns true. (**^VAL**)

Arguments

resv1

resv2

Reserved for internal use. (**a**)

container_id

The ID of the Toolkit container window for the ActiveX control. This must be passed by value: ^VAL(*container_id*). (**i**)

event

The event code for the event that invokes the extension. This must be D_EVENT_LEFT_DBLCLK, and it must be passed by value: ^VAL(D_EVENT_LEFT_DBLCLK). (**i**)

control_id

The ID of the ActiveX control. This must be passed by value: ^VAL(*control_id*). (**i**)

col_cell

The horizontal position (x axis) as a cell column number. This must be passed by value: ^VAL(*col_cell*). (**i**)

row_cell

The vertical position (y axis) as a cell row number. This must be passed by value: ^VAL(*row_cell*). (**i**)

col_pix

The horizontal position (x axis) as a pixel column number. This must be passed by value: ^VAL(*col_pix*). (**i**)

row_pix

The vertical position (y axis) as a pixel row number. This must be passed by value:

`^VAL(row_pix)`. (i)

add_info

Additional mouse click information. This must be passed by value: `^VAL(add_info)`. (i)

D_AREA_CLIENT	Click was in client area.
D_AREA_NONCLIENT	Click was in non-client area.
D_BUTTON_LEFT	Left mouse button was down during click.
D_BUTTON_MIDDLE	Middle mouse button was down during click.
D_BUTTON_RIGHT	Right mouse button was down during click.
D_KEY_CONTROL	Control key was down during click.
D_KEY_SHIFT	Shift key was down during click.
D_LSTCLK_FOOTER	Click was in footer section.
D_LSTCLK_HEADER	Click was in header section.
D_LSTCLK_ONLINE	Click was on a line.

Discussion

If you passed the *prefix* argument to `%AX_TKSINGLE` or `%GRID_CREATE`, and a routine with the name *prefix*_TKEVENT_LEFT_DBLCLK was available when the container was created, `%AX_TKEVENT_LEFT_DBLCLK` calls your extension routine for the `D_EVENT_LEFT_DBLCLK` event. If you didn't pass the *prefix* argument, `%AX_TKEVENT_LEFT_DBLCLK` does nothing.

This routine is part of the method set that's registered when you use `%AX_TKSINGLE` or `%GRID_CREATE`.

See also

- ▶ [%UWNDEVENTS_METHOD](#) on page 16-237.
- ▶ [%AX_TKSINGLE](#) on page 4-28.
- ▶ [%GRID_CREATE](#) on page 4-36.
- ▶ “Appendix D: Methods”.

%AX_TKEVENT_MAXIMIZE – Call extension for maximize event

WT			

value = %AX_TKEVENT_MAXIMIZE (*resv1*, *resv2*, ^VAL(*container_id*), ^VAL(*event*))

Return value

value

The return value for the method. On Windows, this function always returns true unless the extension returns false. On UNIX, this function will cause an error if invoked. On OpenVMS, it returns true. (^VAL)

Arguments

resv1

resv2

Reserved for internal use. (a)

container_id

The ID of the Toolkit container window for the ActiveX control. This must be passed by value: ^VAL(*container_id*). (i)

event

The event code for the event that invoke the extension. This must be D_EVENT_MAXIMIZE, and it must be passed by value: ^VAL(D_EVENT_MAXIMIZE). (i)

Discussion

If you passed the *prefix* argument to %AX_TKSINGLE or %GRID_CREATE, and a routine with the name *prefix_TKEVENT_MAXIMIZE* was available when the container was created, %AX_TKEVENT_MAXIMIZE calls your extension routine for the D_EVENT_MAXIMIZE event. If you didn't pass the *prefix* argument, %AX_TKEVENT_MAXIMIZE does nothing.

This routine is part of the method set that's registered when you use %AX_TKSINGLE or %GRID_CREATE.

See also

- ▶ [%UWNDEVENTS_METHOD](#) on page 16-237.
- ▶ [%AX_TKSINGLE](#) on page 4-28.
- ▶ [%GRID_CREATE](#) on page 4-36.
- ▶ “Appendix D: Methods”.

%AX_TKEVENT_MIDDLE_CLICK – Call extension for middle click event

WT			

```
value = %AX_TKEVENT_MIDDLE_CLICK(resv1, resv2, ^VAL(container_id), ^VAL(event),
&      ^VAL(col_cell), ^VAL(row_cell), ^VAL(col_pix), ^VAL(row_pix),
&      ^VAL(add_info) )
```

Return value

value

The return value for the method. On Windows, this function always returns true unless the extension returns false. On UNIX, this function will cause an error if invoked. On OpenVMS, it returns true. (**^VAL**)

Arguments

resv1

resv2

Reserved for internal use. (**a**)

container_id

The ID of the Toolkit container window for the ActiveX control. This must be passed by value: **^VAL(container_id)**. (**i**)

event

The event code for the event that invokes the extension. This must be **D_EVENT_MIDDLE_CLICK**, and it must be passed by value: **^VAL(D_EVENT_MIDDLE_CLICK)**. (**i**)

col_cell

The horizontal position (x axis) as a cell column number. This must be passed by value: **^VAL(col_cell)**. (**i**)

row_cell

The vertical position (y axis) as a cell row number. This must be passed by value: **^VAL(row_cell)**. (**i**)

col_pix

The horizontal position (x axis) as a pixel column number. This must be passed by value: **^VAL(col_pix)**. (**i**)

row_pix

The vertical position (y axis) as a pixel row number. This must be passed by value:
^VAL(*row_pix*). (i)

add_info

Additional mouse click information. This must be passed by value: ^VAL(*add_info*). (i)

D_AREA_CLIENT	Click was in client area.
D_AREA_NONCLIENT	Click was in non-client area.
D_BUTTON_LEFT	Left mouse button was down during click.
D_BUTTON_MIDDLE	Middle mouse button was down during click.
D_BUTTON_RIGHT	Right mouse button was down during click.
D_KEY_CONTROL	Control key was down during click.
D_KEY_SHIFT	Shift key was down during click.
D_LSTCLK_FOOTER	Click was in footer section.
D_LSTCLK_HEADER	Click was in header section.
D_LSTCLK_ONLINE	Click was on a line.

Discussion

If you passed the *prefix* argument to %AX_TKSINGLE or %GRID_CREATE, and a routine with the name *prefix_TKEVENT_MIDDLE_CLICK* was available when the container was created, %AX_TKEVENT_MIDDLE_CLICK calls your extension routine for the D_EVENT_MIDDLE_CLICK event. If you didn't pass the *prefix* argument, %AX_TKEVENT_MIDDLE_CLICK does nothing.

This routine is part of the method set that's registered when you use %AX_TKSINGLE or %GRID_CREATE.

See also

- ▶ [%UWNDEVENTS_METHOD](#) on page 16-237.
- ▶ [%AX_TKSINGLE](#) on page 4-28.
- ▶ [%GRID_CREATE](#) on page 4-36.
- ▶ [“Appendix D: Methods”](#).

%AX_TKEVENT_MIDDLE_DBLCLK – Call extension for middle double-click event

WT			

```
value = %AX_TKEVENT_MIDDLE_DBLCLK(resv1, resv2, ^VAL(container_id),
&      ^VAL(event), ^VAL(col_cell), ^VAL(row_cell), ^VAL(col_pix), ^VAL(row_pix),
&      ^VAL(add_info) )
```

Return value

value

The return value for the method. On Windows, this function always returns true unless the extension returns false. On UNIX, this function will cause an error if invoked. On OpenVMS, it returns true. (^VAL)

Arguments

resv1

resv2

Reserved for internal use. (a)

container_id

The ID of the Toolkit container window for the ActiveX control. This must be passed by value: ^VAL(container_id). (i)

event

The event code for the event that invokes the extension. This must be D_EVENT_MIDDLE_DBLCLK, and it must be passed by value: ^VAL(D_EVENT_MIDDLE_DBLCLK). (i)

col_cell

The horizontal position (x axis) as a cell column number. This must be passed by value: ^VAL(col_cell). (i)

row_cell

The vertical position (y axis) as a cell row number. This must be passed by value: ^VAL(row_cell). (i)

col_pix

The horizontal position (x axis) as a pixel column number. This must be passed by value: ^VAL(col_pix). (i)

row_pix

The vertical position (y axis) as a pixel row number. This must be passed by value:
 ^VAL(*row_pix*). (i)

add_info

Additional mouse click information. This must be passed by value: ^VAL(*add_info*). (i)

D_AREA_CLIENT	Click was in client area.
D_AREA_NONCLIENT	Click was in non-client area.
D_BUTTON_LEFT	Left mouse button was down during click.
D_BUTTON_MIDDLE	Middle mouse button was down during click.
D_BUTTON_RIGHT	Right mouse button was down during click.
D_KEY_CONTROL	Control key was down during click.
D_KEY_SHIFT	Shift key was down during click.
D_LSTCLK_FOOTER	Click was in footer section.
D_LSTCLK_HEADER	Click was in header section.
D_LSTCLK_ONLINE	Click was on a line.

Discussion

If you passed the *prefix* argument to %AX_TKSINGLE or %GRID_CREATE, and a routine with the name *prefix*_TKEVENT_MIDDLE_DBLCLK was available when the container was created, %AX_TKEVENT_MIDDLE_DBLCLK calls your extension routine for the D_EVENT_MIDDLE_DBLCLK event. If you didn't pass the *prefix* argument, %AX_TKEVENT_MIDDLE_DBLCLK does nothing.

This routine is part of the method set that's registered when you use %AX_TKSINGLE or %GRID_CREATE.

See also

- ▶ [%UWNDEVENTS_METHOD](#) on page 16-237.
- ▶ [%AX_TKSINGLE](#) on page 4-28.
- ▶ [%GRID_CREATE](#) on page 4-36.
- ▶ [“Appendix D: Methods”](#).

%AX_TKEVENT_MINIMIZE – Call extension routine for minimize event

WT			

value = %AX_TKEVENT_MINIMIZE (*resv1*, *resv2*, ^VAL(*container_id*), ^VAL(*event*))

Return value

value

The return value for the method. On Windows, this function always returns true unless the extension returns false. On UNIX, this function will cause an error if invoked. On OpenVMS, it returns true. (^VAL)

Arguments

resv1

resv2

Reserved for internal use. (a)

container_id

The ID of the Toolkit container window for the ActiveX control. This must be passed by value: ^VAL(*container_id*). (i)

event

The event code for the event that invokes the extension. This must be D_EVENT_MINIMIZE, and it must be passed by value: ^VAL(D_EVENT_MINIMIZE). (i)

Discussion

If you passed the *prefix* argument to %AX_TKSINGLE or %GRID_CREATE, and a routine with the name *prefix_TKEVENT_MINIMIZE* was available when the container was created, %AX_TKEVENT_MINIMIZE calls your extension routine for the D_EVENT_MINIMIZE event. If you didn't pass the *prefix* argument, %AX_TKEVENT_MINIMIZE does nothing.

This routine is part of the method set that's registered for %AX_TKSINGLE or %GRID_CREATE.

See also

- ▶ [%UWNDEVENTS_METHOD](#) on page 16-237.
- ▶ [%AX_TKSINGLE](#) on page 4-28.
- ▶ [%GRID_CREATE](#) on page 4-36.
- ▶ [“Appendix D: Methods”](#).

%AX_TKEVENT_MOVE – Call extension routine for move event

WT			

```
value = %AX_TKEVENT_MOVE (resv1, resv2, ^VAL (container_id), ^VAL (event),
& ^VAL (col_cell), ^VAL (row_cell), ^VAL (col_pix), ^VAL (row_pix) )
```

Return value

value

The return value for the method. On Windows, this function always returns true unless the extension returns false. On UNIX, this function will cause an error if invoked. On OpenVMS, it returns true. (**^VAL**)

Arguments

resv1

resv2

Reserved for internal use. (**a**)

container_id

The ID of the Toolkit container window for the ActiveX control. This must be passed by value: ^VAL(*container_id*). (**i**)

event

The event code for the event that invokes the extension. This must be D_EVENT_MOVE, and it must be passed by value: ^VAL(D_EVENT_MOVE). (**i**)

col_cell

The horizontal position (x axis) as a cell column number. This must be passed by value: ^VAL(*col_cell*). (**i**)

row_cell

The vertical position (y axis) as a cell row number. This must be passed by value: ^VAL(*row_cell*). (**i**)

col_pix

The horizontal position (x axis) as a pixel column number. This must be passed by value: ^VAL(*col_pix*). (**i**)

row_pix

The vertical position (y axis) as a pixel row number. This must be passed by value: ^VAL(*row_pix*). (**i**)

Discussion

If you passed the *prefix* argument to %AX_TKSINGLE or %GRID_CREATE, and a routine with the name *prefix_TKEVENT_MOVE* was available when the container was created, %AX_TKEVENT_MOVE calls your extension routine for the D_EVENT_MOVE event. If you didn't pass the *prefix* argument, %AX_TKEVENT_MOVE does nothing.

This routine is part of the method set that's registered when you use %AX_TKSINGLE or %GRID_CREATE.

See also

- ▶ [%UWNDEVENTS_METHOD on page 16-237.](#)
- ▶ [%AX_TKSINGLE on page 4-28.](#)
- ▶ [%GRID_CREATE on page 4-36.](#)
- ▶ [“Appendix D: Methods”.](#)

%AX_TKEVENT_RESTORE – Call extension routine for restore event

WT			

value = %AX_TKEVENT_RESTORE (*resv1*, *resv2*, ^VAL(*container_id*), ^VAL(*event*))

Return value

value

The return value for the method. On Windows, this function always returns true unless the extension returns false. On UNIX, this function will cause an error if invoked. On OpenVMS, it returns true. (^VAL)

Arguments

resv1

resv2

Reserved for internal use. (a)

container_id

The ID of the Toolkit container window for the ActiveX control. This must be passed by value: ^VAL(*container_id*). (i)

event

The event code for the event that invokes the extension. This must be D_EVENT_RESTORE, and it must be passed by value: ^VAL(D_EVENT_RESTORE). (i)

Discussion

If you passed the *prefix* argument to %AX_TKSINGLE or %GRID_CREATE, and a routine with the name *prefix*_TKEVENT_RESTORE was available when the container was created, %AX_TKEVENT_RESTORE calls your extension routine for the D_EVENT_RESTORE event. If you didn't pass the *prefix* argument, %AX_TKEVENT_RESTORE does nothing.

This routine is part of the method set that's registered for %AX_TKSINGLE or %GRID_CREATE.

See also

- ▶ [%UWNDEVENTS_METHOD on page 16-237.](#)
- ▶ [%AX_TKSINGLE on page 4-28.](#)
- ▶ [%GRID_CREATE on page 4-36.](#)
- ▶ [“Appendix D: Methods”.](#)

%AX_TKEVENT_RIGHT_CLICK – Call extension routine for right click event

WT			

```
value = %AX_TKEVENT_RIGHT_CLICK(resv1, resv2, ^VAL(container_id), ^VAL(event),
&      ^VAL(col_cell), ^VAL(row_cell), ^VAL(col_pix), ^VAL(row_pix),
&      ^VAL(add_info) )
```

Return value

value

The return value for the method. On Windows, this function always returns true unless the extension returns false. On UNIX, this function will cause an error if invoked. On OpenVMS, it returns true. (^VAL)

Arguments

resv1

resv2

Reserved for internal use. (a)

container_id

The ID of the Toolkit container window for the ActiveX control. This must be passed by value: ^VAL(container_id). (i)

event

The event code for the event that invokes the extension. This must always be D_EVENT_RIGHT_CLICK, and it must be passed by value: ^VAL(D_EVENT_RIGHT_CLICK). (i)

col_cell

The horizontal position (x axis) as a cell column number. This must be passed by value: ^VAL(col_cell). (i)

row_cell

The vertical position (y axis) as a cell row number. This must be passed by value: ^VAL(row_cell). (i)

col_pix

The horizontal position (x axis) as a pixel column number. This must be passed by value: ^VAL(col_pix). (i)

row_pix

The vertical position (y axis) as a pixel row number. This must be passed by value:
^VAL(*row_pix*). (i)

add_info

Additional mouse click information. This must be passed by value: ^VAL(*add_info*). (i)

D_AREA_CLIENT	Click was in client area.
D_AREA_NONCLIENT	Click was in non-client area.
D_BUTTON_LEFT	Left mouse button was down during click.
D_BUTTON_MIDDLE	Middle mouse button was down during click.
D_BUTTON_RIGHT	Right mouse button was down during click.
D_KEY_CONTROL	Control key was down during click.
D_KEY_SHIFT	Shift key was down during click.
D_LSTCLK_FOOTER	Click was in footer section.
D_LSTCLK_HEADER	Click was in header section.
D_LSTCLK_ONLINE	Click was on a line.

Discussion

If you passed the *prefix* argument to %AX_TKSINGLE or %GRID_CREATE, and a routine with the name *prefix_TKEVENT_RIGHT_CLICK* was available when the container was created, %AX_TKEVENT_RIGHT_CLICK calls your extension routine for the D_EVENT_RIGHT_CLICK event. If you didn't pass the *prefix* argument, %AX_TKEVENT_RIGHT_CLICK does nothing.

This routine is part of the method set that's registered when you use %AX_TKSINGLE or %GRID_CREATE.

See also

- ▶ [%UWNDEVENTS_METHOD](#) on page 16-237.
- ▶ [%AX_TKSINGLE](#) on page 4-28.
- ▶ [%GRID_CREATE](#) on page 4-36.
- ▶ "Appendix D: Methods".

%AX_TKEVENT_RIGHT_DBLCLK – Call extension for right double-click event

WT			

```
value = %AX_TKEVENT_RIGHT_DBLCLK(resv1, resv2, ^VAL(container_id), ^VAL(event),
&      ^VAL(col_cell), ^VAL(row_cell), ^VAL(col_pix), ^VAL(row_pix),
&      ^VAL(add_info) )
```

Return value

value

The return value for the method. On Windows, this function always returns true unless the extension returns false. On UNIX, this function will cause an error if invoked. On OpenVMS, it returns true. (^VAL)

Arguments

resv1

resv2

Reserved for internal use. (a)

container_id

The ID of the Toolkit container window for the ActiveX control. This must be passed by value: ^VAL(*container_id*). (i)

event

The event code for the event that invokes the extension. This must be D_EVENT_RIGHT_DBLCLK, and it must be passed by value: ^VAL(D_EVENT_RIGHT_DBLCLK). (i)

col_cell

The horizontal position (x axis) as a cell column number. This must be passed by value: ^VAL(*col_cell*). (i)

row_cell

The vertical position (y axis) as a cell row number. This must be passed by value: ^VAL(*row_cell*). (i)

col_pix

The horizontal position (x axis) as a pixel column number. This must be passed by value: ^VAL(*col_pix*). (i)

row_pix

The vertical position (y axis) as a pixel row number. This must be passed by value:
^VAL(*row_pix*). (i)

add_info

Additional mouse click information. This must be passed by value: ^VAL(*add_info*). (i)

D_AREA_CLIENT	Click was in client area.
D_AREA_NONCLIENT	Click was in non-client area.
D_BUTTON_LEFT	Left mouse button was down during click.
D_BUTTON_MIDDLE	Middle mouse button was down during click.
D_BUTTON_RIGHT	Right mouse button was down during click.
D_KEY_CONTROL	Control key was down during click.
D_KEY_SHIFT	Shift key was down during click.
D_LSTCLK_FOOTER	Click was in footer section.
D_LSTCLK_HEADER	Click was in header section.
D_LSTCLK_ONLINE	Click was on a line.

Discussion

If you passed the *prefix* argument to %AX_TKSINGLE or %GRID_CREATE, and a routine with the name *prefix*_TKEVENT_RIGHT_DBLCLK was available when the container was created, %AX_TKEVENT_RIGHT_DBLCLK calls your extension routine for the D_EVENT_RIGHT_DBLCLK event. If you didn't pass the *prefix* argument, %AX_TKEVENT_RIGHT_DBLCLK does nothing.

This routine is part of the method set that's registered when you use %AX_TKSINGLE or %GRID_CREATE.

See also

- ▶ [%UWNDEVENTS_METHOD](#) on page 16-237.
- ▶ [%AX_TKSINGLE](#) on page 4-28.
- ▶ [%GRID_CREATE](#) on page 4-36.
- ▶ [“Appendix D: Methods”](#).

%AX_TKEVENT_SCROLL – Call extension routine for scroll event

WT			

value = %AX_TKEVENT_SCROLL(*resv1*, *resv2*, ^VAL(*container_id*), ^VAL(*event*),
& ^VAL(*col_cell*), ^VAL(*row_cell*), ^VAL(*col_offset*), ^VAL(*row_offset*))

Return value

value

The return value for the method. On Windows, this function always returns true unless the extension returns false. On UNIX, this function will cause an error if invoked. On OpenVMS, it returns true. (^VAL)

Arguments

resv1

resv2

Reserved for internal use. (a)

container_id

The ID of the Toolkit container window for the ActiveX control. This must be passed by value: ^VAL(*container_id*). (i)

event

The event code for the event that invokes the extension. This must be D_EVENT_SCROLL, and it must be passed by value: ^VAL(D_EVENT_SCROLL). (i)

col_cell

The display area's horizontal position (x axis) as a cell column number. This must be passed by value: ^VAL(*col_cell*). (i)

row_cell

The display area's vertical position (y axis) as a cell row number. This must be passed by value: ^VAL(*row_cell*). (i)

col_offset

The offset (number of pixels) between the window's horizontal (x axis) position and the display area's horizontal position. This must be passed by value: ^VAL(*col_offset*). (i)

row_offset

The offset (number of pixels) between the window's vertical (y axis) position and the display area's vertical position. This must be passed by value: ^VAL(*row_offset*). (i)

Discussion

If you passed the *prefix* argument to %AX_TKSINGLE or %GRID_CREATE, and a routine with the name *prefix_TKEVENT_SCROLL* was available when the container was created, %AX_TKEVENT_SCROLL calls your extension routine for the D_EVENT_SCROLL event. If you didn't pass the *prefix* argument, %AX_TKEVENT_SCROLL does nothing.

This routine is part of the method set that's registered when you use %AX_TKSINGLE or %GRID_CREATE.

See also

- ▶ [%UWNDEVENTS_METHOD](#) on page 16-237.
- ▶ [%AX_TKSINGLE](#) on page 4-28.
- ▶ [%GRID_CREATE](#) on page 4-36.
- ▶ [“Appendix D: Methods”](#).

%AX_TKEVENT_SIZE – Size the Toolkit container window

WT			

```
value = %AX_TKEVENT_SIZE (resv1, resv2, ^VAL(container_id), ^VAL(event),
& ^VAL(win_width), ^VAL(win_height), ^VAL(client_width), ^VAL(client_height) )
```

Return value

value

The return value for the method. On Windows, this function always returns true unless the extension returns false. On UNIX, this function will cause an error if invoked. On OpenVMS, it returns true. (^VAL)

Arguments

resv1

resv2

Reserved for internal use. (a)

container_id

The ID of the Toolkit container window for the ActiveX control. This must be passed by value: ^VAL(*container_id*). (i)

event

The event code for the event that invokes the extension. This must be D_EVENT_SIZE, and it must be passed by value: ^VAL(D_EVENT_SIZE). (i)

win_width

Width of the logical window (number of columns). This must be passed by value: ^VAL(*win_width*). (i)

win_height

Height of the logical window (number of rows). This must be passed by value: ^VAL(*win_height*). (i)

client_width

Width of the client area (number of pixels). This must be passed by value: ^VAL(*client_width*). (i)

client_height

Height of the client area (number of pixels). This must be passed by value: ^VAL(*client_height*). (i)

Discussion

%AX_TKSINGLE and %GRID_CREATE automatically register a %UWNDEVENTS_METHOD set that includes %AX_TKEVENT_SIZE as the routine for the D_EVENT_SIZE event. When the D_EVENT_SIZE event is triggered (programmatically or by the user), %U_WNDEVENTS calls this routine. If *prefix* (an argument for %AX_TKSINGLE and %GRID_CREATE) was passed and

- ▶ there's an extension for the D_EVENT_SIZE event, %AX_TKEVENT_SIZE calls the extension. If the extension routine returns FALSE, %AX_TKEVENT_SIZE returns true without performing any other function. If the extension returns true, %AX_TKEVENT_SIZE resizes the ActiveX container and control by setting their "Ext_Rect" properties to match the client area of the Toolkit container window.
- ▶ there is no extension for D_EVENT_SIZE, %AX_TKEVENT_SIZE resizes the ActiveX container and control by setting their "Ext_Rect" properties to match the client area of the Toolkit container window and returns true.

(For information on "Ext_Rect", see ["Extended properties"](#) in the "Synergy ActiveX API" chapter of the *Synergy DBL Language Reference Manual*.)

If you assign your own method set to the window, you can call %AX_TKEVENT_SIZE from the routine you've supplied for the D_EVENT_SIZE event.

See also

The ["Synergy ActiveX API"](#) chapter of the *Synergy DBL Language Reference Manual*.

5

Button Routines

Using the Button Subroutines 5-2

Discusses general button appearance and behavior and provides an overview of the routines you'll use to create, modify, and delete buttons in Windows environments.

B_BUTTON – Add, delete, or modify a button on a window or tab set.....	5-4
B_BUTTONSET – Define the placement of a set of buttons on a window or tab set	5-14
%B_DISABLE – Disable a button in a window or tab set.....	5-16
%B_ENABLE – Enable a button in a window or tab set.....	5-17
%B_INFO – Return information about the buttons in a window	5-19

Using the Button Subroutines

On Windows systems, you can add buttons to a window, list, or tab set. Buttons are a standard way of enabling users to terminate input (OK and Cancel buttons are typical examples) or signify that any action should take place. Note the following:

- ▶ Every button must be part of a button set, and there can be only one button set per window, list, or tab set.
- ▶ A button can display either text or a graphic on its face. (See [“Toolkit button graphics” on page 2-15](#) for supported graphic formats and for information on transparent button graphics.)
- ▶ When a button has focus, a thin dotted line is drawn around the text portion of the button, and the button can be “pressed” using either the spacebar or the ENTER key. Whether or not a button has focus, the button can be clicked with the left mouse button.
- ▶ When a button is pressed or clicked, a method subroutine is invoked or a menu entry is signaled. If a menu entry is signaled, **g_select** is set to true, and **g_entnam** is set to the name of the button. (**G_select** and **g_entnam**, global variables defined in **tools.def**, are automatically set by Toolkit when a menu entry is signaled.)
- ▶ When in **I_INPUT** for a window or tab set, buttons are included in the container’s tabbing order. (This is not true of buttons in a list.) **TAB** moves forward and **SHIFT+TAB** moves backward through the buttons.
- ▶ The maximum number of buttons allowed in a container is 127. A fatal error occurs if this maximum is exceeded.
- ▶ You can use **I_NEXT** to move focus to a button.

Except for **%B_INFO**, the routines documented in this chapter can be used only for buttons on a window or tab set. For routines for buttons on a list, see [chapter 9, “List Routines.”](#)

Creating buttons

To create buttons, use the **.BUTTON** script command or, to create them at runtime, use **B_BUTTON**, **IB_BUTTON**, or **L_BUTTON**.

- ▶ The **.BUTTON** script command creates buttons on windows. See [.BUTTON on page 2-13](#) for information.
- ▶ **B_BUTTON** creates buttons on windows and tab sets at runtime. For information, see [B_BUTTON on page 5-4](#). (You can use **IB_BUTTON** to create buttons on input windows at runtime, and you can use the **DTS_BUTTON** subfunction to **%TS_TABSET** to create buttons on tab sets at runtime, but **B_BUTTON** works for tab sets and all windows.)
- ▶ **L_BUTTON** creates buttons on lists. For information, see [L_BUTTON on page 9-55](#).

Defining button placement and the default button

You can define a button set's location within a window (bottom or right), its justification, the maximum number of buttons per row or column, and the default button. To do this, use the `.BUTTON_SET` script command or, at runtime, use `B_BUTTONSET`, `IB_BUTTONSET`, or `L_BUTTONSET`. (To define a button's location in the button set, use `B_BUTTON`.)

- ▶ Use the `.BUTTON_SET` script command for button sets on windows. For more information, see [.BUTTON_SET on page 2-17](#).
- ▶ Use `B_BUTTONSET` for button sets on windows and tab sets. For information, see [B_BUTTONSET on page 5-14](#). (You can use `IB_BUTTONSET` for button sets on input windows and the `DTS_BUTTONSET` subfunction to `%TS_TABSET` for button sets on tab sets, but `B_BUTTONSET` can be used in both of these situations.)
- ▶ Use `L_BUTTONSET` for button sets on lists. See [L_BUTTONSET on page 9-62](#).

Note that you can use these to specify a default button, but this default won't be applied unless you also pass `E_STATE` with the `D_ON` and `D_RETURNBTN` options. See the `E_STATE` [Discussion on page 7-41](#) for more information.

Enabling and disabling buttons

In general, the buttons for a window are enabled when the window is being processed, and they are disabled when the window is not being processed. (For details and exceptions, see the `.BUTTON` [Discussion on page 2-14](#).)

To enable or disable a button, use `%B_ENABLE` and `%B_DISABLE` for buttons on windows and tab sets. Use `%L_BUTTONSTATE` for buttons on lists. For information on how buttons can be automatically enabled and disabled, see the `.BUTTON` [Discussion on page 2-14](#).

Deleting buttons

To delete a button on a window or tab set, use the `DSB_DELETE` subfunction of `%B_BUTTON`. For buttons on lists, use the `DSB_DELETE` subfunction of `%L_BUTTON`.

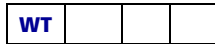
Modifying buttons

To modify a button on a window or tab set, use the `DSB_MODIFY` subfunction of `%B_BUTTON` routine. For buttons on lists, use the `DSB_MODIFY` subfunction of `%L_BUTTON`.

Getting information on buttons

You can also get information on buttons and button sets for a window, tab set, or list by using `%B_INFO`. This function has subfunctions that return the number of buttons, the index for the default button, the enabled state of a button, method information, and so forth. See `%B_INFO` on [page 5-19](#).

B_BUTTON – Add, delete, or modify a button on a window or tab set



```
xcall B_BUTTON (container_id, [subfunction,] args)
```

Arguments

container_id

The ID of the window or tab set that contains or will contain the button. **(n)**

subfunction

(optional) One of the following: **(n)**

DSB_ADD Add a button to the window or tab set. (default)

DSB_DELETE Delete a button from the window or tab set.

DSB_MODIFY Modify a button in the window or tab set.

args

One or more arguments, depending on *subfunction*.

Discussion

The B_BUTTON subroutine creates, deletes, or modifies a Windows-style button on a window or tab set. See the individual subfunctions for details.

This routine has no effect on applications running on UNIX or OpenVMS or in Synergy .NET. It does, however, save button information on these platforms, so if the window is saved to a window library (by using U_SAVE) and later loaded from that library on a Windows platform in traditional Synergy, the button will be on the window (assuming ISAM-compatible platforms).

See also

“Using the Button Subroutines” on page 5-2 for general information on buttons and button sets on windows, tab sets, and lists.

DSB_ADD

```
xcall B_BUTTON (container_id, [DSB_ADD,] name, [type], [type_data], [method], [elb],
&      [select_char], [position][, tooltip])
```

Arguments

container_id

The ID of the window or tab set you want to add the button to. **(n)**

name

The name of the button to add (a maximum of 30 characters). **(a)**

type

(optional) The button face type: **(n)**

DSB_TEXT Text. (default)

DSB_BITMAP Graphic.

type_data

(optional) Text for the button face if *type* is DSB_TEXT. A filename if *type* is DSB_BITMAP. **(a)**

method

(optional) The name of the method subroutine to invoke. **(a)**

elb

(optional) The name of the executable library (**.elb**) that contains *method*. **(a)**

select_char

(optional) The quick-select character. If not passed or blank, the button will not have a quick-select character. **(a)**

position

(optional) The position the button will be inserted in the button set. **(a or n)**

tooltip

(optional) The text of a ToolTip to be displayed when the mouse pointer rests over the button. **(a)**

Discussion

The `DSB_ADD` subfunction of `B_BUTTON` enables you to add a button to a window or tab set. If there are no existing buttons, a default button set is created. By default, button sets are placed at the bottom of the window or tab set, are left-justified, and can have as many buttons on a row as will fit before the row wraps. (You can change these defaults with `B_BUTTONSET`.)

The window or tab set will expand downward or to the right to accommodate the buttons, which may cause your window to extend beyond the bottom or right side of the application window. Also note that buttons are a standard size; text that exceeds that size will not be visible.

The *type* and *type_data* arguments enable you to specify text or a graphic for the button face.

- ▶ If you pass *type* as `DSB_TEXT`, use *type_data* to specify the text to display on the button face.
- ▶ If you pass *type* as `DSB_BITMAP`, use *type_data* to specify the name of a graphic file to display on the button face. See [“Toolkit button graphics” on page 2-15](#) for information on supported graphic formats and for information on creating transparent button graphics.

If you don’t pass *type_data*, the button’s name will be displayed on the button’s face.

If you pass *method*, the method subroutine you pass will be invoked when the button is pressed. The arguments passed to the method are the window ID, button name, and the 20 optional method data arguments passed to `I_INPUT`, `L_INPUT`, or any of the other input routines that allow method data arguments (`L_SELECT`, for example). If you don’t pass *method*, the first ten characters of the button’s name are used to signal a menu entry when the button is pressed.

If you want Toolkit to look in a specific ELB (executable library) for the method specified by *method*, pass the filename of the ELB as the *elb* argument. You can pass the *elb* argument only if you pass the *method* argument.

Specifying a quick select character

Select_char is case insensitive. The character specified by *select_char* does not have to exist in the button’s text, but if it does exist in the button’s text, the first occurrence of *select_char* will be underlined.

If *select_char* is specified in a call to `B_BUTTON`, when the window is being processed by `C_PROCESS`, `I_INPUT`, `TS_PROCESS`, `T_VIEW`, `T_EDIT`, `AX_INPUT`, `L_INPUT`, or `L_SELECT`, in many cases pressing `ALT+select_char` will have the effect of selecting the button. However, this works only for enabled buttons on the currently active window or tab set, which means that a button can be activated only when the application is processing the window or tab set that contains the button. (For information on how `ALT+select_char` works with composite windows, see [“ALT+key button activation” on page 6-7](#).)

To avoid confusing the user, we recommend that no two buttons have the same quick-select character. However, if they do, the following rules apply:

- ▶ If the buttons with the conflicting quick-select character are on the same window, the quick-select character is assigned to the last button defined. The other buttons will not have a quick-select character.

- ▶ Within a tab set, if the tab set container and a contained window define buttons with the same quick-select character, the contained window's button is assigned the quick-select character, and the button on the tab set will have its quick-select character removed. When a different tab is active, one that does not contain a button with a conflicting quick-select character, the quick-select character will be re-enabled on the tab set button.
- ▶ In the case of nested windows, where an input routine is called for a second window within a method for the first window, the quick-select characters for buttons in the first window will still work as long as they are enabled and do not conflict with buttons on the second window. If there is a conflict, the buttons on the second window get the quick-select character until the input process on the second window is exited, at which point any overridden quick-select characters for the first window will be restored.

In the case where the `ALT+select_char` combination for a button matches that for a menu column, the button gets the quick-select character. Note, however, that pressing ALT and releasing it, followed by the quick-select character, activates the menu column.

Specifying the button's position

The *position* argument enables you to insert the button at a given point in the button set:

- ▶ To add the button above or to the left of an existing button (depending on the button set's orientation), pass the name of the existing button. The *position* argument is not case sensitive.
- ▶ To specify a numeric position among existing buttons, pass a position number, which is an index (base 1) that specifies the numeric position (left-to-right or top-to-bottom) among existing buttons. For example, if you pass 1, the button is added as the left-most or top button. If you pass 2, the button will be added as the second button to the left or the second button from the top.
- ▶ To add the button to the end of the existing buttons (to the far right or bottom of the button set), pass *position* as a zero-length or blank alphanumeric value.

Toolkit generates a fatal error if you pass a name for *position* that doesn't match any of the existing button names or if you pass a number that's less than 1 or greater than the number of buttons on the window plus 1.

To add a ToolTip to the button, pass a non-blank value with a length that is greater than zero for *tooltip*. Note that a ToolTip's length can be no longer than the operating system's maximum length for an alphanumeric field. (See [DSB_TOOLTIP on page 5-34](#) for information on retrieving ToolTip text.)

Examples

The following example adds an OK button named `I_OK` with a quick-select character `K`. When `ALT+K` is pressed, a click of the OK button is simulated. If there are other buttons on the window, this button will be added to the end of the button set. Note that we didn't include `DSB_ADD`. You can omit `DSB_ADD` when you add a button because `DSB_ADD` is the default.

```
xcall b_button(wndid, "I_OK", DSB_TEXT, "OK", , , "K")
```

Button Routines

B_BUTTON

The next example creates a button named **lookup** with a graphic (**lookup.bmp**) on its face. Clicking this button will invoke the **lookup** subroutine located in the **button.elb** library, and moving the mouse pointer over this button will display the ToolTip “Enter a customer name.” This button will be added to the beginning of the button set because the value 1 is passed as the *position* argument.

```
xcall b_button(wndid, DSB_ADD, "lookup", DSB_BITMAP, "lookup.bmp",  
& "lookup", "button.elb",, 1, "Enter a customer name")
```

DSB_DELETE

```
xcall B_BUTTON(container_id, DSB_DELETE, name)
```

Arguments

container_id

The ID of the window or tab set that contains the button you want to delete. **(n)**

name

The name of the button to delete. **(a)**

Discussion

The DSB_DELETE subfunction of B_BUTTON deletes a button from a window or tab set. Remaining buttons are repositioned as if the deleted button had never existed. For example, if the button is the second button of three in a button set, the third button will be moved to the second button position.

Name is not case sensitive. If no button with that name exists, Toolkit generates a fatal error.

Examples

The following example deletes a button named **lookup**.

```
xcall b_button(wndid, DSB_DELETE, "lookup")
```

DSB_MODIFY

```
xcall B_BUTTON (container_id, DSB_MODIFY, name, [type], [type_data], [method], [elb],  
&          [select_char], [position][, tooltip])
```

Arguments

container_id

The ID of the window or tab set that contains the button you want to modify. **(n)**

name

The name of the button to modify. **(a)**

type

(optional) The new type of button face. (If you don't pass this argument, the previous value is the default.) **(n)**

DSB_TEXT Text.

DSB_BITMAP Graphic.

type_data

(optional) New text for the button if *type* is DSB_TEXT. If *type* is DSB_BITMAP, this is the filename for the graphic. **(a)**

method

(optional) The name of a new method subroutine to invoke. **(a)**

elb

(optional) The name of the executable library (**.elb**) that contains *method*. **(a)**

select_char

(optional) A new quick-select character. **(a)**

position

(optional) A position you want the button moved to in the button set. **(a or n)**

tooltip

(optional) The text of a ToolTip to be displayed when the mouse pointer rests over the button. **(a)**

Discussion

The `DSB_MODIFY` subfunction of `B_BUTTON` modifies a button in a window or tab set. Use the *name* argument to specify the name (case-insensitive) of an existing button. (If *name* isn't the name of an existing button, Toolkit will generate a fatal error.)

If a button has no method, you can use the *method* argument to add a method. If a button does have a method, you can use the *method* argument to change the method (by passing the new method name) or remove the method (by passing *method* as blank or zero-length), which causes Toolkit to signal the button name as a menu entry. Note that if you change the method for a button and an ELB has been specified for the previous method, Toolkit will look for the new method in the same ELB unless you also pass the *elb* argument.

The *elb* argument enables you to specify a new ELB for a button's method or instruct Toolkit not to use a previous *elb* specification. If you pass a new library name as the *elb* argument, Toolkit will look only in this library (and any libraries linked to it) to find the method specified by *method*. If you pass *elb* as blank or zero-length, Toolkit will look in any available module or open ELB (rather than the ELB specified in a previous *elb* specification). So results depend on the application's context (which libraries are open). For the `DSB_MODIFY` subfunction, you can pass *elb* without passing *method* and vice versa.

The *type* and *type_data* arguments enable you to change what appears on the button's face. The *type_data* argument can be passed without also passing *type*, and *type* can be passed without *type_data*. But if you pass *type*, you may need to pass *type_data* to prevent a caption from being interpreted as the filename for a graphic or vice versa. For information on supported graphic formats and creating transparent graphics, see [“Toolkit button graphics” on page 2-15](#).

The *select_char* argument enables you to specify a new quick-select character for a button or remove a quick-select character from a button. To specify a new quick-select character, pass the character as *select_char*. To remove a quick-select character from a button, pass *select_char* as blank or zero-length. See [“Specifying a quick select character” on page 5-6](#) for more information.

The *position* argument enables you to move the button to a new position in the button set.

- ▶ To move the button above or to the left of an existing button (depending on the button set's orientation), pass the name of the existing button. The *position* argument is not case sensitive.
- ▶ To specify a numeric position among existing buttons, pass a position number, which is an index (base 1) that specifies the new numeric position (left to right or top to bottom) among existing buttons. For example, if you pass 1, the button will be moved to the left-most or top position in the button set. If you pass 2, the button will be moved to the second position to the left or the second position from the top in the button set.
- ▶ To move the button to the end of existing buttons (to the far right or bottom of the button set), pass *position* as a zero-length or blank alphanumeric value.

Note that Toolkit generates a fatal error if you pass a name for *position* that doesn't match any of the existing button names or if you pass a number that's less than 1 or greater than the number of buttons on the window plus 1.

To modify the ToolTip for a button, pass a non-blank value with a length that is greater than zero for *tooltip*. Note that a ToolTip's length can be no longer than the operating system's maximum length for an alphanumeric field. (See [DSB_TOOLTIP](#) on page 5-34 for information on retrieving ToolTip text.)

Examples

The following example changes the button face for the lookup button. It will display a new graphic named **lookup2.png**. (We're assuming that the button previously had a graphic displayed on the button face. If it instead had text on the button face, this example would need to pass **DSB_BITMAP** as the *type* argument to prevent Toolkit from interpreting "lookup2.png" as a text string rather than a filename.)

```
xcall b_button(wndid, DSB_MODIFY, "lookup",, "lookup2.png")
```

The following example changes the button face for the **lookup** button to the text "Search All".

```
xcall b_button(wndid, DSB_MODIFY, "lookup", DSB_TEXT, "Search All")
```

The next example changes the method name and the ELB for the **lookup** button.

```
xcall b_button(wndid, DSB_MODIFY, "lookup",, "search_all",  
& "button2.elb")
```

The next example changes the button so that it no longer calls a method when clicked. Instead, Toolkit will use the first ten characters of the button name to signal a menu entry.

```
xcall b_button(wndid, DSB_MODIFY, "lookup",, "")
```

The next example changes the button so that Toolkit no longer looks in a specific ELB for the method.

```
xcall b_button(wndid, DSB_MODIFY, "lookup",,,, "")
```

The next example changes the quick-select character for the button to "s". When the user presses ALT+S, the button will be activated.

```
xcall b_button(wndid, DSB_MODIFY, "lookup",,,,, "s")
```

The next example moves the **lookup** button to the beginning of the button set. (This will be the far left or top depending the orientation of the button set.)

```
xcall b_button(wndid, DSB_MODIFY, "lookup",,,,,, 1)
```

The next example moves the **lookup** button to the position that precedes the **I_CANCEL** button. Depending on the button set orientation, it will be above or to the left of the **I_CANCEL** button.

```
xcall b_button(wndid, DSB_MODIFY, "lookup",,,,,, "I_CANCEL")
```


The next example moves the **lookup** button to the third position in the button set. Depending on the orientation of the button set, it will be either the third button from the top or the third button from the left.

```
xcall b_button(wndid, DSB_MODIFY, "lookup",,,,,, 3)
```

The next example moves the **lookup** button to the end of the button set. Depending on the orientation of the button set, it will either be at the far right or the bottom of the button set.

```
xcall b_button(wndid, DSB_MODIFY, "lookup",,,,,, "")
```

The next example sets the ToolTip text to “Enter vendor number”:

```
xcall b_button(wndid, DSB_MODIFY, "lookup",,,,,, "Enter vendor number")
```

B_BUTTONSET – Define the placement of a set of buttons on a window or tab set



```
xcall B_BUTTONSET (container_id, [placement], [per_row_col], [justification]
&                [, button_name])
```

Arguments

container_id

The ID of the window or tab set. **(n)**

placement

(optional) One of the following: **(n)**

DSB_BOTTOM The set of buttons will be placed horizontally along the bottom of the container with a maximum of *per_row_col* buttons per row. (default)

DSB_RIGHT The set of buttons will be placed vertically along the right side of the container with a maximum of *per_row_col* buttons per column.

per_row_col

(optional) The maximum number of buttons to place on a row if DSB_BOTTOM is specified, or on a column if DSB_RIGHT is specified. The default behavior is to display as many buttons as will fit and wrap to the next row or column. **(n)**

justification

(optional) One of the following: **(n)**

DSB_BEGIN The buttons will be left-justified if DSB_BOTTOM is specified, or positioned towards the top if DSB_RIGHT is specified. (default)

DSB_CENTER The buttons will be centered along the designated side.

DSB_END The buttons will be right-justified if DSB_BOTTOM is specified, or positioned towards the bottom if DSB_RIGHT is specified.

button_name

(optional) The name of the default button or “*NONE*” to clear explicitly set button default for the window. (The latter must be uppercase and enclosed in quotation marks.) **(a)**

Discussion

B_BUTTONSET defines the placement of a set of buttons within a window or tab set. (Use the L_BUTTONSET subroutine to define a button set on a list.)

If button set information is specified for a container that already has a button set, any new information will supersede the previous information, but unspecified arguments will not be modified.

Every button is contained within a set, and only one button set is allowed per window or tab set. Because of this, a call to B_BUTTONSET affects all buttons within a container.

If the E_STATE option D_RETURNBTN is set, the default button will be selected if you press RETURN, no matter what field you're on in the window. If *button_name* is not specified, the first button in the window is the default.

This routine has no effect on applications running on UNIX or OpenVMS or in Synergy .NET. It does, however, save button information on these platforms, so if the window is saved to a window library (by using U_SAVE) and later loaded from that library on a Windows platform in traditional Synergy, the button will be on the window (assuming ISAM-compatible platforms).

See also

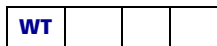
- ▶ [“Using the Button Subroutines” on page 5-2](#) for general information on buttons and button sets on windows, tab sets, and lists.
- ▶ [.BUTTON_SET on page 2-17](#) for more information on defining the placement of a window's button set in a script file.
- ▶ [L_BUTTONSET on page 9-62](#) for more information on defining a button set within a list.
- ▶ [IB_BUTTONSET on page 8-90](#) for more information on defining a button set when building a new input window at runtime.
- ▶ [DTS_BUTTONSET on page 13-17](#) for more information about specifying button placement on a tab set. (DTS_BUTTONSET is a subfunction of %TS_TABSET.)
- ▶ [E_STATE on page 7-41](#) for more information about the ENTER key selecting the default button.

Examples

The following example places the window's buttons on the right side of the window, centered, with at most three buttons per column.

```
xcall b_buttonset(wndid, DSB_RIGHT, 3, DSB_CENTER)
```

%B_DISABLE – Disable a button in a window or tab set



```
state = %B_DISABLE (container_id, button_name)
```

or

```
xcall B_DISABLE (container_id, button_name)
```

Return value

state

On Windows, the previous state of the button: (^VAL)

D_DISABLED Disabled.

D_ENABLED Enabled.

D_INPUTONLY Enabled only within input (the default state; see below).

On UNIX and OpenVMS, always returns false.

Arguments

container_id

The ID of the window or tab set that contains the button to disable. (**n**)

button_name

The name of the button to disable. (**a**)

Discussion

The %B_DISABLE function prevents a button from being enabled and, if the button is enabled, disables the button (which prevents the button from receiving focus). (Use %L_BUTTONSTATE to disable a button on a list.) By default, a button is enabled only when its window is being processed. (See [.BUTTON on page 2-13](#) for details and exceptions.) To restore this default behavior (which %B_DISABLE prevents), call %B_ENABLE with the *input_only* argument set to true.

If *container_id* is invalid or the window does not contain the specified button, an error occurs.

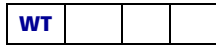
This routine has no effect on applications running on UNIX or OpenVMS or in Synergy .NET. It does, however, save button information on these platforms, so if the window is saved to a window library (by using U_SAVE) and later loaded from that library on a Windows platform in traditional Synergy, the button will be on the window (assuming ISAM-compatible platforms).

Examples

The following example disables the OK button in window **wndid**.

```
prev_state = %b_disable(wndid, "OK")
```

%B_ENABLE – Enable a button in a window or tab set



```
prev_state = %B_ENABLE (container_id, button_name[, input_only])
```

or

```
xcall B_ENABLE (container_id, button_name[, input_only])
```

Return value

prev_state

On Windows, the previous state of the button: (^VAL)

D_DISABLED Disabled.

D_ENABLED Enabled.

D_INPUTONLY Enabled only within input (the default state; see below).

On UNIX and OpenVMS, always returns false.

Arguments

container_id

The ID of the window or tab set that contains the button to enable. (**n**)

button_name

The name of the button to enable. (**a**)

input_only

(optional) The true/false flag that specifies whether the button is to be enabled only when being processed. This is the default state for all buttons. (**n**)

Discussion

The %B_ENABLE function enables a button in a window and either restores or prevents automatic button enabling and disabling. (Use %L_BUTTONSTATE to enable a button on a list.)

By default, a button is automatically enabled when the window that contains the button is processed; otherwise, it is automatically disabled. There is an exception: if a button is in a composite window, the button will be automatically enabled when the composite window is processed (even if the button is on a child window that is not active) and disabled otherwise.

If you omit *input_only* or pass it as false, %B_ENABLE prevents the button from being automatically disabled and, if the button is disabled, enables it.

Button Routines

%B_ENABLE

If you pass *input_only* as true, %B_ENABLE restores the default behavior, which, incidentally, enables the button if the window or tab set (or composite window) for the button is being processed.

If *container_id* is invalid or the window does not contain the specified button, an error occurs.

This routine has no effect on applications running on UNIX or OpenVMS or in Synergy .NET. It does, however, save button information on these platforms, so if the window is saved to a window library (by using U_SAVE) and later loaded from that library on a Windows platform in traditional Synergy, the button will be on the window (assuming ISAM-compatible platforms).

Examples

The following example enables the OK button in window **wndid**.

```
prev_state = %b_enable(wndid, "OK")
```

The next example enables the Exit button in window **inpid**, but enables it only when **inpid** is being processed.

```
prev_state = %b_enable(inpid, "Exit", TRUE)
```

%B_INFO – Return information about the buttons in a window



return = %B_INFO (*window_id*, *subfunction*, *args*)

Return value

return
A value that is specific to the subfunction. (^VAL)

Arguments

window_id
The ID of a window (see Discussion below). (n)

subfunction
One of the following: (n)

DSB_COUNT	Return the number of buttons in a window.
DSB_DEFAULT	Return the index of the default button.
DSB_ENABLED	Return the enabled state of a button.
DSB_INDEX	Return the index (position in button set) of a button.
DSB_JUSTIFICATION	Return the justification setting for a window’s button set.
DSB_METHOD	Return method information for a button.
DSB_NAME	Return a button’s name by passing its index.
DSB_PER_ROW_COL	Return the maximum number of buttons that can be displayed in a button set row or column for a window.
DSB_PLACEMENT	Return the placement (at the bottom of the window or along the right side) for a window’s button set.
DSB_SELECTCHAR	Return the quick-select character for a button.
DSB_TYPE	Return the button face type (text or graphic) for a button.
DSB_TOOLTIP	Return the text and length for a button ToolTip.

args
Arguments specific to *subfunction*.

Discussion

On Windows and UNIX, the %B_INFO function enables you to return information on a window's buttons. (On UNIX, buttons are not displayed, but they may be defined for a window.) See the individual subfunctions for details.

Note the following:

- ▶ For lists, pass the ID of the list's containing window as *window_id*. (The ID of a list's containing window is returned by the D_LCTRID subfunction for L_STATUS.)
- ▶ For tab sets, pass the window ID for the tab set.
- ▶ For composite windows, passing the composite container window's ID instructs %B_INFO to return information about the buttons for that window only. For information on buttons for a child window, pass ID of the child window as *window_id*.

On OpenVMS, this function ignores all parameters and returns a value of 0.

DSB_COUNT

```
count = %B_INFO (window_id, DSB_COUNT)
```

Return value

count

The number of buttons in the specified window or a value of 0 if the window has no buttons. (^**VAL**)

Arguments

window_id

The ID of a window. (See the %B_INFO [Discussion on page 5-20.](#)) (**n**)

Discussion

The DSB_COUNT subfunction returns the number of buttons in the window specified by *window_id*. If the window has no buttons, DSB_COUNT returns a value of 0.

Examples

The following example returns the number of buttons in **wndid**:

```
btn_num = %b_info(wndid, DSB_COUNT)
```

Button Routines

%B_INFO

DSB_DEFAULT

index = %B_INFO(*window_id*, DSB_DEFAULT)

Return value

index

The index of the default button, or a value of 0 if no default button has been specified for the window. (^**VAL**)

Arguments

window_id

The ID of a window. (See the %B_INFO [Discussion on page 5-20.](#)) (**n**)

Discussion

The DSB_DEFAULT subfunction tells you whether the specified window has a default button (if it doesn't, DSB_DEFAULT returns a value of 0) and returns the index for the default button if there is one. The index (base 1) specifies the numeric position (from left to right or from top to bottom) of the button in relation to other buttons in the window.

Examples

The following example returns the index for the default button for **wndid**:

```
btn_dflt = %b_info(wndid, DSB_DEFAULT)
```

DSB_ENABLED

state = %B_INFO(*window_id*, DSB_ENABLED, *button*)

Return value

state

One of the following: (^VAL)

TRUE The button is enabled.

FALSE The button is disabled.

DSB_INPUT_ONLY The button is enabled only when its window or composite container is being processed.

Arguments

window_id

The ID of a window. (See the %B_INFO [Discussion on page 5-20.](#)) (**n**)

button

The button name (if the passed value is alpha) or the index for the button (if passed value is numeric). (**a** or **n**)

Discussion

The DSB_ENABLED subfunction returns the enabled state of the button specified with *button*.

You can specify a button by name (case-insensitive) or by numeric position. If you pass a number for *button*, it is interpreted as an index (base 1) that specifies the position of the button in relation to the other buttons for the window (from left to right, or from top to bottom). Note that a fatal error will occur if you pass a number that is less than 1, a number that is greater than the number of buttons in the window, or an alpha value that is not the name of a button in the window.

For information on the input-only state, see [%B_ENABLE on page 5-17.](#)

Examples

In the following example, %B_INFO returns the enabled state of the first button in the window's button set:

```
btn_state = %b_info(wndid, DSB_ENABLED, 1)
```

Button Routines

%B_INFO

DSB_INDEX

index = %B_INFO (*window_id*, DSB_INDEX, *name*)

Return value

index

The index of the button specified by *name*, or 0 if the specified button is not found. (**^VAL**)

Arguments

window_id

The ID of a window. (See the %B_INFO [Discussion on page 5-20.](#)) (**n**)

name

The name (case-insensitive) of a button on the window specified by *window_id*. (**a**)

Discussion

The DSB_INDEX subfunction returns the index of the button specified by *name*, or it returns a value of 0 if there is no such button. The index (base 1) specifies the numeric position (from left to right or from top to bottom) of the button in relation to other buttons in the window.

Examples

In the following example, %B_INFO returns the index for a button named lookup:

```
btn_index = %b_info(widid, DSB_INDEX, "lookup")
```

DSB_JUSTIFICATION

justification = %B_INFO(*window_id*, DSB_JUSTIFICATION)

Return value

justification

One of the following: (^VAL)

DSB_BEGIN Buttons are left-justified if the button set is placed at the bottom of the window. If the button set is placed vertically along the right side of the window, buttons are top-justified.

DSB_CENTER Buttons are centered along the bottom or along the right side of the window.

DSB_END Buttons are right-justified if the button set is placed at the bottom of the window. If the button set is placed vertically along the right side of the window, buttons are bottom-justified.

Arguments

window_id

The ID of a window. (See the %B_INFO [Discussion on page 5-20.](#)) (n)

Discussion

The DSB_JUSTIFICATION subfunction returns the justification setting for a window's button set. For information on the justification setting for button sets, see [.BUTTON_SET on page 2-17.](#)

See also

- ▶ [B_BUTTONSET on page 5-14.](#)
- ▶ [L_BUTTONSET on page 9-62.](#)

Examples

In the following example, %B_INFO returns the justification setting for the window's button set:

```
btn_just = %B_INFO(wndid, DSB_JUSTIFICATION)
```

Button Routines

%B_INFO

DSB_METHOD

address = %B_INFO (*window_id*, DSB_METHOD, *button*, [*method_name*][, *elb_path*])

or

xcall B_INFO (*window_id*, DSB_METHOD, *button*, [*method_name*][, *elb_path*])

Return value

address

The address of the button's method, or 0 if the button has no method. (^VAL)

Arguments

window_id

The ID of a window. (See %B_INFO [Discussion on page 5-20.](#)) (n)

button

The button name (if passed as alpha) or the button index (if passed as numeric). (a or n)

method_name

(optional) Returned name of the method (or blank if there is no method for the button). (a)

elb_path

(optional) Returned path for the method's ELB (or blank if there is no method for the button or if there is no ELB specified for the method). (a)

Discussion

The DSB_METHOD subfunction tells you if a button has a method and returns the address for the method if there is one. If there is no method, DSB_METHOD returns 0. Optionally, it returns the name (*method_name*) of the method and the path (*elb_path*) for the method's ELB.

You can specify a button by name (case-insensitive) or by numeric position. If you pass a number for *button*, it is interpreted as an index (base 1) that specifies the position of the button in relation to the other buttons for the window (from left to right, or from top to bottom). A fatal error will occur if you pass a number that is less than 1, a number that is greater than the number of buttons in the window, or an alpha value that is not the name of a button in the window.

See also

[.BUTTON on page 2-13.](#)

Examples

In this example, %B_INFO returns the button method address for the button specified by **btn_index** and retrieves the button's name (**meth_name**), as well as the path (**elb_path**) to the ELB that contains the button method:

```
meth_addr = %B_INFO(wndid, DSB_METHOD, btn_index, meth_name, elb_path)
```

DSB_NAME

```
success = %B_INFO (window_id, DSB_NAME, index, name)
```

or

```
xcall B_INFO (window_id, DSB_NAME, index, name)
```

Return value

success

True if *index* corresponds to a button; false if there is no button for *index*. (^VAL)

Arguments

window_id

The ID of a window. (See the %B_INFO [Discussion on page 5-20.](#)) (n)

index

The index of the button to access. This is an index (base 1) that specifies the numeric position (from left to right or from top to bottom) of the button in relation to other buttons in the window. (If *index* is non-numeric, a fatal Toolkit error will occur.) (n)

name

The returned name of the button specified by *index*. If no button is found for *index*, *name* is filled with spaces. (a)

Discussion

The DSB_NAME subfunction indicates whether there is a button that corresponds to a specified index and, if there is, retrieves the name of the button:

- ▶ If there is a button for *index*, DSB_NAME returns true and passes the name of the button in *name*.
- ▶ If there is no button for *index*, DSB_NAME returns false and fills *name* with spaces.

Examples

In the following example, %B_INFO tests to for the existence of a button in the window and, if there is, retrieves the name of the first button in the button set:

```
btn_index = 1
if (%B_INFO(wndid, DSB_NAME, btn_index, btn_name))
.
.
.
```


DSB_PER_ROW_COL

number = %B_INFO(*window_id*, DSB_PER_ROW_COL)

Return value

number

The maximum number of buttons per row or column, or 0 if no maximum has been set for the specified window's button set. (^VAL)

Arguments

window_id

The ID of a window. (See the %B_INFO [Discussion on page 5-20.](#)) (n)

Discussion

The DSB_PER_ROW_COL subfunction returns the maximum number of buttons that will be displayed in a row or column for the specified window's button set. If no maximum has been set for the button set, this function returns zero. For information on the maximum setting for rows and for columns, see [.BUTTON_SET on page 2-17.](#)

Examples

In the following example, %B_INFO returns the maximum number of buttons that will be displayed in a row (if the button set for **wndid** is positioned at the bottom of the window) or in a column (if the button set for **wndid** is positioned vertically along the right side of the window):

```
num_btns = %B_INFO(wndid, DSB_PER_ROW_COL)
```

DSB_PLACEMENT

placement = %B_INFO(*window_id*, DSB_PLACEMENT)

Return value

placement

One of the following: (^VAL)

DSB_BOTTOM Buttons are placed horizontally along the bottom of container.

DSB_RIGHT Buttons are placed vertically along the right side of the container.

Arguments

window_id

The ID of a window. (See the %B_INFO [Discussion on page 5-20.](#)) (n)

Discussion

The DSB_PLACEMENT subfunction returns the placement setting for a window's button set, telling you whether the buttons are placed horizontally along the bottom of the window or vertically along the right side of the window.

See also

- ▶ [“Defining button placement and the default button” on page 5-3](#) for information on setting button placement.
- ▶ [“.BUTTON_SET” on page 2-17](#) for information on setting button placement for a window.
- ▶ [B_BUTTONSET on page 5-14](#) and [L_BUTTONSET on page 9-62.](#)

Examples

In the following example, %B_INFO returns the placement setting for the window specified by **wndid**:

```
btn_plc = %b_info(wndid, DSB_PLACEMENT)
```

DSB_SELECTCHAR

value = %B_INFO(*window_id*, DSB_SELECTCHAR, *button*[, *char*])

or

xcall B_INFO(*window_id*, DSB_SELECTCHAR, *button*[, *char*])

Return value

value

The ASCII value of the quick-select character for the button (specified with *button*), or 0 if the button has no quick-select character. (^VAL)

Arguments

window_id

The ID of a window. (See the %B_INFO [Discussion on page 5-20](#).) (n)

button

The button name (if passed as alpha) or the button index (if passed as numeric). (a or n)

char

(optional) The returned quick-select character for the specified button, or %char(0) if the button has no quick-select character. (a)

Discussion

The DSB_SELECTCHAR function tells you if a button has a quick-select character and returns the ASCII value (*value*) of the character, if there is one. If there is no quick-select character, it returns 0. It optionally returns the quick-select character itself in *char*.

You can specify a button by name (case-insensitive) or by numeric position. If you pass a number for *button*, it is interpreted as an index (base 1) that specifies the position of the button in relation to the other buttons for the window (from left to right, or from top to bottom). Note that a fatal error will occur if you pass a number that is less than 1, a number that is greater than the number of buttons in the window, or an alpha value that is not the name of a button in the window.

See also

- ▶ [.BUTTON](#) on page 2-13.
- ▶ [B_BUTTONSET](#) on page 5-14 and [L_BUTTONSET](#) on page 9-62.

Button Routines

%B_INFO

Examples

In the following example, %B_INFO tests for a select character for the first button in the window's button set and retrieves the quick-select character (in **sel_char**) if there is one:

```
btn_index = 1
if (%B_INFO(wndid, DSB_SELECTCHAR, btn_index, sel_char)) then
.
.
.
```

DSB_TYPE

```
type = %B_INFO(window_id, DSB_TYPE, button[, type_data])
```

or

```
xcall B_INFO(window_id, DSB_TYPE, button[, type_data])
```

Return value

type

One of the following: (^VAL)

DSB_TEXT Text.

DSB_BITMAP Graphic.

Arguments

window_id

The ID of a window. (See the %B_INFO [Discussion on page 5-20](#).) (**n**)

button

The button name (if passed as alpha) or the button index (if passed as numeric). (**a** or **n**)

type_data

(optional) The returned type-specific data, which is the text for the button face (if the type is DSB_TEXT) or the image path (if the type is DSB_BITMAP). (**a**)

Discussion

The DSB_TYPE subfunction returns the type of the button face (text or graphic) and optionally retrieves either the text for the button or the path for the button's graphic.

You can specify a button by name (case-insensitive) or by numeric position. If you pass a number for *button*, it is interpreted as an index (base 1) that specifies the position of the button in relation to the other buttons for the window (from left to right, or from top to bottom). A fatal error will occur if you pass a number that is less than 1, a number that is greater than the number of buttons in the window, or an alpha value that is not the name of a button in the window.

See also

- ▶ [“Toolkit button graphics” on page 2-15](#) for information on supported graphic formats.
- ▶ [DSB_ADD on page 5-5](#) for B_BUTTON and [DSB_ADD on page 9-56](#) for L_BUTTON.

Examples

In the following example, %B_INFO returns the button type for the button named “lookup” and retrieves either the button's text or the path for the button's graphic in **btn_data**:

```
btn_type = %b_info(wndid, DSB_TYPE, "lookup", btn_data)
```

DSB_TOOLTIP

```
length = %B_INFO(window_id, DSB_TOOLTIP, button[, text])
```

or

```
xcall B_INFO(window_id, DSB_TOOLTIP, button[, text])
```

Return value

length

The length of the button's ToolTip text, or 0 if the button has no ToolTip. (^VAL)

Arguments

window_id

The ID of a window. (See the %B_INFO [Discussion on page 5-20.](#)) (n)

button

The button name (if passed as alpha) or the button index (if passed as numeric). (a or n)

text

(optional) The returned text of the button's ToolTip. (a)

Discussion

The DSB_TOOLTIP subfunction enables you to determine if a button on a window or a list has a ToolTip. It returns the length of a button's ToolTip text (or zero if the button has no ToolTip). It optionally returns the text of the button's ToolTip in *text* (or all blanks if the button has no ToolTip).

You can specify a button by name (case-insensitive) or by numeric position. If you pass a number for *button*, it is interpreted as an index (base 1) that specifies the position of the button in relation to the other buttons for the window (from left to right, or from top to bottom). Note that a fatal error will occur if you pass a number that is less than 1, a number that is greater than the number of buttons in the window, or an alpha value that is not the name of a button in the window.

Note the following for *text*:

- ▶ If *text* is not large enough to contain the ToolTip text for the button, that text shall be truncated to the size of *text*. (*Length* will still be returned with the full length of the ToolTip text.)
- ▶ If *text* is larger than the ToolTip text for the button, the remainder of *text* will contain spaces.
- ▶ If the button has no ToolTip, *text* is returned as all blanks.

Examples

In the following example, %B_INFO returns the ToolTip text (**ttip_text**) and the length of the ToolTip text (**ttip_len**) for a button named "lookup":

```
ttip_len= %B_INFO(wndid, DSB_TOOLTIP, "lookup", ttip_text)
```

6

Composite Window Routines

Creating and Processing Composite Windows 6-2

Describes composite windows, how they work, and how you can create and use them.

%C_CONTAINER – Perform composite window operations 6-10

C_PROCESS – Process a composite window 6-42

The Composite Window Processing Methods 6-46

C_METHAX – Process a child ActiveX window 6-47

C_METHCTR – Process a child composite window 6-49

C_METHINP – Process a child input window 6-51

C_METHLST – Process a child list 6-54

C_METHNET – Process a child .NET form window 6-57

C_METHNOP – Prevent a child from gaining focus 6-59

C_METHSEL – Process a child selection window 6-61

C_METHTS – Process a child tabbed dialog 6-63

C_METHTXT – Process a child text window 6-65

Creating and Processing Composite Windows

The composite window routines and methods enable you to create and process composite windows, which are windows that consist of a parent window and child windows and/or lists. For example, you could create a composite window with input fields at the top for header information, a table in the middle for line item entry, a multi-line “notes” field that supports rich text editing through an ActiveX control, buttons, and so forth. A composite window functions and appears to the user as a single window.

The parent window in a composite window is the *composite container window*. Windows or lists within a composite container window are *child windows* or *child lists*. A child window or list can be any type of UI Toolkit window or list, including ActiveX windows, tab set windows, ActiveX Toolkit lists, .NET form windows, and other composite windows. (A composite window can have a maximum of 2047 children.) Note the following:

- ▶ On Windows, child windows and lists are wholly contained within the composite container window.
- ▶ On UNIX and OpenVMS, the display area for a child window or list is circumscribed if necessary to make it fit within the display area of the composite container window. Subsequent moving or resizing, however, may cause a child to exceed the boundaries of the container.

A composite window not only appears to users as a single, unified window, it is also treated in many ways as a single window by the application. Calls you make that affect the container affect the composite window as a whole:

- ▶ When a container is placed, all placed child windows and lists become visible.
- ▶ When a container is moved, all child windows and lists are moved with it.
- ▶ When a container is removed, all child windows and lists become invisible. (When a container is deleted, though, Toolkit automatically removes placed child windows and lists. It does not delete them.)

To create a composite window,

1. Create the composite container window with the DC_CREATE subfunction for %C_CONTAINER. See [DC_CREATE](#) on page 6-32.
2. Add child windows or lists or both to the container window. As you do this you can set properties for the child windows and lists, including placement, tabbing position, a menu column to be loaded when the child is processed, and the name of a routine to be used in place of the default method for the child. See [DC_ADD](#) on page 6-21.

Placing, removing, and deleting windows

When you place a child, the child remains invisible until the composite container window is placed. When you place a composite window (by placing its container), only child windows and lists that have already been placed will appear on the window. Note that a container and its child windows and lists don't have to be logged in the same environment. When a container is placed, placed windows and lists will be displayed, even if they are logged in different environments.

- ▶ To instruct Toolkit to place a child, either pass both the *row* and *col* arguments in the DC_ADD call that adds the child to the composite window, or use the D_PLACE operation for U_WINDOW or L_PLACE to place the child after it has been added to a composite window.
- ▶ To place the composite container window, pass the window ID for the container to the D_PLACE operation for U_WINDOW.

To instruct Toolkit to remove a child from the screen, use the D_REMOVE operation for U_WINDOW or use L_REMOVE. Note the following:

- ▶ If the composite container window has not yet been placed, the remove operation will have no visible effect because the child is already invisible. However, the child will not become visible when the container is placed.
- ▶ Removing a child with D_REMOVE or L_REMOVE removes the child from the display but does not remove (disassociate) the child from the container. The DC_ADD call that added it to the container still stands, and the child retains its position in the tabbing order for the composite window.
- ▶ To remove a child from the display *and* from the container, use the DC_REMOVE subfunction for %C_CONTAINER.

To remove a composite window from the screen, pass the composite container window ID to the D_REMOVE operation for U_WINDOW. Removing the composite container window makes all child windows and lists invisible, no matter which environments they are logged in.

To delete a composite window, delete its composite container window. To delete a composite container window or a child, leave the environment in which it was logged or use the D_DELETE operation for U_WINDOW. Note, however, that deleting the container removes all child windows and lists but does not delete them.

Processing composite windows

Once you have created and placed a composite window, you can process it by calling C_PROCESS, which in turn calls methods that process the child windows and lists. Toolkit includes default processing methods, but you can instruct C_PROCESS to call your own custom methods instead. See [C_PROCESS on page 6-42](#) and [“The Composite Window Processing Methods” on page 6-46](#).

Controlling tabbing, focus, and input context

When a composite window is being processed, tabbing moves through placed child windows and lists according to the composite window's tabbing order, which by default is the order they were added to the composite window. Then, after child windows and lists, tabbing moves through any enabled buttons added to the composite container window.



Every child window and list for a composite window has a tabbing index. This number specifies the child's position in the tabbing order. Tabbing indices are base 1, so the first child for a composite window has a tabbing index of 1. To get the tabbing index of a child, use the DC_CHILDINDEX subfunction for %C_CONTAINER. (Note that buttons on the composite container window also have tabbing indices, which are also base 1.)

You can do the following to control the tabbing order or activate a specific child or button:

- ▶ To control a child's place in the tabbing order, add child windows and lists in the order that you want users to tab through them. (When you add a child, it's added to the end of the tabbing index by default.) You can, however, use the *tabindex* argument for DC_ADD to insert a child into a different position when you add it.
- ▶ To change the active child, use one of the C_ menu entries (see the C_PROCESS [Discussion on page 6-42](#)) or use the DC_CONTEXT subfunction for %C_CONTAINER. (You can also use DC_CONTEXT to determine which child is active in a composite window.)
- ▶ To change focus to a specified button on the composite container window, use the DC_BTNFOCUS subfunction for %C_CONTAINER. (You can also use this function to determine whether a button on the composite container window has focus.)

Tabbing within child windows and lists

When processing a child window or list with L_INPUT, L_INPFLD, S_SELECT, T_EDIT, T_VIEW, or U_CHR, pressing TAB (or otherwise signaling the C_NEXT menu entry) moves focus to the first button for the window. Subsequent tabbing advances through any remaining buttons for the window and then, if the window is on a tab, advances through buttons on the tab set. Finally, tabbing causes the input routine to return with **g_select** set to true and **g_entnam** set to "C_NEXT", which instructs C_PROCESS to move focus to the next child window. SHIFT+TAB works the same way, but in reverse. It returns **g_select** set to true, but it sets **g_entnam** to "C_PREV" to move to the preceding child.

The above also applies to I_INPUT and I_INPFLD, except that

- ▶ these include input fields in the tabbing order.
- ▶ signaling "C_NEXT" or "C_PREV" while in the tabbing order of the window itself (which includes buttons for the window and, if the button is on a tab, buttons for the tab set) does not move from field to field or button to button. Instead, these menu entries are returned to C_PROCESS. To move from among fields and buttons in the window (and tab set if applicable), use the I_NEXTCTL and I_PREVCTL menu entries.

In any case, users won't be able to tab out of a window or list if the child processing method changes the value of **g_entnamtab** or **g_entnamstab** (defined in **tkctl.def**), if the ActiveX control for a child ActiveX window processes TAB and SHIFT+TAB as an accelerator without returning it as a keystroke (see “[ActiveX controls and tabbing](#)” below).

Additionally, Toolkit includes several routines that enable you to control the context of a child window or list. All the following except **I_NEXT** and **L_NEXT** determine context based on the reason the child received focus. This enables child processing methods (the default methods and any methods you create) to provide an interface where users can smoothly tab through and click into composite windows.

- ▶ To control context for a child composite window, use the **DC_CTRCONTEXT** subfunction for **%C_CONTAINER**. See [DC_CTRCONTEXT](#) on page 6-33 for more information.
- ▶ To control context for a window or list on a child tab set, use the **DTS_CTRCONTEXT** subfunction for **%TS_TABSET**. See [DTS_CTRCONTEXT](#) on page 13-36 for more information.
- ▶ To control context for a child input window, use **I_NEXT** or **I_CTRCONTEXT**. See [I_NEXT](#) on page 8-65 or [I_CTRCONTEXT](#) on page 8-9 for more information.
- ▶ To control context for a child list, use **L_NEXT** or **L_CTRCONTEXT**. See [L_NEXT](#) on page 9-107 or [L_CTRCONTEXT](#) on page 9-80 for more information.
- ▶ To control context for other types of windows, use **U_CTRCONTEXT**. See [U_CTRCONTEXT](#) on page 16-24 for more information.

Note that when processing tab sets, CTRL+TAB and CTRL+SHIFT+TAB apply to the innermost tab set. For example, if a tab set contains a composite window, and that composite window in turn contains a tab set, these key combinations apply to the inner tab set when the inner tab is being processed. When Toolkit is processing any other window in the composite window on the tab, they apply to the outer tab set. See [%TS_TABSET](#) on page 13-7 for more information on navigating tab sets.

For Windows, note the following:

- ▶ If an enabled tab on an inactive child window is clicked, Toolkit activates the child window and moves focus to the tab (instead of signaling the **TS_TAB n** menu entry). It then signals the “**C_RESET**” menu entry (see the **C_PROCESS** [Discussion](#) on page 6-42).
- ▶ If a disabled tab on an inactive child window is clicked, Toolkit activates the child window, moves focus to the default tab, and then signals the “**C_RESET**” menu entry (instead of signaling the **TS_TAB n** menu entry).
- ▶ For selection windows that are part of a composite window, TAB and SHIFT+TAB move focus to the next or previous child (unless TAB or SHIFT+TAB is a shortcut on a placed menu column). TAB instructs **S_SELECT** to signal “**C_NEXT**”, and SHIFT+TAB instructs **S_SELECT** to signal “**C_PREV**”.

ActiveX controls and tabbing

There are a couple of situations in which tabbing may not automatically work seamlessly when a composite window includes an ActiveX control. The mechanisms for working around these issues are illustrated in [ActivexChildTabbing.zip](#), available from Synergy CodeExchange in the Resource Center on the Synergex web site.

- ▶ If the ActiveX control uses TAB or SHIFT+TAB internally (i.e., processes them as accelerators), users might not be able to tab out of the child unless you create a mechanism for this. To do this, code the control to invoke a method that signals the “C_NEXT” or “C_PREV” menu entry at the point a user would tab out of the control. (For information on “C_NEXT” and “C_PREV,” see the [C_PROCESS Discussion on page 6-42](#).) Note that often, however, this isn’t necessary. If a control uses TAB and SHIFT+TAB internally, but returns them as unprocessed when it reaches the end of its own tabbing order, TAB and SHIFT+TAB do tab out of the child. This is usually how controls created with VB6 work.
- ▶ If the ActiveX control consists of more than one control and has its own tabbing order, focus within the ActiveX control might not be where you want it when the user tabs to the ActiveX control. For example, if the ActiveX control consists of two edit controls, focus within the ActiveX control will generally be on the edit control that last had focus. To work around this, create a method for the child (a composite window processing method) that in turn uses a method in the ActiveX control to set focus.

Buttons in composite windows

To add buttons to a composite window, you can use `B_BUTTON` to add buttons to the container, and you can use `B_BUTTON` or `L_BUTTON` to add buttons to a child window or list. Note the following:

- ▶ Buttons on a composite window are enabled when the composite window is being processed and disabled when it’s not. This is true even of buttons in child windows and lists that are not active. See the [.BUTTON Discussion on page 2-14](#) for details.
- ▶ You can create a panel of buttons that can be placed anywhere in the client area of a composite window. To do this, create an input window with buttons but with an empty input set (see [.SET on page 2-97](#) or [IB_SET on page 8-109](#)). You can then place the input window wherever you want the button panel to appear in the composite window’s client area.

The default button

The following rules determine which button is the default for a composite window as a whole. Note that for usability and aesthetics, we recommend that you specify only one default button anywhere within a composite window—in other words, only one child window or list should have a default button (unless a default button is set for the composite container window). This prevents Toolkit from switching the default as context changes.

1. If the window being processed specifies a default button, and that button is enabled, that button becomes the default button for the composite window.
2. Otherwise, if the container for the active window specifies an enabled default button, that button becomes the default. If the container doesn't specify an enabled default button, but is itself contained, the default button for its container shall be the default, and so forth.
3. If no default button can be determined by the above, the default for another descendant window of a common container may become the default for the composite window. This, of course, will only come into play if there are several layers of containment (composite windows within composite windows). In this case, Toolkit starts with the innermost child and, if it is unable to establish a default from that, moves to its container, its container's container, and so forth.
4. If none of the above results in a default (i.e., if no default button is specified on any window with a common ancestor), the first button for the composite window becomes the default. The first button is determined according to the same hierarchy discussed in the previous rules. If the active window has a button, the first button on that window will be the default. If the active window doesn't have a button, the first button on the active window's container will be the default. If the parent has no button, the first button on the grandparent becomes the default, and so on. If none of these have buttons, the first button on a window that shares a common ancestor to the current window will be the default.

See the `BUTTON_SET` [Discussion on page 2-17](#) for more information on default buttons.

ALT+key button activation

The following rules determine which button, if any, will be activated by an ALT+key combination.



For usability and aesthetics, we recommend that only one button for each ALT+key combination be specified anywhere within a container. This prevents Toolkit from switching key associations as context changes.

1. If the window being processed contains an enabled button that specifies the entered key combination, Toolkit activates that button.
2. Otherwise, if the container for the active window has an enabled button that specifies the entered combination, Toolkit activates that button. If the container doesn't have an enabled button for the key combination, but is itself contained, Toolkit uses the enabled button for the key combination on the container's container, and so forth.

3. If no button is activated as a result of the above, Toolkit may activate a button that specifies the key combination in another descendant window of a common container. In this case, Toolkit starts with the innermost child and, if no button specifies the key combination in that window, moves to its container, its container's container, and so forth.

Positioning composite windows, child windows, and lists

To position a composite window (as a whole), position the composite container window by using one of the following:

- ▶ `D_PLACE` operation for `U_WINDOW` when placing the container
- ▶ `D_MOVE` operation for `U_WINDOW` to move the container after it has been placed

To control the placement of a child window or list, use one of the following. In all cases, settings are relative to the client area of the immediate parent.

- ▶ The `row` and `col` arguments for `DC_ADD` (a subfunction for `%C_CONTAINER`) when adding a child to the composite window
- ▶ `.PLACEMENT` when defining a child window or the window for a child list. This will work only if you use the `D_MOVE` operation for `U_WINDOW` to place the child.
- ▶ `L_CREATE` when creating a child list. This will work only if the `row` and `col` arguments are omitted in the `DC_ADD` call and the `row` and `column` arguments are omitted when using `L_PLACE` to place the child.
- ▶ The `D_PLACE` operation for `U_WINDOW` or `L_PLACE` when placing a child
- ▶ The `D_MOVE` operation for `U_WINDOW` or the `WP_POSITION` function for `W_PROC` to move a child after it has been placed

Getting composite window information

The `%C_CONTAINER` function includes several subfunctions that enable you to get child and composite container window information. With `%C_CONTAINER` you can

- ▶ determine how many child windows and lists are associated with a composite container window (`DC_NUMCHILD`).
- ▶ determine if a child window or list is active (`DC_CONTEXT`).
- ▶ determine the tabbing index for a specified child window or list (`DC_CHILDINDEX`).
- ▶ determine if a child with a specified tabbing index exists, and/or get the ID and type of the child (`DC_CHILD`).
- ▶ determine if a window or list is a child (`DC_CONTAINER`, `DC_CHILDINDEX`).
- ▶ get the ID of the associated container (`DC_CONTAINER`).

- ▶ determine if focus is on a button for the container (not a button for a child window or list) and, if so, which button ([DC_BTNFOCUS](#)).
- ▶ get the address of a method routine for a child window/event pair ([DC_GETEVENT](#)).

Additionally, you can

- ▶ find out if a window is a composite container window by using the `D_WINDOW_TYPE` subfunction for `%E_INFO`.
- ▶ get the method set ID for a child by using the `D_GETEVENTS` subfunction for `%U_WNDEVENTS` or by using the `D_LGETEVENTS` flag for `L_STATUS`.
- ▶ get the address of a method routine for a given event in a method set by using the `D_GETEVENT` subfunction for `%U_WNDEVENTS`.
- ▶ get the ID of the container window for a window or list that's part of a composite window on a tab set by using the `DTS_TABSET` subfunction for `%TS_TABSET`.

Event methods

When a window or list is added to a composite window (even if it's part of tab set that is added to a composite window), its `%UWNDEVENTS_METHOD` set is replaced by one that handles mouse clicks in composite windows. The previously registered method set, however, is registered as a set of extensions to the new method set if there is a difference between the two. See the [DC_EVENT Discussion on page 6-37](#) for information on the default behavior and for information on replacing it. Note the following:

- ▶ You can use the `DC_EVENT` subfunction for `%C_CONTAINER` to replace these extensions.
- ▶ You can use the `D_GETEVENTS` subfunction for `%U_WNDEVENTS` to determine if a method set has been registered for a UI Toolkit window and, if it has, what the ID for the method is.
- ▶ You can use the `D_GETEVENT` subfunction for `%U_WNDEVENTS` to retrieve the address of a method routine for the specified event (*event*) in the specified method set (*method_set*).

When a child is removed from its container, the `%UWNDEVENTS_METHOD` set for the child will remain as it was before the `DC_REMOVE` call, but the click event will no longer alter the container's context.

%C_CONTAINER – Perform composite window operations



value = %C_CONTAINER (*subfunction*, *args*)

or

xcall C_CONTAINER (*subfunction*, *args*)

Return value

value

A value specific to the subfunction.

Arguments

subfunction

One of the following: (n)

DC_ADD

Add a child window or list to a composite window.

DC_BTNFOCUS

Determine whether a button on a composite window has focus and return the tabbing index for the button and/or change focus to a specified button. (Windows only)

DC_CHILD

Determine whether a child exists for a given tabbing index and, if so, return the type and ID of the child.

DC_CHILDINDEX

Determine whether a window or list is a child and, if it is, return the tabbing index for the child.

DC_CONTAINER

Return the ID of the container window associated with a child or, if the window or list is not a child, indicate this by returning zero.

DC_CONTEXT

Use a tabbing index to change the active child for a composite window, and indicate which child is active by returning the tabbing index for that child.

DC_CREATE

Create a composite container window.

DC_CTRCONTEXT

Set the active child or button based on a code that indicates the reason the container received focus.

DC_ENABLE

Determine if a child window or list is enabled or disabled and/or enable or disable the child window or list.

DC_EVENT

Register an extension to the standard event methods for a child window or list.

DC_GETEVENT

Return the address for a currently registered extension for an event method for a child window or list.

DC_NUMCHILD

Return the number of children (windows and lists) associated with a composite container window.

DC_REMOVE

Remove a child window or list from a composite window.

args

One or more arguments, depending on *subfunction*.

Discussion

The %C_CONTAINER function enables you to create a composite container window, add child windows and lists, set options, and get information about options for the composite window (button focus, input context for a child input window, and so forth). See the individual subfunctions for details.

Examples

The following example creates and processes a composite window that includes a list and an input window to display and accept contact information. It also uses an ActiveX control to display an image and another for text editing. Note that the window script file and repository schema are included at the end of this example.

```
main composite
;
; Example of composite window usage
;

.include "WND:tools.def"
.align
stack record
    id_col      ,i4      ;ID of the general menu column

proc
    xcall u_start("composite", 1)
    xcall m_ldcol(id_col,, "c_general",,,, D_GLOBAL)
    xcall e_sct("Composite window example", D_HEADER)
    do
        xcall composite_example
    until (g_select)
    xcall u_finish
.end

subroutine composite_example
;
; Description: XCALLable example of composite window processing
;
```

Composite Window Routines

%C_CONTAINER

```
; Arguments:   None
;
.include "DBLDIR:windows.def"
.include "WND:tools.def"
.align
stack record
    chan_contacts    ,i4 ;Channel for updating the contacts database
    chan_icontacts   ,i4 ;Channel for loading the contacts
    id_container     ,i4 ;ID of the container
    id_listwnd       ,i4 ;ID of the list input window
    id_list          ,i4 ;ID of the list
    id_selcol        ,i4 ;ID of a selection column for non-windows
    id_contact       ,i4 ;ID of the contact input window
    id_inpcol        ,i4 ;ID of an input column for non-windows
    id_text          ,i4 ;ID of a text window
    id_editcol       ,i4 ;ID of an edit column for non-windows
    id_image         ,i4 ;ID of the ActiveX image window
    id_imgbtn        ,i4 ;ID of the "Image..." button window
    request          ,i4 ;List request
    read_one         ,i4 ;Did we read a record?
    err              ,i4 ;Error encountered

.include "contact" repository, record="list_contact"
.include "contact" repository, record="inp_contact"

.define D_KEYSIZ ,(^size(list_contact.lastname) +
^size(list_contact.firstname))

proc
    xcall e_enter
    xcall e_state(D_RETURNBTN)
    xcall e_method(D_METH_ENTRST, "entrst")
    call do_setup
    do
        call do_process
    until (g_select)
    xcall e_exit
    xreturn

;
; do_setup - create and place windows
;
do_setup,
    ;Connect to the database.
    xcall u_open(chan_contacts, "U:I", "contacts",,, err)
    if (err)
        begin
            if (err == $ERR_FNF) then
```

```

        begin
            xcall isamc("contacts", ^size(list_contact), 1,
                & "start=1,length="+%string(D_KEYSIZ))
            goto do_setup
        end
    else
        xcall u_abort("Unable to open Contacts database,
            & error = " + %string(err))
    end

clear read_one ;No file context yet

;Open again read-only for loading.
xcall u_open(chan_icontacts, "I:I", "contacts",,, err)

;Create the container and add the desired buttons.
id_container = %c_container(DC_CREATE, "", g_bdysiz,
    & %w_info(WIF_SCOLS))
xcall b_button(id_container, DSB_ADD, "O_ADD", DSB_TEXT, "Add",,,
"a")
xcall b_button(id_container, DSB_ADD, "O_DEL", DSB_TEXT, "Delete"
& ,,, "d")
xcall b_button(id_container, DSB_ADD, "O_EXIT", DSB_TEXT, "Exit"
& ,,, "x")

;Next, create the list and add it to the container.
xcall i_ldinp(id_listwnd,, "il_contacts")
xcall l_create(id_list, id_listwnd, list_contact, g_utlib,
    & "l_contacts",,,,,1, %isinfo(chan_icontacts,"NUMRECS"))
xcall l_sect(id_list, "Contacts", D_HEADER)
.ifdef D_GUI
    clear id_selcol, id_inpcol
.else
    xcall m_ldcol(id_selcol,, "c_listsel")
    xcall m_ldcol(id_inpcol,, "c_input")
.endc
xcall c_container(DC_ADD, id_container, DC_LIST, id_list, 1, 1, 3
    & ,,, id_selcol)

;Next, the input window
xcall i_ldinp(id_contact,, "i_contact")
xcall i_frames(id_contact)
xcall c_container(DC_ADD, id_container, DC_WINDOW, id_contact, 8, 40,
    & 4,,, id_inpcol)

;Next, a text window for editing notes
.ifdef D_GUI
    ;On Windows, we'll use a nice multi-line edit control

```

Composite Window Routines

%C_CONTAINER

```
id_text = %ax_tksingle("", 6, 40, "SYNNOTES.SynNotesCtrl.1",
& "SynNotes.ocx",, "notes_")
clear id_editcol
.else
;On non-Windows, we'll use a T_EDIT window
xcall w_proc(WP_CREATE, id_text, "", 6, 40)
xcall u_logwnd(id_text)
xcall t_setup(id_text)
xcall m_ldcol(id_editcol,, "c_edit")
.endc
;Regardless, C_PROCESS knows what method to use by default.
xcall c_container(DC_ADD, id_container, DC_WINDOW, id_text, 13,
40,,,
& id_editcol)

.ifdef D_GUI
;Now an image control
id_image = %ax_tksingle("", 6, 19, "SYNIMAGE.SynImageCtrl.1",
& "SynImage.ocx")
;We have no need to put this into the tabbing order, so we use method
; c_methnop.
xcall c_container(DC_ADD, id_container, DC_WINDOW, id_image, 1, 40,,
& "c_methnop")

xcall i_ldinp(id_imgbtn,, "i_imagebtn")
xcall c_container(DC_ADD, id_container, DC_WINDOW, id_imgbtn, 4,
& 61, 4)
.endc

;Place the container and maximize it.
xcall u_window(D_PLACE, id_container, 1, 1)
xcall w_proc(WP_STATE, id_container, WPS_MAXIMIZED)
return

;
; do_process - run the input loop
;
do_process,
xcall c_process(id_container, chan_icontacts, chan_contacts,
& list_contact, inp_contact)
if (g_select)
begin
using g_entnam select
("O_ADD "), call add
("O_DEL "), call delete
("O_SEL "), call select
("O_IMAGE "), call image
("O_EXIT "), begin
```

```

        call update
        g_select = TRUE      ;Force exit from loop.
      end
    endusing
  end
  return

;
; load_contact - load windows from inp_contact
;
load_contact,
  xcall i_display(id_contact,, inp_contact)
.ifdef D_GUI
  xcall ax_tkset(id_image, "ImagePath", %atrim(inp_contact.image))
  xcall ax_tkset(id_text, "Text", %atrim(inp_contact.notes))
.else
  xcall w_area(id_text, WA_COPY, WAC_WTOP)
  xcall w_info(WI_XFR, id_text, WIX_DPUT, inp_contact.notes)
.endc
  return

;
; add - add a new, empty contact
;
add,
  clear g_select
  xcall l_findspec(id_list, inp_contact,,,,, D_LFND_CURR)
  call update
  clear inp_contact
  call load_contact
  xcall i_init(id_contact)
  xcall c_container(DC_CONTEXT, id_container, 2)      ;Focus input window.
  xcall l_selstyle(id_list, D_OFF)
  return

;
; delete - delete the current contact
;
delete,
  clear g_select
  if (read_one && (%u_msgbox("Really delete this contact?", D_MYESNO)
    &                == D_MIDYES))
    begin
      delete(chan_contacts) [err=delerr]
      clear read_one, inp_contact
.ifdef D_GUI
      xcall ax_tkset(id_text, "Text", "")
.else
      xcall w_area(id_text, WA_COPY, WAC_WTOP, WA_FILL, ' ')

```

Composite Window Routines

%C_CONTAINER

```
.endc
    end
    call add
    return

delerr,
    xcall u_msgbox("Could not delete, error = " + %string(%ernum))
    clear read_one
    return

;
; select - select the contact in the list
;
select,
    clear g_select
    xcall l_findspec(id_list, list_contact,,,,, D_LFND_CURR)
    call update
    read(chan_contacts, inp_contact, list_contact(1:D_KEYSIZ))
    read_one = TRUE
    call load_contact
    return

;
; image - browse for an image file
;
image,
    clear g_select
    if (%u_getfilename(inp_contact.image, "Select image file",,,
    &                               ,inp_contact.image))
        begin
            xcall ax_tkset(id_image, "ImagePath", %atrim(inp_contact.image))
        end
    return

;
; update - update the current contact
;
update,
.ifdef D_GUI
    xcall ax_tkget(id_text, "Text", inp_contact.notes)
.else
    xcall w_area(id_text, WA_COPY, WAC_WTOP)
    xcall w_info(WI_XFR, id_text, WIX_DGET, inp_contact.notes)
.endc
    if (inp_contact)
        begin
            if (read_one)
                delete(chan_contacts)
            store(chan_contacts, inp_contact) [dup=duperr, err=write_err]
```

```

        end

update_cont,
    ;xcall u_update(FALSE)
    xcall l_restart(id_list, 1, %isinfo(chan_icontacts,"NUMRECS"))
    xcall l_process(id_list, request=D_LFIND, list_contact,,
        & chan_icontacts)
    clear g_select, read_one
    return

duperr,
    xcall u_msgbox("Duplicate name, changes not saved")
    goto update_cont

write_err,
    xcall u_msgbox("Unable to save changes, error = " + %string(%ernum))
    goto update_cont
endsubroutine

subroutine load_contacts, reentrant
;
; Description: Load method for contacts list
;
; Arguments:
;
    a_listid      ,n      ;ID of the list

    a_request     ,n      ;List load request

    group a_data ,a      ;Data structure
.include "contact" repository, norecord
endgroup

    a_inpid      ,n      ;ID of the list's input window

    a_dsa        ,n      ;Item disabled flag

    a_index      ,n      ;Item index

    a_chan       ,n      ;Channel for loading

.align
stack record
    name          ,a35

proc
    if (a_index == 1) then
        read(a_chan, a_data, ^FIRST) [err=eof]

```

Composite Window Routines

%C_CONTAINER

```
    else
        reads(a_chan, a_data, eof)
        name = a_data.lastname
        if (a_data.firstname)
            name = %atrim(name) + ", " + a_data.firstname
        xcall i_dspfld(a_inpid, "name", name)
        xreturn

eof,                ;EOF encountered
    a_request = D_LEOF
    xreturn
endsubroutine

subroutine arrive_contact    ,reentrant
;
; Description:  List item arrive method for contacts
;
; Arguments:    None used here
;
.include "WND:tools.def"

proc
    if (!g_select)
        xcall m_signal("O_SEL")    ;Select this one.
    xreturn
endsubroutine

subroutine entrst    ,reentrant
;
; Description:  EENTRST_METHOD
;
; Arguments:
;
    a_nodata    ,n

    a_reset    ,n

proc
    if (!a_nodata)
        clear a_reset
    xreturn
endsubroutine

function notes_click    ,^val, reentrant
;
; Description:  Stock click event for the notes control
;
```



```
; Arguments:
;
    a_ctl      ,i      ;Control ID

.align
stack record
    wndid      ,i4      ;ID of the containing window

proc
    wndid = %ax_getint(a_ctl, D_AXSINGLE_WNDID)      ;Get window ID.
    if (wndid)
        xcall u_wndevents(D_SIGNAL, wndid, D_EVENT_LEFT_CLICK,
            &                1, 1, 1, 1, 0)
        freturn TRUE
    endifunction
endfunction
```

The following is the window script for the above example:

```
.script

.column c_general, "General"
.entry o_exit, "Exit", key(f4), select(x)
.entry c_next, "Next component", key(^n)
.entry c_prev, "Previous component", key(^b)
.end

.input il_contacts, 1, 35
.field name, a35
.end

.listclass l_contacts, header(1), load_method("load_contacts"), -
    arrive_method("arrive_contact")
.end

.column c_listsel, "Select"
.entry s_up, "Up", key(up)
.entry s_down, "Down", key(down)
.entry s_top, "First", key(^f)
.entry s_bottom, "Last", key(^l)
.end

.input i_contact, 5, 40
.repository_structure contact
.field firstname, pos(1,1), prompt("First name: "), required, fpos(1,15)
.field lastname, pos(2,1), prompt("Last name: "), fpos(2,15)
.field phone, pos(3,1), prompt("Phone number: "), fpos(3,15)
.field email, pos(4,1), prompt("E-mail: "), fpos(4,15), view_length(25)
.text t1, position(5,1)
```

Notes:

.end

```
.column c_input, "Input"
.entry i_nextctl, "Next", key(down)
.entry i_prevctl, "Previous", key(up)
.end
```

```
.column c_edit, "Edit"
.entry e_up, "Up", key(up)
.entry e_down, "Down", key(down)
.entry e_left, "Left", key(left)
.entry e_right, "Right", key(right)
.end
```

```
.input i_imagebtn, 1, 1
.button o_image, text("Image..."), select('i')
.button_set right(1)
.end
```

The following is the schema for the repository used in the above example:

```
Structure CONTACT    DBL ISAM    Modified 20061108152400
    Description "Contacts"
```

```
Field LASTNAME    Type ALPHA    Size 20
    Description "Contact's last name"
```

```
Field FIRSTNAME    Type ALPHA    Size 15
    Description "Contact's first name"
```

```
Field PHONE    Type ALPHA    Size 15
    Description "Contact's phone number"
```

```
Field EMAIL    Type ALPHA    Size 64
    Description "E-mail address"
```

```
Field IMAGE    Type ALPHA    Size 256
    Description "Location of picture of contact"
```

```
Field NOTES    Type ALPHA    Size 1024
    Description "Notes on contact"
```

DC_ADD

```
tab_indx = %C_CONTAINER (DC_ADD, container_id, type, child_id, [row], [col], [recordindex],
& [method], [tabindex][, col_id])
```

Return value

tab_indx

The tabbing index of the child within the composite container window. (n)

Arguments

container_id

The ID of the composite container window. This must be the ID of a composite container window (created with the DC_CREATE subfunction). Any other value will cause a fatal Toolkit error. (n)

type

The type of child. One of the following (defined in **tools.def**): (n)

DC_LIST A standard Toolkit list or an ActiveX Toolkit list.

DC_WINDOW A Toolkit window.

child_id

The ID for an existing window or list to be associated with the container. See the [Discussion](#) below. (n)

row

(optional) Placement row for child window or list. See the [Discussion](#) below. (n)

col

(optional) Placement column for child window or list. See the [Discussion](#) below. (n)

recordindex

(optional) An index that determines which *a_methoddata* argument passed in C_PROCESS calls will be used as the data record or field to accept input data for the method. (n)

method

(optional) Method for processing the child. See the [Discussion](#) below. (a or n)

tabindex

(optional) Tabbing index (base 1) for the child, which enables you to insert the child at any position in the tabbing order. (n)

col_id

(optional) ID of a menu column to place when processing this child. (n)

Discussion

The `DC_ADD` subfunction of `%C_CONTAINER` adds an existing UI Toolkit window or list to a composite window. `DC_ADD` can add any type of Toolkit window or list, including an ActiveX window, an ActiveX Toolkit list, a .NET form window, or another composite window. In all cases, the added window or list becomes a child of the composite container window specified with *container_id*. Note the following:

- ▶ When a child is added to a composite window, the child's border is turned off. You can, however, turn it back on with `W_BRDR` or `L_BORDER`. See [W_BRDR](#) in the “Synergy Windowing API” chapter of the *Synergy DBL Language Reference Manual* and [L_BORDER](#) on page 9-54 for more information.
- ▶ By default, child windows and lists are enabled for processing. For information on disabling or re-enabling a child, see [DC_ENABLE](#) on page 6-35.
- ▶ When a window or list is added to a composite window, its `%UWNDEVENTS_METHOD` set is replaced by one that better handles mouse clicks in composite windows. The previously registered method set, however, will be registered as extensions to the new method set if it does not match the new method set. See the [DC_EVENT Discussion](#) on page 6-37 for more information.
- ▶ On Windows, if a child is too large to fit in its composite container window, Toolkit reduces the child's display area to fit in the container and adds scroll bars to the child. To prevent scroll bars from being added to the child, either make the container large enough for the display area defined for the child or call `W_BRDR` with `WB_SBOFF` for the child after calling `DC_ADD`. For example:

```
xcall c_container(DC_ADD, ctrid, chldid, 1, 1)
xcall w_brdr(chldid, WB_SBOFF)
```

On UNIX and OpenVMS, if a child is too large to fit in its composite container window, the display area is circumscribed to make it fit within the display area of the container window.

- ▶ When a container is added to a tab set or another container, it may be resized to fit the display area of the container, which affects the amount of space available for subsequently added child windows and lists.

Child_id must be the ID of an existing Toolkit window or list. You can add any type of window or list, but the window or list cannot be the child of another container. Additionally, a window or list must be logged with Toolkit environment processing (current, global, or another environment) before it can be added as a child.

Row and *col* optionally specify the position of the upper left corner of the child window or list. These settings are relative to the composite container window's client area, so if you pass 1 for *row* and 1 for *col*, the child will be placed in the upper left corner of the client area. Additionally, passing these arguments instructs Toolkit to place the child window or list. Omitting one or both of these arguments prevents the child from being placed.



If *row* is passed, *col* must also be passed. Attempting to pass *row* without *col* will cause a fatal Toolkit error. If *row* is not passed, the window or list will not be placed and *col*, if passed, will be ignored.

Recordindex specifies which *a_methoddata* argument passed in C_PROCESS calls will be passed as the *record* argument to the processing method for the child. When C_PROCESS calls the method for the child, it passes on all supplied *a_methoddata* arguments, and it passes as the *record* argument the *a_methoddata* argument that corresponds to the *recordindex* value. (The default methods that use *record* use it as the data record or field that accepts input for the child. See “[The Composite Window Processing Methods](#)” on page 6-46 for more information.) In the following, for example, because the value 2 is passed as *recordindex* in the DC_ADD call, the second *a_methoddata* argument (data2) in the C_PROCESS call will be passed as the *record* argument (and as the second *a_methoddata* argument) to the method for the child. If the method is the default input method (C_METHINP), for example, the value for data2 will be passed as the third argument (*data_area*) in the I_INPUT call for the method.

```
xcall c_container(DC_ADD, cnt_id1, DC_WINDOW, input_id, 16, 1, 2)
.
.
.
xcall c_process(cnt_id1, data1, data2)
```

Note the following for *recordindex*:

- ▶ *Recordindex* is interpreted as an index. Passing a value of 1 as *recordindex* specifies the first *a_methoddata* argument in a C_PROCESS call, passing a value of 2 specifies the second, and so forth. If *recordindex* is not passed, is less than 1, or specifies an *a_methoddata* argument that is not passed in a C_PROCESS call, the *record* argument will not be passed to the method.
- ▶ Because the *record* argument is used by only three of the default methods (C_METHINP for input windows, C_METHLST for lists, and C_METHSEL for selection windows), *recordindex* is required only if one of these methods (or a method you create that requires it) will be invoked for the child.
- ▶ The same *recordindex* value can be passed for more than one method. For instance, you could have two child input windows that operate on different fields within the same data record.

If *method* is alphanumeric, it specifies the name of a method to invoke to process the child window. If *method* is numeric and non-zero, it specifies the address (as returned by %XADDR) of a method to invoke to process the child window.

If you specify a *method*, the method must be available when the window or list is associated with the container. If *method* is blank or zero, the default method is used to process the child window or list. See [C_PROCESS](#) on page 6-42.

Generally, you'll want to add child windows and lists to the container in the order that you want users to tab through them. However, you can use *tabindex* to specify a position (base 1) in the tabbing order. Indices for siblings at or beyond *tabindex* are incremented by 1.

- ▶ If you omit *tabindex* or pass it as zero, Toolkit adds the child at the end of the tabbing order.
- ▶ If you pass a value that is less than zero or greater than the number of children in the container plus 1, a fatal Toolkit error is generated.

Col_id, if passed and non-zero, specifies the ID of a menu column to be loaded when `C_PROCESS` is called for the container. If `g_plc_col_args` is true, `C_PROCESS` places the column prior to invoking a method and removes it after the method returns.

See also

[“Controlling tabbing, focus, and input context” on page 6-4](#) for information on tabbing indices.

DC_BTNFOCUS

old_button = %C_CONTAINER (DC_BTNFOCUS, *container_id*[, *new_button*])

or

xcall C_CONTAINER (DC_BTNFOCUS, *container_id*[, *new_button*])

Return value

old_button

The tabbing index of the button that has focus when this function is called, or 0 if no button has focus. (**n**)

Arguments

container_id

The ID of the composite container window. This must be the ID of a composite container window (created with the DC_CREATE subfunction). Any other value will cause a fatal Toolkit error. (**n**)

new_button

(optional) The index or name of a button that you want to receive focus. (**a** or **n**)

Discussion

This subfunction is only supported on Windows.

The DC_BTNFOCUS subfunction of %C_CONTAINER does one or both of the following:

- ▶ Indicates whether focus is on a button on the composite container window, and if so, indicates which button has focus. If *old_button* is returned as a value of 0, no button on the container window has focus. If *old_button* is returned as a non-zero value, that value indicates which button has focus.
- ▶ Sets focus to a specified button (*new_button*) on the composite container window. In this case, *old_button* returns the index for the button that had focus *before* this function was called (or a value of 0 if no button had focus).

If you pass *new_button*, and it is not zero or blank, this function does both of the above.

Additionally, the context for the container is automatically set to zero (to focus buttons), and the context reason is set to “C_SET”. (See the C_PROCESS [Discussion on page 6-42](#) for more information on “C_SET”.) If no further context changes are made, the next call to C_PROCESS moves focus to the specified button.

A non-blank alphanumeric value is interpreted as a button name. If you pass an alphanumeric value, it must match a button name for the container, or it will cause a fatal Toolkit error. A non-zero numeric value is interpreted as a button index (base 1). If you pass a non-zero numeric value, it must match an index for a button on the container, or it will cause a fatal Toolkit error.

If you do not pass *new_button* (or pass a value of 0 or blank), this function does not set focus to a button, but only indicates (with *old_button*) whether a button has focus and, if so, returns the tabbing index for that button.



If focus is on a child window or list when you call DC_BTNFOCUS, context for the container is automatically set to 0, but focus will not be set to the specified button until the next call to C_PROCESS. To do this without exiting the current C_PROCESS processing, signal a menu entry, such as "C_RESET", in the method after the DC_BTNFOCUS call.

On UNIX and OpenVMS, DC_BTNFOCUS does nothing unless *new_button* is passed. If *new_button* is passed, DC_BTNFOCUS causes C_PROCESS to process the first child window or list if a button that corresponds to *new_button* is defined for the window. If *new_button* does not correspond to any button defined for the window, there is a fatal error.

See also

- ▶ [I_CTRCONTEXT on page 8-9](#) and [I_NEXT on page 8-65](#) for information on setting focus on a button for a child window or list.
- ▶ [“Controlling tabbing, focus, and input context” on page 6-4](#) for information on tabbing indices.

DC_CHILD

status = %C_CONTAINER (DC_CHILD, *container_id*, *tabindex*, *type*, *child_id*)

or

`xcall C_CONTAINER (DC_CHILD, container_id, tabindex, type, child_id)`

Return value

status

True if the specified tabbing index (*tabindex*) exists for the composite window; false if not. (n)

Arguments

container_id

The ID of the composite container window. This must be the ID of a composite container window (created with the DC_CREATE subfunction). Any other value will cause a fatal Toolkit error. (n)

tabindex

The tabbing index for the child window or list. (n)

type

The returned child type. One of the following (defined in **tools.def**): (n)

DC_LIST A standard Toolkit list or an ActiveX Toolkit list.

DC_WINDOW A Toolkit window.

child_id

The returned ID of the child window or list. (n)

Discussion

The DC_CHILD subfunction of %C_CONTAINER indicates whether a child exists for a given tabbing index (*tabindex*) and, if so, returns the type and ID for the child:

- ▶ If there is a child window or list for *tabindex*, *type* returns the type (DC_LIST or DC_WINDOW) of the child, and *child_id* returns the ID of the child.
- ▶ If there is no child window or list for *tabindex*, *type* and *child_id* are returned as 0, and the function returns *status* as false.

See also

[“Controlling tabbing, focus, and input context” on page 6-4](#) for information on tabbing indices.

DC_CHILDINDEX

tabindex = %C_CONTAINER (DC_CHILDINDEX, *container_id*, *type*, *child_id*)

Return value

tabindex

The tabbing index of a child window or list, or 0 if the specified window or list is not a child of this container. **(n)**

Arguments

container_id

The ID for the composite container window. This must be the ID of a composite container window (created with the DC_CREATE subfunction). Any other value will cause a fatal Toolkit error. **(n)**

type

The type of the child whose tabbing index you want returned. One of the following (defined in **tools.def**): **(n)**

DC_LIST A standard Toolkit list or an ActiveX Toolkit list.

DC_WINDOW A Toolkit window.

child_id

The ID of the child window or list. **(n)**

Discussion

The DC_CHILDINDEX subfunction of %C_CONTAINER enables you to determine if a window or list is a child of a specified container and, if it is, returns its tabbing index.

Use both *type* and *child_id* (which must be a valid Toolkit window or list ID) to specify the window or list.

See also

[“Controlling tabbing, focus, and input context” on page 6-4](#) for information on tabbing indices.

DC_CONTAINER

container_id = %C_CONTAINER (DC_CONTAINER, *type*, *child_id*)

Return value

container_id

The ID of the composite container window for a specified window or list, or 0 if the specified window or list is not part of a composite window. (n)

Arguments

type

The type of the child whose container ID you want returned. One of the following (defined in **tools.def**): (n)

DC_LIST A standard Toolkit list or an ActiveX Toolkit list.

DC_WINDOW A Toolkit window.

child_id

The ID of the child window or list. (n)

Discussion

The DC_CONTAINER subfunction of %C_CONTAINER returns the ID of the composite container window associated with a child or, if the window or list specified by *child_id* is not a child, indicate this by returning zero. Note that for a composite window within another composite window, DC_CONTAINER returns the ID for the immediate container.

Use both *type* and *child_id* (which must be a valid Toolkit window or list ID) to specify the window or list.

DC_CONTEXT

old_context = %C_CONTAINER (DC_CONTEXT, *container_id*, [*new_context*][, *reason*])

or

xcall C_CONTAINER (DC_CONTEXT, *container_id*, [*new_context*][, *reason*])

Return value

old_context

The tabbing index for the child window or list that was active when this function was called, or 0 if one of the buttons on the composite container window was active. (n)

Arguments

container_id

The ID for the composite container window. This must be the ID of a composite container window (created with the DC_CREATE subfunction). Any other value will cause a fatal Toolkit error. (n)

new_context

(optional) The tabbing index for the child window or list that you want to be active, or a value of 0 if you want focus to move to the container, which will have the effect of focusing the first button on the next call to C_PROCESS, or the first child if there is no button. (n)

reason

(optional) The reason for the context change. This can be blank (indicating no change), a program-supplied reason passed when changing context programmatically, or one of the following: (a)

C_FIRST	Attempt to reach first child.
C_LAST	Attempt to reach last child.
C_NEXT	Context shift from previous child (e.g., TAB).
C_PREV	Context shift from next child (e.g., SHIFT+TAB).
C_SET	Context explicitly placed on this child (e.g., mouse click or programmatically without a specified reason).

If you pass *reason*, up to the first 30 characters of this argument are passed to the method for the new active child window or list when the method for that child is invoked.

Discussion

The DC_CONTEXT subfunction of %C_CONTAINER does one or both of the following:

- ▶ Indicates which child window or list is active by returning the tabbing index of the active child, or indicates that a button on the composite container window is active by returning a value of 0.
- ▶ Makes the child window or list specified by *new_context* active. Or, if *new_context* is passed as a value of 0, makes the first button active or the first child active if there is no button.

If you do pass *new_context*, this function does both of the above. If no further context changes are made, the next call to C_PROCESS moves focus to the specified child window or list. Or, if you passed *new_context* as zero, it moves focus either to the current button for the container (if the container has buttons) or to the first child (if the container does not have buttons). Note, however, that if *new_context* is passed and is less than zero or greater than the number of children for the container, a fatal Toolkit error will result.

If you do not pass *new_context*, this function does not make another child window or list active, but only returns the index for the currently active child or returns zero if a button on the container window is active.

See also

- ▶ [C_PROCESS on page 6-42](#) for information on processing child windows and lists.
- ▶ [“Controlling tabbing, focus, and input context” on page 6-4](#) for information on tabbing indices.

Composite Window Routines

`%C_CONTAINER`

DC_CREATE

container_id = `%C_CONTAINER (DC_CREATE, name, rows, cols[, title])`

Return value

container_id

The ID for the created composite container window. **(n)**

Arguments

name

A name for the new composite container window. **(a)**

rows

The number of rows in the container. **(n)**

cols

The number of columns in the container. **(n)**

title

(optional) The title for the composite container window. **(a)**

Discussion

The `DC_CREATE` subfunction of `%C_CONTAINER` creates a composite container window, which is a UI Toolkit window that can be associated with other Toolkit windows and lists to create a composite window. Note that when you create a composite container window, it is automatically logged with the current Toolkit environment. You can, however, promote it to global with `U_GBLWND`.

If you pass a null string (“”) for *name*, Toolkit assigns a unique name in the form `_W_nnn`, where *nnn* is the container ID.

If *title* is not passed, the container window will not have a title. (You can add one later using the `WB_TITLE` subfunction of `W_BRDR`.)

See also

[“Creating and Processing Composite Windows” on page 6-2.](#)

DC_CTRCONTEXT

new_context = %C_CONTAINER (DC_CTRCONTEXT, *container_id*, *reason*)

Return value

new_context

The tabbing index for the child window or list that became active as a result of this function, or 0 if focus moves to a button. **(n)**

Arguments

container_id

The ID for the composite container window. This must be the ID of a composite container window (created with the DC_CREATE subfunction). Any other value will cause a fatal Toolkit error. **(n)**

reason

The reason for the context change, which determines which child will become active (see the [Discussion](#) below). This can be blank (indicating no change), a program-supplied reason passed when changing context programmatically, or one of the following: **(a)**

C_FIRST	Attempt to reach first child.
C_LAST	Attempt to reach last child.
C_NEXT	Context shift from previous child (e.g., TAB).
C_PREV	Context shift from next child (e.g., SHIFT+TAB).
C_SET	Context explicitly placed on a child or button on the composite container window (e.g., mouse click or programmatically without a specified reason).

Discussion

The DC_CTRCONTEXT subfunction of %C_CONTAINER enables you to use the context change reason (*reason*) to activate a child on a composite window. Typically, this routine is used in the method for contained composite windows (see [C_METHCTR](#) on page 6-49). You can, however, pass “C_LAST” to move to the last button or child without having to know how many of either one exist in the container:

```
c_container(DC_CTRCONTEXT, ctrid, "C_LAST")
```

DC_CTRCONTEXT does one of the following based on the content of *reason*, which is typically the *reason* argument passed to the method from C_PROCESS:

- ▶ If *reason* is “C_FIRST” or “C_NEXT”, it moves focus to the first child window or list. If there are buttons but no child windows or lists, it sets focus on the first button for the container.

Composite Window Routines

%C_CONTAINER

- ▶ If *reason* is “C_LAST” or “C_PREV”, it moves focus to the last button (if the container has buttons), or the last child window or list (if the container has no buttons).
- ▶ If *reason* is set to any other value (“C_SET”, blank, etc.), focus does not change.

DC_ENABLE

old_state = %C_CONTAINER (DC_ENABLE, *container_id*, *tabindex*[, *new_state*])

or

xcall C_CONTAINER (DC_ENABLE, *container_id*, *tabindex*[, *new_state*])

Return value

old_state

1 if the specified child is enabled when this function is called, or 0 if the child window or list is disabled when this function is called. **(n)**

Arguments

container_id

The ID for the composite container window with a child whose enabled/disabled state you want to query and/or change. This must be the ID of a composite container window (created with the DC_CREATE subfunction). Any other value will cause a fatal Toolkit error. **(n)**

tabindex

The tabbing index for the child whose enabled/disabled state you want to query and/or change. **(n)**

new_state

Either a non-zero value to enable the child window or list or zero to disable the child window or list. **(n)**

Discussion

The DC_ENABLE subfunction of %C_CONTAINER enables you to enable or disable a child window or list (specified with *tabindex*). It also returns the state (enabled or disabled) of the child when DC_ENABLE was called (prior to any change made by the call). If a child window or list is disabled, C_PROCESS will not invoke its method, and on Windows mouse clicks will not trigger events for the window or list.

DC_EVENT

return = %C_CONTAINER (DC_EVENT, *container_id*, *tabindex*, *event*, *method*)

or

xcall C_CONTAINER (DC_EVENT, *container_id*, *tabindex*, *event*, *method*)

Return value

return

Always returns true. (n)

Arguments

container_id

The ID for the composite container window with a child window or list whose event method extension you want to replace. This must be the ID of a composite container window created with the DC_CREATE subfunction. Any other value will cause a fatal Toolkit error. (n)

tabindex

The tabbing index for the child window or list whose event method extension you want to replace. (n)

event

The code for the event whose method extension will be replaced. One of the following (defined in **tools.def**): (n)

D_EVENT_LEFT_CLICK	Left mouse button event.
D_EVENT_MIDDLE_CLICK	Middle mouse button event.
D_EVENT_RIGHT_CLICK	Right mouse button event.
D_EVENT_LEFT_DBLCLK	Double-click left mouse button.
D_EVENT_MIDDLE_DBLCLK	Double-click middle mouse button.
D_EVENT_RIGHT_DBLCLK	Double-click right mouse button.
D_EVENT_MOVE	Move window event.
D_EVENT_SIZE	Size window event.
D_EVENT_MINIMIZE	Minimize window event.
D_EVENT_MAXIMIZE	Maximize window event.
D_EVENT_RESTORE	Restore window event.
D_EVENT_SCROLL	Scroll window event.
D_EVENT_CLOSE	Close window event.

method

The name or address for the routine that will replace any automatically registered %UWNDEVENT_METHOD method for the specified event. (**a** or **n**)

Discussion

The DC_EVENT subfunction of %C_CONTAINER registers a routine (*method*) as an extension to the standard method for an event for a child window or list. This replaces any previously registered extension for the event.

The standard method set contains only one method, a left-click method that sets the context of the container to the clicked child window or list:

- ▶ For an input window, if the left click occurs on a field, drilldown button, or hyperlink prompt within the window, this method sets a state for the window indicating that the field was clicked. Then, when I_INPUT processes the window, context is established on that item, and the click is simulated again (as long as the clicked item is in the current input set, and as long as context has not been changed for that window in the meantime).
- ▶ For a list, if the left click occurs on an item, the standard left click method queues a request to select the item. (If the click occurs elsewhere, such as in the header, this method only returns.)
- ▶ For a composite container window (a contained container), this method sets the context of the container to the clicked window or list and then signals “C_RESET” to reset focus for all containers.

You can register an extension for any event, including the left click event. Note the following:

- ▶ Extension routines must accept the arguments and return a value as specified in [%UWNDEVENTS_METHOD on page 16-237](#).
- ▶ If a routine has already been registered for the event, DC_EVENT replaces that routine with the method specified with *method*.
- ▶ When a child window or list is added to a container window, Toolkit automatically registers any %UWNDEVENTS_METHOD methods associated with that window as extensions to the standard method set for the child window. You can use DC_EVENT to replace these extensions.
- ▶ If you register an extension for the left click event, the extension is called first, and then the standard left click event is called unless the extension returns a value of 0, in which case it is not invoked.
- ▶ The standard left click event won’t do any of the actions listed above if the clicked window is the currently active window in the currently active container (and its parent window, grandparent window, and so forth, are also the active windows in their containers). However, any extension registered with DC_EVENT will be invoked.

See also

- ▶ [%UWNDEVENTS_METHOD on page 16-237](#).
- ▶ “Controlling tabbing, focus, and input context” on page 6-4 for information on tabbing indices.

DC_GETEVENT

method_addr = %C_CONTAINER (DC_GETEVENT, *container_id*, *tabindex*, *event*)

Return value

method_addr

The returned method address, or 0 if no routine is registered as the extension for the specified child window / event pair. (**D_ADDR**)

Arguments

container_id

The ID for the composite container window. This must be the ID of a composite container window (created with the DC_CREATE subfunction). Any other value will cause a fatal Toolkit error. (**n**)

tabindex

The tabbing index for the child window or list. (**n**)

event

The code for the event whose method extension address you want to retrieve. One of the following (defined in **tools.def**): (**n**)

D_EVENT_LEFT_CLICK	Left mouse button event.
D_EVENT_MIDDLE_CLICK	Middle mouse button event.
D_EVENT_RIGHT_CLICK	Right mouse button event.
D_EVENT_LEFT_DBLCLK	Double-click left mouse button.
D_EVENT_MIDDLE_DBLCLK	Double-click middle mouse button.
D_EVENT_RIGHT_DBLCLK	Double-click right mouse button.
D_EVENT_MOVE	Move window event.
D_EVENT_SIZE	Size window event.
D_EVENT_MINIMIZE	Minimize window event.
D_EVENT_MAXIMIZE	Maximize window event.
D_EVENT_RESTORE	Restore window event.
D_EVENT_SCROLL	Scroll window event.
D_EVENT_CLOSE	Close window event.

Discussion

The `DC_GETEVENT` subfunction of `%C_CONTAINER` returns the address for the currently registered extension for an event method for a child window.

See also

- ▶ [DC_EVENT](#) on page 6-36 for information on event extensions.
- ▶ [“Controlling tabbing, focus, and input context”](#) on page 6-4 for information on tabbing indices.

Composite Window Routines

%C_CONTAINER

DC_NUMCHILD

number = %C_CONTAINER (DC_NUMCHILD, *container_id*)

Return value

number

The number of child windows and lists associated with a composite container window. If the composite container window has no children, or if the window is not a composite container window, returns value of 0. **(n)**

Arguments

container_id

The ID for the composite container window. This must be the ID of a composite container window (created with the DC_CREATE subfunction). Any other value will cause a fatal Toolkit error. **(n)**

Discussion

The DC_NUMCHILD subfunction of %C_CONTAINER returns the number of child windows and lists associated with a composite container window. For a container that is a child of another container, only immediate children are counted.

DC_REMOVE

`xcall C_CONTAINER (DC_REMOVE, container_id, tabindex)`

Arguments

container_id

The ID for the composite container window. This must be the ID of a composite container window (created with the DC_CREATE subfunction). Any other value will cause a fatal Toolkit error. (n)

tabindex

The tabbing index for the child window or list to be removed. (n)

Discussion

The DC_REMOVE subfunction of %C_CONTAINER removes a child window or list from a composite window by disassociating it from the composite container window.

When a child is removed from its container the following are true:

- ▶ The tabbing indices for subsequent children are decremented.
- ▶ The child will no longer be placed.
- ▶ The child's border will retain the state it had before the DC_REMOVE call.
- ▶ The %UWNDEVENTS_METHOD set for the child will remain as it was before the DC_REMOVE call, but the left click event for child windows or lists will be disabled.

If *tabindex* is less than 1 or greater than the number of child windows and lists, a fatal Toolkit error will result.

If %C_CONTAINER is called as a function with DC_REMOVE, it always returns true.

See also

[“Controlling tabbing, focus, and input context” on page 6-4](#) for information on tabbing indices.

C_PROCESS – Process a composite window



```
xcall C_PROCESS (container_id [, a_methoddata1, ..., a_methoddata20])
```

Arguments

container_id

The ID for the container window. This must be the ID of a composite container window (created with the DC_CREATE subfunction of %C_CONTAINER). Any other value will cause a fatal Toolkit error. (**n**)

a_methoddata1 through *a_methoddata20*

(optional) Up to 20 additional data arguments. See the Discussion. (**any**)

Discussion

The C_PROCESS subroutine processes composite windows by looping through and processing child windows and child lists as well as enabled buttons on the composite container window.

- ▶ To process a child window or list, C_PROCESS calls the method for the child (see the “Processing Methods” table on page 6-43).
- ▶ To process a button on the container, C_PROCESS merely waits for a user response.

If there are no placed and enabled child windows, child lists, or buttons on the container, C_PROCESS immediately returns control to the routine that called it.

C_PROCESS starts on whatever child currently has context. By default, it starts with the first child window or list in the tabbing order, which is the child with the tabbing index of 1. (If the window has no child, C_PROCESS starts with the first button on the container window.) However, if the container has previously been processed, it starts with the child that was active when C_PROCESS last exited. Additionally, if the DC_CONTEXT subfunction for %C_CONTAINER has been used to specify a child, C_PROCESS starts with that child.

When the processing for a button or child has finished, C_PROCESS checks **g_entnam** to see if a C_ menu entry that it recognizes is active and, if one is, moves focus as instructed by the C_ menu entry. (Recognized menu entries are listed below.) It then continues the loop by processing the newly active child or button. However, if a processing method or button on the container returns after signaling a menu entry that C_PROCESS does *not* recognize, C_PROCESS returns control to the routine that called it, and the menu entry is passed on to that routine.

When C_PROCESS is first called, it does not check **g_entnam**, so using %M_SIGNAL to signal one of the C_ menu entries before calling C_PROCESS will not work. Use the DC_CONTEXT subfunction for %C_CONTAINER to instruct C_PROCESS to start on a child or button other than the default.

Note the following:

- ▶ If a processing method returns without signaling a menu entry (and doesn't otherwise change focus to another child or button before returning), C_PROCESS calls the method again, resulting in a possibly endless loop.
- ▶ C_PROCESS clears **g_select** after processing a menu entry that it recognizes. Otherwise it leaves this value as is. Additionally, C_PROCESS does not in itself change the values of **g_setsts**, **g_fldnam**, or **g_entnam**. However, a button or the processing method for a child may change these. For example, a child method will set these variables as documented in the [I_INPUT Discussion on page 8-56](#) if the method calls I_INPUT and then returns without signaling another menu entry. (Such a method would need to change focus in some way to another child, or C_PROCESS will again signal the input method for the same child. See note above.)
- ▶ If context moves to a disabled child, the method for the child is not called. Rather, context is automatically moved to the next or previous child or button. If the context reason is "C_PREV" or "C_LAST", the previous enabled child or button is activated. Otherwise, the next enabled child or button is activated.
- ▶ If a button is enabled on a child window or list that is not active and the user clicks that button, the signaled menu entry will be seen first by the method for the child that is active. If that method does not clear **g_select**, the menu entry is passed out to C_PROCESS. If C_PROCESS doesn't recognize it, it is passed out to the routine that called C_PROCESS. Consequently, the method for the window or list that contains the button will not see the menu entry, so we recommend that you do one or more of the following:
 - ▶ Handle button-generated menu entries in the outermost loop that calls C_PROCESS.
 - ▶ Handle button-generated menu entries in a common routine that all methods can call.
 - ▶ Disable buttons explicitly when your application is not prepared to process them. See the [.BUTTON Discussion on page 2-14](#) for information on disabling and enabling buttons.

To process a child window or list, C_PROCESS calls either the processing method when the child was added (with the DC_ADD subroutine for %C_CONTAINER) or the default Toolkit-supplied method as listed in the table below.

Processing Methods	
Child type	Default processing method
ActiveX window	C_METHAX on page 6-47
Composite window	C_METHCTR on page 6-49
Input window	C_METHINP on page 6-51
List	C_METHLST on page 6-54
.NET form window	C_METHNET on page 6-57

Processing Methods	
Child type	Default processing method
Selection window	C_METHSEL on page 6-61
Tab set window	C_METHTS on page 6-63
Text window	C_METHTXT on page 6-65
Other	C_METHNOP on page 6-59

C_PROCESS recognizes the following menu entries:

- ▶ “C_FIRST” makes the first child (window or list) active, or it makes the first button on the composite container window active if there is no child. Note, however, that if the composite window is the child of another composite window, this menu entry is passed to the outer composite window.
- ▶ “C_LAST” makes the last child on the composite window active if there are no buttons on the composite container window. If there are buttons, “C_LAST” makes the last button active. Note, however, that if the composite window is the child of another composite window, this menu entry is passed to the outer composite window.
- ▶ “C_NEXT” makes the next child active or, if the last child is currently active, it makes the first button on the composite container window active. If the last child or button is active, and the composite window is *not* the child of another composite window, “C_NEXT” makes the first child active (or button if there is no child). If the composite window is the child of another composite window, however, and the last button is active (or child if there are no buttons), the menu entry is passed to the outer composite window.
- ▶ “C_PREV” makes the previous child window, child list, or button on the container active. If the first child is active (or button if there is no child), “C_PREV” makes the last button active, or if there are no buttons, it makes the last child active. Note, however, that if the composite window is the child of another composite window, and focus is on the first child (or button if there is no child), this menu entry is passed to the outer composite window.
- ▶ “C_RESET” takes into account any programmatic context change (for example, with DC_CONTEXT or DC_BTNFOCUS) and resets the context in a composite window to the currently active child window, child list, or button. Note, however, that if the composite window is itself a child of another composite window, this menu entry is passed to the outer composite window.
- ▶ “C_SETnnnn” makes the child window or list with the tabbing index *nnnn* active. (If *nnnn* is less than 1 or greater than the number of children, the menu entry is ignored.)

For the child processing methods (default or otherwise), C_PROCESS passes the following: the container ID, the type of child (window or list), the child ID, the reason the child was activated, and as many method data arguments as are passed in the C_PROCESS call. Additionally, note that

- ▶ the data record argument may also be passed and is required only if one of the following is invoked: C_METHINP, C_METHLST, C_METHSEL, or a method you created that requires it.
- ▶ the data record argument will not be passed to the method if *recordindex* was not passed in the DC_ADD call for the child, if it was passed as a value that was less than 1, or if the *a_methoddata* argument that corresponds to *recordindex* is not passed in the C_PROCESS call.

The *a_methoddata* arguments are optional and are ignored unless C_PROCESS invokes one of the following: the default method for input windows (C_METHINP), the default method for lists (C_METHLST), the default method for selection windows (C_METHSEL), or a method you have created that requires these arguments. Note that for C_METHINP, C_METHLST, and C_METHSEL,

- ▶ the only *a_methoddata* argument required is the one specified by the *recordindex* value passed in the DC_ADD call for the window or list. For example, if the value passed as *recordindex* in the DC_ADD call is 3, you must include the third *a_methoddata* argument in the C_PROCESS call.
- ▶ the *a_methoddata* argument you pass must be the data record or field for input for the child window or list. C_PROCESS will not only pass this *a_methoddata* argument on to the method, it will also pass this argument as the *record* argument for the method. This in turn is passed as the *data_area* argument to I_INPUT (called by C_METHINP), the *data* argument passed to L_SELECT (called by C_METHLST), or the *field_data* argument passed to S_SELECT (called by C_METHSEL).

See also

- ▶ [“Controlling tabbing, focus, and input context” on page 6-4](#) for information on tabbing indices.
- ▶ [%C_CONTAINER on page 6-10](#) for information on this function and its subroutines.

Examples

See the [%C_CONTAINER Examples on page 6-11](#).

The Composite Window Processing Methods

To process a child window or list, `C_PROCESS` calls either the default processing method for the child or the processing method assigned to the child when it was added with `DC_ADD`. (See [C_PROCESS on page 6-42](#) for details.) This section documents the default Toolkit-supplied processing methods for composite windows and the arguments and return values needed for any processing methods you write for child windows or lists.



You must declare method arguments exactly as documented. Toolkit provides strong prototype validation for composite window processing methods. Incorrectly declared arguments will cause undefined results.

`C_METHAX` – Process a child ActiveX window

`C_METHCTR` – Process a child composite window

`C_METHINP` – Process a child input window

`C_METHLST` – Process a child list

`C_METHNET` – Process a child .NET form window

`C_METHNOP` – Prevent a child from gaining focus

`C_METHSEL` – Process a child selection window

`C_METHTS` – Process a child tabbed dialog

`C_METHTXT` – Process a child text window

See also

- ▶ The `C_PROCESS` [Discussion on page 6-42](#), which documents how `C_PROCESS` calls a method and how methods control the `C_PROCESS` loop by signaling menu entries.
- ▶ The `ctrmethods.dbl` file included in your UI Toolkit distribution. This file includes the code for the default methods and serves as an example of how to create your own composite window processing methods. (Note that modifying or compiling `ctrmethods.dbl` will not change the default methods; it is distributed simply to supply examples.)

C_METHAX – Process a child ActiveX window

WT			

```

subroutine C_METHAX, reentrant
in      req container_id      , n
in      req type              , n
in      req child_id         , n
in      req reason           , a
opt record                   , a
opt a_methoddata1           , any
.
.
opt a_methoddata20          , any

```

Arguments

container_id

This argument is not used by the default processing methods. However, it passes the ID of the container window in case a method you supply needs this information. **(n)**

type

This argument is not used by the default processing methods. However, it passes one of the following (defined in **tools.def**) in case a method you supply needs this information: **(n)**

DC_LIST A standard Toolkit list or an ActiveX Toolkit list.

DC_WINDOW A Toolkit window.

child_id

The ID for the Toolkit container for the ActiveX control. (See “[Containers for ActiveX controls](#)” on page 4-4 for information.) **(n)**

reason

The reason the child was activated, which can be blank (indicating that the child was already activated), a program-supplied reason passed because the child was activated programmatically, or one of the following: **(a)**

C_FIRST Focus moved in attempt to reach first child.

C_LAST Focus moved in attempt to reach last child.

C_NEXT Focus moved from previous child (e.g., TAB).

C_PREV Focus moved from next child (e.g., SHIFT+TAB).

C_SET Child explicitly activated. For example, this could be the result of a mouse click, or it could be that the child was activated programmatically without a specified reason.

record

(optional) This argument is not used by the default method C_METHAX. However, this argument passes a copy of the *a_methoddata* argument specified by *recordindex* (a DC_ADD argument) for other methods that need this information. **(a)**

a_methoddata1 through *a_methoddata20*

(optional) These arguments are not used by the default method C_METHAX. However, up to 20 additional data arguments are passed for other methods that need this information. **(any)**

Discussion

C_METHAX is a Toolkit-supplied processing method that C_PROCESS calls by default when processing a child ActiveX window. If you want something other than the default processing described below, you can instruct C_PROCESS to call a different routine by passing the routine's name in the DC_ADD call for the window.

C_METHAX does the following:

1. Calls U_CTRCONTEXT to set context within the ActiveX window.
2. Calls AX_INPUT to process input.



To see the code for this method, see the **ctrmeths.dbl** file included in your UI Toolkit distribution. This file is distributed simply as a reference and a starting point for writing your own composite window processing methods; modifying or compiling this file will not change the default methods.

See also

- ▶ [“Controlling tabbing, focus, and input context” on page 6-4](#) for information on tabbing indices.
- ▶ [DC_ADD on page 6-21](#).
- ▶ [C_PROCESS on page 6-42](#).
- ▶ [“Overloading UI Toolkit Routines” on page 18-8](#) for information on overloading this subroutine.

C_METHCTR – Process a child composite window



```
subroutine C_METHCTR, reentrant
in      req container_id      , n
in      req type              , n
in      req child_id         , n
in      req reason           , a
opt record                   , a
opt a_methoddata1           , any
.
.
opt a_methoddata20          , any
```

Arguments

container_id

This argument is not used by the default processing methods. However, it passes the ID of the container window in case a method you supply needs this information. **(n)**

type

This argument is not used by the default processing methods. However, it passes one of the following (defined in **tools.def**) in case a method you supply needs this: **(n)**

DC_LIST A standard Toolkit list or an ActiveX Toolkit list.

DC_WINDOW A Toolkit window.

child_id

The ID for the child container window. **(n)**

reason

The reason the child was activated, which can be blank (indicating that the child was already activated), a program-supplied reason passed because the child was activated programmatically, or one of the following: **(a)**

C_FIRST Focus moved in attempt to reach first child.

C_LAST Focus moved in attempt to reach last child.

C_NEXT Focus moved from previous child (e.g., TAB).

C_PREV Focus moved from next child (e.g., SHIFT+TAB).

C_SET Child explicitly activated. For example, this could be the result of a mouse click, or it could be that the child was activated programmatically without a specified reason.

record

(optional) This argument is not used by the default method C_METHCTR. However, this argument passes a copy of the *a_methoddata* argument specified by *recordindex* (a DC_ADD argument) for other methods that need this information. (**a**)

a_methoddata1 through *a_methoddata20*

(optional) Up to 20 additional data arguments. (**any**)

Discussion

C_METHCTR is a Toolkit-supplied processing method that C_PROCESS calls by default when processing a child window that is itself a composite window. If you want something other than the default processing described below, you can instruct C_PROCESS to call a different routine by passing the routine's name in the DC_ADD call for the child composite window.

C_METHCTR does the following:

1. Establishes focus in the child composite window by calling the DC_CTRCONTEXT subfunction of %C_CONTAINER and passing the ID of the child container and *reason*.
2. Calls C_PROCESS to process the child composite window, passing the ID of the composite container window and the *a_methoddata* arguments.



To see the code for this method, see the **ctrmeths.dbl** file included in your UI Toolkit distribution. This file is distributed simply as a reference and a starting point for writing your own composite window processing methods; modifying or compiling this file will not change the default methods.

See also

- ▶ “Controlling tabbing, focus, and input context” on page 6-4 for information on tabbing indices.
- ▶ DC_ADD on page 6-21.
- ▶ C_PROCESS on page 6-42.
- ▶ “Overloading UI Toolkit Routines” on page 18-8 for information on overloading this subroutine.

C_METHINP – Process a child input window



```

subroutine C_METHINP, reentrant
in      req container_id      , n
in      req type              , n
in      req child_id         , n
in      req reason           , a
opt record                   , a
opt a_methoddata1           , any
.
.
opt a_methoddata20          , any

```

Arguments

container_id

This argument is not used by the default processing methods. However, it passes the ID of the container window in case a method you supply needs this information. **(n)**

type

This argument is not used by the default processing methods. However, it passes one of the following (defined in **tools.def**) in case a method you supply needs this: **(n)**

DC_LIST A standard Toolkit list or an ActiveX Toolkit list.

DC_WINDOW A Toolkit window.

child_id

The ID for the child input window. **(n)**

reason

The reason the child was activated, which can be blank (indicating that the child was already activated), a program-supplied reason passed because the child was activated programmatically, or one of the following: **(a)**

C_FIRST Focus moved in attempt to reach first child.

C_LAST Focus moved in attempt to reach last child.

C_NEXT Focus moved from previous child (e.g., TAB).

C_PREV Focus moved from next child (e.g., SHIFT+TAB).

C_SET Child explicitly activated. For example, this could be the result of a mouse click, or it could be that the child was activated programmatically without a specified reason.

record

(optional) The data area for child window input. (**a**)

a_methoddata1 through *a_methoddata20*

(optional) Up to 20 additional data arguments. (**any**)

Discussion

C_METHINP is a Toolkit-supplied processing method that C_PROCESS calls by default when processing a child input window. If you want something other than the default processing described below, however, you can instruct C_PROCESS to call a different routine by passing the routine's name in the DC_ADD call for the input window.

C_METHINP does the following:

1. Sets focus for the child input window by calling I_CTRCONTEXT and passing the following: the ID of the input window (*child_id*), “*CURR*” for the input set name, and the context change reason (*reason*) that was passed to this method.

Note that the I_CTRCONTEXT is designed specifically to be used in processing methods for child input windows. See [I_CTRCONTEXT on page 8-9](#) for more information.

2. Processes the input window by calling I_INPUT and passing the following: the ID of the input window (*child_id*), “*CURR*” for the input set name, the *record* argument passed to this method (see below), and all the *a_methoddata* arguments passed to this method. It also passes the D_NOTERM argument to avoid the input window completion logic that displays the “Make corrections or press RETURN to continue” message.
3. If **g_setsts** is false when control is returned to C_METHINP (indicating that input is complete), C_METHINP then calls %M_SIGNAL, passing “C_NEXT” as the menu entry name to signal. This makes the next child active (or the first button for the composite container window, if there are no more children). If **g_setsts** is true, context remains on the same child window. For information on **g_setsts**, see the I_INPUT [Discussion on page 8-56](#).

Note that for *record*, C_PROCESS passes the *a_methoddata* argument specified by the *recordindex* argument in the DC_ADD call for the window. For details, see the information on *recordindex* in the DC_ADD [Discussion on page 6-22](#), and see the information on the *a_methoddata* arguments in the C_PROCESS [Discussion on page 6-42](#).



To see the code for this method, see the **ctrmeths.dbl** file included in your UI Toolkit distribution. This file is distributed simply as a reference and a starting point for writing your own composite window processing methods; modifying or compiling this file will not change the default methods.

See also

- ▶ [“Controlling tabbing, focus, and input context” on page 6-4](#) for information on tabbing indices.
- ▶ [C_PROCESS on page 6-42](#).
- ▶ [“Overloading UI Toolkit Routines” on page 18-8](#) for information on overloading this subroutine.

C_METHLST – Process a child list



```

subroutine C_METHLST, reentrant
in      req  container_id      , n
in      req  type              , n
in      req  child_id         , n
in      req  reason           , a
opt record                      , a
opt a_methoddata1              , any
.
.
opt a_methoddata20             , any

```

Arguments

container_id

This argument is not used by the default processing methods. However, it passes the ID of the container window in case a method you supply needs this information. **(n)**

type

This argument is not used by the default processing methods. However, it passes one of the following (defined in **tools.def**) in case a method you supply needs this information: **(n)**

DC_LIST A standard Toolkit list or an ActiveX Toolkit list.

DC_WINDOW A Toolkit window.

child_id

The ID for the child list. **(n)**

reason

This argument is not used by the default method C_METHLST. However, it is passed for methods that do need the reason the child was activated. *Reason* can be blank (indicating that the child was already activated), a program-supplied reason passed because the child was activated programmatically, or one of the following: **(a)**

C_FIRST Focus moved in attempt to reach first child.

C_LAST Focus moved in attempt to reach last child.

C_NEXT Focus moved from previous child (e.g., TAB).

C_PREV	Focus moved from next child (e.g., SHIFT+TAB).
C_SET	Child explicitly activated. For example, this could be the result of a mouse click, or it could be that the child was activated programmatically without a specified reason.

record

(optional) The data area for input for the child list. (**a**)

a_methoddata1 through *a_methoddata20*

(optional) Up to 20 additional data arguments. (**any**)

Discussion

C_METHLST is a Toolkit-supplied processing method that C_PROCESS calls by default when processing a child that is a standard Toolkit list or an ActiveX Toolkit list. If you want something other than the default processing described below, you can instruct C_PROCESS to call a different routine by passing the routine's name in the DC_ADD call for the list.

For *record*, C_PROCESS passes the *a_methoddata* argument specified by the *recordindex* argument in the DC_ADD call for the window. For details, see the information on *recordindex* in the DC_ADD [Discussion on page 6-22](#), and see the information on the *a_methoddata* arguments in the C_PROCESS [Discussion on page 6-42](#).

C_METHLST does the following:

1. Sets the highlight style for the selected item to highlight (as opposed to outline or off) by calling %L_SELSTYLE with D_ON.
2. If on UNIX or OpenVMS, calls L_ENABLE. (See note below.)
3. Calls L_CTRCONTEXT to set focus to either the list or a button for the list, depending on *reason*.
4. Processes the list by calling L_SELECT and passing the following: the ID of the list (*child_id*), a D_LNOP request code, the *record* argument passed to this method, and all the *a_methoddata* arguments passed to this method.
5. If **g_select** is false when control is returned to C_METHLST (indicating that an item was selected), C_METHLST then calls %M_SIGNAL, passing "C_NEXT" as the menu entry name to signal. This passes context to the next child (or the first button for the composite container window, if there are no more children). If **g_select** is true, context remains on the list.
6. If on UNIX or OpenVMS, calls L_DISABLE. (See note below.)
7. Sets the highlight style for the selected item to outline by calling %L_SELSTYLE with D_OUTLINE.

To visually indicate when a child list is active on Windows, the default method sets the highlight style for the selected item to highlight ([step 1](#) above) when the child list is active and to outline ([step 7](#) above) when the child list is not active. This is generally effective for standard Toolkit lists in Windows environments, but not in UNIX and OpenVMS environments, where this doesn't result

in any visual difference. (On UNIX and OpenVMS, D_OUTLINE has the same effect as D_ON). So for UNIX and OpenVMS, this method enables the list when active ([step 2](#) above) and disables the list when inactive ([step 6](#) above).

ActiveX Toolkit lists ignore %L_SELSTYLE settings, but automatically have a bold frame when active.



To see the code for this method, see the **ctrmeths.dbl** file included in your UI Toolkit distribution. This file is distributed simply as a reference and a starting point for writing your own composite window processing methods; modifying or compiling this file will not change the default methods.

See also

- ▶ [“Controlling tabbing, focus, and input context” on page 6-4](#) for information on tabbing indices.
- ▶ [C_PROCESS on page 6-42](#).
- ▶ [“Overloading UI Toolkit Routines” on page 18-8](#) for information on overloading this subroutine.

C_METHNET – Process a child .NET form window

WT			

```

subroutine C_METHNET, reentrant
in      req container_id      , n
in      req type              , n
in      req child_id         , n
in      req reason           , a
opt record                   , a
opt a_methoddata1           , any
.
.
opt a_methoddata20          , any

```

Arguments

container_id

This argument is not used by the default processing methods. However, it passes the ID of the container window in case a method you supply needs this information. **(n)**

type

This argument is not used by the default processing methods. However, it passes one of the following (defined in **tools.def**) in case a method you supply needs this information: **(n)**

DC_LIST A standard Toolkit list or an ActiveX Toolkit list.

DC_WINDOW A Toolkit window.

child_id

The ID for the Toolkit container window for the .NET form. See [%DOTNET_TKWIN on page 11-12](#). **(n)**

reason

The reason the child was activated, which can be blank (indicating that the child was already activated), a program-supplied reason passed because the child was activated programmatically, or one of the following: **(a)**

C_FIRST Focus moved in attempt to reach first child.

C_LAST Focus moved in attempt to reach last child.

C_NEXT Focus moved from previous child (e.g., TAB).

C_PREV Focus moved from next child (e.g., SHIFT+TAB).

C_SET Child explicitly activated. For example, this could be the result of a mouse click, or it could be that the child was activated programmatically without a specified reason.

record

(optional) This argument is not used by the default method C_METHNET. However, this argument passes a copy of the *a_methoddata* argument specified by *recordindex* (a DC_ADD argument) for other methods that need this information. **(a)**

a_methoddata1 through *a_methoddata20*

(optional) These arguments are not used by the default method C_METHNET. However, up to 20 additional data arguments are passed for other methods that need this information. **(any)**

Discussion

C_METHNET is a Toolkit-supplied processing method that C_PROCESS calls by default when processing a child .NET form window. If you want something other than the default processing described below, you can instruct C_PROCESS to call a different routine by passing the routine's name in the DC_ADD call for the window.

C_METHNET does the following:

1. Calls U_CTRCONTEXT to set context within the .NET form window.
2. Calls DOTNET_TKINPUT to process input.



To see the code for this method, see the **ctrmeths.dbl** file included in your UI Toolkit distribution. This file is distributed simply as a reference and a starting point for writing your own composite window processing methods; modifying or compiling this file will not change the default methods.

See also

- ▶ [“Controlling tabbing, focus, and input context” on page 6-4](#) for information on tabbing indices.
- ▶ [C_PROCESS on page 6-42](#).
- ▶ [“Overloading UI Toolkit Routines” on page 18-8](#) for information on overloading this subroutine.
- ▶ [Chapter 11, “.NET Routines,”](#) for information on .NET form windows.

C_METHNOP – Prevent a child from gaining focus



```

subroutine C_METHNOP, reentrant
in      req container_id      , n
in      req type              , n
in      req child_id         , n
in      req reason           , a
opt record                   , a
opt a_methoddata1           , any
.
.
opt a_methoddata20          , any

```

Arguments

container_id

This argument is not used by the default processing methods. However, it passes the ID of the container window in case a method you supply needs this information. **(n)**

type

This argument is not used by the default processing methods. However, it passes one of the following (defined in **tools.def**) in case a method you supply needs this information: **(n)**

DC_LIST A standard Toolkit list or an ActiveX Toolkit list.

DC_WINDOW A Toolkit window.

child_id

This argument is not used by the default implementation of C_METHNOP. However it passes the ID for the child for other methods that need this information. **(n)**

reason

The reason the child was activated, which can be blank (indicating the child was already activated), a program-supplied reason passed because the child was activated programmatically, or one of the following: **(a)**

C_FIRST Focus moved in attempt to reach first child.

C_LAST Focus moved in attempt to reach last child.

C_NEXT Focus moved from previous child (e.g., TAB).

C_PREV Focus moved from next child (e.g., SHIFT+TAB).

C_SET

Child explicitly activated. For example, this could be the result of a mouse click, or it could be that the child was activated programmatically without a specified reason.

record

(optional) This argument is not used by the default method C_METHNOP. However, this argument passes a copy of the *a_methoddata* argument specified by *recordindex* (a DC_ADD argument) for other methods that need this information. **(a)**

a_methoddata1 through *a_methoddata20*

(optional) These arguments are not used by the default method C_METHNOP. However, up to 20 additional data arguments are passed for other methods that need this information. **(any)**

Discussion

C_METHNOP is a Toolkit-supplied processing method that C_PROCESS calls by default when processing a child window that doesn't fit into one of the general classifications that invoke other default processing methods (see the C_PROCESS [Discussion on page 6-42](#)) and for which no method was assigned when it was added to the composite window. For example, this method will be used for a child window created with .WINDOW, assuming no method was assigned in its DC_ADD call and assuming that T_SETUP has not been called for it (which would cause C_PROCESS to use C_METHTXT).

If you want something other than the default processing for such windows—which is simply to skip over the window (C_METHNOP calls %M_SIGNAL with “C_PREV” or “C_NEXT”)—you can instruct C_PROCESS to call a different routine when processing the window by passing the routine's name in the DC_ADD call for the window. Additionally, you can assign this method to any child window or list that you don't want to be activated.



To see the code for this method, see the **ctrmeths.dbl** file included in your UI Toolkit distribution. This file is distributed simply as a reference and a starting point for writing your own composite window processing methods; modifying or compiling this file will not change the default methods.

See also

- ▶ [“Controlling tabbing, focus, and input context” on page 6-4](#) for information on tabbing indices.
- ▶ [C_PROCESS on page 6-42](#).
- ▶ [“Overloading UI Toolkit Routines” on page 18-8](#) for information on overloading this subroutine.

C_METHSEL – Process a child selection window



```

subroutine C_METHSEL, reentrant
in      req container_id      , n
in      req type              , n
in      req child_id         , n
in      req reason           , a
opt record                   , a
opt a_methoddata1            , any
.
.
opt a_methoddata20           , any

```

Arguments

container_id

This argument is not used by the default processing methods. However, it passes the ID of the container window in case a method you supply needs this information. **(n)**

type

This argument is not used by the default processing methods. However, it passes one of the following (defined in **tools.def**) in case a method you supply needs this information: **(n)**

DC_LIST A standard Toolkit list or an ActiveX Toolkit list.

DC_WINDOW A Toolkit window.

child_id

The ID for the child selection window. **(n)**

reason

The reason the child was activated, which can be blank (indicating that the child was already activated), a program-supplied reason passed because the child was activated programmatically, or one of the following: **(a)**

C_FIRST Focus moved in attempt to reach first child.

C_LAST Focus moved in attempt to reach last child.

C_NEXT Focus moved from previous child (e.g., TAB).

C_PREV Focus moved from next child (e.g., SHIFT+TAB).

C_SET Child explicitly activated. For example, this could be the result of a mouse click, or it could be that the child was activated programmatically without a specified reason.

record

(optional) The data area for input for the child list. **(a)**

a_methoddata1 through *a_methoddata20*

(optional) These arguments are not used by the default method C_METHSEL. However, up to 20 additional data arguments are passed for other methods that need this information. **(any)**

Discussion

C_METHSEL is a Toolkit-supplied processing method that C_PROCESS calls by default when processing a child selection window. If you want something other than the default processing described below, you can instruct C_PROCESS to call a different routine by passing the routine's name in the DC_ADD call for the selection list.

C_METHSEL does the following:

1. Calls U_CTRCONTEXT to set context within the selection window.
2. Processes the selection window by calling S_SELECT and passing the ID of the selection window (*child_id*) and the *record* argument passed to this method.
3. If **g_select** is false when control is returned to C_METHSEL (indicating that an item was selected), C_METHSEL then calls %M_SIGNAL, passing "C_NEXT" as the menu entry name to signal. This moves context to the next child (or to the first button for the composite container window if there are no more children). If **g_select** is true, context remains on the selection window.

Note that for *record*, C_PROCESS passes the *a_methoddata* argument specified by the *recordindex* argument in the DC_ADD call for the window. For details, see the information on *recordindex* in the DC_ADD [Discussion on page 6-22](#), and see the information on the *a_methoddata* arguments in the C_PROCESS [Discussion on page 6-42](#).



To see the code for this method, see the **ctrmeths.dbl** file included in your UI Toolkit distribution. This file is distributed simply as a reference and a starting point for writing your own composite window processing methods; modifying or compiling this file will not change the default methods.

See also

- ▶ [“Controlling tabbing, focus, and input context” on page 6-4](#) for information on tabbing indices.
- ▶ [C_PROCESS on page 6-42](#).
- ▶ [“Overloading UI Toolkit Routines” on page 18-8](#) for information on overloading this subroutine.

C_METHTS – Process a child tabbed dialog



```
subroutine C_METHTS, reentrant
in    req container_id      , n
in    req type              , n
in    req child_id         , n
in    req reason           , a
opt record                 , a
opt a_methoddata1         , any
.
.
opt a_methoddata20        , any
```

Arguments

container_id

This argument is not used by the default processing methods. However, it passes the ID of the container window in case a method you supply needs this information. **(n)**

type

This argument is not used by the default processing methods. However, it passes one of the following (defined in **tools.def**) in case a method you supply needs this: **(n)**

DC_LIST A standard Toolkit list or an ActiveX Toolkit list.

DC_WINDOW A Toolkit window.

child_id

The ID for the child tabbed dialog. **(n)**

reason

The reason the child was activated, which can be blank (indicating that the child was already activated), a program-supplied reason passed because the child was activated programmatically, or one of the following: **(a)**

C_FIRST Focus moved in attempt to reach first child.

C_LAST Focus moved in attempt to reach last child.

C_NEXT Focus moved from previous child (e.g., TAB).

C_PREV Focus moved from next child (e.g., SHIFT+TAB).

C_SET Child explicitly activated. For example, this could be the result of a mouse click, or it could be that the child was activated programmatically without a specified reason.

record

(optional) This argument is not used by the default method C_METHAX. However, this argument passes a copy of the *a_methoddata* argument specified by *recordindex* (a DC_ADD argument) for other methods that need this information. (**a**)

a_methoddata1 through *a_methoddata20*

(optional) Up to 20 additional data arguments. (**any**)

Discussion

C_METHTS is a Toolkit-supplied processing method that C_PROCESS calls by default when processing a child window that is a tabbed dialog. If you want something other than the default processing described below, you can instruct C_PROCESS to call a different routine by passing the routine's name in the DC_ADD call for the list.

C_METHTS does the following:

1. Establishes focus for the tab set by passing the *reason* argument on to the DTS_CTRCONTEXT subfunction of %TS_TABSET. (Note that the DTS_CTRCONTEXT subfunction for %TS_TABSET is designed specifically to be used in processing methods for child windows that are tabbed dialogs. See [DTS_CTRCONTEXT](#) on page 13-36 for more information.)
2. Processes the tabbed dialog by calling %TS_PROCESS with the ID of the child window and all the *a_methoddata* arguments passed in the C_PROCESS call.



To see the code for this method, see the **ctrmeths.dbl** file included in your UI Toolkit distribution. This file is distributed simply as a reference and a starting point for writing your own composite window processing methods; modifying or compiling this file will not change the default methods.

See also

- ▶ [“Controlling tabbing, focus, and input context”](#) on page 6-4 for information on tabbing indices.
- ▶ [DC_ADD](#) on page 6-21.
- ▶ [C_PROCESS](#) on page 6-42.
- ▶ [“Overloading UI Toolkit Routines”](#) on page 18-8 for information on overloading this subroutine.

C_METHTXT – Process a child text window



```
subroutine C_METHTXT, reentrant
in      req container_id      , n
in      req type              , n
in      req child_id         , n
in      req reason           , a
opt record                   , a
opt a_methoddata1           , any
.
.
opt a_methoddata20          , any
```

Arguments

container_id

This argument is not used by the default processing methods. However, it passes the ID of the container window in case a method you supply needs this information. **(n)**

type

This argument is not used by the default processing methods. However, it passes one of the following (defined in **tools.def**) in case a method you supply needs this: **(n)**

DC_LIST A standard Toolkit list or an ActiveX Toolkit list.

DC_WINDOW A Toolkit window.

child_id

The ID for the child text window. **(n)**

reason

The reason the child was activated, which can be blank (indicating that the child was already activated), a program-supplied reason passed because the child was activated programmatically, or one of the following: **(a)**

C_FIRST Focus moved in attempt to reach first child.

C_LAST Focus moved in attempt to reach last child.

C_NEXT Focus moved from previous child (e.g., TAB).

C_PREV Focus moved from next child (e.g., SHIFT+TAB).

C_SET Child explicitly activated. For example, this could be the result of a mouse click, or it could be that the child was activated programmatically without a specified reason.

record

(optional) This argument is not used by the default method C_METHTXT. However, this argument passes a copy of the *a_methoddata* argument specified by *recordindex* (a DC_ADD argument) for other methods that need this information. **(a)**

a_methoddata1 through *a_methoddata20*

(optional) These arguments are not used by the default method C_METHTXT. However, up to 20 additional data arguments are passed for other methods that need this information. **(any)**

Discussion

C_METHTXT is a Toolkit-supplied processing method that C_PROCESS calls by default when processing a child text window. If you want something other than the default processing described below, you can instruct C_PROCESS to call a different routine by passing the routine's name in the DC_ADD call for the text window.

C_METHTXT does the following:

1. Calls U_CTRCONTEXT to set context within the selection window.
2. Calls T_EDIT for the window.



To see the code for this method, see the **ctrmeths.dbl** file included in your UI Toolkit distribution. This file is distributed simply as a reference and a starting point for writing your own composite window processing methods; modifying or compiling this file will not change the default methods.

See also

- ▶ [“Controlling tabbing, focus, and input context” on page 6-4](#) for information on tabbing indices.
- ▶ [DC_ADD on page 6-21](#).
- ▶ [C_PROCESS on page 6-42](#).
- ▶ [“Overloading UI Toolkit Routines” on page 18-8](#) for information on overloading this subroutine.

7

Environment Routines

Using Environments 7-2

Explains the concept of an environment and the terminology associated with it.

E_ENTER – Enter a new environment.....	7-3
E_EXIT – Exit to the previous environment	7-5
%E_FONT – Modify the font for the header, footer, or information line.....	7-6
%E_INFO – Return information about environment processing	7-8
E_METHOD – Register an environment method	7-35
E_SECT – Place text into a section of the display screen	7-38
E_STATE – Modify the state of the current environment	7-41

The Environment Methods 7-45

%EAPPMOVE_METHOD – Perform application move processing	7-46
%EAPPSIZE_METHOD – Perform application resize processing	7-47
%EAPPSTATE_METHOD – Perform application state processing	7-49
%ECENTURY_METHOD – Perform century processing	7-50
ECHKFLD_METHOD – Perform user field validation.....	7-53
ECLOSE_METHOD – Perform application close processing.....	7-59
EDSPFLD_METHOD – Perform user field display	7-62
EEDTDSP_METHOD – Perform user field editing	7-67
EENTRST_METHOD – Perform input override processing	7-70
EFKEY_METHOD – Perform key mapping	7-72
EHELP_METHOD – Perform help processing.....	7-73
ESCRIPTERR_METHOD – Perform script compilation error processing	7-80
EUTILS_METHOD – Perform utilities processing	7-82

Using Environments

An *environment* is a program state that consists of the current definition of the screen, any terminal settings and processing options, and open channels.

When you create (enter) a new environment, UI Toolkit saves the old environment and copies its attributes to the current one. Any change to an environment item supersedes the previous contents. When you delete (exit) the current environment, the previous environment is restored.

The advantage to using environments is to enable you to easily create and maintain “component” programs. For example, consider an inventory management application as a component. If inventory management enters a new environment upon entry, and exits it when returning to the calling program, the component is completely modular and it doesn’t matter who calls it. You can call the component not only from the main menu but also from the inventory receivings component or any other program.

A program can have more than one environment. An application moves from one environment to another with `E_ENTER`. Each new environment is at a new and deeper environment level. You can have a maximum number of 32 environment levels. You can control the maximum number of environment levels with the `U_START` subroutine. The default number of levels is 16.

When you exit an environment with the `E_EXIT` subroutine, UI Toolkit closes its current state (deletes locally created windows, closes locally opened files, deletes any new menus, and so forth). The current environment is then reset to the previous environment.

The current environment is maintained by UI Toolkit and owns the header, information, and footer lines. It also owns channels, state flags, windows, menu columns, lists, and possibly toolbars. Note that only toolbars created with `%TB_TKCREATE` or logged with `TB_TKLOG` are under the control of the environment.

The current display owns the menu bar and body of the screen as well as placement of the windows, menu columns, toolbars, and lists. The current display can be shared by more than one environment.

Windows, menu columns, channels, and toolbars are either global or local to an environment. These two statuses primarily determine deletion privileges: a local item is owned by the environment that defines it and can only be deleted by that particular environment, while a global item can be deleted at any environment level. Global items are owned by the environment system rather than a specific environment level. All local elements are automatically deleted when you exit the environment that owns them.

A display can also be designated as shared, which means that it is accessible by multiple environment levels (although it is still owned by the environment that defined it). A new environment whose display is defined as shared can access the previous environment’s menu bar, menu columns, and screen body. If you make changes to a shared display, those changes will be reflected in all environment levels that share the display.

E_ENTER – Enter a new environment



```
xcall E_ENTER (/D_SHARE | D_NEW/)
```

Arguments

D_SHARE

(optional) Shared display. **(n)**

D_NEW

(optional) New display. (default) **(n)**

Discussion

The E_ENTER subroutine defines a new environment and determines whether the display is shared.

Passing D_SHARE flags the display as shared. This means that the new environment accesses the previous environment's menu bar, menu columns, toolbars, and screen body. However, the display is owned by the environment that defined it, and although the new environment can modify the menu bar and toolbars, it cannot delete local menu columns, toolbars, or windows. If you modify a shared display, that modification will affect the previous environment.

If D_NEW is passed (or neither D_NEW nor D_SHARE is passed), a display is also created with the previous environment's menu bar, menu columns, toolbars, and screen body. However, if you make a modification (for example, removing a menu column or toolbar), that modification will not affect the previous environment.

E_ENTER performs the following operations:

- ▶ Saves the description of the current environment
- ▶ Defines new display screen sections and copies the contents of the previous environment's sections into the new sections
- ▶ If the display is not shared, defines a new menu bar and remembers which windows, toolbars, and columns were placed (also remembers the enabled/disabled state of menu columns and entries and toolbar buttons)
- ▶ Copies the previous environment's state flags
- ▶ Initializes the new local window and local channel lists to empty

You do not need to call E_ENTER after a U_START, as the U_START subroutine automatically enters the first environment.

Environment Routines

E_ENTER

Note that even though you define a new environment, all previous environments still exist. Suppose, for example, that you have passed some global file channels and windows created in the caller's current environment to a routine. Even if you enter a new environment, you can use the passed channels and windows. Although they are not owned by the new environment, they still exist and are accessible. The non-owning environment, however, cannot delete windows and channels.

Examples

This example defines a new environment with a shared display.

```
xcall e_enter(D_SHARE)
```

E_EXIT – Exit to the previous environment



```
xcall E_EXIT
```

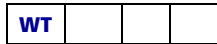
Discussion

The E_EXIT subroutine exits to the previous environment. E_EXIT performs the following operations:

- ▶ Deletes local windows, menu columns, and toolbars
- ▶ Purges local channels
- ▶ If the display was not shared with the previous level, restores the previous level's placed windows, menu columns, and toolbars. Note, however, that this does not apply to toolbar buttons. If a toolbar button is deleted, leaving the environment will not restore it.
- ▶ If a toolbar button was forced down (with the TBB_SELECT subfunction of %TB_BUTTON or because it is a member of a group) in a prior environment and then was popped up in the current environment, exiting the environment pushes the button down again. This also invokes the button's method and/or keystrokes associated with the left mouse button if any are registered.
- ▶ Restores the state of menu columns and menu entries, whether enabled or disabled, and the state of toolbar buttons, whether enabled or disabled. (Note that if a button was selected in a prior environment and then was deselected in the current environment, exiting the environment selects the button. This invokes the button's method and any registered keystrokes that are associated with the left mouse button.)
- ▶ Restores the previous level's header, footer, and information line

The UI Toolkit utilities remember only the placement and order of a placed window. If you modify any other characteristics (for example, color or contents), they will remain modified.

%E_FONT – Modify the font for the header, footer, or information line



```
prev_font = %E_FONT (section[, font_id])
```

or

```
xcall E_FONT (section[, font_id])
```

Return value

prev_font

On Windows, the font handle of the previously selected font for the *section*. On UNIX and OpenVMS, always returns zero. (^VAL)

Arguments

section

The section identifier: (**n**)

D_HEADER Header section.

D_FOOTER Footer section.

D_INFO Information line.

font_id

(optional) The name of a font palette entry, a font handle, or one of the following. (**a** or **n**)

DF_CURRENT Current font.

DF_DEFAULT Global font.

DF_ALTERNATE Alternate font.

DF_DEBUGGER Debugger font.

Discussion

The %E_FONT function modifies the font for the header, footer, or information line. This font information is used for the current environment only. It is not saved in **synergy.ini**. On Windows, the Windows theme in use determines the font for the title. (Toolkit settings for titles are ignored on Windows.)

If *font_id* is not passed, the font is not changed.

The initial values for the header, footer, and information line can be specified in **synergy.ini** or in the environment by setting the **FONT_HEADER**, **FONT_FOOTER**, and **FONT_INFO** environment variables. If no initial values are set, a font palette entry named “Status” is used. If a font palette entry named “Status” is not defined, one is created using typeface “Arial,” point size 10, and sizing character “A.” Note that the sizing character is not used for the font for these sections, but it would affect the font’s reuse in other Synergy windows.

Changing the font of a displayed screen section at runtime will cause the application window to be resized. If more than one of these fonts or the application window font are to be changed at the same time, we recommend that you call **U_UPDATE(FALSE)** prior to **%E_FONT** and **U_UPDATE(TRUE)** afterwards to avoid excessive repainting.

See also

- ▶ **FONT_HEADER**, **FONT_FOOTER**, and **FONT_INFO** in the “Environment Variables” chapter of *Environment Variables & System Options* for information on setting these environment variables in **synergy.ini**.
- ▶ “Using Fonts on Windows” on page 18-40 for information on fonts and font palettes.

Examples

The following example sets the font for the footer section to be **footer_font**, where **footer_font** is a font defined in **synergy.ini** or **synuser.ini** or defined at runtime by calling **%U_WNDFONT(D_SETFONT)**.

```
old_font = %e_font(D_FOOTER, "footer_font")
```

The following example sets the font for the information line to be the font specified by **font_hndl**. **Font_hndl** would have been returned from a call to **%U_WNDFONT(D_SETFONT)** when the font palette entry was created.

```
old_info_font = %e_font(D_INFO, font_hndl)
```

%E_INFO – Return information about environment processing



return = %E_INFO (*subfunction* [, *arg*, ...])

Return value

return

A value specific to the subfunction. (^VAL)

Arguments

subfunction

The specific function to perform. It must be one of the following: (n)

D_CHANNEL_LEVEL	Return the environment level at which a channel is logged.
D_CHANNEL_MAX	Return the highest channel number that Toolkit can use.
D_CHANNELS	Return the number of channels logged in a level.
D_DSP_LEVEL	Return the current display level.
D_ENV_LEVEL	Return the current environment level.
D_LIST_LEVEL	Return the environment level at which a list is logged.
D_LISTS	Return the number of lists logged in a level.
D_MAX_ENV_LEVEL	Return the maximum environment level.
D_METHOD	Return the address of the current method routine.
D_MONTH_ABBREV	Return true if successful; otherwise, false.
D_MONTH_NUMBER	Return the month number (1-12).
D_SECT_ROWS	Return the number of rows available in a section.
D_SECT_TEXT	Return the length of text in a section.
D_STATE	Return a true/false flag indicating whether a state code is on or off.
D_TOOLBAR_LEVEL	Return the environment level for a toolbar.
D_TOOLBARS	Return a list of toolbars at a given environment level.
D_WINDOW_LEVEL	Return the environment level at which a window is logged.
D_WINDOW_TYPE	Return the type of a window.
D_WINDOWS	Return the number of windows logged in a level.

arg

(optional) One or more arguments, depending on *subfunction*.

Discussion

The %E_INFO function is used to return information about the current environment. See the individual subfunctions for details.

Examples

The first example does the following: retrieves the total number of all windows in order to allocate an array to hold their IDs, determines which windows are input windows, and then displays all the IDs of the input windows.

```
.align
stack record
    array_hndl    ,i4          ;Handle to an allocated array
    nmwnds        ,i4          ;Number of windows in the array
    ix            ,i4          ;An index into the array
    wndid         ,i4          ;A window ID

structure window_array          ;Structure for the allocated memory
    id            ,i4          ;A window ID

proc
    nmwnds = %e_info(D_WINDOWS,,, D_ALL) ;Get number of windows
    array_hndl = %mem_proc(DM_ALLOC, nmwnds * ^size(window_array))
                                ;Allocate the array
    xcall e_info(D_WINDOWS, nmwnds, ^m(window_array.id, array_hndl),
    &          D_ALL)           ;Retrieve the IDs
    for ix from 1 thru nmwnds   ;Loop through the ID's
    begin
        wndid = ^m(window_array[ix].id, array_hndl);Get one
                                ;Is it an input window?
        if (%e_info(D_WINDOW_TYPE, wndid) .eq. D_WTYPE_INPUT)
            xcall u_message("ID " + %string(wndid) +
            &          " is an input window")
        end
    xcall mem_proc(DM_FREE, array_hndl) ;Free the array
```

The following example is similar to the first, but it uses the maximum array size.

```
.include "WND:system.def"
.align
stack record
    array        , [D_MAXWND] i4      ;Array of window IDs
    nmwnds        , i4                ;Number of windows
    ix            , i4                ;Index into the array
```

Environment Routines

%E_INFO

```
proc
                                ;Retrieve the IDs and the count
nmwnds = %e_info(D_WINDOWS, D_MAXWND, array, D_ALL)
for ix from 1 thru nmwnds      ;Loop thru the IDs
begin
                                ;Is it an input window?
if (%e_info(D_WINDOW_TYPE, array[ix]) .eq. D_WTYPE_INPUT)
begin
    xcall u_message("ID " + %string(array[ix]) +
        & " is an input window")
end
end
end
```

In the next example, an EUTILS_METHOD environment method calls any previously registered method if it does not know how to handle a menu entry.

```
subroutine my_utils_method,reentrant
;
; Description: A utility menu entry method that can inherit from
;             the previously registered method.
;
; Arguments:
;     a_record      ,a          ;(Optional) record passed to I_INPUT
;
.include "WND:tools.def"

.align
stack record
    my_method      ,D_ADDR      ;Address of this routine
    prev_method    ,D_ADDR      ;Address of previous level routine
    lvl            ,i4          ;An environment level index
proc
    using g_entnam select      ;Do we know this entry?
("U_ENTRY1 "),call entry1
("U_ENTRY2 "),call entry2
()          ,call unknown      ;No, let's pass it back to previous method
    endusing
    xreturn

entry1,
                                ;(Code to process entry U_ENTRY1)
    return

entry2,
                                ;(Code to process entry U_ENTRY2)
    return

unknown,
                                ;First, get our address to compare against
    my_method = %e_info(D_METHOD, D_METH_UTILS)
```

```
;Now, beginning with the previous level, find the method that was
; registered prior to this one
for lvl from (%e_info(D_ENV_LEVEL) - 1) thru 1 by -1
begin
    prev_method = %e_info(D_METHOD, D_METH_UTILS, lvl)
                                ;Get the method for level lvl
    if (prev_method .ne. my_method)
        begin                    ;Back to before my_method was registered
            if (prev_method) ;Zero means "no method"
                xcall xsubr(prev_method, a_record)
                                ;Invoke previous method, if one was
                                ; registered
            exitloop            ;Don't invoke any others
        end
    end
    return
endsubroutine
```

Environment Routines

%E_INFO

D_CHANNEL_LEVEL

env_level = %E_INFO (D_CHANNEL_LEVEL, *channel*)

Return value

env_level

The environment level at which the channel is logged or one of the following: (^VAL)

D_FREE No channel is logged with the specified ID.

D_GLOBAL The channel is global.

D_SYS The channel is owned by the Toolkit system processing.

Arguments

channel

The channel number. (**n**)

Discussion

The D_CHANNEL_LEVEL subfunction obtains the environment level at which the specified channel is logged.

If the *channel* is invalid, a fatal error occurs.

D_CHANNEL_MAX

max_channel = %E_INFO (D_CHANNEL_MAX)

Return value

max_channel

The highest channel number that Toolkit can use. (^VAL)

Discussion

The D_CHANNELS subfunction returns the highest channel number that Toolkit can use. This limit is set by U_START.

D_CHANNELS

channels_logged = %E_INFO (D_CHANNELS, [*channels_req*], [*channels*][, *env_level*])

or

xcall E_INFO (D_CHANNELS, [*channels_req*], [*channels*][, *env_level*])

Return value

channels_logged

The number of channels logged. (^VAL)

Arguments

channels_req

(optional) The number of channels requested. (n)

channels

(optional) The returned array of channel numbers. (n)

env_level

(optional) The environment level (a value between 1 and the current environment), or one of the following. (n)

D_ALL All environment levels including global.

D_GLOBAL The global environment level.

D_LOCAL The current environment level. (default)

Discussion

The D_CHANNELS subfunction obtains the number of channels logged.

Because *channels_req* and *channels* are optional and the *channels_logged* return value indicates the size needed to contain *channels*, you can call D_CHANNELS once without passing *channels_req* and *channels* to get the necessary size. Then, allocate the required space before calling the subfunction again, passing *channels_req* and *channels* to obtain the *array of channel numbers*.

In the case of an array return argument like *channels*, the number of elements requested can be specified in the *channels_req* argument. This allows you to avoid overrunning the array boundary if the array passed is smaller than the total number of items. The array reference passed should refer to the array's first element. %E_INFO will treat this as the base of a pseudo array into which to write the returned channels.

Instead of sizing the *channels* array dynamically, the calling routine can allocate the maximum possible size by passing D_MAXCHN for *channels_req*. D_MAXCHN is defined in **system.def**.

If *env_level* is not passed or 0, D_LOCAL is assumed. If a value is passed that is greater than the current environment level, a fatal error occurs.

D_DSP_LEVEL

dsp_level = %E_INFO (D_DSP_LEVEL)

Return value

dsp_level

The current display level. (^VAL)

Discussion

The D_DSP_LEVEL subfunction obtains the current display level.

Environment Routines

%E_INFO

D_ENV_LEVEL

```
env_level = %E_INFO(D_ENV_LEVEL)
```

Return value

env_level

The current environment level, or 0 if Toolkit is not active. (^VAL)

Discussion

The D_ENV_LEVEL subfunction enables you to determine if Toolkit is active (i.e., if it has been initialized with U_START but not terminated with U_FINISH). And if Toolkit is active, this subfunction returns the current environment level.

Examples

The following example checks whether or not you can enter a new environment.

```
if (%e_info(D_ENV_LEVEL) .le. %e_info(D_MAX_ENV_LEVEL)) then
  xcall e_enter
else
  xcall u_message("Cannot enter a new environment - " +
    & "maximum levels already in use")
```

The next example determines whether Toolkit is active and initializes it if it's not active:

```
if (!%e_info(D_ENV_LEVEL)) then
  xcall u_start
.
.
.
```

Also see [Examples on page 7-9](#).

D_LIST_LEVEL

env_level = %E_INFO (D_LIST_LEVEL, *list_id*)

Return value

env_level

The environment level at which *list_id* is logged or one of the following: (^VAL)

D_FREE No list is logged with the specified ID.

D_GLOBAL The list is global.

D_SYS The list is owned by the Toolkit system processing.

Arguments

list_id

The list ID. (n)

Discussion

The D_LIST_LEVEL subfunction obtains the environment level at which the specified list is logged.

If *list_id* is invalid, a fatal error occurs.

D_LISTS

lists_logged = %E_INFO(D_LISTS, [*lists_req*], [*list_id*][, *env_level*])

or

xcall E_INFO(D_LISTS, [*lists_req*], [*list_id*][, *env_level*])

Return value

lists_logged

The number of lists logged at the specified level. (^VAL)

Arguments

lists_req

(optional) The number of lists requested. (n)

list_ids

(optional) The returned array of list IDs. (n)

env_level

(optional) The environment level, or one of the following. (n)

D_ALL All environment levels including global.

D_GLOBAL The global environment level.

D_LOCAL The current environment level. (default)

Discussion

The D_LISTS subfunction obtains the number of lists logged in a level.

Because the *lists_req* and *list_ids* arguments are optional and the *lists_logged* return value indicates the size needed to contain the *list_ids* argument, %E_INFO(D_LISTS) can be called once without passing *lists_req* and *list_ids* to get the necessary size. Then, allocate the required space before calling the D_LISTS subfunction again, passing *lists_req* and *list_ids* to obtain the *array of list IDs*.

In the case of an array return argument like *list_ids*, the number of elements requested can be specified in the *lists_req* argument. This allows you to avoid overrunning the array boundary if the array passed is smaller than the total number of items. The array reference passed should refer to the array's first element. %E_INFO will treat this as the base of a pseudo array into which to write the returned lists.

Instead of sizing the *list_ids* array dynamically, the calling routine can also allocate the maximum possible size by passing D_MAXLST for *lists_req*. D_MAXLST is defined in **WND: system.def**.

D_MAX_ENV_LEVEL

max_env_level = %E_INFO(D_MAX_ENV_LEVEL)

Return value

max_env_level

The maximum environment level. (^VAL)

Discussion

The D_MAX_ENV_LEVEL subfunction obtains the maximum environment levels specified at the time that U_START was called. This is usually D_DFLTENV, but it can be as high as D_MAXENV. Both of these constants are defined in **WND: system.def**.

Examples

The following example checks whether or not you can enter a new environment.

```
if (%e_info(D_ENV_LEVEL) .le. %e_info(D_MAX_ENV_LEVEL)) then
    xcall e_enter
else
    xcall u_message("Cannot enter a new environment - " +
        & "maximum levels already in use")
```

D_METHOD

method_addr = %E_INFO (D_METHOD, *method_code*[, *env_level*])

Return value

method_addr

The address (D_ADDR) of the specified method routine if registered, or 0 if not registered. (^VAL)

Arguments

method_code

Get one of the following addresses: (n)

D_METH_APPCLOSE	Application close method.
D_METH_APPMOVE	Application move method.
D_METH_APPSIZE	Application resize method.
D_METH_APPSTATE	Application state method.
D_METH_CENTURY	Century method.
D_METH_CHKFLD	User field validation method.
D_METH_DSPFLD	User field display method.
D_METH_EDTDS	User field edit display method.
D_METH_FKEY	Key mapping method.
D_METH_HELP	Help method.
D_METH_SCRIPTERR	Script compilation error method.
D_METH_UTILS	Utilities method.

env_level

(optional) Either the environment level or **D_LOCAL** (for current environment level; default). (n)

Discussion

The D_METHOD subfunction obtains the address of the routine defined in *method_code*.

This subfunction is useful for nested components where each component provides a method for an environment-level event. In this case, the innermost component can add functionality without overloading the method of the outer component. Prior to registering the method for the new environment, the component can save off the address of the previous method, then invoke it within the new method using %XSUBR. Note also that this address can be used to re-register the previous method without invoking E_EXIT.

Env_level can be a value between 1 and the current environment level or D_LOCAL. If *env_level* is not passed or 0, D_LOCAL is assumed. If a value is passed that is greater than the current environment level, a fatal error occurs.

See also

[E_METHOD on page 7-35](#) for information on registering methods.

Examples

See [Examples on page 7-9](#).

D_MONTH_ABBREV

status = %E_INFO (D_MONTH_ABBREV, *month_num*, *month_abbrev*)

Return value

status

True if successful; false (0) if not. (**^VAL**)

Arguments

month_num

The month number (1-12). (**n**)

month_abbrev

The returned month abbreviation as defined by UI Toolkit. (**a**)

Discussion

The D_MONTH_ABBREV subfunction obtains the month abbreviation for a specified month number.

If *month_num* is not between 1 and 12, this subfunction returns false and no month abbreviation is returned.

See also

[U_MONTHS on page 16-101](#) for assigning alpha characters to represent months.

D_MONTH_NUMBER

month_num = %E_INFO (D_MONTH_NUMBER, *month_abbrev*)

Return value

month_num

The month number (1-12) if found; otherwise, returns 0. (^**VAL**)

Arguments

month_abbrev

The month abbreviation as defined by UI Toolkit. (**a**)

Discussion

The D_MONTH_NUMBER subfunction obtains the month number for a specified month abbreviation.

If *month_abbrev* is not a valid month abbreviation as defined by UI Toolkit, this subfunction returns 0.

See also

[U_MONTHS on page 16-101](#) for assigning alpha characters to represent months.

Environment Routines

%E_INFO

D_SECT_ROWS

rows = %E_INFO (D_SECT_ROWS, *section*)

Return value

rows

The number of rows available in a section, or 0 if the section doesn't exist. (^**VAL**)

Arguments

section

One of the following sections: (**n**)

D_CAPTION Reference a Microsoft Windows caption.

D_FOOTER Reference the footer section.

D_HEADER Reference the header section.

D_INFO Reference the information line.

Discussion

The D_SECT_ROWS subfunction obtains the number of rows available in a specified section of the display screen.

If *section* doesn't exist in the display screen, this subfunction returns 0.

D_SECT_TEXT

text_length = %E_INFO (D_SECT_TEXT, [*text_buffer*], *section*, [*row*][, *env_level*])

or

xcall E_INFO (D_SECT_TEXT, [*text_buffer*], *section*, [*row*][, *env_level*])

Return value

text_length

The length of text in a section if successful; otherwise, returns 0. (^VAL)

Arguments

text_buffer

(optional) The returned text buffer. (a)

section

One of the following sections: (n)

D_CAPTION Reference a Microsoft Windows caption.

D_FOOTER Reference the footer section.

D_HEADER Reference the header section.

D_INFO Reference the information line.

row

(optional) The row number. (n)

env_level

(optional) Either the environment level or **D_LOCAL** (for current environment level; default). (n)

Discussion

The D_SECT_TEXT subfunction returns the entire text for a specified section within an environment.

In traditional Synergy on Windows the text may not be displayed as one string (for example, left-justified, centered, and right-justified sections), but within UI Toolkit it is stored internally as one logical string.

Because the *text_buffer* argument is optional and the *text_length* return value indicates the size needed to contain the *text_buffer* argument, %E_INFO (D_SECT_TEXT) can be called once without passing *text_buffer* to get the necessary size. Then, allocate the required space before calling the D_SECT_TEXT subfunction again, passing *text_buffer* to obtain the text.

If *section* doesn't exist in the display screen, this subfunction returns 0.

Environment Routines

%E_INFO

Env_level can be a value between 1 and the current environment level or D_LOCAL. If *env_level* is not passed or 0, D_LOCAL is assumed. If a value is passed that is greater than the current environment level, a fatal error occurs.

See also

[E_SECT on page 7-38](#) for information on placing text into a section of the display screen.

D_STATE

state_flag = %E_INFO (D_STATE, *state*[, *env_level*])

Return value

state_flag

A true/false flag indicating whether a state code is on or off. (^VAL)

Arguments

state

One of the following: (n)

D_CBADVANCE Advance context for combo boxes.

D_CURSOR Display a screen cursor.

D_ECHO Echo terminal input.

D_EURO Follow European formatting conventions.

D_INPPRCRND Apply the Input Field Processing or Read-Only Input Field Processing renditions to the field that has focus during input processing.

D_INPRNDOVER Override field-level rendition settings (color and attribute settings) with Input Field Processing or Read-Only Input Field Processing renditions.

D_INTR Allow interrupt characters.

D_LOCKWAIT Wait for locked records.

D_LOWER Allow lowercase terminal input.

D_MSGWAIT Wait for messages.

D_RETURNBTN RETURN (ENTER) key simulates a key press of the default button.

D_VALSTCHG Validate immediately on a state change of certain field types.

env_level

(optional) Either the environment level (a value between 1 and the current environment level) or **D_LOCAL** (for current environment level; default). (n)

Discussion

The D_STATE subfunction obtains the on/off status for a specified state code.

If *env_level* is not passed or 0, D_LOCAL is assumed. If the passed value is greater than the current environment level, a fatal error occurs.

See also

[E_STATE on page 7-41](#) for information on modifying the state of the current environment.

Environment Routines

%E_INFO

D_TOOLBAR_LEVEL

env_level = %E_INFO (D_TOOLBAR_LEVEL, *toolbar_id*)

Return value

env_level

The environment level at which a toolbar is logged, or one of the following: (^VAL)

D_FREE No toolbar is logged with the specified ID.

D_GLOBAL The toolbar is global.

Arguments

toolbar_id

The toolbar ID. (**n**)

Discussion

The D_TOOLBAR_LEVEL subfunction returns the environment level at which the specified toolbar is logged. If the toolbar was neither created nor logged with Toolkit, D_FREE is returned.

If *toolbar_id* is invalid, a fatal error occurs.

D_TOOLBARS

```
num_toolbars = %E_INFO (D_TOOLBARS, [toolbars_req], [toolbars_ids][, env_level])
```

or

```
xcall E_INFO (D_TOOLBARS, [toolbars_req], [toolbars_ids][, env_level])
```

Return value

num_toolbars

The number of toolbars logged for a given environment level. (^VAL)

Arguments

toolbars_req

(optional) The number of toolbars requested or **D_MAXTB** to allocate the maximum possible size for the returned array. (n)

toolbar_ids

(optional) The returned array of toolbar IDs. (n)

env_level

(optional) The environment level, or one of the following: (n)

D_ALL	All environment levels including global.
D_GLOBAL	The global environment level.
D_LOCAL	The current environment level. (default)

Discussion

The **D_TOOLBARS** subfunction obtains the number of toolbars logged for a given environment level.

Because the *toolbars_req* and *toolbar_ids* arguments are optional and the *num_toolbars* return value indicates the size needed to contain the *toolbar_ids* argument, **%E_INFO (D_TOOLBARS)** can be called once without passing *toolbars_req* and *toolbar_ids* to get the necessary size. Then, allocate the required space before calling the **D_TOOLBARS** subfunction again, passing *toolbars_req* and *toolbar_ids* to obtain the array of toolbar IDs.

The number of elements requested in the *toolbar_ids* array can be specified in the *toolbars_req* argument. This allows you to avoid overrunning the array boundary if the array passed is smaller than the total number of items. However, the array should have at least as many elements as the number passed for *toolbars_req*. The array reference passed should refer to the array's first element. **%E_INFO** will then treat this as the base of a pseudo array into which to write the returned toolbar IDs.

The calling routine can also allocate the maximum possible size by passing `D_MAXTB` for the *toolbars_req* argument and sizing the array to this same maximum. `D_MAXTB` is defined in **WND: system.def**.

Env_level can be a value (between 1 and the current environment level), `D_ALL`, `D_GLOBAL`, or `D_LOCAL`. If *env_level* is not passed or is passed as 0, `D_LOCAL` is assumed. If *env_level* is greater than the current environment level, a fatal error occurs.

D_WINDOW_LEVEL

env_level = %E_INFO (D_WINDOW_LEVEL, *window_id*)

Return value

env_level

The environment level at which the window is logged, or one of the following: (^VAL)

D_FREE No window is logged with the specified ID.

D_GLOBAL The window is global.

D_SYS The window is owned by the Toolkit system processing.

Arguments

window_id

The window ID. (**n**)

Discussion

The D_WINDOW_LEVEL subfunction obtains the environment level at which the specified window is logged.

If *window_id* is invalid, a fatal error occurs.

D_WINDOW_TYPE

window_type = %E_INFO (D_WINDOW_TYPE, *window_id*)

Return value

window_type

An identifier that specifies the window's type: (^VAL)

D_WTYPE_ACTIVEX	ActiveX container.
D_WTYPE_COLUMN	A menu column or subcolumn.
D_WTYPE_CONTAINER	A composite container window.
D_WTYPE_DOTNET	A .NET form container.
D_WTYPE_LISTCLASS	A list class.
D_WTYPE_INPUT	An input window.
D_WTYPE_SELECT	A selection window.
D_WTYPE_TABS	A tab set container.
D_WTYPE_TEXT	A text window.
D_WTYPE_OTHER	None of the above.

Arguments

window_id

The window ID whose type you want to retrieve. (n)

Discussion

The D_WINDOW_TYPE subfunction obtains the window type for a specified window ID.

More window types may be added in the future to the list of possible window types. Thus, D_WTYPE_OTHER could at some point encompass different types of windows and should therefore only be used as an “else” case.

Examples

See [Examples on page 7-9](#).

D_WINDOWS

```
num_windows = %E_INFO (D_WINDOWS, [windows_req], [window_ids][, env_level])
```

or

```
xcall E_INFO (D_WINDOWS, [windows_req], [window_ids][, env_level])
```

Return value

num_windows

The number of windows logged in a level. (^VAL)

Arguments

windows_req

(optional) The number of windows requested. (n)

window_ids

(optional) The returned array of window IDs. (n)

env_level

(optional) The environment level, or one of the following: (n)

D_ALL All environment levels including global.

D_GLOBAL The global environment level.

D_LOCAL The current environment level. (default)

Discussion

The D_WINDOWS subfunction obtains the number of windows logged in a level.

Because *windows_req* and *window_ids* are optional and the *num_windows* return value indicates the size needed to contain *window_ids*, you can D_WINDOWS once without passing *windows_req* and *window_ids* to get the necessary size. Then, allocate the required space before calling the subfunction again, passing *windows_req* and *window_ids* to obtain the *array of window IDs*.

In the case of an array return argument like *window_ids*, the number of elements requested can be specified in the *windows_req* argument. This allows you to avoid overrunning the array boundary if the array passed is smaller than the total number of items. The array reference passed should refer to the array's first element. %E_INFO will treat this as the base of a pseudo array into which to write the returned windows.

Instead of sizing the *window_ids* array dynamically, the calling routine can also allocate the maximum possible size by passing D_MAXWND for the *windows_req* argument. D_MAXWND is defined in **WND: system.def**.

Environment Routines

%E_INFO

Env_level can be a value between 1 and the current environment level, D_ALL, D_GLOBAL, or D_LOCAL. If *env_level* is not passed or 0, D_LOCAL is assumed. If a value is passed that is greater than the current environment level, a fatal error occurs.

Examples

See [Examples on page 7-9](#).

E_METHOD – Register an environment method



```
xcall E_METHOD (event_method_pair[, ...])
```

Arguments

event_method_pair

Has the format:

method_option[, method|library]

where *method_option* is one of the following environment method options:

D_METH_APPCLOSE	Specify the application close method, which is the subroutine that is called when U_FINISH is called or when the application is closed using the System menu, close box, or ALT+F4.
D_METH_APPMOVE	Specify the application move method, which is the function that is called when the application window is moved.
D_METH_APPSIZE	Specify the application size method, which is the function that is called when the application window is resized.
D_METH_APPSTATE	Specify the application state method, which is the function that is called when the application window state is changed.
D_METH_CENTURY	Specify the century method, which is the function that is called by U_DCDDAT whenever it needs to supply a default century. (Note that U_DCDDAT is called within the input processor to handle date defaulting.) If not specified, the default is the current century.
D_METH_CHKFLD	Specify the user field validation method, which is the subroutine that is called to validate the contents of the current user-defined input field.
D_METH_DSPFLD	Specify the user field display method, which is the subroutine that is called to display the contents of the current field if it is a user-defined input field.
D_METH_EDTDSP	Specify the user field edit display method, which is the subroutine that is called to display the editable contents of a user-defined field differently than both the way it is stored and the way the D_METH_DSPFLD method displays it.
D_METH_ENTRST	Specify the input override method, which is the subroutine that is called when an input field is exited due to a menu entry.

D_METH_FKEY	Specify the key mapping method, which is the subroutine that is called whenever a function key or control character is entered by the user.
D_METH_HELP	Specify the help method, which is the subroutine that is called when the user selects the menu entry with the internal name “O_HELP”.
D_METH_LIBRARY	Specify the library for in which Toolkit will search for methods for subsequent event method pairs.
D_METH_SCRIPTERR	Specify the script compilation error method, which is the subroutine to be called whenever the Script compiler external routine interface encounters a script compilation error.
D_METH_UTILS	Specify the utilities method, which is the subroutine that is called when the user selects a menu entry whose internal name begins with “U_”.

method

(optional) The name or integer address of the method to be used for the environment-level event specified with *method_option*, or a value of 0, which resets the method pair so that no method is associated with the specified method. (Used for all *method_option* options except D_METH_LIBRARY, which works with *library*.) (**D_ADDR**)

library

(optional) The ELB or shared image that contains the methods for subsequent event method pairs. (Used only with D_METH_LIBRARY.) (**a**)

Discussion

E_METHOD registers method subroutine names for environment-level events. Use E_METHOD if you want to do additional processing when an environment-level event is initiated. When that event occurs, Toolkit will call *method*, which is a routine you write, and if *library* is specified prior to the method, Toolkit will look in the specified library for the method.

You may pass multiple environment method options in a call, each followed by its optional *method* and *library*. See the examples below.

If passed, *method* must be either the name or the address of a routine. If the integer address is specified, such as the return value from %XADDR, it must be of type D_ADDR (which is defined in **dbl.def** as either **i4** or **i8**, depending on the system). Note that Toolkit immediately locates the method routine (unlike input field methods and list methods, which aren’t located until they need to be called).

If you omit *method* from an *event_method_pair* (any but a D_METH_LIBRARY pair), Toolkit resets the method pair, so that no method is associated with the specified event.

To enable the application to be closed with the System menu, the close box, or ALT+F4, you must either specify an application close method (D_METH_APPCLOSE) or use E_STATE to set the D_INTR state to allow interrupt characters.

Note the following for D_METH_LIBRARY and *library*:

- ▶ *Library* is passed on to %XADDR as the *elb* argument, which means that on Windows and UNIX you can use an environment variable to specify the path—for example, “ELB_DIR:elb_name”. On OpenVMS, however, *library* can be either the filename of a shared image located in SYS\$SHARE or a logical name that refers to a shared image, which may reside anywhere (so if *library* consists of a path and a name, similar to the example above, the path is ignored). For more information, see %XADDR in the “System-Supplied Subroutines and Functions” chapter of the *Synergy DBL Language Reference Manual*.
- ▶ A *library* specification applies only to methods specified by name after the *library* specification in the same E_METHOD call. (*Library* is not used to look up *method* arguments passed as addresses, which represent routines that have already been located.) If there’s more than one *library* setting in a call, each one affects only the methods specified between it and the next *library* setting.
- ▶ If you don’t pass D_METH_LIBRARY, Toolkit uses the default mechanism for finding methods: it selects the first method it finds with the correct name in the program image or in a previously opened ELB or shared image.
- ▶ If you pass D_METH_LIBRARY without passing *library*, Toolkit returns to the default mechanism (described above) for finding methods for subsequently specified event method pairs. (In other words, it ignores previously specified D_METH_LIBRARY pairs.)

See also

- ▶ “The Environment Methods” on page 7-45.
- ▶ Chapter 2, “Script,” for information on Script compilation errors.
- ▶ “Appendix D: Methods”.

Examples

The following example registers “ap_close” as the application close method. Because no library is specified, Toolkit will look for that routine in the program image or in a previously opened ELB.

```
xcall e_method(D_METH_APPCLOSE, "ap_close")
```

The next example specifies the method by passing its address.

```
xcall e_method(D_METH_APPCLOSE, %xaddr("ap_close"))
```

The following example registers “mydspfld” as the user field display method subroutine. This subroutine is to be found in the **userlib.elb** executable library.

```
xcall e_method(D_METH_LIBRARY, "userlib.elb", D_METH_DSPFLD, "mydspfld")
```

E_SECT – Place text into a section of the display screen



```
xcall E_SECT (text[, text_flag, ...])
```

Arguments

text

Text to display. (**a**)

text_flag

(optional) One or more of the following text flags: (**n**)

D_HEADER	Reference the header section.
D_CAPTION	Reference a Microsoft Windows caption.
D_FOOTER	Reference the footer section.
D_INFO	Reference the information line. (default)
D_POSITION , <i>row</i> , <i>col</i>	Specify the text's position by row and column.
D_NOCLEAR	Don't clear the line of text.
D_CLEAR	Clear entire line of text. (default)
D_CLREOL	Clear to end of line.
D_CLRBOL	Clear to beginning of line.
D_DISPLAY	Display after last text. (default)
D_LEFT	Left-justify the text.
D_CENTER	Center the text.
D_RIGHT	Right-justify the text.
D_VLINE	Convert “ ” characters to vertical lines. (default)
D_NOVLINE	Don't convert “ ” characters to vertical lines.

Discussion

The E_SECT subroutine places text into the various display screen sections (header, footer, and information line) in the current environment.

The *text* argument contains the text that will be displayed in one of the screen sections. The *text_flag* arguments define where and how that text will be displayed. You can have a maximum of six *text_flag* arguments.

WIN

In a Windows environment, if your header section is exactly one line in size, the application's window caption is used to display the header. Use `D_HEADER` or `D_CAPTION` to set or modify the header text. For compatibility between all environments, we recommend that you always use `D_HEADER`. The maximum visible length of header or caption text is 78 characters.

If your header section is more than one line in size, an area below the application's window caption will be used for displaying the header. Use `D_HEADER` to access this area. Use `D_CAPTION` to set and modify an additional line of text in the application's window caption. The Windows theme in use determines the location of the title (Toolkit settings for titles are ignored on Windows).

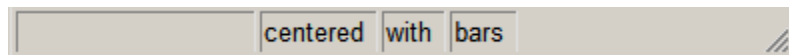
Microsoft Windows-style status bars are used for the information line and footer in the application window. These status bars are created outside the client area, allowing more space for your application. The information line and footer float with the bottom of the application window when the window is resized. Only one-line footers are supported.

The `D_LEFT`, `D_RIGHT`, and `D_CENTER` options divide the status bar into sections and specify the justification for text. Text is justified according to the option, but the status bar is divided differently depending on whether the text includes vertical bars and the `D_VLINE` option is set:

- ▶ If the `D_VLINE` option is set, each section whose text contains vertical bars is divided with borders that are close to the left side of text. For example,

```
e_sect("centered|with|bars", D_INFO, D_CENTER, D_CLEAR)
```

will result in a status line with something like the following:



- ▶ If the `D_VLINE` option is not set or the text does not contain vertical bars, the status bar is generally divided evenly in half or thirds if you specify two or all three of these options.

The information line is loaded by default if `D_HEADER`, `D_CAPTION` or `D_FOOTER` is not specified.

The `D_POSITION` argument redefines the current position of the text by specifying the section row and column where the text should begin. This redefinition occurs before any clearing or justification. The default display position (`D_DISPLAY`) for the section text is after the last text placed into that section.

After positioning, the routine takes care of clearing the text. The entire line of text is cleared if none of the clearing function codes (`D_NOCLEAR`, `D_CLEAR`, `D_CLREOL`, or `D_CLRBOL`) are specified.

Justification occurs next. If `D_LEFT`, `D_CENTER`, or `D_RIGHT` is not specified, the text is displayed at the current display position (`D_DISPLAY`).

Next, if `D_NOVLINE` is not passed, “|” characters will be converted to vertical lines.

Finally, the routine outputs text at the current position and updates the current position. The text is displayed in the current default rendition for the specified screen section.

If you attempt to place text in a section that does not exist (because it was not defined at the `U_START` call), a fatal error occurs.

Examples

The following example displays the quoted text at the left side of the information line (if there was no prior display to the information line). The line is cleared to blanks and the “i” character is converted to a vertical line.

```
xcall e_sect("RETURN to continue|^Z to exit")
```

This example centers the contents of **nam(i:j)** in the second line of the header section. The line is cleared to blanks and the “i” character is converted to a vertical line if it occurs in **nam**.

```
xcall e_sect(nam(i:j), D_HEADER, D_CENTER, D_POSITION, 2, 1)
```

The next example places the contents of **pagnm** at display row 2 and column 60 of the footer section. The line that will contain the text is not cleared, and any “i” characters are converted to vertical lines.

```
xcall e_sect(pagnm, D_FOOTER, D_NOCLEAR, D_POSITION, 2, 60)
```

The following example changes the Windows environment application caption to “Accounts Receivable” whether or not the caption is being used for the header section. This statement would be ignored in all other environments.

```
xcall e_sect("Accounts Receivable", D_CAPTION)
```


E_STATE – Modify the state of the current environment



```
xcall E_STATE([state, ...])
```

Arguments

state

(optional) One or more of the following codes. The meaning of each is dependent on the preceding arguments. (**n**)

D_ON	Turn on (set) options. (default)
D_OFF	Turn off (reset) options.
D_CBADVANCE	Advance context for combo boxes.
D_CURSOR	Display a screen cursor.
D_ECHO	Echo terminal input.
D_EURO	Allow European formatting conventions.
D_INPPRCRND	Apply the Input Field Processing or Read-Only Input Field Processing renditions to the field that has focus during input processing.
D_INPRNDOVER	Override field-level rendition settings (color and attribute settings) with Input Field Processing or Read-Only Input Field Processing renditions. (D_INPPRCRND must also be set to D_ON for this to work.)
D_INTR	Allow interrupt characters.
D_LOCKWAIT	Wait for locked records.
D_LOWER	Allow lowercase terminal input.
D_MSGWAIT	Wait for messages.
D_RETURNBTN	RETURN (ENTER) key simulates a key press of the default button.
D_VALSTCHG	Validate immediately on a state change of certain field types.

Discussion

E_STATE sets or modifies the following states or behavior for the current environment:

- ▶ Synergy DBL FLAGS subroutine states
- ▶ Synergy DBL DFLAG subroutine states
- ▶ Synergy low-level window system states
- ▶ UI Toolkit behavior

The D_ON and D_OFF codes indicate whether the codes that follow in the argument list are to be turned on or off. D_ON is the default for any code that is passed as a *state* argument. (For example, if you pass D_INPRNDOVER without passing D_ON or D_OFF, D_INPRNDOVER is set to D_ON, even though it is turned off by default.)

U_START sets all environment states to their defaults. Use E_STATE after calling U_START to modify these states.

Codes D_EURO, D_LOWER, D_ECHO, and D_INTR refer to controls 1, 2, 5, and 8 of the FLAGS subroutine, respectively. The codes D_MSGWAIT and D_LOCKWAIT refer to controls 2 and 3 of the DFLAG subroutine, respectively. The D_CURSOR code refers to the WP_CURSOR function state of the W_PROC subroutine.

WIN

When D_CBADVANCE is turned on, which it is by default, context will advance when the user selects an entry in a combo box by using its drop-down list. If the list is not dropped down, or if D_CBADVANCE is turned off, the context will remain on that field. If a user presses ENTER or the TAB key while in a combo box (the field or the drop-down list), context is advanced to the next field regardless of the D_CBADVANCE setting. If you always want the context to advance, regardless of how the user selects the item, you will need to handle this by using I_NEXT.

If you disable interrupt characters using the following combination: D_OFF, D_INTR, the close box and the “Close” entry (in the Windows system menu), your Windows application will also be disabled. To provide true Windows look-and-feel in your application, you should *not* disable interrupt characters, and you should support an application close method that performs a clean shutdown of your application.

D_RETURNBTN determines the behavior the RETURN (ENTER) key will have on Windows systems. It is ignored on UNIX and OpenVMS systems. If D_RETURNBTN is set, pressing the RETURN (ENTER) key will simulate pressing the default button. The default button is visually indicated by a thicker border around the button. If the default button is not specified, it will be the first button in the tabbing order. In this case, when a window has no buttons but its tab set container does, the first button on the tab set is the default button. A tab set cannot have a default button when a list is on the current tab.

If a tab set container and its contained window both have buttons, the following rules are used to determine which button is the default:

- ▶ If only one, either the contained window or the tab set container, defines a default button, the specified button is the default button.
 - ▶ If neither the contained window nor the tab set container defines a default button, the first button on the contained window is the default button.
 - ▶ If both the contained object and the tab set container define a default button, the default button of the contained window is the default button.
-

When D_INPPRCRND is set to D_ON, Toolkit applies the Input Field Processing or Read-Only Input Field Processing renditions to the currently active input field during I_INPUT or I_INPFLD processing. When D_INPPRCRND is set to D_OFF, Toolkit does not change renditions for the active field. (By default, D_INPPRCRND is set to D_OFF on Windows and D_ON on UNIX and OpenVMS.)

For information on renditions see [“Renditions” on page 18-25](#).

Note the following:

- ▶ To instruct Toolkit to override field-level rendition settings (.FIELD, I_FLDMOD, and IB_FIELD settings) with Input Field Processing renditions, set D_INPRNDOVER and D_INPPRCRND to D_ON. By default, D_INPRNDOVER is set to off, so field-level rendition settings override the Input Field Processing renditions.
- ▶ On Windows, D_INPPRCRND does not affect screen updates caused by calls to W_DISP. (So neither Input Field Processing nor Read-Only Input Field Processing renditions are used in this case.)

The D_VALSTCHG code allows immediate validation of check boxes, radio buttons, and combo boxes. Immediate validation occurs when a check box is toggled, the selected entry of a group of radio buttons is changed or the selected item in a combo box is changed. Validation means that the entry will pass through any Toolkit tests as well as any change method specified, and, if it passes, the associated data area will be updated with the new contents and any break processing would then occur. Focus will remain on this field unless the change method or break processing forces a change of focus (for instance, by calling I_NEXT) or D_CBADVANCE is specified. Likewise, leave and arrive methods will not be invoked unless such a change of focus occurs.

To enable or disable this option *per* field, toggle the option in the field’s arrive method.

E_STATE can have a total of 10 *state* arguments.

A special case occurs when you don’t pass any arguments to this subroutine. In this case, all environment states are assumed to be unknown, and they are set or reset according to the current environment level.

You must use this subroutine exclusively to manipulate the specified states. If you directly manipulate them in any other way, the states as they are “known” by the UI Toolkit environment may be incorrect.

The E_STATE subroutine does not affect displays during input processing. (.FIELD controls the displays in this situation.) For example, during input processing, the echo state won’t be altered.

Any E_STATE option can be set within private environments, and E_EXIT will revert the option to the value defined in the previous environment level.

See also

- ▶ [E_METHOD](#) on page 7-35, [U_FINISH](#) on page 16-46, and [ECLOSE_METHOD](#) on page 7-59 for more information about shutdown operations in Windows environments.
- ▶ [FLAGS](#) and [DFLAG](#) in the “System-Supplied Subroutines and Functions” chapter of the *Synergy DBL Language Reference Manual*.
- ▶ [W_PROC](#) in the “Synergy Windowing API” chapter of the *Synergy DBL Language Reference Manual*.
- ▶ [.BUTTON_SET](#) on page 2-17, [B_BUTTONSET](#) on page 5-14, [IB_BUTTONSET](#) on page 8-90, [L_BUTTONSET](#) on page 9-62, or [%TS_TABSET](#) on page 13-7 for specifying the default button.

Examples

This example disallows lowercase on terminal input and turns the screen cursor off.

```
xcall e_state(D_OFF, D_LOWER, D_CURSOR)
```

This example disables terminal interrupt characters (CTRL+C on most systems), turns the screen cursor on, and enables lowercase input.

```
xcall e_state(D_OFF, D_INTR, D_ON, D_CURSOR, D_LOWER)
```

The Environment Methods

This section contains specification pages for environment method subroutines. These method subroutines are written by you, and their names specified as arguments to E_METHOD. UI Toolkit then calls them at the appropriate time.

Method subroutines are important in applications that will run in a Windows environment.

[%EAPPMOVE_METHOD](#) – Perform application move processing

[%EAPPSIZE_METHOD](#) – Perform application resize processing

[%EAPPSTATE_METHOD](#) – Perform application state processing

[%ECENTURY_METHOD](#) – Perform century processing

[ECHKFLD_METHOD](#) – Perform user field validation

[ECLOSE_METHOD](#) – Perform application close processing

[EDSPFLD_METHOD](#) – Perform user field display

[EEDTDSP_METHOD](#) – Perform user field editing

[EENTRST_METHOD](#) – Perform input override processing

[EFKEY_METHOD](#) – Perform key mapping

[EHELP_METHOD](#) – Perform help processing

[ESCRPTERR_METHOD](#) – Perform script compilation error processing

[EUTILS_METHOD](#) – Perform utilities processing

%EAPPMOVE_METHOD – Perform application move processing

WT			

```
function EAPPMOVE_METHOD, ^val
    a_x          , n
    a_y          , n
```

Return value

This function should return true (1), so that other return values may be used in the future. **(n)**

Arguments

a_x

The pixel position in the x-direction. **(n)**

a_y

The pixel position in the y-direction. **(n)**

Discussion

%EAPPMOVE_METHOD is a function that you write *and name*. The name of your function is registered with UI Toolkit using the E_METHOD subroutine.

UI Toolkit calls your %EAPPMOVE_METHOD function when the application window is moved.

The pixel coordinates *a_x* and *a_y* correspond to the position of the upper-left corner of the application window's frame, relative to the entire screen.

Because Toolkit invokes this function as a direct callback, the code in this function should be kept to a minimum, and operations that could conflict with concurrent processing avoided. For example, any action causing the rebuilding of an input set (such as I_SETDEL) must be avoided because this would give the user the opportunity to create a fatal error condition in I_INPUT. Instead, use %M_SIGNAL to synchronize processing for all such cases.

See also

- ▶ [“Overloading UI Toolkit Routines” on page 18-8](#) for information on creating this function.
- ▶ [“Appendix D: Methods”](#).

%EAPPSIZE_METHOD – Perform application resize processing

WT			

```
function EAPPSIZE_METHOD, ^val
    a_columns      ,n
    a_rows         ,n
    a_width        ,n
    a_height       ,n
```

Return value

This function should return true (1), so that other return values may be used in the future. (n)

Arguments

a_columns

The number of columns visible in the application window. (n)

a_rows

The number of rows visible in the application window. (n)

a_width

The number of horizontal pixels visible in the application window. (n)

a_height

The number of vertical pixels visible in the application window. (n)

Discussion

%EAPPSIZE_METHOD is a function that you write *and name*. The name of your function is registered with UI Toolkit using the E_METHOD subroutine.

UI Toolkit calls your %EAPPSIZE_METHOD function when the application window is resized.

A_columns and *a_rows* specify the visible area in the application window. Note that this could be different from the value returned by %W_INFO(WIF_SCOLS), which returns the logical size of the application window.

A_width and *a_height* report the size *inside* the application window's client area. This does not include the size of the frame, caption bar, menu bar, toolbars, information line, nor footer.

Because Toolkit invokes this function as a direct callback, the code in this function should be kept to a minimum, and operations that could conflict with concurrent processing avoided. For example, any action causing the rebuilding of an input set (such as I_SETDEL) must be avoided because this would give the user the opportunity to create a fatal error condition in I_INPUT. Instead, use %M_SIGNAL to synchronize processing for all such cases.

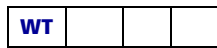
Environment Routines

%EAPPSIZE_METHOD

See also

- ▶ [“Overloading UI Toolkit Routines” on page 18-8](#) for more information on creating this function.
- ▶ [“Appendix D: Methods”](#).

%EAPPSTATE_METHOD – Perform application state processing



```
function EAPPSTATE_METHOD, ^val
    a_state      , n
```

Return value

This function should return true (1), so that other return values may be used in the future. (n)

Arguments

a_state

One of the following states defined in **tools.def**: (n)

WPS_NORMAL	Window was restored.
WPS_MINIMIZED	Window was minimized.
WPS_MAXIMIZED	Window was maximized.
WPS_HIDDEN	Window was hidden.

Discussion

%EAPPSTATE_METHOD is a function that you write *and name*. The name of your function is registered with UI Toolkit using the E_METHOD subroutine.

UI Toolkit calls your %EAPPSTATE_METHOD function when the state of the application window changes.

Because Toolkit invokes this function as a direct callback, the code in this function should be kept to a minimum, and operations that could conflict with concurrent processing avoided. For example, any action causing the rebuilding of an input set (such as I_SETDEL) must be avoided because this would give the user the opportunity to create a fatal error condition in I_INPUT. Instead, use %M_SIGNAL to synchronize processing for all such cases.

See also

- ▶ [“Overloading UI Toolkit Routines” on page 18-8](#) for more information on creating this function.
- ▶ [“Appendix D: Methods”](#).

%ECENTURY_METHOD – Perform century processing

WT	WN	U	V

```
function ECENTURY_METHOD, ^val
    a_data      , a
    a_type      , n
    a_today     , n
    a_year      , n
    a_month     , n
    a_day       , n
```

Return value

The two-digit century to use by default (for example, 19 or 20). **(n)**

Arguments

a_data

The data being decoded. **(a)**

a_type

One of the following: **(n)**

D_DATE_Y2P Date, format YYPP.

D_DATE_Y2J Date, format YYJJJ.

D_DATE_Y2MD Date, format YYMMDD.

D_DATE_Y4P Date, format YYYYPP.

D_DATE_Y4J Date, format YYYYJJJ.

D_DATE_Y4MD Date, format YYYYMMDD.

a_today

The default to today flag. **(n)**

a_year

The two-digit year entered. **(n)**

a_month

The month entered. **(n)**

a_day

The day entered. **(n)**

Discussion

%ECENTURY_METHOD is a function that you write *and name*. The name of your function is registered with UI Toolkit using the E_METHOD subroutine. Your %ECENTURY_METHOD function must be declared as a ^VAL function.

%ECENTURY_METHOD is called whenever UI Toolkit needs to supply a default century.

You will never call %ECENTURY_METHOD directly; it is called only by UI Toolkit.

A_data is the string passed to U_DCDDAT. For input processing, a_data is the string typed in by the user.

A_type is one of the definitions defined in **tools.def**.

If a_today is true, the date defaults to today's date.

A_year, a_month, and a_day are the decoded year, month, and day entered. Note that a_year is a two-digit year.

See also

- ▶ [SYNCENTURY](#) in the “Environment Variables” chapter of *Environment Variables & System Options* for more information about defining a sliding window used to determine the default century.
- ▶ [“Overloading UI Toolkit Routines” on page 18-8](#) for more information on creating this function.
- ▶ [“Appendix D: Methods”](#).

Examples

The following example shows the default century method behavior when SYNCENTURY is used.

```
function mycentury, ^val, reentrant
;
; Description: Century method implementing SYNCENTURY behavior
;
; Arguments:
    a_data          ,a      ;The date as entered
    a_type          ,n      ;The type of date field
    a_today         ,n      ;The "default to today" flag
    a_year          ,n      ;The two-digit year parsed from input
    a_month         ,n      ;The month parsed
    a_day           ,n      ;The day parsed

; Return value:
;     The two-digit century
;
```

Environment Routines

%ECENTURY_METHOD

```
.align
stack record
    length            ,i4      ;Length of contents of SYNCENTURY
                                ; logical
    syncentury        ,d2      ;Value contained in SYNCENTURY
    first_century     ,d2      ;First of the two centuries in our
                                ; sliding range
    group todays_ds   ,a20     ;%datetime
    group today       ,d8      ;The date portion thereof
    group yyyy        ,d4      ;The year out of that
    cc                ,d2      ;The century part
    yy                ,d2      ;The year (non-century)
    endgroup
    mm                ,d2      ;The month
    dd                ,d2      ;The day
    endgroup
endgroup
string                ,a80     ;String buffer for SYNCENTURY contents

proc
    todays_ds = %datetime      ;Load the current date to get the year
                                ;Is SYNCENTURY defined?
    xcall getlog("SYNCENTURY", string, length)
    if (length) then          ;Yes, use it
        begin
            onerror ($ERR_DIGIT) nosyncentury ;Catch conversion errors
            syncentury = string ;Convert from alpha to numeric
        end
    else                      ;No, SYNCENTURY is not defined
        begin                ; or is defined non-numeric
nosyncentury,
            clear syncentury ;Default to zero
        end
    offerror

    if (yy .ge. syncentury) then ;Are we on or after SYNCENTURY?
        first_century = cc      ;Yes, later numbers are in this
                                ; century
    else
        first_century = cc - 1 ;No, later numbers are in
                                ; previous century

    if (a_year .ge. syncentury) ;Entered a year on or after?
        freturn first_century ;Yes, return the earlier century

        freturn (first_century + 1) ;No, return the later century
endfunction
```

ECHKFLD_METHOD – Perform user field validation



```
subroutine ECHKFLD_METHOD
  a_input      , a
  a_type       , a
  a_text       , a
  a_status     , n
  a_inpinfo    , a
```

Arguments

a_input

The contents of the input field. **(a)**

a_type

The *user_type* string specified in the .FIELD USER qualifier. **(a)**

a_text

(optional) The *user_text* string specified in the .FIELD USER qualifier. **(a)**

a_status

Must be returned with a value of zero if the input is valid, and with a non-zero value if the input is invalid. Refer to **tools.def** for the list of valid return values. **(n)**

a_inpinfo

The structure of input information. **(a)**

Discussion

ECHKFLD_METHOD is a subroutine that you write *and name*. The name of your subroutine is registered with UI Toolkit using the E_METHOD subroutine. This subroutine replaces the default field validation that Toolkit does for the current field..



We recommend using an input field change method (%ICHANGE_METHOD) instead of ECHKFLD_METHOD. Unlike ECHKFLD_METHOD, your %ICHANGE_METHOD adds to Toolkit functionality rather than replaces it. %ICHANGE_METHOD also provides access to the structure defined in **inpinf.def** and to the 20 optional arguments passed to I_INPUT and L_INPUT. See “%ICHANGE_METHOD” on page 8-118 for more information.

The UI Toolkit input routine calls your ECHKFLD_METHOD subroutine to validate the contents of the current input field. It is only called if the field definition uses the *user_type* argument from the .FIELD USER qualifier.

You will never call ECHKFLD_METHOD directly; it is called only by UI Toolkit.

Note that the length of the text passed in *a_input* is defined by the input length for the field. It is not limited by either the display length or the view length.

If *a_status* is returned non-zero, UI Toolkit will display the appropriate error message. (See ECHKFLD_METHOD section of the **tools.def** file for the error flags.) If you return an *a_status* value of D_EMITTEDERR, no error message will be emitted, but input will be performed again for the field. This assumes that you have emitted any appropriate messages from within ECHKFLD_METHOD.

The **inpinf.def** file defines *a_inpinf*. See [IARRIVE_METHOD on page 8-114](#) for information.

S/DE Repository allows you to specify a subtype (or class) for user-defined fields. The default subtype is alpha, but you can optionally specify numeric or date. To access this subtype within the Toolkit user-defined processing routines, use the window ID and field number contained within the **inputinfo** structure (the *a_inpinf* argument). Pass these two values to the I_FLDINF subroutine to retrieve the **gs_inpfld** structure (the *controls* argument) which contains the **gs_class** (user subtype) value. For user-defined *alpha* fields; **gs_class** = 0; for user-defined *numeric* fields, **gs_class** = 1; for user-defined *date* fields, **gs_class** = 2.

If ECHKFLD_METHOD doesn't exist, your code will still run but UI Toolkit will not check whether the data entered by the user meets any user-defined criteria.

The ECHKFLD_METHOD subroutine works together with the EDSPFLD_METHOD and EEDTDSP_METHOD subroutines (which you write also). See the EDSPFLD_METHOD [Discussion on page 7-63](#) for more information on the steps that occur when a user-defined field is displayed, edited, and validated. If you have both *Exxx_METHODs* and an *%ICHANGE_METHOD* specified, the *%ICHANGE_METHOD* will be called after the *Exxx_METHODs* for the user-defined field.

Be sure that you write ECHKFLD_METHOD in such a way that it returns the validated data in storage format. EDSPFLD_METHOD, which is called by ECHKFLD_METHOD, needs the data in proper storage format to display the data correctly.

The following is a sample window script for both the following example and the example for the EDSPFLD_METHOD subroutine:

```
.input window, 1, 17
.placement 5, 29
.title " Zip Test "
.field zip, a10, pos(1, 2), prompt("Zip   "), -
  user(, "TYP_ZIPCODE"), uc
.end
```

See also

- ▶ “Overloading UI Toolkit Routines” on page 18-8 for more information on creating this subroutine.
- ▶ “Appendix D: Methods”.

Examples

This following example shows how to use ECHKFLD_METHOD, EDSPFLD_METHOD, and EEDTDSP_METHOD to validate a ZIP Code field.

The ECHKFLD_METHOD example checks the length of the input. If the length is either five or nine digits, the value is a U.S. code. If the length is six characters, it is a Canadian postal code. The routine verifies that U.S. ZIP Codes contain only numbers.

If the ZIP Code contains any errors, the subroutine sets the *a_status* return variable to D_EMITTEDERR (a non-zero value). This tells the Toolkit input routine that the input is invalid and needs to be re-entered. When the subroutine verifies that the input is valid, it returns a value of zero in *a_status*, which tells the Toolkit input routine that the input is valid and to call EDSPFLD_METHOD to display the data.



For an example that shows how you can use ECHKFLD_METHOD and EDSPFLD_METHOD to handle user-defined field types, see the EDSPFLD_METHOD Examples on page 7-64.

```
subroutine echkfld_method
  a_input   ,a      ;Characters input by user
  a_type    ,a      ;User-defined designation of this field
  a_text    ,a      ;Optional user string associated with this field
  a_status  ,n      ;Returned with zero if input valid
  a_inpinfo ,a      ;Input information

  .include "WND:tools.def"

record
  len      ,i4
  num      ,i4

proc
  clear a_status      ;Assume valid at outset
  case (a_type) of
    begincase
  "TYP_ZIPCODE":      ;Accept either 5 or 9 digits only
    begin
      len = %trim(a_input)
      if (len.eq.1 .and. a_input.eq." ")
        len = 0      ;Trim returns 1 if all blanks
```

```

        if (len.eq.5 .or. len.eq.9) then
            begin
                onerror bad_digit
                num = a_input
                offerror
                a_input = num,"xxxxxxxxx"
                exit
            bad_digit,
                xcall u_message("Only 5 or 9 digits are valid", D_ERROR)
                a_status = D_EMITTEDERR
            end
            else if (len.eq.6)then          ;Canadian ZIP Code format
                nop                        ;All characters valid, do nothing
            else if (len.ne.0)              ;Ignore empty field
            begin
                xcall u_message("Invalid U.S. or Canadian zip", D_ERROR)
                a_status = D_EMITTEDERR
            end
        end
                                ;Enter code of other user-types here

    endcase
xreturn
endsubroutine

```

The EDSPFLD_METHOD example puts a space between the numbers in a Canadian postal code (for example, **1G3 234**) and inserts a hyphen in nine-digit U.S. ZIP Codes. To get this space, it manipulates the input (**1G3234**) and then calls Synergy DBL routines to display the data. It then passes the displayed result into the *a_dest* argument.

```

subroutine edspfld_method
    a_buf      ,a      ;Characters input by user
    a_typ      ,a      ;User-defined designation of this field
    a_usr      ,a      ;Optional user string associated with this field
    a_inpid    ,n      ;Window ID
    a_fld      ,n      ;Window field number
    a_dest     ,a      ;Optional destination area in storage record.
                        ; If present, this routine must fill with the
                        ; storage representation.
    a_attr     ,n      ;Attribute for display
    a_color    ,n      ;Color for display
    a_inpinfo  ,a      ;Input information

    .include "WND:tools.def"
    .include "WND:windows.def"

    record
        dsptxt      ,a20
        dsp10       ,a10  @dsptxt
        dspcz1      ,a3   @dsptxt

```



```

        dspczd      ,a1      @dspcz1+3
        dspcz2      ,a3      @dspczd+1
numzip      ,d9
firstfour    ,d4      @numzip
canzip       ,a6
        canz1       ,a3      @canzip
        canz2       ,a3      @canz1+3

proc
    clear numzip, canzip, dsptxt
    case (a_typ) of
        begincase

"TYP_ZIPCODE ":
        begin
            onerror (20) canadian      ;Assume numeric,
            numzip = a_buf              ; otherwise Canadian
            if (.not.firstfour) then    ;Either 5 or 9 digits
                begin
                    dsp10 = numzip(5:5), "XXXXX" [left]
                    numzip=numzip*10000
                end
            else
                dsp10 = numzip, "XXXXX-XXXX"
            if (^passed(a_dest))
                ^d(a_dest) = numzip
            if (FALSE)                  ;Enter here if error #20 from above
                begin
canadian,      canzip = a_buf          ;Canadian ZIP Code
                dspcz1 = canz1          ;Load first 3 characters
                dspcz2 = canz2          ;Load second 3 characters
                if (^passed(a_dest))
                    a_dest = canzip
                end
            end
        ;Enter code for other user-types here
        endcase
        xcall w_flds(a_inpid, WF_PUT, a_fld, dsptxt, WF_ATTR,
&          a_fld, a_attr, WF_COLOR, a_fld, a_color)
        xreturn
    endsubroutine

```

This EEDTDSP_METHOD example left-justifies editable numeric text.

```

subroutine eedtdsp_method
    a_data      ,a          ;Data as stored
    a_type      ,a          ;Data type
    a_text      ,a          ;User text
    a_edit      ,a          ;Returned edit data

```

```
        a_inpinfo ,a                ;Input information

.align
record
        numeric,i4                ;Numeric test field

proc
        case a_type of
        begincase
            "TYP_ZIPCODE ": call zipcode
        endcase
        else
            a_edit = a_data
        xreturn

zipcode,
        onerror (20) canzip
        numeric = a_data
        offererror
        a_edit = numeric [left]    ;Left-justify numeric
        return

canzip,
        offererror
        a_edit = a_data            ;Just move alphanumeric
        return
endsubroutine
```

ECLOSE_METHOD – Perform application close processing

WT	WN	U	V

```
subroutine ECLOSE_METHOD
  a_noclr      ,n
  a_cancel     ,n
  a_type       ,n
```

Arguments

a_noclr

Returned with any *no_clear* flag passed to U_FINISH. (n)

a_cancel

(optional) The cancel shutdown flag which is passed to U_FINISH. (n)

a_type

The flag defining the type of shutdown in progress: (n)

- 0** U_FINISH was called directly.
- 1** The user is closing the application.
- 2** An interrupt (^C) was received.

Discussion

ECLOSE_METHOD is a subroutine that you write *and name*. The name of your subroutine is registered with UI Toolkit using the E_METHOD subroutine.

UI Toolkit calls your ECLOSE_METHOD subroutine to allow customized shutdown processing for your application. On Windows, ECLOSE_METHOD is called when

- ▶ U_FINISH is called.
- ▶ the user closes the application using the System menu, close box, or Close menu entry. (This is applicable only if you have previously called U_START and interrupt is enabled.)

On UNIX and OpenVMS, ECLOSE_METHOD is invoked only when U_FINISH is called (so *a_type* is always 0 on these platforms).

If *a_cancel* is passed, this method can set it to true to cancel application shutdown if you passed the cancel argument to U_FINISH.

See also

- ▶ [E_METHOD](#) on page 7-35 for registering your ECLOSE_METHOD with the environment.
- ▶ [E_STATE](#) on page 7-41 for more information on enabling interrupt characters.
- ▶ [U_FINISH](#) on page 16-46 for more information about managing your application's shutdown procedures.
- ▶ [“Overloading UI Toolkit Routines”](#) on page 18-8 for more information on creating this subroutine.
- ▶ [“Appendix D: Methods”](#).

Examples

The following method would have to be registered in the transaction routine's environment.

```
subroutine trx_close
;
; Description: Close method for transaction entry
;
; Arguments:
;
;         a_noclr      ,n      ;"No clear" option for U_FINISH
;
;         a_cancel     ,n      ;(Optional) cancel shutdown flag
;
;         a_type       ,n      ;Type of shutdown:
;                               ; 1 = Close
;                               ; 2 = Interrupt

.include "trxgbis.def"      ;Global fields (g_changes, etc.)
;
.align
record
;
;         style        ,i4      ;Message box style
proc
;
;         if (g_changes)      ;Are there unsaved changes
;                               ; to the transactions?
;
;         begin
;             if (^passed(a_cancel)) then          ;Can we cancel
;                                                     ; shutdown?
;                 style = D_MYESNOCANCEL           ;Yes, include
;                                                     ; cancel button
;
;             else
;                 style = D_MYESNO                  ;No, only allow
;                                                     ; yes/no
;                 style = style + D_MICONEXCLAM     ;Put an exclamation
;                                                     ; icon on
```

```

        using %u_msgbox("Save changes to transactions?",
&                style) select
        (D_MIDYES),    call save                ;Save the transactions
        (D_MIDNO),    nop                    ;No action required
        (D_MIDCANCEL),a_cancel = TRUE        ;Cancel shutdown
        endusing
    end
xreturn

save,
    ;Insert code here to save all changes to
    ; transaction file
    return
endsubroutine

```

The following code shows how you would use E_METHOD to register this method:

```

proc
    xcall e_enter
    xcall e_method(D_METH_APPCLOSE, "trx_close")
    .
    .
    .
    xcall e_exit
    xreturn
.end

```

EDSPFLD_METHOD – Perform user field display



```

subroutine EDSPFLD_METHOD
  a_input      , a
  a_type       , a
  a_text       , a
  a_window     , n
  a_fldnum     , n
  a_dest       , a
  a_attr       , n
  a_color      , n
  a_inpinfo    , a

```

Arguments

a_input

The contents of the input field. **(a)**

a_type

The *user_type* string specified in the .FIELD USER qualifier. **(a)**

a_text

(optional) The *user_text* string specified in the .FIELD USER qualifier. **(a)**

a_window

The ID of the input window being processed. **(n)**

a_fldnum

Contains the internal field number of the current display field. **(n)**

a_dest

(optional) The data area in which to store the displayed data. If it is present, EDSPFLD_METHOD must load *a_dest* in storage format. **(a)**

a_attr

The attribute to apply to the display. **(n)**

a_color

The color to apply to the display. **(n)**

a_inpinfo

The structure of input information. **(a)**

Discussion

EDSPFLD_METHOD is a subroutine that you write *and name*. The name of your subroutine is registered with UI Toolkit using the E_METHOD subroutine.



We recommend using an input field display method (%IDISPLAY_METHOD) instead of EDSPFLD_METHOD. Unlike EDSPFLD_METHOD, your %IDISPLAY_METHOD adds to Toolkit functionality rather than replaces it. %IDISPLAY_METHOD also provides access to the structure defined in **inpinf.def** and to the 20 optional arguments passed to I_INPUT and L_INPUT. See [%IDISPLAY_METHOD on page 8-122](#) for more information.

EDSPFLD_METHOD cannot populate a field that has a view length that's shorter than the display length.

The UI Toolkit input subroutine calls your EDSPFLD_METHOD subroutine to display the contents of the current input field. It is also called after the ECHKFLD_METHOD subroutine validates the user-defined input to display the validated data in the specified format. It is only called if the field definition uses the *user_type* argument in the .FIELD USER qualifier.

You will never call EDSPFLD_METHOD directly; it is called only by UI Toolkit.

The **inpinf.def** file defines *a_inpinfo*. See [IARRIVE_METHOD on page 8-114](#) for information.

S/DE Repository enables you to specify a subtype (or class) for user-defined fields. The default subtype is alpha, but you can also specify numeric or date. To access this subtype within the Toolkit user-defined processing routines, use the window ID and field number contained within the **inputinfo** structure (the *a_inpinfo* argument). Pass these two values to the I_FLDINF subroutine to retrieve the **gs_inpfld** structure (the *controls* argument) which contains the **gs_class** (user subtype) value. For user-defined *alpha* fields; **gs_class** = 0; for user-defined *numeric* fields, **gs_class** = 1; for user-defined *date* fields, **gs_class** = 2.

To display the user-defined input in a special format, you must call the Synergy DBL subroutine W_FLDS. The subroutine call can look like this:

```
xcall w_flds(a_window, WF_PUT, a_fldnum, display_text, WF_ATTR, a_fldnum, a_attr,
&           WF_COLOR, a_fldnum, a_color)
```

display_text

The specially formatted input to display in the input field. (a)

EDSPFLD_METHOD works in conjunction with the ECHKFLD_METHOD and EEDTDSP_METHOD subroutines. The following steps occur for a user-defined field:

1. When your program calls I_DISPLAY or I_DSPFLD, UI Toolkit also calls EDSPFLD_METHOD and passes it the field's data in the format it was stored. EDSPFLD_METHOD displays the data in the field with the formatting you have specified in the EDSPFLD_METHOD subroutine.

For example, if you have written EDSPFLD_METHOD so that it formats all case "A" numbers with a dollar sign and a decimal point, it will display \$350.00 when passed the stored data "350A."

2. When the user selects the field for editing, UI Toolkit calls EEDTDSP_METHOD and passes it the data in storage format. The supplied EEDTDSP_METHOD displays the editable data in the stored format. If you want the editable data to display differently, you must write your own version of EEDTDSP_METHOD.

For example, once the user has selected the “350A” data for editing, your EEDTDSP_METHOD might re-display the data as “350.” If you used the supplied version of EEDTDSP_METHOD, the editable data would display as “350A.”

3. After the user edits the data, UI Toolkit calls ECHKFLD_METHOD to validate the new data, determining if it meets all of your specified criteria. If ECHKFLD_METHOD validates the data, EDSPFLD_METHOD is called again to display the new data including the proper formatting.

If the user has changed “350” to “150,” ECHKFLD_METHOD is called and checks to make sure “150” is an acceptable entry for that field. If it is, EDSPFLD_METHOD displays “\$150.00” in the field.

See also

- ▶ [I_DISPLAY on page 8-13](#) for more information on specifying a display method for an input field.
- ▶ [“Overloading UI Toolkit Routines” on page 18-8](#) for more information on creating this subroutine.
- ▶ [“Appendix D: Methods”](#).

Examples

The following example shows how you can use ECHKFLD_METHOD and EDSPFLD_METHOD to handle user-defined field types. See the ECHKFLD_METHOD [Discussion on page 7-53](#) for a sample window script that corresponds with this example.



See the ECHKFLD_METHOD [Examples on page 7-55](#) for an example that uses ECHKFLD_METHOD, EDSPFLD_METHOD, and EEDTSP_METHOD to validate and display a ZIP Code.

```
; Description:  Routines for handling user-defined field types
;
; Reprinted by permission of BancTec, Inc.
;
; Routines:
;     ECHKFLD_METHOD - Validation of user-defined types
;     EDSPFLD_METHOD - Display and storage of user-defined types
;
subroutine ECHKFLD_METHOD
;; Description: Validate user-defined data types
;; Arguments:
;     a_input      ,a      ;Text as input
```



```

        a_type      ,a      ;User-defined data type
        a_text      ,a      ;User text for field
        a_status    ,n      ;Returned status
        a_inpinfo   ,a      ;Input information

.include "WND:tools.def"

.align
record
        sts         ,i4     ;Decoding status
        yr          ,i4
        mo          ,i4
        da          ,i4

proc
case a_type of
    begincase
        "MDYDATE ":call mdydate
    endcase
else
    begin
        xcall u_message("Unrecognized user type "+a_type)
    end
xreturn
;
; Description: Handle MM/DD/YY date format
;
mdydate,
    xcall u_dcddat(sts, a_input, 3,, yr, mo, da)
    if (sts) then
        a_status = D_DATERR
    else
        begin
            clear a_status
            a_input = %string(mo, "XX")+%string(da, "XX")+%string(yr, "XX")
        end
    return
endsubroutine

subroutine EDSPFLD_METHOD
;
; Description: Handle display and storage of user-defined data types
;
; Arguments:
    a_input      ,a      ;Contents of field
    a_type       ,a      ;User-defined type
    a_text       ,a      ;User text
    a_window     ,n      ;Input window ID

```

Environment Routines

EDSPFLD_METHOD

```

    a_fldnm      ,n      ;Display field number
    a_dest       ,a      ;Destination (optional)
    a_attr       ,n      ;Display attribute
    a_color      ,n      ;Display color
    a_inpinfo    ,a      ;Input information
;
.include "WND:tools.def"
.include "WND:windows.def"
;
record
    group yymmdd      ,d
        YY            ,d2
        mm            ,d2
        dd            ,d2
    endgroup
    buf              ,a8
;
proc
    case a_type of
        begincase
            "MDYDATE ":call mdydate
        endcase
    else
        begin
            xcall u_message("Unrecognized user type "+a_type)
        end
    xreturn
;
; Description: Display and store MM/DD/YY dates
;
mdydate,
    yy = a_input(5:2)
    mm = a_input(1:2)
    dd = a_input(3:2)
    xcall u_fmtat(3, buf, yy, mm, dd)
    xcall w_flds(a_window, WF_PUT, a_fldnm, buf, WF_ATTR,
&      a_fldnm, a_attr, WF_COLOR, a_fldnm, a_color)
    if (^passed(a_dest))
        a_dest(1:6) = a_input      ;Note: (1:6) to avoid clobbering following
                                   ;fields. Defined as d8, but stored as d6.
    return
endsubroutine
```

EEDTDSP_METHOD – Perform user field editing



```
subroutine EEDTDSP_METHOD
  a_data      , a
  a_type      , a
  a_text      , a
  a_edit      , a
  a_inpinfo   , a
```

Arguments

a_data

The data as stored in the field. **(a)**

a_type

The *user_type* string specified in the .FIELD USER qualifier. **(a)**

a_text

(optional) The *user_text* string specified in the .FIELD USER qualifier. **(a)**

a_edit

The returned text to be edited. **(a)**

a_inpinfo

The structure of input information. **(a)**

Discussion

EEDTDSP_METHOD is a subroutine that you write *and name*. The name of your subroutine is registered with UI Toolkit using the E_METHOD subroutine.



We recommend using an input field edit format method (%IEDITFMT_METHOD) instead of EEDTESP_METHOD. Unlike EEDTDSP_METHOD, your %IEDITFMT_METHOD adds to Toolkit functionality rather than replaces it. %IEDITFMT_METHOD also provides access to the structure defined in **inpinf.def** and to the 20 optional arguments passed to I_INPUT and L_INPUT. See [%IEDITFMT_METHOD on page 8-127](#) for more information.

The UI Toolkit input subroutine calls the EEDTDSP_METHOD subroutine to display the editable contents of a user-type field differently than both the way it is stored and the way EDSPFLD_METHOD displays it (with formatting). If an edit format method is specified for the input field (using %IEDITFMT_METHOD), the edit format method is called after the EEDTDSP_METHOD subroutine. Use EEDTDSP_METHOD if you need the text of user-defined data type fields to display in another way than the way they are stored.

Note that the length of the text passed in *a_edit* is defined by the input length for the field. It is not limited by either the display length or the view length.

You will never call EEDTDSP_METHOD directly; it is called only by UI Toolkit.

The **inpinf.def** file defines *a_inpinfo*. See [IARRIVE_METHOD](#) on page 8-114 for information.

S/DE Repository allows you to specify a subtype (or class) for user-defined fields. The default subtype is alpha, but you can optionally specify numeric or date. To access this subtype within the Toolkit user-defined processing routines, use the window ID and field number contained within the **inputinfo** structure (the *a_inpinfo* argument). Pass these two values to the **I_FLDINF** subroutine to retrieve the **gs_inpfld** structure (the *controls* argument) which contains the **gs_class** (user subtype) value. For user-defined *alpha* fields; **gs_class** = 0; for user-defined *numeric* fields, **gs_class** = 1; for user-defined *date* fields, **gs_class** = 2.

Although you can manage the display and validation of user-defined fields using **EDSPFLD_METHOD** and **ECHKFLD_METHOD**, when the field is edited (with the *E_xxx* menu entries) the field is treated as alphanumeric. Thus if the storage form of the data differs from the manner in which it was entered, confusion can result. EEDTDSP_METHOD can solve this problem.

Toolkit's default method is named **TKP_EDTDSP** and is listed below:

```
subroutine tkp_edtdsp
;
; Description:
; Format the editable text for a user-defined data type
;
; Arguments:
;   a_data           ,a      ;Data as stored in the field
;   a_type           ,a      ;User-defined data type
;   a_text           ,a      ;(Optional) user text string
;   a_edit           ,a      ;Returned text to be edited
;   a_inpinfo        ,a      ;Input information
;
; Notes:
;
;   Overload this routine if you need to have user-defined data types
;   display data other than as stored when the user begins to edit the
;   field.
;
;   This default version merely moves the stored data into the editing
;   buffer. You should retain this functionality for any unknown user data
;   types. If you remove the semi-colons which "comment out" the case
;   statement, then replace the "MYTYPE" case shown with one or more of
;   your own, the "else" on the endcase will handle the default condition.
```

```
proc
; case a_usertype of
;   begincase
;     "MYTYPE ":call format_my_type
;   endcase
; else
      a_edit = a_data
      xreturn
endsubroutine
```

See also

- ▶ [%IEDITFMT_METHOD on page 8-127](#) for more information on specifying an edit format method for an input field.
- ▶ [“Overloading UI Toolkit Routines” on page 18-8](#) for more information on creating this subroutine.
- ▶ [“Appendix D: Methods”](#).

Examples

See the ECHKFLD_METHOD [Examples on page 7-55](#) for an example that uses ECHKFLD_METHOD, EDSPFLD_METHOD, and EEDTSP_METHOD to validate and display a ZIP Code.

EENTRST_METHOD – Perform input override processing



```
subroutine EENTRST_METHOD
  a_nomod      , n
  a_reset      , n
```

Arguments

a_nomod

The true/false flag that specifies whether the data was modified. **(n)**

a_reset

The true/false flag that determines whether to reset the field's contents. **(n)**

Discussion

When the user selects a menu entry, the default behavior is to cancel pending input. EENTRST_METHOD enables you to override this default behavior, so you can retain the pending input.

EENTRST_METHOD is a subroutine that you write *and name*. The name of your subroutine is registered with UI Toolkit using the E_METHOD subroutine.

UI Toolkit calls your EENTRST_METHOD subroutine when an input field is exited due to a menu entry being invoked. This occurs when %M_SIGNAL is called or when the user selects a menu entry, presses a menu entry shortcut, or presses a standard button. On Windows, this includes I_CANCEL; EENTRST_METHOD overrides I_CANCEL.

When this subroutine is called, if the user has not modified the input field contents, *a_nomod* is passed as true; otherwise it's passed as false.

When this subroutine is called, *a_reset* is always passed as true unless the field is a check box or a multi-line text field, in which case it's passed as false. On Windows, *a_reset* is also passed as false if the field is a radio button or combo box (selection window).

If this subroutine sets *a_reset* to false, Toolkit will accept and validate the pending data. If *a_reset* is set to false when *a_nomod* is true, this means that the existing contents of the field will be validated as if they were entered for the first time, even though no change has been made. Note that this can cause validation errors on existing data, for example an error will occur on a required field that is blank. Also, setting *a_reset* to false can cause a break to occur on a field.

If using I_INPFLD, validation does not occur until you leave the field, so a button with a method does not trigger the EENTRST_METHOD unless the button method signals a menu entry.

See also

- ▶ [“Overloading UI Toolkit Routines” on page 18-8](#) for more information on creating this subroutine.
- ▶ [“Appendix D: Methods”](#).

Examples

Note that the IF statement in the following example tests for the I_CANCEL menu entry and prevents the EENTRST_METHOD from overriding the cancel, which it would do on Windows systems. (You can create similar tests for other menu entries for which you do not want to accept pending data.)

```
subroutine eentrst_method
    a_nomod          ,n
    a_reset          ,n

    .include 'WND:tools.def'

proc

    if (g_entnam.ne.'I_CANCEL'.and..not.a_nomod)
        a_reset = 0
    xreturn

endsubroutine
```

EFKEY_METHOD – Perform key mapping

	WN	U	V

```
subroutine EFKEY_METHOD
    a_key_code    , n
```

Arguments

a_key_code

The key value to map. (**n**)

Discussion

EFKEY_METHOD is a subroutine that you write *and name*. The name of your subroutine is registered with UI Toolkit using the E_METHOD subroutine. (On Windows, this routine is ignored.)

UI Toolkit calls EFKEY_METHOD whenever a function key or control character is entered by the user, enabling you to assign a key to a reserved function. The version of EFKEY_METHOD supplied with Toolkit does nothing. You can use it, however, to remap any function code you desire.

The distributed include file **inpctl.def** defines the function codes for key mapping.

EFKEY_METHOD is passed *a_key_code*, which is the key code that resulted from any interpretation of the key map. You can modify this function code and change it to any other code defined in **inpctl.def** or zero to have nothing happen.

At the end of the list of defined codes, you may notice definitions for DELETE_KEY, PROCESS_MENU, SCREEN_REDRAW, and EOF_KEY. To assign these functions to a different key, all you need to change in your EFKEY_METHOD subroutine is the usual code generated by the desired keystroke to one of these codes.

See also

- ▶ [“Overloading UI Toolkit Routines” on page 18-8](#) for more information on creating this subroutine.
- ▶ [“Appendix D: Methods”](#).

EHELP_METHOD – Perform help processing

WT	WN	U	V

```
subroutine EHELP_METHOD
  a_ident , a
```

Arguments

a_ident

(optional) The help identifier. (**a**)

Discussion

EHELP_METHOD is a subroutine that you write *and name*. The name of your subroutine is registered with UI Toolkit using the E_METHOD subroutine.

EHELP_METHOD is called by UI Toolkit when a menu entry with the internal name O_HELP is selected. This routine will not be called unless that entry name is present on one of your menu columns.

UI Toolkit provides a help processing subroutine named TKP_HELP. You can overload it with a more robust or application-specific version.

EHELP_METHOD can be passed one argument, *a_ident*. UI Toolkit determines whether it will pass this argument, and if so, what it will pass in it. This is determined as follows:

- ▶ If the user has a menu column dropped down and selects O_HELP, UI Toolkit will set **g_enthlp** to the internal menu entry name of the highlighted menu entry. The *a_ident* argument will not be passed.
- ▶ If the user has no menu column dropped down and selects O_HELP while on an input field that has a help identifier specified (either in the .FIELD script or with IB_FIELD), UI Toolkit will pass help identifier as the *a_ident* argument.
- ▶ If the user has no menu column dropped down and is not on an input field with a help identifier specified, but the current subroutine was passed a help identifier, UI Toolkit will pass that help identifier value as the *a_ident* argument. The following subroutines can be passed a help identifier:

```
IB_FIELD
L_CHR
L_SELECT
M_PROCESS
S_SELECT
S_SELECTCB
T_EDIT
T_VIEW
```

```

U_CHR
U_EDITKEYS
U_EDITREND
U_FLD
%U_GETFILENAME
U_MODREND
%U_MSGBOX

```

- ▶ In all other cases, UI Toolkit will not pass the *a_ident* argument. (This includes the case where O_HELP is selected while the input termination message is displayed.)

The help identifier associated with an input field or passed to one of the above routines can be used to support the mechanism you choose to provide help for the user. The help identifier can be the name of a help window or, perhaps, the key for an ISAM record that contains help text.

When writing your EHELP_METHOD, be sure to consider whether you want to support a second O_HELP request (for example, enabling the user to select Help while they are viewing the first Help window).

The TKP_HELP subroutine distributed with UI Toolkit goes through the following steps:

1. If **g_enthlp** is non-blank, UI Toolkit will search the window library for the window *h_name* where *name* is the contents of **g_enthlp**. This window will be displayed, if found, and UI Toolkit will skip to step 4. If not found, UI Toolkit proceeds to step 2.
2. If **g_enthlp** is blank or if the window *h_name* (in step 1) was not found, and if *a_ident* was passed, UI Toolkit will search the library for a window *name*, where *name* is the contents of *a_ident*. (If *a_ident* was not passed, UI Toolkit goes to step 3.) This window will be displayed, if found, and UI Toolkit will skip to step 4. If the window is not found, Toolkit will issue the message “No help available for identifier *name*” and proceed to step 3. (This message is in the text message file, facility DTK, mnemonic NOHELP. See “[The Synergy UI Toolkit Control Panel](#)” in the “General Utilities” chapter of *Synergy Tools* for information on customizing messages using the Synergy UI Toolkit Control Panel.)
3. If **g_enthlp** is blank or if the window *h_name* (in step 1) was not found, and *a_ident* was not passed, the window library is searched for a window named *h_general*. If this window cannot be found, Toolkit will issue the message “No help available for identifier ‘h_general.’”

If a window could be loaded, then TKP_HELP carries out steps 4 through 7.

4. It loads the information line with the message found in text message facility DTK, mnemonic HELPINFO, and a column named *c_help* is sought in the library opened on **g_utlib**. If found, TKP_HELP places the column. If not, no error occurs, but the user probably won’t be able to scroll the text. (Some of these actions are only applicable for scrollable windows. See the comments within the TKP_HELP source listed in the [Examples on page 7-75](#).)
5. TKP_HELP calls U_HELPREND and applies any help renditions to the window. TKP_HELP also places the window at this time.

6. The help window is displayed:
 - ▶ If in a Windows environment, TKP_HELP calls U_WINDOW and uses a Windows-style pop-up window with an OK button.
 - ▶ If in a UNIX or OpenVMS environment and the entire window is visible (i.e., the display area matches the window area), TKP_HELP displays a message requesting the user to press RETURN (ENTER) when done viewing the window.
 - ▶ If in a UNIX or OpenVMS environment and if TKP_HELP can show only part of the window, the window is considered scrollable, and T_VIEW is used to enable the user to browse the window.
7. If the user makes another selection of O_HELP, TKP_HELP looks for and displays a second help window named h_help (if found). When the user exits from this second help window, they will return to the first help window. (TKP_HELP will then return to the calling routine when one of the first two scenarios is true.)
8. TKP_HELP returns to the calling routine when
 - ▶ the entire window is visible and the user presses RETURN or, in Windows, the user clicks on the OK button.
 - ▶ the window is scrollable and the user selects any menu entry that T_VIEW does not recognize (other than O_HELP). If the entry is O_EXIT, **g_select** is returned false to the caller in order to avoid further exiting. Any other menu entry is returned to the caller.

See also

- ▶ [“The Synergy UI Toolkit Control Panel”](#) in the “General Utilities” chapter of *Synergy Tools* for information on customizing the HELPINFO and NOHELP text messages.
- ▶ [S_SELECT on page 12-10](#) for an example of how a help identifier is passed.
- ▶ [U_HELPREND on page 16-70](#) for help window renditions.
- ▶ [“Overloading UI Toolkit Routines” on page 18-8](#) for more information on creating this subroutine.
- ▶ [“Appendix D: Methods”](#).

Examples

The following is a listing of the TKP_HELP subroutine supplied with UI Toolkit. It includes comments that explain the operation of the subroutine. While you review it, note that **g_select** must be cleared so that the menu processing subroutine (M_PROCESS or I_INPUT) continues without interruption.

```
subroutine tkp_help
;
; Description: Default version of help utility
;
      a_id      ,a              ;(Optional) help identifier
```

```

;
; Notes:
;
;       This routine is called from within the higher-level input
;       routines whenever the menu entry "O_HELP" is selected.
;       Developers are encouraged to replace this version with
;       whatever help mechanisms are desired.
;
;       If a_id is passed, it is a help identifier, which is either
;       associated with an input field or was passed into the
;       routine which performed input. If no such help identifier
;       exists, a_id is not passed.
;
;       If g_enthlp is non-blank, it contains the name of a menu
;       entry which was highlighted on a dropped-down menu column
;       when the shortcut for O_HELP was pressed.
;
;       This default version operates as follows:
;
; 1. Determine the help identifier to use
;    A. If g_enthlp is non-blank, use "h_%a", where %a is
;       the contents of g_enthlp.
;    B. Otherwise
;       1. If a_id is passed, use it
;       2. Otherwise, use "h_general"
; 2. Attempt to load a window from g_utlib named after the
;    identifier
;    A. If unsuccessful and 1.A. was used above, go to
;       1.B.
;    B. If unsuccessful and 1.B. was used above, issue an
;       error and exit.
; 3. Apply help renditions to the window.
; 4. Compare the display area to the size of the window
;    A. If the entire window is visible, ask the user to
;       press return.
;    B. If part of the window is not visible, perform text
;       viewing.
;       1. Load an information line from the text message
;          file.
;       2. Attempt to load column "c_help" from g_utlib,
;          ignoring errors.
;       3. Setup the window for text viewing.
;       4. View the text with T_VIEW.
;       5. Exit on any non-text-viewing menu entry,
;          clearing any "O_EXIT" to avoid further exiting.
; 5. Return to the calling routine.
;
; This routine also enters a private environment and exits it
; before returning.

```

```
.include "WND:tools.def"
.include "UTL:dtkmsg.def"
.include "WND:windows.def"
.include "UTL:inpctl.def"
.align
record
    len      ,i4          ;Length of message line
    wnd      ,2i4         ;Help window ID
    ndx      ,i4          ;Index into help windows
    col      ,i4          ;Help column ID
    err      ,i4          ;Error return
    srch     ,i4          ;Search flag for window loads
    mnu      ,i4          ;Are we attempting menu entry help?
    line     ,a132         ;Message line
    helpid   ,a15          ;Help identifier to use
    chr      ,a1           ;Character typed
    wareas   ,8d3          ;Window areas from W_INFO
    display ,a12 @wareas   ;Display area coordinates
    proc     ,a12 @wareas+12 ;Processing area coordinates

proc
    clear g_select, ndx      ;Assume we will satisfy O_HELP
    if (g_enthlp) then      ;Were we sitting on a menu entry?
        begin
            mnu = TRUE       ;Yes, attempt menu help
            xcall s_bld(helpid, , "h_%a", g_enthlp)
        end
    else
        begin
use_id,      clear mnu      ;No, use any passed identifier
            if (^passed(a_id)) then
                helpid = a_id
            else
                helpid = "h_general" ;Or h_general if none
            end
        end
        call do_help
        xreturn

do_help,
    xcall e_enter            ;Enter a private environment
    xcall e_state(D_OFF, D_ECHO, D_CURSOR)
    incr ndx
                                ;Attempt to load the window
    xcall u_ldwnd(wnd(ndx), g_utlib, helpid,, srch, err,, TRUE)
    if (err)
        begin
            ;Could not find it
            if (mnu)         ;If we were attempting menu entry
```

```

        begin
            decr ndx
            xcall e_exit
            call use_id           ;try any passed identifier instead
            return
        end
        xcall u_msg("DTK", DTK_NOHELP, D_ERROR, , helpid)
        goto exit               ;Issue error message and exit
    end
    xcall u_helprend(wnd(ndx))  ;Apply help renditions to window

.ifdef D_DTKGUI
    xcall u_window(D_REMOVE, wnd(ndx))
    xcall synmsw_popup(wnd(ndx))
.else
    xcall w_area(wnd(ndx), WA_COPY, WAC_WTOP)      ;Test window areas
    xcall w_info(WI_AREAS, wnd(ndx), wareas)
    if (display .ne. proc) then                    ;Scrollable?
        begin
            xcall u_text(g_txtlib, "DTK", DTK_HELPINFO2, line, len)
            xcall e_sect(line(1:len), D_INFO, D_CENTER);Load info from msg
                                                    ;file
                                                    ;Attempt to load column, ignoring errors
            xcall m_ldcol(col, g_utlib, "c_help", D_NOPLC, srch, err)
            xcall t_setup(wnd(ndx))                 ;Setup for text viewing
            repeat
                begin
                    xcall t_view(wnd(ndx), col, err) ;View the text
                    if (g_select) then
                        begin
                            case g_entnam of
                                begincase
                                    "O_EXIT ":    clear g_select
                                    "O_ABANDON ": clear g_select
                                    "O_HELP ":     begin
                                        clear g_select
                                        if (helpid .ne. "h_help ")
                                            begin
                                                helpid = "h_help"
                                                clear mnu
                                                call do_help
                                            end
                                        end
                                    nextloop
                                end
                            endcase
                        exitloop
                    end
                else

```

```

        if (err) then
            xcall u_beep
        else
            exitloop                                ;On return
        end
    end
else
    begin
        xcall u_text(g_txtlib, "DTK", DTK_HELPINFO2, line, len)
        ;Load info line from our message file
        xcall e_sect(line(1:len), D_INFO, D_CENTER)
repeat
                                                ;Full window
        begin
            xcall u_chr_p(chr)
            if (g_select) then
                begin
                    case g_entnam of
                    begincase
                    "O_EXIT ":      clear g_select
                    "O_ABANDON ":  clear g_select
                    "O_HELP ":      begin
                                    clear g_select
                                    if (helpid .ne. "h_help")
                                    begin
                                        helpid = "h_help"
                                        clear mnu
                                        call do_help
                                    end
                                end
                    nextloop
                    end
                    endcase
                    exitloop
                end
            else
                if (gi_sts .eq. RETURN_KEY) then
                    exitloop
                else
                    xcall u_beep
                end
            end
        end
    .endc

exit,
    decr ndx
    xcall e_exit                                ;Exit the private environment
    return                                    ;Return to caller
endsubroutine

```

ESCRIPERR_METHOD – Perform script compilation error processing

WT		U	V

```
subroutine ESCRIPERR_METHOD
  a_type      , n
  a_message   , a
  a_filename  , a
  a_linenum   , n
  a_textline  , a
  a_tokennum  , n
  a_tokenlen  , n
```

Arguments

a_type

The flag defining the type of error message: **(n)**

D_ERROR Error.

D_ALERT Warning.

D_INFO Informational text.

a_message

The text of the message. **(a)**

a_filename

(optional) The name of the file in which the error occurred. **(a)**

a_linenum

(optional) The line number within the file, at which the error occurred. **(n)**

a_textline

(optional) The text of the line at which the error occurred. **(a)**

a_tokennum

(optional) The index of the token within *a_textline*, at which the error occurred. **(n)**

a_tokenlen

(optional) The length of *a_tokennum*. **(n)**

Discussion

ESCRIPTErr_METHOD is a subroutine that you write *and name*. The name of your subroutine is registered with UI Toolkit using the E_METHOD subroutine.

UI Toolkit calls your ESCRIPTERR_METHOD subroutine to allow customized script compilation error processing for your application, when using the external routine interface of the Script compiler only.

Once registered, ESCRIPTERR_METHOD will be called when an error occurs while using any of the following routines, defined in [chapter 2, “Script.”](#)

- ▶ %SCR_OPENLIBRARY
- ▶ %SCR_CLOSELIBRARY
- ▶ %SCR_PROCESS
- ▶ %SCR_ERRORCOUNT

If an ESCRIPTERR_METHOD is not registered, the U_START subroutine registers a default routine that displays a message box for each error.

If *a_type* is set to D_INFO and *a_message* is blank, *a_textline* contains the output resulting from a .LOG script command.

There is no maximum length for *a_message*. The ESCRIPTERR_METHOD routine is responsible for wrapping or truncating the message appropriately.

A_filename can contain up to 256 characters, and is one of the following:

- ▶ The name of the script file, if a script file open or syntax error occurred
- ▶ The name of the library if a library file open or creation error occurred
- ▶ Not passed if not one of the above

A_linenum, *a_textline*, *a_tokennum*, and *a_tokenlen* are only passed if a script syntax error occurred.

A_textline can contain up to 255 characters.

See also

- ▶ [“Accessing the Script compiler using the external routine interface” on page 2-120](#) for more information about using the Script compiler’s external routine interface.
- ▶ [“Overloading UI Toolkit Routines” on page 18-8](#) for more information on creating this subroutine.
- ▶ [“Appendix D: Methods”](#).

EUTILS_METHOD – Perform utilities processing



```
subroutine EUTILS_METHOD
    a_record      , a
    a_inpinfo     , a
    a_methoddata1 , any
    .
    .
    a_methoddata20, any
```

Arguments

a_record

(optional) The data area associated with the input window of the calling program. **(a)**

a_inpinfo

(optional) The structure of input information. **(a)**

a_methoddata1 through *a_methoddata20*

(optional) Up to 20 additional data arguments. **(any)**

Discussion

EUTILS_METHOD is a subroutine that you write *and* name. You must use E_METHOD to register the name of your subroutine.

The EUTILS_METHOD subroutine is called by UI Toolkit when the user selects a menu entry whose internal name begins with “U_”.

The *a_record* argument is passed only if EUTILS_METHOD is called from I_INPUT or L_INPUT.

The *a_inpinfo* argument is passed only if EUTILS_METHOD is called from I_INPUT, I_INPFLD, L_INPUT, or L_INPFLD. The **inpinf.def** file defines *a_inpinfo*. See [IARRIVE_METHOD](#) on [page 8-114](#) for information.

The *a_methoddata* arguments are passed only if they are passed to the routine within which the EUTILS_METHOD method is called.

UI Toolkit’s default utility processing routine is named TKP_UTILS, which generates the error “Unknown utility entry xxxx received” (where xxxx is the name of a utility entry) if the user selects a U_ menu entry and no overloaded EUTILS_METHOD subroutine is registered.

Examples

In the following example, because EUTILS_METHOD is called only when a U_ menu entry is selected, **g_entnam** will always contain the entry name of a U_ entry. (There's no need to check **g_select**.)

After the entry is processed, **g_select** must be cleared so the menu processing subroutine (M_PROCESS or I_INPUT, for example) will continue as if no menu selection were made. If **g_select** is true on return, the menu entry is returned to the calling routine.

```
subroutine my_eutils_method
  a_record      ,a          ; optional data area
  .include "WND:tools.def"

proc
  case g_entnam of
    begincase
      "U_COMM " :      xcall comment
      "U_PUZZ " :      xcall puzzle
      "U_SCRPRN " :     xcall scrprn
    endcase
  else
    xcall u_message("Unrecognized utility menu entry")
    clear g_select
  xreturn
endsubroutine
```

See also

- ▶ [“Overloading UI Toolkit Routines” on page 18-8](#) for more information on creating this subroutine.
- ▶ [“Appendix D: Methods”](#).

8

Input Routines

Using Input Windows 8-3

Defines and discusses the basic concepts of input windows, as well as the IB_xxx subroutines, which enable you to build input windows “on-the-fly.”

I_ACTIVATE – Specify characters for input termination	8-8
I_CTRCONTEXT – Establish input context for child input window	8-9
I_DISABLE – Disable input fields.....	8-11
I_DISPLAY – Display data to input fields.....	8-13
I_DSPAREA – Set the display area for an input window	8-15
I_DSPFLD – Display data to one input field.....	8-17
I_ENABLE – Enable input fields	8-19
%I_FLDDIM – Retrieve an input field’s dimension or number of elements.....	8-21
I_FLDINF – Return input field information.....	8-22
I_FLDMOD – Modify an input field’s attributes at runtime.....	8-25
%I_FLDPREC – Retrieve an input field’s precision.....	8-36
I_FLDSEL – Assign a new selection window to an input field	8-37
%I_FLDSIZE – Retrieve an input field’s size.....	8-39
%I_FLDSTRPOS – Retrieve an input field’s structure position.....	8-40
I_FORCE – Force data to or from an input field.....	8-41
I_FRAMES – Display frames on input fields.....	8-43
I_GETFLD – Get data from an input field’s data area.....	8-44
%I_GETSTRING – Retrieve a string referenced by an input window data pointer	8-46
%I_INFO – Return general information about an input window	8-48
I_INIT – Initialize input fields.....	8-49
I_INPFLD – Process an input field	8-51
I_INPUT – Process an input set.....	8-55
I_LDINP – Load an input window from a library	8-62
I_NEXT – Specify the next input field or button to be processed.....	8-65

I_PROMPT – Set a variable prompt	8-69
I_PUTFLD – Put data into an input field.....	8-70
I_READONLY – Set read-only state for input fields.....	8-72
I_SETDEL – Delete fields from an input set.....	8-77
I_SETINF – Retrieve information about an input set.....	8-80
I_SNAPSHOT – Save/restore an input window’s controls	8-82
I_TXTPOS – Get or set the input position for a multi-line text field	8-84
I_USER – Retrieve/load the user-defined string for an input field.....	8-87
IB_BUTTON – Include a Windows-style button	8-88
IB_BUTTONSET – Define the placement of a set of buttons	8-90
IB_DSP_AREA – Set the display area for a new input window	8-92
IB_END – Define the end of a new input window	8-93
IB_FIELD – Add an input field to a new input window	8-95
IB_INPUT – Create a new input window	8-103
IB_LOCAL – Set the current input window structure to local	8-105
IB_PAINT – Set the paint character for a new input window	8-106
IB_RPS_STRUCTURE – Specify the repository structure for a new input window.....	8-107
IB_SET – Add a set definition to a new input window	8-109
IB_STRUCTURE – Define a local structure within a new input window	8-111

The Input Methods 8-113

IARRIVE_METHOD – Perform input field arrive processing	8-114
%ICHANGE_METHOD – Perform input field change processing	8-118
%IDISPLAY_METHOD – Perform input field display processing.....	8-122
IDRILL_METHOD – Perform drilldown button processing	8-125
%IEDITFMT_METHOD – Perform input field “edit format” processing.....	8-127
IHYPER_METHOD – Perform hyperlink prompt processing	8-130
ILEAVE_METHOD – Perform input field leave processing	8-132

Using Input Windows

UI Toolkit addresses an application's need to process terminal input through an input window. An input window is a standard window that can contain text, input fields, and buttons. Input windows are defined with the `.INPUT` script command or at runtime with the `IB_INPUT` subroutine (see [“Building Input Windows at Runtime”](#) on page 8-6).

Note that you can also create windows that accept input and that are made up of input windows, lists, ActiveX controls, and/or other types of windows. These windows are processed by Toolkit and appear to the user as a single window. See [chapter 6, “Composite Window Routines.”](#)

Input fields

Each input field is associated with a position and length within the window, as well as a set of qualifiers that defines how the field should be processed.

Qualifiers

Input field qualifiers enable you to define input field characteristics, which include the following:

- ▶ Name for the field
- ▶ Type and length of the data
- ▶ Number of digits to the right of the decimal point (if numeric)
- ▶ Redisplay format (if applicable)
- ▶ Defaults that occur as a result of omitted data
- ▶ Help message to display
- ▶ Selection window to use
- ▶ Valid range or set of values (if applicable)

Organization of fields

You define and determine the order of the input fields and field qualifiers in an input window script using the `.FIELD` command. (See [chapter 2, “Script,”](#) for more information on input window script commands.) Because the maintenance of your field qualifiers takes place in a script file (outside of your application), you can modify the characteristics of your input window or input fields without ever touching your actual code.

The following rules apply to field organization:

- ▶ Fields can be placed anywhere in an input window.
- ▶ Fields are organized into one or more data areas.
- ▶ A specific field can only be specified in one data area definition.
- ▶ Each data area can be associated with one or more input sets.
- ▶ An input set can consist of one or more fields from a specific data area.

Processing and editing field input

The input utilities support menu entries for moving between and editing input fields (`I_xxx` menu entries), for editing text (`E_xxx` menu entries), and for moving around selection windows (`S_xxx`). These menu entries are often referred to as reserved menu entries.

Applications on UNIX and OpenVMS require that you place the corresponding reserved menu entry on a placed menu column to make the function available to the user. Refer to [“Appendix B: Reserved Menu Entries”](#) for a list of the specific menu entries available. On Windows, these reserved menu entries are ignored, as functions are already a part of the Windows environment.

When UI Toolkit processes input for an empty field, it clears the field to blanks and then displays the default value for the field, if one exists. The user can do one of the following:

- ▶ Accept the default by pressing ENTER.
- ▶ Edit the default by selecting any editing function from the menu.
- ▶ Enter a new value in the field.

As soon as the user types any character in the field, the field is cleared and the typed character is the first character of the input. If the field doesn't have a default, input proceeds normally.

Field editing is automatically performed during all input processing. Therefore, if the user moves to a field that already contains data, the user can edit it.

For example, in a UNIX or OpenVMS environment, if the entry `“E_RIGHT”` is present on the menu and the user selects it, UI Toolkit will move the cursor to the right one character and place the input position at that character. The user can go into edit mode by selecting any one of the `“E_”` reserved menu entries listed in the [I_INPUT Discussion on page 8-56](#).

Let's assume a user of an application on UNIX or OpenVMS originally typed a telephone number as `“3334545,”` and the number was reformatted and displayed in the field as `“333-4545.”` If the user moved back to the field and selected `“E_RIGHT,”` `“333-4545”` would be cleared from the field and `“3334545”` would be redisplayed, left-justified. The user could then edit this data.

If the user presses the BACKSPACE or rubout key at a field that contains data, the field is cleared and the input is redone. If the user presses any other character, the field is cleared and the typed character is used as the first data of the input.

When the user types data in an input field and presses ENTER, that data goes into the data area associated with the input field in your program. If the user terminates the input field by selecting a menu entry (for example, a search function) instead of pressing ENTER, the field reverts back to its original contents. You must use the `I_FORCE` subroutine to force the data from the internal input field buffer into the associated data area in your program if this is the action desired. (See [I_FORCE on page 8-41](#) for information on overriding this default behavior.)

By default, Toolkit sets no time limit for input; it waits until I/O processing is completed before returning to the calling subroutine. You can, however, change the default, and you can override the default for a field. Then, if the user does not complete input within the time you specify, input times out. For more information, see the `.FIELD` qualifier [WAIT on page 2-72](#).

To facilitate redirection of input, UI Toolkit enables you to configure the number of successive end-of-file characters to allow on input. The field **g_eof_max** (in **tkctl.def**), which is set at 100 by default, specifies this maximum. After **g_eof_max** successive EOF characters (CTRL+D on UNIX, CTRL+Z on OpenVMS) have been encountered, a message is displayed on the screen and the program stops. You can set this field to zero to disable this behavior.

Text fields

Your input windows can contain dimensioned alpha fields that offer full editing capabilities similar to text windows. (Note that only single-dimension alpha arrays are supported. See [chapter 14, “Text Routines,”](#) for more information about text windows.) This feature is ideal for any field in which the user must enter a significant amount of text. For example, if a window in your application has a field for comments, the user would probably appreciate being able to edit the text that she types into that field.

The primary difference between edit processing in a text field and edit processing in a text window is that pressing ENTER in a text field terminates the field’s input (rather than inserting a new line at the cursor position). We made text input fields work this way so that they would terminate in the same way as all other input fields. But this shouldn’t be a problem; words automatically wrap to a new line during data entry. Also, you can override this action by modifying the value of **g_txt_rtrn** (defined in the **tools.def** file).

On Windows, arrayed text fields are displayed as a multi-line edit control. Because the sizing of this field includes extra leading for each line, the edit control will contain extra space into which no text can be entered. You can minimize the amount of extra space by setting the environment variable **MINIMIZE_LEADING** to 1 and using fixed fonts. When using proportional fonts, if the text field uses a smaller font than the input window, the input window’s background color will extend beyond the bottom of the edit control.

On Unix and OpenVMS, when the user begins performing input to a text field, any specified text editing menu column is placed on the menu bar, and the user can access these editing commands as if the field were a text window. A menu column entry that inserts a new line is available, if you choose to put it in your text editing menu column. You may decide to make only a subset of the editing commands available on the menu for text field entry, since some of the entries are already available for normal field editing, and the user may not need such functions as paragraph movement or top and bottom buffer movement for small amounts of text. See [“Appendix B: Reserved Menu Entries”](#) for a complete list of reserved menu entries.

Understanding display, input, and view lengths

Input fields have the following length settings:

- ▶ display length—the maximum number of characters that can be displayed in the field
- ▶ input length—the maximum number of characters that the user can enter in the field
- ▶ view length—the amount of space (width) on the screen that the field will take

Toolkit automatically calculates the display length and the input length by looking at field settings like size, type, and the format string. Toolkit then uses the greater of the display length or input length to determine the width of the field, which is referred to as the view length.

Typically, Toolkit's default calculations are sufficient, but you can override them if necessary.

- ▶ If you want a field to take up less room on the screen than the greater of the input or display length, set an explicit view length. On Windows this makes the contents of the field scrollable. On UNIX and OpenVMS, the contents of a field will be truncated to the view length if you move the cursor in the field or modify the contents of the field.
- ▶ Because the input length determines how many characters are passed to a change method, you may want to override the input length to enable users to input more or fewer characters than the field can store. For example, if you store a temperature in Celsius, but allow users to enter a temperature followed by either "F" or "C" to indicate Fahrenheit or Celsius, you could set the input length to 4, while the storage size might allow for only 2 digits. The change method could then convert Fahrenheit temperatures to Celsius before storing the value. See [INPUT_LENGTH](#), [NOINPUT_LENGTH](#) on page 2-54 for more information on overriding the input length.
- ▶ If you have a display method that requires a different display length than the default, you can override the display length. You could set this override, for example, if your display method formats data based on runtime considerations (so you can't use a format string) and you know you'll need more or fewer characters than the input length. See [DSP_LENGTH](#), [NODSP_LENGTH](#) on page 2-45 for more information on overriding the display length.

View length, display length, and input length can be specified in Repository, in a window script (as .FIELD qualifiers), or at runtime (with I_FLDMOD or IB_FIELD).

For more information on these settings, including information on their default values, see [DSP_LENGTH](#), [NODSP_LENGTH](#) on page 2-45, [INPUT_LENGTH](#), [NOINPUT_LENGTH](#) on page 2-54, and [VIEW_LENGTH](#), [NOVIEW_LENGTH](#) on page 2-71.

Building Input Windows at Runtime

In addition to generating input window definitions from script files, you can generate them at runtime. The subroutines that begin with "IB_" (for "input build") support runtime generation of input windows. These subroutines provide an alternative to defining your input windows in script files.

We recommend that you only use these subroutines when you cannot determine the contents of your input window until runtime. Loading windows from window libraries is faster and more modular than generating windows at runtime.

There is almost a one-to-one correspondence between the `IB_XXX` subroutines and the script commands that build an input window (along with `I_LDINP`). For example:

- ▶ `IB_INPUT` provides essentially the same functionality as the `.INPUT` script command.
- ▶ `IB_FIELD` gives you the same results as the `.FIELD` script command.
- ▶ `IB_PAINT` corresponds to the `.PAINT` script command.

These subroutines are also called in the same order as the input window script commands (`IB_INPUT`, `IB_FIELD`, ..., `IB_END`).

Each subroutine must be passed a *build_id* variable, which is initially returned by `IB_INPUT`. This variable should be an aligned **i4**.

Multiple fields, sets, and structures are supported, but the fields used in a set or structure must be defined before you define a set or structure. Also, the structure used in a set must be defined before the set.

`IB_RPS_STRUCTURE` must be called prior to the `IB_FIELD` calls for fields to be drawn from a repository structure. `IB_STRUCTURE` can only be used for local fields (fields not drawn from the Repository). Before either `IB_RPS_STRUCTURE` or `IB_LOCAL` is called, the default state is local.

`IB_INPUT` must be called once at the beginning and `IB_END` once at the end to complete the input window. After `IB_END` returns successfully, the *build_id* may be discarded. At this point, the input window is ready to use. Note that build warnings for `IB_XXX` subroutines display only if **g_dtkbounds** is set to 1 or 2. And if `IB_FIELD` creates a text field that is too large for a window, a warning will display only if **g_dtkbounds** is set to 2 or greater. See [g_dtkbounds on page 18-3](#).

See the “`IB_`” subroutine specification pages later in this chapter for the syntax, arguments, and discussion of these routines.

I_ACTIVATE – Specify characters for input termination



```
xcall I_ACTIVATE ([activation,] character[, ...])
```

Arguments

activation

(optional) One of the following options: **(n)**

D_ACTIVATE Activate specified character(s). (default)

D_DEACTIVATE Deactivate specified character(s).

character

One or more numeric values, separated by commas, that represent ASCII character(s). Valid values are 0 through 255. **(n)**

Discussion

The I_ACTIVATE subroutine enables you to activate a character or set of characters. This instructs Toolkit to treat the characters as ENTER. You can also deactivate characters, which instructs Toolkit *not* to treat them as ENTER. You can include both D_ACTIVATE and D_DEACTIVATE in the same I_ACTIVATE call. (See Examples below.)

I_ACTIVATE calls the ACCHR or DACHR Synergy DBL subroutine for each specified character. On subsequent field input, characters activated by this subroutine will behave like the ENTER key and terminate input for the field. Deactivated characters are returned to their normal state. They are accepted as printable characters unless they're entered as part of a shortcut. The activation state of a character remains constant until it's changed by another call to I_ACTIVATE; it is not affected by entering or exiting an environment.



%RDTRM returns the numeric value for an activated character if tested on a field break or within ECHKFLD_METHOD.

Examples

The following I_ACTIVATE call instructs UI Toolkit to treat the period key as ENTER.

```
xcall i_activate(%decml('.'))
```

The following example deactivates the period and hyphen keys. This returns these characters to their normal state as input characters. In addition, this example activates the TAB key. After this call, TAB will terminate a field as if the user pressed ENTER, and %RDTRM will return 9.

```
xcall i_activate(D_DEACTIVATE, %decml('.'), %decml('-'), D_ACTIVATE, 9)
```

I_CTRCONTEXT – Establish input context for child input window



```
xcall I_CTRCONTEXT (inp_id, [set_name], reason)
```

Arguments

inp_id

The ID of the child input window. **(n)**

set_name

(optional) The name of the input set to receive input context or ***CURR*** to use the currently active input set. ***CURR*** must be in capital letters and enclosed in quotation marks. **(a)**

reason

The reason the input window was activated, which determines what I_CTRCONTEXT will move input context to. This argument can be passed as blank (indicating that it was already activated), a program-supplied reason passed because it was activated programmatically, or one of the following: **(a)**

C_FIRST	Focus moved in attempt to reach the first child in the composite window. This instructs I_CTRCONTEXT to move context to the first active field in the input set. If there is no active field, context moves to the first button for the window.
C_LAST	Focus moved in attempt to reach the last child in the composite window. This instructs I_CTRCONTEXT to move context to the last active button on the tab set (if the window is on a tab). If there is no such button, it moves context to the last active button for the window. If there's no active button for the window, it moves context to the last input field in the input set.
C_NEXT	Focus moved from the previous child in the composite window—e.g., by pressing TAB. This instructs I_CTRCONTEXT to move context to the first active field in the input set. If there is no active field, context moves to the first button for the window.
C_PREV	Focus moved from the next child in the composite window—e.g., by pressing SHIFT+TAB. This instructs I_CTRCONTEXT to move focus to the last active button on the tab set (if the window is on a tab). If there is no such button, it moves focus to the last active button for the window. If there's no active button for the window, it moves focus to the last input field in the input set.
C_SET	Child explicitly activated. For example, this could be the result of a mouse click, or it could be that the child was activated programmatically

without a specified reason. With this reason, I_CTRCONTEXT does not change focus.

If *reason* is blank or any value other than the above, I_CTRCONTEXT does not change input context.

Discussion

Like I_NEXT, the I_CTRCONTEXT subroutine sets input context for an input window. Unlike I_NEXT, however, it uses a *reason* to determine where to set input context. This is because I_CTRCONTEXT is designed for use in child input window methods (C_METHINP by default), where you likely want input context to initially reflect the reason the input window received focus. For example, if you have a composite window with two input windows for children, and the user presses SHIFT+TAB to move from the second input window back to the first, you may want the last button or field in the first input window to receive input context.

To use in a child method, pass the reason passed to the child method (by C_PROCESS) as the *reason* argument for I_CTRCONTEXT. See the **ctrmeths.dbl** file included in your UI Toolkit distribution for an example. This file includes code for the C_METHINP method, which uses I_CTRCONTEXT.

The *set_name* argument determines which input set will be used to set input context.

- ▶ If the name of an input set in the window is passed as *set_name*, I_CTRCONTEXT uses this input set to determine input context.
- ▶ If *set_name* is passed as *CURR*, I_CTRCONTEXT uses the most recently accessed input set for the window. If no input set has been accessed, input context defaults to the first input set for the window to determine input set context.
- ▶ If *set_name* is not passed, I_CTRCONTEXT uses the first input set to determine input context.

Note that in a utility subroutine where you do not know the name of the currently active set, you can use *CURR* to specify the set that was most recently accessed in the window.

See also

- ▶ [C_METHINP](#) on page 6-51.
- ▶ [“Controlling tabbing, focus, and input context”](#) on page 6-4 for information on tabbing indices.
- ▶ [DC_CTRCONTEXT](#) on page 6-33 for information on moving focus to a specific child window.

I_DISABLE – Disable input fields



```
xcall I_DISABLE ([D_FLDS,] window_id, field_list, ...)
```

```
xcall I_DISABLE (D_SET, window_id, set_name)
```

```
xcall I_DISABLE (D_ALL, window_id)
```

Arguments

D_FLDS

(optional) Disables the specified input fields. (default)

D_SET

Disables all input fields in a set.

D_ALL

Disables all input fields in a window.

window_id

The ID of the input window containing the field(s) to be disabled. (**n**)

field_list

One or more field specifications. (**a**)

set_name

The name of the set containing the field(s) to be disabled. (**a**)

Discussion

The I_DISABLE subroutine disables one or more fields in an input window.

On Windows, the disabled fields will be grayed and unable to receive keyboard focus either by mouse click or by pressing TAB/ARROW/RETURN (ENTER).

You can specify up to nine field lists (*field_list* arguments), each of which can have from one to nine field specifications. If a *field_list* has multiple field specifications, separate the field specifications with any character *except* alpha characters, numeric characters, dollar sign (\$), underscore (_), and square brackets ([or]), which are used to enclose dimension specifications for repository arrayed fields). For example:

```
"name"
```

or

```
"name, birthday, soc_sec_no, employer, phone, policy_no, conditions[2]"
```

The maximum length of a *field_list* is 99 characters. Additional characters are ignored. For information on field specifications, see [“Field specifications” on page 2-35](#).

If any of the designated fields are not contained within the window, a fatal error occurs.

When using D_SET, it is important to remember that the field remains disabled even when you are not processing the set to which you applied D_SET. This is because enablement is a field-specific attribute. When using D_ALL, it is important to note that buttons aren't disabled. To disable buttons you should use the B_DISABLE subroutine.

Calling I_INPFLD on a disabled field returns immediately.

Note that when a break occurs in input processing the input set context for the next input has already been determined. Thus, altering the enabled state of a field in the set during break processing will not alter the input set context, even if you are enabling the field that would have been processed next. You must call I_NEXT to alter the context in any way. For example, consider the following scenario: An input set consists of fields A, B, and C. Field C is a break field. The user keys a value into field C and terminates by selecting the menu entry I_PREV, which creates a break. In the break processing, field B is disabled. The next field accessed is field C again, because the I_PREV logic had targeted field B as the next field to process. When it came time to process it, field B was disabled, thus field C was accessed instead. If, on the other hand, field B was disabled within the leave method for field C, then field A would be accessed next, because the input set context is determined after the leave method is called.

See also

- ▶ [I_ENABLE on page 8-19](#) for more information on enabling input fields.
- ▶ [I_NEXT on page 8-65](#) for more information on setting input set context when there are disabled fields.

Examples

The following example disables the fields **state**, **addr[2]**, and **zip**.

```
xcall i_disable(wndid, "state, addr[2], zip")
```


I_DISPLAY – Display data to input fields



```
xcall I_DISPLAY (window_id, [set_name], data_area, [field_list1], [field_list2],
&                [field_list3], [field_list4], [field_list5], [field_list6], [field_list7],
&                [field_list8], [field_list9][, a_methoddata1, ..., a_methoddata20])
```

Arguments

window_id

The ID of the input window containing the fields to be displayed. **(n)**

set_name

(optional) The name of the input set associated with the fields. **(a)**

data_area

The data area for the structure associated with the input set that contains the data to display. **(a)**

field_list1 through *field_list9*

(optional) Up to nine field specifications, or ***ENABLED*** to display enabled fields only. **(a)**

a_methoddata1 through *a_methoddata20*

(optional) Up to 20 additional data arguments. **(any)**

Discussion

The **I_DISPLAY** subroutine loads data in input fields according to the structure associated with the input set. For example, suppose you want to display some support information such as item name, sales amount, or delivery date while a transaction is being input. You can create an input field for each of the information displays (defining their data types and display formats) and then place these fields in their own input set. If you set breaks in the transaction input set at appropriate input fields, you can use **I_DISPLAY** to perform your information displays. (An additional benefit of this method is that it also places the controls for the information displays, along with the input fields, outside of the programs.)

If *set_name* is not present, **I_DISPLAY** defaults to the first input set.

I_DISPLAY loads data from *data_area* to the corresponding field(s) in the window and marks the field(s) “not empty.” Note that by default, Toolkit does not ensure that *data_area* is large enough for the fields it references. Toolkit generates an error for this condition only if **g_dtkbounds** is set to 1 or greater. See [g_dtkbounds on page 18-3](#).

The fields referenced by this subroutine must be part of an input set for the window.

You can specify up to nine field lists (*field_list* arguments), each of which can have from one to nine field specifications. If a *field_list* has multiple field specifications, separate the field specifications with any character *except* alpha characters, numeric characters, dollar sign (\$), underscore (_), and square brackets ([or] , which are used to enclose dimension specifications for repository arrayed fields). For example:

```
"name"
```

or

```
"name, birthday, soc_sec_no, employer, phone, policy_no, conditions[2]"
```

The maximum length of a *field_list* is 99 characters. Additional characters are ignored. For information on field specifications, see [“Field specifications” on page 2-35](#).

If you don’t specify any fields, all fields within the set will be displayed.

A_methoddata1 through *a_methoddata20* are up to 20 additional optional arguments that can be passed to I_DISPLAY. I_DISPLAY, in turn, passes them directly to %IDISPLAY_METHOD. This allows you to communicate additional information to your display method. Be careful to specify method data arguments only after you have passed *all* the field lists, so that *a_methoddata1* is the 13th argument passed, with any subsequent method data arguments following.

Note that if I_DISPLAY is used with an unframed field after the input window for the field has been added to a tab set, renditions will be applied to the field that may not match the renditions for the window. See [%TS_TABSET on page 13-7](#) for more information.

See also

- ▶ [.SET on page 2-97](#) for defining an input set in a script file.
- ▶ [.STRUCTURE on page 2-99](#) for defining a structure in a script file.
- ▶ [I_SETINF on page 8-80](#) for retrieving the names of input sets.

Examples

The following example loads all input fields in input set **maindcs** with data from the **dcsmast** data area.

```
xcall i_display(dcswnd, "maindcs", dcsmast)
```

The next example loads the input fields **name**, **addr[2]**, **city**, **st**, and **zip** using the data in the **cusmas** data area, and these fields reside in the **cusmas** input set.

```
xcall i_display(inpid, "cusmas", cusmas, "name, addr[2], city, st, zip")
```

The following example displays only the enabled fields in the input set **inpset**, and does not alter disabled fields. It passes **method_data** as the method data argument.

```
xcall i_display(inpid, "inpset", record, "*ENABLED*",,,,,,,  
& method_data)
```

I_DSPAREA – Set the display area for an input window



```
xcall I_DSPAREA (window_id, [orig_row], [orig_col], [height][, width])
```

Syntax

window_id

The ID of the input window to affect. **(n)**

orig_row

(optional) The new origin row for the display area. **(n)**

orig_col

(optional) The new origin column for the display area. **(n)**

height

(optional) The new height for the display area in number of rows. **(n)**

width

(optional) The new width for the display area in number of columns. **(n)**

Discussion

The I_DSPAREA subroutine sets the display area for an input window.

If any of the optional arguments are omitted, their corresponding display area coordinates are unaffected. The resulting values are passed directly to the Synergy DBL W_AREA subroutine, where any limitations are applied.

You should use this subroutine when setting the display area for an input window. After changing the actual display area, the input window's internal control data is updated to reflect the resulting display area. This ensures that the input processor will move the display area as needed to make input fields visible.

If you use this subroutine for a non-input window, a fatal error occurs. For non-input windows, directly call the W_AREA subroutine.

See also

[W_AREA](#) in the “Synergy Windowing API” chapter of the *Synergy DBL Language Reference Manual*.

Examples

The following example sets the display area for the input window whose ID is contained in **inpid**. The new display area will begin at row 1, column 1 in the window, and will be 10 rows high and 78 columns wide.

```
xcall i_dsparea(inpid, 1, 1, 10, 78)
```

The next example moves the display area origin row to row 10 of the window. The origin column, height, and width remain unchanged.

```
xcall i_dsparea(inpid, 10)
```

I_DSPFLD – Display data to one input field



```
xcall I_DSPFLD (window_id, field_spec, data[, a_methoddata1, ..., a_methoddata20])
```

Arguments

window_id

The ID of the input window containing the field to display. (**n**)

field_spec

The specification for the field that I_DSPFLD will display data to. (See “[Field specifications](#)” on [page 2-35](#) for information.) (**a**)

data

The data to display. (**a** or **n**)

a_methoddata1 through *a_methoddata20*

(optional) Up to 20 additional data arguments. (**any**)

Discussion

The I_DSPFLD subroutine displays *data* to an input field, regardless of any input set associations.

If either *window_id* or *field_spec* is invalid, a fatal error occurs.

Data is interpreted and displayed according to the input field’s definition, regardless of the actual type of *data*. The *data* is not put into any structure associated with the field, and a field does not have to be part of an input set. You can use this subroutine to access a field whether or not it is associated with any set.

Note that by default, Toolkit does not require the size of *data* to match the size of the referenced field (*field_spec*). If the sizes don’t match, Toolkit generates an error only if **g_dtkbounds** is set to 2 or greater (see [g_dtkbounds on page 18-3](#)). And note that Toolkit also generates a fatal error with this setting if an integer value is passed for a non-integer field or vice versa (though no error is generated for user-defined data types).

A_methoddata1 through *a_methoddata20* are up to 20 additional optional arguments that can be passed to I_DSPFLD. I_DSPFLD, in turn, passes them directly to %IDISPLAY_METHOD. This allows you to communicate additional information to your display method.

Note that if I_DSPFLD is used with an unframed field after the input window for the field has been added to a tab set, renditions will be applied to the field that may not match the renditions for the window. See [%TS_TABSET on page 13-7](#) for more information.

See also

- ▶ [I_INPUT on page 8-55](#) for more information on putting data into a field.
- ▶ [I_FORCE on page 8-41](#) for more information on forcing data to a field.

Examples

In the following example, the data “Smith” is loaded into the field **LSTNAME**:

```
last_name="Smith"  
xcall i_dspfld(wndid, "LSTNAME", last_name)
```

In the next example, the data “AB101” is loaded into the field **ORDNO[3]**:

```
ordno="AB101"  
xcall i_dspfld(wndid, "ORDNO[3]", ordno)
```

I_ENABLE – Enable input fields



```
xcall I_ENABLE ([D_FLDS,] window_id, field_list, ...)
```

```
xcall I_ENABLE (D_SET, window_id, set_name)
```

```
xcall I_ENABLE (D_ALL, window_id)
```

Arguments

D_FLDS

(optional) Enables all specified input fields. (default)

D_SET

Enables all input fields in a set.

D_ALL

Enables all input fields in a window.

window_id

The ID of the input window containing the field(s) to be enabled. **(n)**

field_list

One or more field specifications. **(a)**

set_name

The name of the set containing the field(s) to be enabled. **(a)**

Discussion

The I_ENABLE subroutine enables one or more fields in an input window. By default, input fields are enabled. Use I_ENABLE to re-enable disabled fields.

You can specify up to nine field lists (*field_list* arguments), each of which can have from one to nine field specifications. If a *field_list* has multiple field specifications, separate the field specifications with any character *except* alpha characters, numeric characters, dollar sign (\$), underscore (_), and square brackets ([or] , which are used to enclose dimension specifications for repository arrayed fields). For example:

```
"name"
```

or

```
"name, birthday, soc_sec_no, employer, phone, policy_no, conditions[2]"
```

The maximum length of a *field_list* is 99 characters. Additional characters are ignored. For information on field specifications, see [“Field specifications” on page 2-35](#).

If any of the designated fields are not contained within the window, a fatal error occurs.

When using `D_SET`, it is important to remember that the field remains enabled even when you are not processing the set to which you applied `D_SET`. This is because enablement is a field-specific attribute. When using `D_ALL`, it is important to note that buttons are not enabled. To enable buttons you should use the `B_ENABLE` subroutine.

Note that when a break occurs in input processing the input set context for the next input has already been determined. Thus, altering the enabled state of a field in the set during break processing will not alter the input set context, even if you are enabling the field that would have been processed next. You must call `I_NEXT` to alter the context in any way. For example, consider the following scenario:

- ▶ An input set consists of fields A, B, and C.
- ▶ Field B is disabled.
- ▶ Field A is a break field.

If a break occurs on field A, and the code executed after the break enables field B, the next field processed is still field C because the context for the next input was determined before the break occurred. To access field B next, `I_NEXT` should be used.

If, on the other hand, field B was enabled within a leave method for field A, field B would be the next field accessed, because the input context for the next input is determined after calling the leave method for a field.

See also

[I_DISABLE on page 8-11](#) for more information on disabling input fields.

Examples

The following example enables the fields **state** and **zip**.

```
xcall i_enable(inpid, "state, zip")
```


%I_FLDDIM – Retrieve an input field’s dimension or number of elements

WT	WN	U	V

elements = %I_FLDDIM(*window_id*, *field_spec*)

Return value

elements

An integer value indicating the number of elements. (^VAL)

Arguments

window_id

The ID of the input window containing the field. (n)

field_spec

The specification for the field whose dimension or number of elements %I_FLDDIM will retrieve. (See “[Field specifications](#)” on page 2-35 for information.) (a)

Discussion

%I_FLDDIM returns the number of elements for a field. If the field is not an array, the return value is 1, and if the field references a repository array field element, the return value is 1. Though this function is useful in a variety of situations, it can be particularly useful as an argument to the L_FINDSPEC subroutine.

If either *window_id* or *field_spec* is invalid, a fatal error occurs.

You do not need to declare this function to use it. It is defined in **tools.def**.

Examples

In this example, the beginning position and length of the find area for the list identified by **lstid** are determined by the structure position and scope (size * dimension) of the field named in **fndfield** within the input window identified by **inpid**. A usual scenario would have **inpid** identify the input window associated with the list, and the input window set’s structure would also double as the data passed to the list processor.

```
xcall l_findspec(lstid,, %i_fldstrpos(inpid, fndfield),
&                %i_fldsize(inpid, fndfield)*%i_flldim(inpid, fndfield))
```

I_FLDINF – Return input field information



```
xcall I_FLDINF (window_id, [field_spec], [field_number], [empty], [controls],
&                [selection_id][, enabled])
```

Arguments

window_id

The ID of the input window containing the field. **(n)**

field_spec

(optional) The specification for the field whose information I_FLDINF will retrieve. (See [“Field specifications” on page 2-35](#) for information.) **(a)**

field_number

(optional) The field number supplied/returned. **(n)**

empty

(optional) The returned empty status. **(n)**

controls

(optional) The returned field controls. **(a)**

selection_id

(optional) The returned selection window ID. **(n)**

enabled

(optional) The returned enabled status. **(n)**

Discussion

The I_FLDINF subroutine returns information about an input field. In order to use it, however, be sure to .INCLUDE these two distributed .def files (in the following order):

- ▶ **WND:system.def**
- ▶ **WND:fldinf.def**

Window_id must contain the ID of a loaded input window.

If *field_spec* is found and non-blank, the input field number is returned in *field_spec*. (If *field_spec* is not found, *field_number* is returned as 0.) If *field_spec* is blank, *field_number* must be non-zero and the field specification is returned in *field_spec*. In either case, *empty*, *controls*, and *selection_id* may optionally be returned.

The *field_number* argument identifies the ordinal number of the .FIELD statement within the specific window script or the ordinal number of the call to IB_FIELD within the program. This does not necessarily correspond to the position of the field within any input set, nor to a low-level window field number (for example, using W_FLDS). If you want to know a field's position within the input set, use I_SETINF to find out. If you want the number of the field in a low-level Synergy window, use **gs_fnum**, contained in the **gs_inpfld** record within the **fldinf.def** file (see “Notes on **gs_inpfld** fields for multi-line text fields” below).

Empty is returned true if the input field is currently empty.

Controls is the input field control record defined in the **gs_inpfld** record (in the file **fldinf.def**). For more information, see the comments in **fldinf.def**, and see “Notes on **gs_inpfld** fields for multi-line text fields” below. Note that to retrieve the strings referenced by the **gs_xxx** options using the function %I_GETSTRING, you must first call I_FLDINF with the *controls* argument to load the control structure. (See %I_GETSTRING on page 8-46 for more information.)

If *selection_id* is returned with a value of 0, the field is not associated with a selection window.

Enabled is returned true if the input field is currently enabled; false if it is disabled.

Notes on **gs_inpfld** fields for multi-line text fields

The fields **gs_edtrow**, **gs_edtcol**, **gs_edtltrow**, and **gs_edtltcol** apply only to multi-line text fields. For all other fields, they will be zero. They are used to establish and record the cursor position within a multi-line text field.

The first two fields, **gs_edtrow** and **gs_edtcol**, dictate the cursor position that is to be established when the field next receives focus. If they are zero, the field will exercise the default behavior of awaiting the first keypress from the user before entering edit mode. On Windows platforms, this means that the entire field contents will be highlighted.

If **gs_edtrow** is non-zero, it specifies the low-level field number of the row to focus. This can be transformed into the ordinal row within the field by subtracting **gs_fnum** (the first field number) and adding 1. Thus, if an input field's first field number (**gs_fnum**) is 4, and **gs_edtrow** is 6, then when the field receives focus, the cursor will be placed in the third row of that field.

If **gs_edtcol** is non-zero, it specifies the column within the field where the cursor will be placed. On Windows, the cursor will be placed before this position. On non-Windows, the cursor is placed on that column.

The **gs_edtltrow** and **gs_edtltcol** fields use the same coordinate system, but they refer to the last cursor position established within the field. Note that if the user accessed the field but never left the default initial state (in other words, they never entered edit mode), these fields will not be modified from their previous value. These fields are used by the Toolkit in two cases:

- ▶ As the reference position for a call to I_TXTPOS when using the D_LAST argument
- ▶ To set **gs_edtrow** and **gs_edtcol** when exiting a field that has the RETAIN_POSITION qualifier

For fields with the RETAIN_POSITION qualifier, **gs_edtrow** and **gs_edtcol** will usually contain the same value as **gs_edtltrow** and **gs_edtltcol**, respectively, unless I_TXTPOS has been called since the last input.

Note that I_TXTPOS never changes the value of **gs_edtltrow/gs_edtltcol**, so D_LAST always refers to the last input position, not the last position established by I_TXTPOS. D_CHANGE, however, refers to **gs_edtrow/gs_edtcol**, so those changes can be made cumulatively.

Examples

The following example returns the field number for field **fld1[2,5]** in the input window associated with the window ID (**id**).

```
xcall i_fldinf(id, "fld1[2,5]", fldnum)
```

The next example returns the name of field number three in **fldname**. If this field has an associated selection window, the selection window ID is returned in **selid**. Otherwise, **selid** contains zero upon return.

```
xcall i_fldinf(id, fldname, 3,,, selid)
```

The following example includes a calculation that determines the row position of the cursor in a multi-line field.

```
xcall i_fldinf(id, fldname,,, gs_inpfld)
row_num = gs_edtltrow - gs_fnum + 1
```

I_FLDMOD – Modify an input field’s attributes at runtime



```
xcall I_FLDMOD (window_id, field_spec, [length], [type][, qualifier, ...])
```

Arguments

window_id

The ID of the input window containing the field to be modified. **(n)**

field_spec

The specification for the field whose attributes I_FLDMOD will modify. See [“Field specifications” on page 2-35](#) for information. **(a)**

length

(optional) The maximum input length. Use *length* to shorten a field or re-lengthen a field after it has been shortened. The modified length must be greater than zero and less than or equal to the original length. **(n)**

type

(optional) The requested field type. Duplicates D_FLD_TYPE; see [“Type and D_FLD_TYPE” on page 8-29](#). **(n)**

qualifier

(optional) One or more of the following field qualifiers: **(a or n)**

D_FLD_DEC, *num_places*

Decimal places.

D_FLD_SIZE, *storage_size*

Storage size. See [“D_FLD_SIZE” on page 8-32](#) for size limits.

D_FLD_TYPE, *type*

Field type. See [“Type and D_FLD_TYPE” on page 8-29](#).

D_ON

Turn on one or more of the following field qualifiers. See [“D_ON and D_OFF” on page 8-29](#) for details. (default)

D_OFF

Turn off one or more of the following field qualifiers. When qualifiers are preceded by D_OFF, flags that normally require an argument do not. See [“D_ON and D_OFF” on page 8-29](#) for details.

D_FLD_ARRIVE, *arrive_method*

Arrive method.

D_FLD_ATTR, *attribute*

Attributes. See [“D_FLD_ATTR” on page 8-30](#) for valid values for *attribute*.

D_FLD_AUTO	Automatic input.
D_FLD_BREAK	Break on change.
D_FLD_BRK_ALWAYS	Break always.
D_FLD_BRK_RTRN	Break on return only.
D_FLD_BZRO	Blank if zero.
D_FLD_CHANGE , <i>change_method</i>	Change method.
D_FLD_CHECKBOX	Check box field.
D_FLD_COLOR , <i>palette#</i>	Color.
D_FLD_COPY	Copy default.
D_FLD_DATE , <i>date_type</i>	Date format. See “ D_FLD_DATE ” on page 8-30 for valid values for <i>date_type</i> .
D_FLD_DECR	Decrement default.
D_FLD_DISABLED	Disabled.
D_FLD_DISPLAY , <i>display_method</i>	Display method.
D_FLD_DFLT , <i>string</i>	Default entry.
D_FLD_DRILL , <i>drill_method</i>	Drill method.
D_FLD_DSLEN , <i>display_len</i>	Display length. Cannot be specified for check boxes, radio buttons, selection windows, or multi-line fields.
D_FLD_EDITFMT , <i>editfmt_method</i>	Edit format method.
D_FLD_ENUM , [<i>dlen</i>],[<i>base</i>],[<i>step</i>]	Enumeration properties. See “ D_FLD_ENUM ” on page 8-30.
D_FLD_FMT , <i>string</i>	Format string.
D_FLD_FONT , <i>font_id</i>	Field font. See “ D_FLD_FONT and D_FLD_PROMPTFONT ” on page 8-31 for valid values for <i>font_id</i> .
D_FLD_FPOS [, D_CHANGE], <i>row</i> , <i>col</i>	Field position. See “ D_FLD_FPOS and D_FLD_POS ” on page 8-31.
D_FLD_HELP , <i>string</i>	Help identifier.
D_FLD_HYPERLINK , <i>hyper_method</i>	Hyperlink method.
D_FLD_INCR	Increment default.
D_FLD_INFO , <i>string</i>	Information text.

D_FLD_INPLEN , <i>input_len</i>	Maximum input length. Cannot be specified for check boxes, radio buttons, selection windows, or multi-line fields.
D_FLD_JUST , <i>just_flag</i>	Justification, where <i>just_flag</i> is D_JST_CENTER, D_JST_LEFT, or D_JST_RIGHT.
D_FLD_LEAVE , <i>leave_method</i>	Leave method.
D_FLD_NEG	Negative allowed.
D_FLD_NEGONLY	Negative allowed only.
D_FLD_NEGORZERO	Negative or zero allowed only.
D_FLD_NODEC	No decimal required.
D_FLD_NOECHO	No echo.
D_FLD_NOECHR , <i>char</i>	Fill character when D_FLD_NOECHO is specified (otherwise ignored).
D_FLD_NOTERM	No termination.
D_FLD_PAINT , <i>char</i>	Paint character.
D_FLD_POS [, D_CHANGE], <i>row, col</i>	Prompt position. See “ D_FLD_FPOS and D_FLD_POS ” on page 8-31.
D_FLD_PROMPTFONT , <i>font_id</i>	Prompt font. See “ D_FLD_FONT and D_FLD_PROMPTFONT ” on page 8-31 for valid values for <i>font_id</i> .
D_FLD_RADIO	Radio buttons.
D_FLD_RANGE , <i>min, max</i>	Range. See “ D_FLD_RANGE ” on page 8-32.
D_FLD_READONLY	Read-only. Not valid for check boxes, radio buttons, selection lists and windows. See “ D_FLD_READONLY ” on page 8-32.
D_FLD_REQUIRED	Required input.
D_FLD_RETPOS	Retain position in text field.
D_FLD_TIME , <i>time_type</i>	Time format. See “ D_FLD_TIME ” on page 8-32 for valid values for <i>time_type</i> .
D_FLD_TIMESTAMP , <i>format</i>	Timestamp. See “ D_FLD_TIMESTAMP ” on page 8-33 for details on this option and valid values for <i>format</i> .
D_FLD_UC	Uppercase only.

D_FLD_USER , <i>string</i>	User-defined text string.
D_FLD_VIEWLEN , <i>view_length</i>	The width of the field (i.e., the width of the area on the screen that will display data for the field). Cannot be specified for check boxes or radio buttons.
D_FLD_WAIT , <i>wait_time</i>	Input time-out for the field, where <i>wait_time</i> is the number of seconds to wait or one of the following:

Discussion

I_FLDMOD modifies various input field attributes at runtime. You can use I_FLDMOD to change the fields in your current input set. Note that I_FLDMOD changes only the field's display, not the internal input buffer layout. In order to prevent invalid data, *field_spec* is always initialized, so alterations to the displayed data are reflected on the next input or display.



I_FLDMOD's options are also available with the .FIELD script command. For more detailed information on I_FLDMOD options, see the corresponding qualifier in ["FIELD qualifiers" on page 2-36](#).

If either *window_id* or *field_spec* is invalid, a fatal error occurs. Any D_FLD_XXX argument flag not listed above will result in the fatal error "Invalid I_FLDMOD code (*code*)," where *code* is the invalid value passed. If you specify a qualifier (other than D_FLD_DEC) that does not apply to the field's destination type, it is ignored.

You can specify methods with the following qualifiers even if the field did not originally define them: D_FLD_ARRIVE, D_FLD_CHANGE, D_FLD_DISPLAY, D_FLD_DRILL, D_FLD_EDITFMT, D_FLD_HYPERLINK, D_FLD_LEAVE.

For all the qualifiers that have text string arguments (such as D_FLD_USER), you can only specify a *string* that is equal to or shorter than the original text string. If you shorten the length, you can re-lengthen or restore the text string for the field as long as it doesn't exceed the length of the original string; if it does exceed the original length, the modification will be ignored.

User-defined field methods are passed the user type and user text strings as arguments. Thus, it is important to know the order in which methods are invoked if the developer modifies either of these strings within a method. The input field methods are invoked in the following order: arrive method, EEDTDSP_METHOD, %IEDITFMT_METHOD, ECHKFLD_METHOD, change method, EDSPFLD_METHOD or %IDISPLAY_METHOD, leave method, and drill or hyperlink method.

The default qualifiers are also inspected when arriving upon the field so the default value can be displayed. Thus, modifying the default value within a method (other than the arrive method) will not cause that new default value to be used if the user presses ENTER.

Because the information line is displayed when arriving on a field, modifying a field's information line within a method (other than the arrive method) may not take visible effect until the next time the field is arrived upon. Otherwise, to immediately affect the information line, use either E_SECT to modify it or %M_SIGNAL to exit the method and force the arrival on the field to occur again.

Type and D_FLD_TYPE

Type is retained for compatibility with earlier versions of Toolkit. It duplicates the functionality of D_FLD_TYPE, except that it does not support D_USERTYPE. A user-defined field can only be specified using D_FLD_TYPE.

Type and D_FLD_TYPE supports the following options for *type*:

D_ALPHA	Alphanumeric
D_DECIMAL	Decimal
D_INTEGER	Integer
D_NUMERIC	Numeric (same as decimal)
D_USERTYPE, <i>type_string</i>	User-defined (D_FLD_TYPE only)

I_FLDMOD will change the field type between alpha, decimal, and integer; requests to change to or from text fields will be ignored.

If you change the field type *from* alpha, the following defaults apply unless you override them:

- ▶ Zero decimal places (D_FLD_DEC, 0)
- ▶ Negative not allowed (D_OFF, D_FLD_NEG)

If you change the field type *to* alpha, the following defaults apply unless you override them:

- ▶ Lowercase allowed (D_OFF, D_FLD_UC)
- ▶ No echo character is blank (D_FLD_NOECHR)
- ▶ No matching case (.FIELD's MATCH(NOCASE))

If you change the field type to D_INTEGER, the storage size is changed to the nearest (but not larger) valid integer storage size. Valid sizes are 1, 2, 4, or 8. Integer fields support all qualifiers that are valid for decimal fields without precision.

If you want the field to become an implied-decimal field, you can pass D_FLD_DEC, *num_places* as a qualifier. Regardless of its original setting, any field type can be converted to implied-decimal. Note that you can specify up to nine digits after the decimal point.

D_ON and D_OFF

The D_ON and D_OFF codes indicate whether the field qualifiers that follow in the argument list are to be turned on or off. D_ON is the default state for any qualifiers listed.

When qualifiers are preceded by D_OFF, flags that normally require an argument do not. For example, the first I_FLDMOD call below turns no echo on. The second call turns it off.

```
xcall i_fldmod(inpid, "myfield",,,, D_FLD_NOECHR, '*')
xcall i_fldmod(inpid, "myfield",,,, D_OFF, D_FLD_NOECHR)
```

Turning off D_FLD_BRK, D_FLD_BRK_ALWAYS, or D_FLD_BRK_RTRN results in a field with no break.

D_FLD_ATTR

D_FLD_ATTR accepts the following values for *attribute* (defined in **tools.def**):

A_ALL	All attributes (same as A_BHRU)
A_NONE	No attributes
A_B A_I	Blinking (UNIX and OpenVMS) or italic (Windows). “I” can be substituted whenever “B” appears in the following flags.
A_BH	Blinking, highlighted
A_BHR	Blinking, highlighted, reverse
A_BHRU	Blinking, highlighted, reverse, underlined
A_BHU	Blinking, highlighted, underlined
A_BR	Blinking, reverse
A_BRU	Blinking, reverse, underlined
A_BU	Blinking, underlined
A_H	Highlighted
A_HR	Highlighted, reverse
A_HRU	Highlighted, reverse, underlined
A_HU	Highlighted, underlined
A_R	Reverse
A_RU	Reverse, underlined
A_U	Underlined

D_FLD_DATE

D_FLD_DATE accepts the following values for *date_type*:

D_DATE_Y2J	YYJJJ date format
D_DATE_Y2MD	YYMMDD date format
D_DATE_Y2P	YYPP date format
D_DATE_Y4J	YYYJJJ date format
D_DATE_Y4MD	YYYYMMDD date format
D_DATE_Y4P	YYYYPP date format

To make the field default to the today’s date, add D_DATE_TODAY to any of the above. For a short date, add D_DATE_SHORT to any of the above. (You can add both of these if desired,)

D_FLD_ENUM

To use D_FLD_ENUM, the field must already be an enumerated field. You cannot use this to turn a non-enumerated field into an enumerated field, and you cannot use D_OFF with this qualifier to change an enumerated field into a non-enumerated field. (D_OFF causes D_FLD_ENUM to be ignored.)

A display length (*dlen*) set with D_FLD_ENUM overrides any previously specified display length (for example, any display length set with D_FLD_DSLEN).

When using D_FLD_ENUM, you must include all comma placeholders for the arguments (even those you don't pass) unless this is the last qualifier for the I_FLDMOD call, in which case you must include only the comma placeholders that precede the last argument specified for D_FLD_ENUM. If you omit an argument, the corresponding enumeration property will remain unchanged.

D_FLD_FONT and D_FLD_PROMPTFONT

The *font_id* argument to D_FLD_FONT and D_FLD_PROMPTFONT can be the name of a font palette entry, a font handle, or one of the following:

DF_ALTERNATE	Alternate font
DF_CURRENT	Current font
DF_DEBUGGER	Debugger font
DF_DEFAULT	Global font

If you change the type of a field (for example, alpha to numeric) and you are using any of the FONT_ initialization settings, be aware that the corresponding font isn't automatically applied. To update the font, use D_FLD_FONT. (Note that if a view length is specified on Windows, changing the font may change the width of the field. See [VIEW_LENGTH, NOVIEW_LENGTH](#) on [page 2-71](#) for information.)

Because a list has only one font setting, using I_FLDMOD to change the font for a field in a list has no effect. With the ActiveX Toolkit list, however, you can manipulate the font for individual cells by using the API Extension methods SetCellFont, SetHeaderCellFont, and SetFooterCellFont.



On Windows, do not use I_FLDMOD to change font settings for a column in a Toolkit list or an ActiveX implementation of a Toolkit list. This doesn't work on Windows: the font will revert to the standard typeface and size, and data in the column will be blanked out. This is because input field fonts are not applied to lists. For an ActiveX Toolkit list, you can use the SetCellFont method to change the font of a specific cell.

I_FLDMOD settings for an ActiveX Toolkit list will not override cell-level fonts or colors specified by the SetCellFont and SetCellColor ActiveX Toolkit list methods.

D_FLD_FPOS and D_FLD_POS

D_FLD_POS modifies the position of the prompt only. It does not affect the field's position (unlike changing the default field position for IB_FIELD or .FIELD). To modify the field position, you must use D_FLD_FPOS. You must also use D_FLD_FPOS to modify the position of a check box prompt on Windows because the position of a check box's prompt is determined by its field position. If either position places any part of a field or prompt outside the window, the positions are automatically adjusted so that the field and prompt fit in the window.

D_FLD_RANGE

D_FLD_RANGE can be specified only for input fields that had a range originally, either using the RANGE qualifier in Script, or using IB_FIELD's D_FLD_RANGE. The field must be numeric, and *min* and *max* must not exceed the field's storage size. If you use variables for *beg* and *end*, they must be alpha variables; decimal variables will not work.

D_FLD_READONLY

If you use D_FLD_READONLY for a field that can't be set to read-only (i.e., a check box, a radio button set, or a field with a selection list or a selection window), the read-only setting will be accepted, but it won't affect the field. (For these field types, use I_DISABLE rather than I_READONLY.) However, if the field is subsequently changed to a regular text field, Toolkit may use this setting because when a field is changed to a text field, the most recent read-only state setting is used to determine whether the field will be read-only or read/write.



You can also set a field's read-only state with I_READONLY. For information on the differences between D_FLD_READONLY and I_READONLY, see [I_READONLY](#) on [page 8-72](#).

For information on changing read-only settings from an input method for the field, see “[Notes on Input Methods](#)” on [page D-3](#).

D_FLD_SIZE

Alpha and alpha array fields can have up to 65,535 characters. Decimal fields (including implied decimal fields) have a maximum size of 28, and for implied-decimal fields, up to 28 digits can follow the decimal point. Integer fields on any operating system can have 1, 2, 4, or 8 digits, depending on size.

D_FLD_TIME

D_FLD_TIME accepts the following values for *time_type*:

D_TIME_HM	HHMM time format
D_TIME_HMS	HHMMSS time format

To make the field default to the current time, add D_TIME_NOW to either of the above. To add AM/PM to the display, add D_TIME_AMPM to either of the above. (You can add both of these if desired.)

D_FLD_TIMESTAMP

If D_OFF is in effect, D_FLD_TIMESTAMP changes a timestamp field to a standard read-only 8-byte integer field. Do not use the *format* argument when D_OFF is in effect. (It will be interpreted as another qualifier.)

If D_ON is in effect, D_FLD_TIMESTAMP *must* be followed by the *format* argument, which enables you to change the formatting used for a timestamp field. The *format* argument can be one or both of the following, or 0 (for neither):

D_TIMESTAMP_AMPM	Use 12-hour notation and AM/PM designation
D_TIMESTAMP_SHORT	Use short display format (no century, seconds, or milliseconds)

To specify both D_TIMESTAMP_AMPM and D_TIMESTAMP_SHORT, use the bitwise OR operator (.BOR. or |) or add them. If a flag isn't passed, it will be reset to its default value. For example, if you don't pass D_TIMESTAMP_SHORT, the long display format will be used even if the field was previously set to use the short format.

For D_ON, D_FLD_TYPE must be set to D_INTEGER, and the field must be a read-only, 8-byte integer. It cannot be a selection field, a radio button set, or a check box. Otherwise, an error will occur.

D_FLD_WAIT

The *wait_time* argument to D_FLD_WAIT can be the number of seconds to wait or one of the following:

D_FOREVER or -1	Wait until I/O processing is completed
D_GLOBAL or -2	Use the globally specified time-out
D_IMMEDIATE or 0	Time out immediately

See also

- ▶ [.FIELD on page 2-31](#) for more information about the various field options.
- ▶ [“Renditions” on page 2-34](#) for information and notes on setting field-level renditions.
- ▶ [“Using Fonts on Windows” on page 18-40](#) for information on fonts and font palettes.

Examples

The following is a sample interface for a room scheduler. This program assumes two different companies in the same building use this scheduler. Company Alpha likes to describe all of its rooms with an alpha field, such as “Conference room,” “Laboratory,” and “Break room.” Company Numeric, however, prefers to use room numbers that are no larger than 9999. Company Alpha does not allow a blank room name, but Company Numeric allows a room #0.

The first field in this input window asks whether Company Alpha or Company Numeric is entering the schedule, and it modifies the room field accordingly.

```
.include "WND:tools.def"
```

```

record
  id          ,i4
  colid       ,i4

record data
  company     ,a30
  room        ,a20
  time        ,a5
  ampm        ,a2

proc

;Set up stuff
xcall u_start("rmsched")
xcall i_ldinp(id, g_utlib, "rmsched")

;Start input processing

do forever
  begin
    xcall i_input(id,, data)
    if (g_setsts) then                ;At company break field
      begin
        if (company .eq. "Numeric Company") then
          xcall i_fldmod(id, "room", 4, D_NUMERIC, D_OFF, D_FLD_REQUIRED)
        else
          xcall i_fldmod(id, "room", 20, D_ALPHA, D_ON, D_FLD_REQUIRED)
        end
      end
    else
      begin                          ;Process the input
        xcall i_init(id)
      end
    end                               ;End of "do forever"
  xcall u_finish
end

```

The input window looks like this:

```

.input rmsched, 10, 65
.field company, a30, req, break, prompt("Company Name "),-
  sel(0, 30, 2, "Alpha Company", "Numeric Company")
.field room, a20, pos(2, 9), prompt("Room ")
.field time, d5, pos(3, 9), prompt("Time "), format("ZX:XX")
.field ampm, a2, sel(0, 2, 2, "am", "pm"), pos(change, 0, 1)
.end

```

The following example changes the **custno** field to type alpha, with uppercase input only.

```

xcall i_fldmod(inpid, "custno",,, D_FLD_TYPE, D_ALPHA, D_FLD_UC)

```

The next example removes the “required” status from the field **creditref**.

```
xcall i_fldmod(inpid, "creditref",,, D_OFF, D_FLD_REQUIRED)
```

The following example modifies the field **date** to default to today’s date.

```
xcall i_fldmod(inpid, "date",,, D_FLD_DATE, D_DATE_Y4MD+D_DATE_TODAY)
```

%I_FLDPREC – Retrieve an input field’s precision

WT	WN	U	V

precision = %I_FLDPREC (*window_id*, *field_spec*)

Return value

precision

This function returns the precision as an integer value. (^VAL)

Arguments

window_id

The ID of the input window containing the field. (n)

field_spec

The field specification for the field whose precision %I_FLDPREC will retrieve. (See “[Field specifications](#)” on page 2-35 for information.) (a)

Discussion

%I_FLDPREC returns the precision (number of places to the right of the decimal point) for the field. If the field is not an implied-decimal field, the return value is zero.

If either *window_id* or *field_spec* is invalid, a fatal error occurs.

You do not need to declare this function to use it. It is defined in **tools.def**.

Examples

The following example returns the number of decimal places for field **amount**.

```
numdec=%i fldprec (wndid, "amount")
```


I_FLDSEL – Assign a new selection window to an input field



```
xcall I_FLDSEL (window_id, field_spec, [selection_id][, row, column])
```

Arguments

window_id

The ID of the input window containing the field. **(n)**

field_spec

The specification for the field that the selection window will be assigned to. (See “[Field specifications](#)” on page 2-35 for information.) **(a)**

selection_id

(optional) The ID of the selection window. **(n)**

row

(optional) The row offset from the field at which to place the upper-left corner of the selection window. **(n)**

column

(optional) The column offset from the field at which to place the upper-left corner of the selection window. **(n)**

Discussion

The I_FLDSEL subroutine assigns a selection window to a field.

If either *window_id* or *field_spec* is invalid, a fatal error occurs.

If *selection_id* is passed and a local selection window is already assigned to the field (SELECT qualifier on . FIELD), it will be deleted. The specified window will be assigned to the field and marked as non-local. It will not be considered owned by the input window. In this situation, the selection window will not be deleted when you delete the input window. Deleting the selection window becomes your responsibility.

If *selection_id* is omitted or has a value of zero, the specified field will not display a selection window. Thus, you have the ability to assign or deassign a selection window at runtime. (For example, you may want to use the same input field to enter an existing code and define a new code. You can assign the selection window to your field with one call to I_FLDSEL, and then remove the selection window from the field with another call to I_FLDSEL.)

If you don't specify *row* and *column*, the new selection window will be placed where the old one was. If there was not a selection window before, the selection window will be placed immediately below the field.

See also

- ▶ [S_SELBLD on page 12-5](#) and [S_SELLD on page 12-17](#) for information about building selection windows dynamically within your programs (“on-the-fly”).
- ▶ [U_WINDOW on page 16-159](#) for information about deleting windows.

Examples

The following example assigns the **yesno** selection window to the **taxable** field in the **arcust** input window. The selection window is placed two rows below and three columns to the right of the beginning of the **taxable** field.

```
xcall i_ldinp(inpwd, g_utlib, "arcust")
xcall u_ldwnd(selwd, g_utlib, "yesno")
xcall i fldsel(inpwd, "taxable", selwd, 2, 3)
```

%I_FLDSIZE – Retrieve an input field’s size



size = %I_FLDSIZE(*window_id*, *field_spec*)

Return value

size

This function returns the field size as an integer value. (^VAL)

Arguments

window_id

The ID of the input window containing the field. (n)

field_spec

The specification for the field whose size %I_FLDSIZE will retrieve. (See “[Field specifications](#)” on page 2-35 for information.) (a)

Discussion

%I_FLDSIZE returns the storage size of the field in bytes. If the field is an array, this is the size of one element.

If either *window_id* or *field_spec* is invalid, a fatal error occurs.

This function may be particularly useful as an argument to the L_FINDSPEC subroutine, although it has a variety of uses.

You do not need to declare this function to use it. It is defined in **tools.def**.

Examples

In this example, the beginning position and length of the find area for the list identified by **lstid** are determined by the structure position and scope (size * dimension) of the field named in **fndfield** within the input window identified by **inpid**. A usual scenario would have **inpid** identify the input window associated with the list, and the input window set’s structure would also double as the data passed to the list processor.

```
xcall l_findspec(lstid,, %i_fldstrpos(inpid, fndfield),
&      %i_fldsize(inpid, fndfield)*%i_flddim(inpid, fndfield))
```

%I_FLDSTRPOS – Retrieve an input field’s structure position

WT	WN	U	V

```
struct_pos = %I_FLDSTRPOS (window_id, field_spec)
```

Return value

struct_pos

This function returns the position as an integer value. (^VAL)

Arguments

window_id

The ID of the input window containing the field. (n)

field_spec

The specification for the field whose structure position %I_FLDSTRPOS will retrieve. (See [“Field specifications” on page 2-35](#) for information.) (a)

Discussion

%I_FLDSTRPOS returns the starting character position of the field within its local or repository structure. If the field is not associated with a structure, the return value is zero. If the field was created from an element in repository array field, %I_FLDSTRPOS returns the starting character position for the element (rather than the starting character position for the beginning of the array).

If either *window_id* or *field_spec* is invalid, a fatal error occurs.

This function may be particularly useful as an argument to the L_FINDSPEC subroutine, although it has a variety of uses.

You do not need to declare this function to use it. It is defined in **tools.def**.

Examples

In this example, the beginning position and length of the find area for the list identified by **lstid** are determined by the structure position and scope (size * dimension) of the field named in **fndfield** within the input window identified by **inpid**. A usual scenario would have **inpid** identify the input window associated with the list, and the input window set’s structure would also double as the data passed to the list processor.

```
xcall l_findspec(lstid,, %i_fldstrpos(inpid, fndfield),
& %i fldsize(inpid, fndfield)*%i flddim(inpid, fndfield))
```

I_FORCE – Force data to or from an input field



```
xcall I_FORCE([new_data][, old_data])
```

Arguments

new_data

(optional) The new data to force. **(a)**

old_data

(optional) Returned with the current contents of the force buffer. **(a)**

Discussion

The I_FORCE subroutine forces a data string as input to the next field. This operation is equivalent to entering the desired data for the field from the terminal. (Note, however, that this routine does not do any of the actual forced data processing. It simply sets up the force buffers so the field input routine can access the data. The data will be the input delivered to the next field processed by I_INPUT or I_INPFLD.)

The force buffer is the location to which UI Toolkit copies the contents of an input field if the user selects a menu entry before pressing ENTER. (Note that the length of the data in the force buffer is not set.) The field is then reset to its initial entering state. You can use I_FORCE with the *old_data* argument to obtain the field's contents at the time the menu entry was selected; if *old_data* is passed, the current contents of the force buffer are copied to *old_data* prior to any new data processing.

If *new_data* is passed, its contents are loaded into the force buffer. If a null string ("") is passed as *new_data*, the force buffer is cleared and any pending I_FORCE is canceled.

If I_FORCE is being used in a selection window field and *new_data* is not a selection in the selection window, a beep will sound and *new_data* will not be accepted into the field. UI Toolkit will then wait for input (with the selection window dropped down if the field is empty).

The force buffer is cleared when a field exits without pending input. As a result, I_FORCE returns only the pending input for the last field accessed.

A call to I_FORCE is valid at any time. If a hot entry is processed by EUTILS_METHOD during input window processing and I_FORCE is called, the input field that was "interrupted" will have the value of the I_FORCE argument.

You can override UI Toolkit's default behavior of canceling pending input by registering an EENTRST_METHOD subroutine. See ["Environment Routines" on page 7-1](#) for more information.

See also

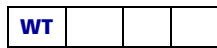
- ▶ [I_PUTFLD on page 8-70](#) for information about putting data into a field.
- ▶ [I_DSPFLD on page 8-17](#) for information about displaying data in a field.

Examples

This example searches for the data string input by the user and places the string it finds back into the input field.

```
xcall i_input(wnd,, data, inpcol, selcol, edtccl)
if (g_select)
  case g_entnam of
    begincase
      "SEARCH":                                ; If input terminated by user
        begin                                  ; and user wants to look up
          xcall i_force(, string)               ; a name, see if user entered
          xcall lookup(g_fldnam, string)        ; a partial name. Go do
          xcall i_force(string)                 ; the search and put the
        end                                     ; found information back
        .                                       ; into the field.
        .
        .
      endcase
```

I_FRAMES – Display frames on input fields



```
xcall I_FRAMES (window_id[, set_name])
```

Arguments

window_id

The ID of the input window. (**n**)

set_name

(optional) The set name containing the fields to frame, or “***NONE***” to remove all frames. (The latter must be uppercase and enclosed in quotation marks.) (**a**)

Discussion

Generally, input fields have frames around them only when the input set they belong to is being processed using I_INPUT. Using I_FRAMES, however, you can display frames on input fields within a given set without calling I_INPUT. (On UNIX and OpenVMS, this routine is ignored.)

Only the fields contained in the specified set will be framed. (This allows for display-only fields that function as labels.) If *set_name* is not passed, then the default (first) will be used. Frames can only be displayed on one set at a time. If *set_name* is the special value “***NONE***”, then all field frames will be removed from the window.

Note that the window must be placed before you pass its ID (as *window_id*) to I_FRAMES.

If you use I_FRAMES to remove the frame from a field that has been displayed or cleared since its input window was added to a tab set, renditions will be applied to the field that may not match the renditions for the window. See [%TS_TABSET on page 13-7](#) for more information.

Examples

The following example displays frames on all the input fields in the input set **set1**.

```
xcall i_frames(id, "set1")
```

The next example clears all the frames from the input window **inpid**.

```
xcall i_frames(inpid, "*NONE*")
```

I_GETFLD – Get data from an input field’s data area

WT	WN	U	V

```
xcall I_GETFLD (window_id, [set_name], data_area, field_spec, data)
```

Arguments

window_id

The ID of the input window containing the field. (**n**)

set_name

(optional) The name of the input set. (**a**)

data_area

The data area for the input set. (**a**)

field_spec

The specification for the field whose data area data will be retrieved from. (See “[Field specifications](#)” on page 2-35 for information.) (**a**)

data

Returned with the obtained data. (**a** or **n**)

Discussion

The I_GETFLD subroutine gets data from an input field’s data area.

If either *window_id* or *field_spec* is invalid, a fatal error occurs.

If you omit *set_name*, it defaults to the first input set.

Data_area must correspond to the structure associated with the input set.

The data is loaded into *data* according to the definitions in the input window, regardless of the actual type of *data*. If the data type is text, *data* is assumed to be the first element of an array dimensioned as specified in the field’s definition.

Note that by default, Toolkit does not ensure that the size of *data_area* or the size of *data* matches the size of the referenced field (*field_spec*) or that they are even large enough for the field.

However, bounds checking can check for the following conditions. (See [g_dtkbounds](#) on page 18-3 for more information.)

- ▶ If **g_dtkbounds** is set to 1, Toolkit generates a fatal error if *data_area* is not large enough for the field.
- ▶ If **g_dtkbounds** is set to 2 or greater, Toolkit generates a fatal error if *data_area* is not large enough (as it does if **g_dtkbounds** is set to 1). And it generates a non-fatal error if *data* is smaller than the field.

Examples

The following example locates the position of the **fax** field in **cusmas** and copies the data to **fax_number**.

```
xcall i_getfld(inpid,, cusmas, "fax", fax_number)
```

The next example locates the position of the **addr** field in **cusmas** and copies the data to **address**.

```
xcall i_getfld(inpid,, cusmas, "addr", address)
```

%I_GETSTRING – Retrieve a string referenced by an input window data pointer



```
string = %I_GETSTRING(window_id, ctrl_pointer[, length])
```

Return value

string

The string referenced by the specified pointer, or a single space if *ctrl_pointer* is null. (a)

Arguments

window_id

The ID of the input window. (n)

ctrl_pointer

One of the following 16-bit pointers, which are defined in the distributed **fldinf.def** and **inpinf.def** files: (i)

gs_hlpptr	Help identifier.
gs_fmtptr	Format.
gs_info	Information line.
gs_dfltptr	Default value or string.
gs_usrptr	User-defined string.
gs_usrtyp	User-defined data type.
gs fldnam	Name of the field.
inp_setnam	Name of current input set.
inp fldnam	Name of current input field.
inp_usertype	User-defined data type.
inp_usertext	User text string.

length

(optional) The returned length of the string. (n)

Discussion

The `%I_GETSTRING` function enables you to retrieve the strings referenced by an input window's internal pointers when you use the control argument in `I_FLDINF`.

The *ctrl_pointer* options from the **gs_pts** array in **fldinf.def** are valid after calling `I_FLDINF` to load the control structure. The *ctrl_pointer* options from **inpinf.def** are valid while within a method subroutine.

In the case of a null pointer, if passed, *length* will be returned as zero.

Ctrl_pointer is assumed to be valid if non-null. Make sure that only pointers created during construction of the input window are used. An invalid pointer cannot be detected by this routine, and its use will have extremely unpredictable results.

You do not need to declare this function to use it. It is declared in **tools.def**.

%I_INFO – Return general information about an input window

WT	WN	U	V

return = %I_INFO (*window_id*, *qualifier*)

Return value

return

A value specific to the requested information. (^VAL)

Arguments

window_id

The ID of the input window whose information you want returned. (n)

qualifier

One of the following: (n)

D_NMFLDS Return the number of input fields in the window.

D_NMSETS Return the number of input sets in the window.

Discussion

The %I_INFO function returns information about an input window.

Window_id must contain the ID of a loaded input window. Otherwise a fatal Toolkit error will occur.

If a value other than those listed above is passed for *qualifier*, a fatal Toolkit error will occur.

Examples

The following example returns the number of input fields in an input window with the window ID **inpid**:

```
num_flds=%i_info(inpid, D_NMFLDS)
```

The next example returns the number of input sets in an input window with the window ID **inpid**:

```
num_sets=%i_info(inpid, D_NMSETS)
```

I_INIT – Initialize input fields



```
xcall I_INIT(window_id, [set_name], [data_area][, field_list, ...])
```

Arguments

window_id

The ID of the input window to process. **(n)**

set_name

(optional) The name of the input set. **(a)**

data_area

(optional) The name of the data area for the input set. **(a)**

field_list

(optional) One or more field specifications, or ***ENABLED*** to initialize enabled fields only. **(a)**

Discussion

The I_INIT subroutine initializes input fields. To initialize means to clear the input field(s) in the window, fill the field(s) with the appropriate paint character, and flag the field(s) as empty. I_INIT does not affect the enabled state of the field(s).

Data_area must correspond to the structure associated with the input set. *Data_area* only applies if you specify a *set_name*. By default, Toolkit does not ensure that *data_area* is large enough for the fields it references. Toolkit generates an error for this condition only if **g_dtkbounds** is set to 1 or greater. See [g_dtkbounds on page 18-3](#).

You can specify up to nine field lists (*field_list* arguments), each of which can have from one to nine field specifications. If a *field_list* has multiple field specifications, separate the field specifications with any character *except* alpha characters, numeric characters, dollar sign (\$), underscore (_), and square brackets ([or] , which are used to enclose dimension specifications for repository arrayed fields). For example:

```
"name"
```

or

```
"name, birthday, soc_sec_no, employer, phone, policy_no, conditions[2]"
```

The maximum length of a *field_list* is 99 characters. Additional characters are ignored. For information on field specifications, see [“Field specifications” on page 2-35](#).

You can do the following:

- ▶ Specify only *window_id*. All fields in the window are initialized. The context is reset to the first field for each set (that is, the next field processed is the first enabled input field in the set).
- ▶ Specify *set_name* without any additional arguments. All fields in the input set are initialized. The context is reset to the first field.
- ▶ Specify *set_name* and *data_area*. All fields associated with the specified input set are initialized. The areas in the data area associated with the fields will also be cleared.
- ▶ Specify one or more fields without *set_name*. All designated fields are initialized, regardless of what input set they're in. Any *data_area* will be ignored.
- ▶ Specify *set_name*, *data_area*, and *field_list*. I_INIT checks to make sure all designated fields reside in the specified input set before initializing them. The areas in the data area associated with the fields are also cleared.



If I_INIT is used to clear an unframed field after the input window for the field has been added to a tab set, renditions applied to the field may not match the renditions for the window. See [%TS_TABSET](#) on page 13-7 for more information.

Examples

The following examples illustrate the five specification possibilities described above.

In this example only a window ID is specified. All fields in the window whose ID is stored in **inpid** will be initialized.

```
xcall i_init(inpid)
```

In this example both a window ID and a set name are specified. All fields in input set **cusmas** will be initialized.

```
xcall i_init(inpid, "cusmas")
```

In this example a window, a set, and a data area are specified. All fields in the data area **cusmas** that are associated with the input set named **cusmas** will be initialized.

```
xcall i_init(inpid, "cusmas", cusmas)
```

In this example window and a list of fields are specified. **fld1** and **fld2[2,7]** will be initialized. Their associated data areas will not be modified.

```
xcall i_init(inpid,, "fld1, fld2[2,7]")
```

The final example specifies a window, a set, a data area, and a field specification containing two fields. The I_INIT routine will initialize the fields **name** and **addr[2]** after verifying that these fields are in input set **cusmas**.

```
xcall i_init(inpid, "cusmas", cusmas, "name, addr[2]")
```

I_INPFLD – Process an input field



```
xcall I_INPFLD (window_id, field_spec, data_location, [input_column]
&               [, select_column][, text_column], [decimal_value], {reserved}
&               [, a_methoddata1, ..., a_methoddata20])
```

Arguments

window_id

The ID of the input window to process. **(n)**

field_spec

The specification for the field to process. (See “[Field specifications](#)” on page 2-35 for information.) **(a)**

data_location

Where to place the input data. **(a or n)**

input_column

(optional) The input column ID. **(n)**

select_column

(optional) The selection column ID. **(n)**

text_column

(optional) The text editing column ID. **(n)**

decimal_value

(optional) The first character to input (specified as a decimal value). **(n)**

{reserved}

Placeholder for a future Toolkit argument. Add an additional comma if specifying any *a_methoddata* arguments.

a_methoddata1 through *a_methoddata20*

(optional) Up to 20 additional data arguments. **(any)**

Discussion

The I_INPFLD subroutine enables a *single input field* to accept user input. All other fields are not modifiable and, on Windows, do not have a frame. If you want to process a set of input fields, see [I_INPUT on page 8-55](#). If you want to process input for a list item, see [L_INPFLD on page 9-98](#).

If either *window_id* or *field_spec* is invalid, a fatal error occurs.

Data_location is loaded according to the input field's definition, directly from the field, regardless of any input set associations. Note that by default, Toolkit does not ensure that the size of *data_location* matches the size of the referenced field (*field_spec*) or that it is even large enough for the field. However, bounds checking can check for these conditions. (See [g_dtkbounds on page 18-3](#) for more information.)

- ▶ If **g_dtkbounds** is set to 1, Toolkit generates a fatal error if *data_location* is not large enough for the field.
- ▶ If **g_dtkbounds** is set to 2 or greater, Toolkit generates a fatal error if *data_location* is not large enough (as it does if **g_dtkbounds** is set to 1), and it generates a non-fatal error if *data_location* is large enough, but isn't the exact size of the field.

UNIX, VMS

Field editing and movement within the field can be automatically performed during input processing if the appropriate reserved menu entries are present on a loaded *input_column*, *select_column*, or *text_column*.

If the *input_column* is present and non-zero, that column will be placed on the menu and removed on exit. If the *select_column* is present and non-zero, it will be placed on the menu when a field with a selection window is processed. If the *text_column* is present and non-zero, it will be placed on the menu when a text field is processed.

I_INPFLD recognizes the following menu entries, which are described in [“Appendix B: Reserved Menu Entries”](#):

E_LEFT	S_UP	I_CANCEL
E_RIGHT	S_DOWN	O_HELP
E_EMOV	S_LEFT	U_xxx
E_BMOV	S_RIGHT	
E_MODE		
E_CDEL		
E_LCLR		
E_XUP		
E_XDOWN		

To edit a field that already contains data, the user can use the “E_” entries above. These “E_” entries are a subset of the text editing functions supported by the T_EDIT subroutine.

Insert mode is the default for editing. When editing text fields (dimensioned alpha fields) all menu entries for T_EDIT are recognized. Text field editing is very similar to text window editing, except that the text field's input is terminated when the user presses ENTER unless **g_txt_rtrn** is true.

The entries that begin with "S_" are for selection windows.

If a user selects the menu entry I_CANCEL, any input being performed will be canceled, and the field will be reset to its initial state (prior to the start of the input).

Note that I_INPFLD does not invoke arrive or leave methods on UNIX or OpenVMS.

WIN

If any of the column arguments (*input_column*, *select_column*, or *text_column*) are passed, Toolkit will not, by default, place or remove the specified columns. These columns usually contain reserved menu entries that are automatically supported by the Windows environments. You can override this default behavior as noted below.

I_INPFLD recognizes the following menu entries, which are described in ["Appendix B: Reserved Menu Entries"](#):

- E_CLEAR
- E_COPY
- E_CUT
- E_MARK
- E_MODE
- E_PASTE
- I_CANCEL

Note that regardless of the available menu entries, the left and right arrow keys and the HOME, END, and DELETE keys are processed within I_INPFLD. Additionally, the up and down arrow keys are reserved and processed within I_INPFLD for multi-line edit controls, radio buttons, and selection fields. Therefore, with the following exception, these keys are never returned to the calling subroutine. However, when a single-line input field has focus, the up and down arrow keys and the HOME, END, and DELETE keys can be mapped to menu entries.

To edit a field that already contains data, the user can use the "E_" entries above. These "E_" entries are the same text editing functions supported by the T_EDIT subroutine.

If a user selects the menu entry I_CANCEL, any input being performed will be canceled, and the field will be reset to its state prior to the start of the input, its initial state. (Note, however, that an EENTRST_METHOD can override I_CANCEL.)

Insert mode is the default for editing. Text field editing is very similar to text window editing, except that the text field's input is terminated when the user presses ENTER unless **g_txt_rtrn** is true.

Note that E_XUP and E_XDOWN operate differently than E_UP and E_DOWN when the cursor is already on the first or last row of a field. See the I_INPUT [Discussion on page 8-56](#) for more information.

If a non-reserved menu entry is selected and the call to I_INPFLD returns, **g_select** will be true and **g_entnam** will hold the name of the menu entry. You will need to clear **g_select** after you satisfy the event.

The default behavior for each environment's handling of columns is controlled by the **g_plc_col_args** flag, which is defined in the distributed **tkctl.def** file. See [g_plc_col_args on page 18-4](#) for information.

I_INPFLD will also call EHELP_METHOD and EUTILS_METHOD on menu selections whose internal names are "O_HELP" or begin with "U_", respectively.

If a time-out occurs, **g_select** is set to true and **g_entnam** is set to the entry name specified in **tkctl.def** for **g_time_entry**. By default, **g_time_entry** is defined as "O_TIMEOUT".

UNIX, VMS

If passed, *decimal_value* is converted to its alpha (ASCII) equivalent and placed into the window as if it were the first character entered by the user. *Decimal_value* is ignored on Windows.

A_methoddata1 through *a_methoddata20* are up to 20 additional optional arguments that can be passed to I_INPFLD. I_INPFLD, in turn, passes them directly to %ICHANGE_METHOD, %IDISPLAY_METHOD, and %IEDITFMT_METHOD. This allows you to communicate additional information to your change, display, and edit format methods.

See also

- ▶ [“Processing and editing field input” on page 8-4](#) for more information about input processing.
- ▶ [“Input fields” on page 8-3](#) and [.DISPLAY_AREA on page 2-24](#) for defining a display area (or “view”) for an input window.
- ▶ [“Appendix B: Reserved Menu Entries”](#) for more information about the text editing functions.

Examples

The following example accepts the data into **whchnam** according to the definition of the input field **which**.

```
xcall i_inpfld(popid, "which", whchnam)
```

I_INPUT – Process an input set



```
xcall I_INPUT (window_id, [set_name], data_area, [input_column], [select_column],
&               [text_column], [D_NOTERM], [decimal_value], {reserved}
&               [, a_methoddata1, ..., a_methoddata20])
```

Arguments

window_id

The ID of the input window to process. **(n)**

set_name

(optional) The name of the input set. **(a)**

data_area

Where to place the input data (a field or record defined within the calling program). **(a)**

input_column

(optional) The input column ID. **(n)**

select_column

(optional) The selection column ID. **(n)**

text_column

(optional) The text editing column ID. **(n)**

D_NOTERM

(optional) Disable termination processing. **(n)**

decimal_value

(optional) The first character to input (specified as a decimal value). **(n)**

{reserved}

A reserved placeholder for a future Toolkit argument. Add an additional comma if specifying any *a_methoddata* arguments.

a_methoddata1 through *a_methoddata20*

(optional) Up to 20 additional data arguments. **(any)**

Discussion

The `I_INPUT` subroutine performs terminal input to the fields in an input set. If the input window has a movable view (or display), the view will be automatically adjusted, if necessary, to include the entire input field. (If you are processing input on a list item, use the `L_INPUT` subroutine instead of `I_INPUT`. See [L_INPUT on page 9-101](#) for more information.)

The `I_INPUT` subroutine can return for one of the following reasons (or a combination of the third and fourth reasons), which set `g_select`, `g_setsts`, `g_fldnam`, and `g_entnam`:

- ▶ Input to the input set is complete. When all active fields in the input set have been processed, `I_INPUT` displays a termination message, which is defined by `g_inp_trm_msg` (see [g_inp_trm_msg on page 18-4](#)).
- ▶ The input set has no active fields and there are no active buttons. (In this case, `g_setsts` is set to false, indicating that input is complete.)
- ▶ Input to a field that has a break specification is complete.
- ▶ A menu entry that the input system does not recognize is signaled (through user-interaction or by `%M_SIGNAL`).
- ▶ TAB or a call to `I_NEXTCTL` from the last button (or field if there is no active button) for an input window that is part of a composite window. (This signals “C_NEXT”, which `I_INPUT` doesn’t recognize but instead passes on to the calling routine.)
- ▶ SHIFT + TAB or a call to `I_PREVCTL` from the first field (or button if there is no active field) for an input window that is part of a composite window. (This signals “C_PREV”, which `I_INPUT` doesn’t recognize, but instead passes on to the calling routine.)

Use the following table to determine the reason(s) for the return.

Reason	<code>g_select</code>	<code>g_setsts</code>	<code>g_fldnam</code>	<code>g_entnam</code>
Input set complete	False	False	n/a	n/a
Break	n/a	True	Name ^a of the break field	n/a
Unrecognized menu entry	True ^b	True ^c	Name ^a of the field that was focused	Name ^a of the selected entry

- a. Field and entry names are always returned in uppercase characters.
- b. An unrecognized menu entry can be signaled through a user interaction or by `%M_SIGNAL`. After you have processed the event, you must clear `g_select`.
- c. If a button was focused while the user exited input, `g_setsts` will be set to 255. This means that while a button was focused, the user did one of the following: pressed the button, but its menu entry was not handled by `I_INPUT`, pressed another button whose entry was not handled by `I_INPUT`, selected a menu entry (or shortcut) that was not handled, or invoked a method (toolbar or input field, etc.), which signaled a menu entry that was not handled.

I_INPUT maintains the “context” of an input set. In other words, it remembers the next field to process in an input set. If a field has a break specified (see [BREAK, NOBREAK on page 2-39](#)), the context changes to the field after the field with the break, and an I_INPUT command will go to the field after the break. When you first load an input window, the context for all sets is reset. If the context is reset, the first field is the next field to be processed. (You can also reset context with I_INIT.)

The context for each input window set is separate from all other input window sets. You can change the context with the I_NEXT subroutine.

You can call I_INPUT at any time, and it will continue where input processing left off in that set. This means that you can process any number of input sets at any time and not lose context.



You should not call I_INPUT from a button method that’s called during I_INPUT processing; this will cause unpredictable results. Instead, you may be able to code the method to call an input method on the original window or signal a menu entry to handle input without calling I_INPUT directly.

If input is “interrupted” by a menu selection (that is, the user selects a menu entry before pressing ENTER), any current contents of that input field are copied into the force buffer (but the length of the data in the force buffer is not set), and the field is reset to its initial entering state (unless you use EENTRST_METHOD to override this behavior). You must call I_FORCE with only the *old_data* argument to obtain the input that was in the field at the time the menu entry was selected.

If *set_name* is not present, I_INPUT will, by default, process the first input set in the window.

The *data_area* is accessed according to the structure specification associated with this input set. This entire *data_area* is passed to the EUTILS_METHOD subroutine when a hot entry is selected, giving it access to fields other than just the current field. Note that by default, Toolkit does not ensure that *data_area* is large enough for the fields it references. Toolkit generates an error for this condition only if **g_dtkbounds** is set to 1 or greater. See [g_dtkbounds on page 18-3](#).

WIN

If any of the column arguments (*input_column*, *select_column*, or *text_column*) are passed, Toolkit will not, by default, place or remove the specified columns. These columns usually contain reserved menu entries that are automatically supported by Windows environments. You can override this default behavior as noted below.

I_INPUT recognizes the following menu entries, which are described in “[Appendix B: Reserved Menu Entries](#)”:

E_CLEAR	E_PASTE	I_FRST	I_NEXTCTL
E_COPY	I_CANCEL	I_FRSTCLR	I_NXTCLR
E_CUT	I_DFLT	I_HYPER	I_OK
E_MARK	I_DONE	I_LAST	I_PREV
E_MODE	I_DRILL	I_NEXT	I_PREVCTL

Note that regardless of the available menu entries, the left and right arrow keys and the HOME, END, and DELETE keys are processed within I_INPUT. Additionally, the up and down arrow keys are reserved and processed within I_INPUT for multi-line edit controls, radio buttons, and selection fields. Therefore, with the following exception, these keys are never returned to the calling subroutine. However, when a single-line input field has focus, the up and down arrow keys and the HOME, END, and DELETE keys can be mapped to menu entries.

If the menu entry I_CANCEL is present on the menu and the user selects it, any input being performed will be canceled, and the field will be reset to its state prior to the start of the input, its initial state. (Note, however, that an EENTRST_METHOD can override I_CANCEL.)

Insert is the default mode for editing. Text field editing is similar to text window editing, except that the text field's input is terminated when the user presses ENTER unless **g_txt_rtrn** is true.

Note the following for multi-line text fields:

- ▶ When pasting within or in front of existing text, text that follows the pasted text will be pushed out of the field (i.e., the data will be truncated) if there is not enough room.
- ▶ I_INPUT's handling of line ends in incoming data is controlled by the **g_txt_softreturn** field, which is defined in **tools.def**. See [g_txt_softreturn on page 18-7](#) for information.

Decimal _value is ignored on Windows.

UNIX, VMS

Field editing and movement within input fields is automatically performed during input processing if the appropriate reserved menu entries are present on a loaded *input_column*, *select_column*, or *text_column*.

If the *input_column* is present and non-zero, that column will be placed on the menu and removed on exit. If the *select_column* is present and non-zero, it will be placed on the menu when a field with a selection window is processed. If the *text_column* is present and non-zero, it will be placed on the menu when a text field is processed. The alternative to specifying any of the “column” arguments is to place the column before calling I_INPUT. (However, then the column may be on the menu even when it has no meaning.)

I_INPUT recognizes the following menu entries, which are described in “[Appendix B: Reserved Menu Entries](#)”. (Note that I_INPUT recognizes the S_ entries only when processing a selection field.)

E_BMOV	I_CANCEL	I_HYPER	I_PREVCTL
E_CDEL	I_DFLT	I_LAST	O_HELP
E_EMOV	I_DONE	I_NEXT	S_DOWN
E_LCLR	I_DRILL	I_NEXTCTL	S_LEFT
E_LEFT	I_EDIT	I_NXTCLR	S_RIGHT
E_MODE	I_FRST	I_OK	S_UP
E_RIGHT	I_FRSTCLR	I_PREV	U_xxx

I_INPUT also recognizes the following when on a multi-line text field:

E_BOTTOM	E_ILIN	E_PDEL	E_WDEL
E_DIR	E_JOIN	E_PMOV	E_WMOV
E_DOWN	E_LDEL	E_TOP	E_XDOWN
E_EDEL	E_LMOV	E_UP	E_XUP

To edit a field that already contains data, the user can move to that field using one of the field movement input entries (such as I_PREV or I_NEXT) listed above. The “E_” entries perform the field editing functions. These “E_” entries are a subset of the text editing functions supported by the T_EDIT subroutine.

Insert mode is the default for editing. When editing text fields (dimensioned alpha fields) all menu entries for T_EDIT are recognized. Text field editing is very similar to text window editing, except that the text field’s input is terminated when the user presses ENTER unless **g_txt_rtrn** is true.

The entries that begin with “S_” are for selection windows.

If the menu entry I_CANCEL is present on the menu and the user selects it, any input being performed will be canceled, and the field will be reset to its initial state (prior to the start of the input).

Additionally, I_INPUT will call EHELP_METHOD and EUTILS_METHOD on menu selections whose internal names are “O_HELP” or begin with “U_”, respectively.

Note that E_XUP and E_XDOWN operate the same as E_UP and E_DOWN, except when the cursor is already on the first row of the field (for E_XUP) or the last row of the field (for E_XDOWN). E_XUP moves the cursor up one row within the field, unless the cursor is already on the top row. In that case, E_XUP is returned to the routine that called I_INPUT. Likewise, E_XDOWN moves the cursor down one row in the field, unless the cursor is already on the bottom row. In that case, E_XDOWN is returned to the calling routine. If you use E_XUP or E_XDOWN, we recommend that you wrap I_INPUT in a routine that handles these menu entries when they are returned.

If I_INPUT encounters an entry name that begins with “I_” that is not in the list above, it generates an error. In other words, if you’re using I_INPUT, none of your application’s menu entry names (except for the reserved ones listed above) should begin with “I_”.

The default behavior for each environment’s handling of columns is controlled by the **g_plc_col_args** flag, which is defined in the distributed **tkctl.def** file. See [g_plc_col_args on page 18-4](#) for information.

If D_NOTERM is present, completion processing is disabled. This means that when an input set is completed, control returns directly to the calling subroutine, and the message specified by **g_inp_trm_msg** is not be displayed.

If a time-out occurs, **g_select** is set to true and **g_entnam** is set to the entry name specified in **tkctl.def** for **g_time_entry**. By default, **g_time_entry** is defined as “O_TIMEOUT”.

UNIX, VMS

If passed, *decimal_value* is converted to its alpha (ASCII) equivalent and placed into the window as if it were the first character entered by the user.

A_methoddata1 through *a_methoddata20* are up to 20 additional optional arguments that can be passed to I_INPUT, which in turn passes them directly to each field's methods. This allows you to communicate additional information to your input field methods.

See also

- ▶ [“Using Fonts on Windows” on page 18-40](#) for information on using proportional and fixed fonts.
- ▶ [“Processing and editing field input” on page 8-4](#) for more information about input processing.
- ▶ [M_LDCOL on page 10-37](#) for information on loading menu columns.
- ▶ [I_LDINP on page 8-62](#) for information on loading input windows.
- ▶ [“Appendix B: Reserved Menu Entries”](#) for more information about the text editing functions.

Examples

The following example processes the input set **cusmas** in the input window stored in **inpid** and places the data in **cusmas**.

```
xcall i_input(inpid, "cusmas", cusmas)
```

The next example processes the input set **artran**, places the input data in variable **artran**, and disables set termination processing.

```
xcall i_input(inpid, "artran", artran,,, D_NOTERM)
```

The following example loads the menu column whose ID is stored in **inpcol** and whose name is **input** from the window library channel that is stored in **g_utlib**. The column will not be placed on the menu. The I_INPUT subroutine processes the input set named **cusmas** and places the data in the record **cusmas**. The column whose ID is stored in **inpcol** will be placed on the menu and removed before I_INPUT returns.

```
xcall m_ldcol(inpcol, g_utlib, "input", D_NOPLC)
.
.
.
xcall i_input(inpid, "cusmas", cusmas, inpcol)
```


The following code fragment illustrates how you can use the I_INPUT subroutine in a program.

```
do
  begin
    xcall i_input(inpid, setname, datarec)
    if (g_setsts) then          ;Will be true for menu entries & break fields
      begin
        if (g_select) then      ;Test for menu entry
          using g_entnam select
            ("ABT"), call abort
            ("EXT"), done=TRUE
          endusing
        else
          using g_fldnam select  ;Test for break processing
            ("FLD1"), call fstbreak
            ("FLD2"), call sndbreak
            ("FLD3"), call thdbreak
          endusing
        end
      else
        done=TRUE                ;Input complete
      end
    end
  until (done)
```

I_LDINP – Load an input window from a library



```
xcall I_LDINP (window_id, [channel], window_name, [D_NOPLC], [search], [error]
&                [, saved_name])
```

Arguments

window_id

Returned with the ID of the input window. **(n)**

channel

(optional) The channel of the window library. If not passed, **g_utlib** is used. **(n)**

window_name

The name of the window to load (a maximum of 15 characters). **(a)**

D_NOPLC

(optional) Don't place the loaded input window on the screen. **(n)**

search

(optional) The search variable. **(n)**

error

(optional) The error return variable. **(n)**

saved_name

(optional) The name of the window when it was originally saved. **(a)**

Discussion

The I_LDINP subroutine loads an input window from a library. An input window must be loaded before it can be referenced by any other I_xxx routine.

When an input window is loaded, all associated selection windows are also loaded. These selection windows are marked as being owned by the input window. When you delete the window (with U_WINDOW or E_EXIT), the selection windows using the .FIELD SELECT qualifier are also deleted. (Those using the .FIELD SELWND qualifier are not.)

If you pass a null string (" ") for *window_name*, Toolkit assigns a unique window name in the form _W_nnn, where *nnn* is the window ID. You must pass *saved_name* in this case.

If you want to create multiple instances of the same window, you can call I_LDINP as many times as you need, using *window_name* to give each succeeding window a new unique name. You must also pass *saved_name*, if *window_name* is different than the name under which the window was saved.

If you place the window (you omit `D_NOPLC`), the window must fit on the screen or you will receive a “Window manager error” when Toolkit loads the window.

If *search* is present, `I_LDINP` searches for an existing window with the specified name before attempting to load the window. If the window is found, a successful load is simulated, *window_id* is set to the ID of the window that was found, and the *search* variable is set to true. If the window is not found, the *search* variable is cleared and loading continues normally.



You can use *search* to make your subroutines more modular. If you have written your routines to call `E_ENTER` on entry and `E_EXIT` before returning, use the *search* variable with `I_LDINP` within your subroutines to load windows.

Search will find the window, if already loaded, or it will enable `I_LDINP` to load it if it can't find it. Since the window will be a local window in the latter case, `E_EXIT` will delete it when it returns to the calling routine. If *search* finds the window already loaded and if it was loaded at a prior level, `E_EXIT` will leave it loaded when it returns.

If *error* is present, it is returned with one of the following values:

- ▶ 0 (false) if the requested window was found in the specified library.
- ▶ 1 (true) if the window was not found.
- ▶ -1 if the window (*window_id*) has an associated selection window that cannot be loaded. You can use `%U_GETWNDERR` to get the text of the error: “Window *name* not found in library *channel*,” where *name* is the name of the missing selection window, and *channel* is the channel number passed to `I_LDINP` (or the value of `g_utlib` if no channel was passed).

A fatal error occurs if *error* is not present and the window (*window_id*) or an associated selection window cannot be found.

See also

- ▶ [M_LDCOL on page 10-37](#) for information on loading menu columns.
- ▶ [U_LDWND on page 16-92](#) for information on loading general windows and for information on directly loading selection windows that may not be associated with input windows.

Examples

In the first example, the subroutine attempts to load the window named **cusmas** from the library opened on channel `g_utlib`. The variable **inpid** is returned with the ID of the input window. If the window isn't in the library, a fatal error occurs.

```
xcall i_ldinp(inpid,, "cusmas")
```

In the second example, `I_LDINP` attempts to load but does not place the window **artran** from the library opened on `libchn`. Again, **inpid** is returned with the ID of the input window. The success of the operation is reported in **errcode**. If the window already exists, an error occurs.

```
xcall i_ldinp(inpid, libchn, "artran", D_NOPLC,, errcode)
```

In the third example, `I_LDINP` searches for the window named **cusmas** before it attempts to load it and place it on the screen. If it finds the window, it will store the window ID in **inpid** and set **find** to true; otherwise, it will load the window named **cusmas**. If the window is loaded successfully, **errcode** is returned with a value of false.

```
xcall i_ldinp(inpid,, "cusmas",, find, errcode)
```

I_NEXT – Specify the next input field or button to be processed



```
xcall I_NEXT(window_id, [set_name][, field_spec])
```

Arguments

window_id

The ID of the input window containing the field. **(n)**

set_name

(optional) The name of the input set to use or ***CURR*** to use the currently loaded set. **(a)**

field_spec

(optional) The specification for the field or one of the keywords listed below (which must be specified in uppercase). *Field_spec* must be enclosed in quotation marks. (For information on field specifications, see “[Field specifications](#)” on page 2-35.) **(a)**

FRST	First enabled field.
LAST	Last enabled field.
NEXT	Next enabled field.
PREV	Previous enabled field.
FCLR	First empty enabled field. (default)
NCLR	Next empty enabled field.
MENU	Set the context based on the contents of g_entnam .
NMBR , <i>field#</i>	<i>Field#</i> if enabled, otherwise next enabled field after <i>field#</i> .
BUTTON , <i>button</i>	Specified button on Windows. <i>Button</i> can be either the name (case-insensitive) or index number of a button.
NONE	No field.
DFLT	Fill default values on all empty fields with defaults and set context to next enabled empty field.
DONE	Complete input on all empty, enabled fields.
LASTCTL	Last enabled control (field or button).
NEXTCTL	Next enabled control (field or button).
PREVCTL	Previous enabled control (field or button).

Discussion

The `I_NEXT` subroutine sets the input context. When processing input, Toolkit moves through fields according to their order in the input set; when a user is finished with a field, Toolkit moves to the next active field in the set. With `I_NEXT`, however, you can determine which field or button (on Windows) Toolkit will advance to. For example, if your application displays an error when a user enters an invalid item code, Toolkit automatically advances to the next field in the input set after the error is displayed. Using `I_NEXT`, however, you can instead instruct Toolkit to go back to the field that caused the break.

These are the general rules for context changes:

- ▶ If a field or button specified with `I_NEXT` is disabled, Toolkit sets the context to the next enabled field or button.
- ▶ If `I_NEXT` specifies a field, but all fields in the input set are disabled, Toolkit moves input context to the first enabled button.
- ▶ If a field or button specified with `I_NEXT` is disabled and is the last field in the input set or the last button in the window, Toolkit sets the context to the first enabled input field in the set.
- ▶ If all fields and all buttons in the input set are disabled, Toolkit proceeds to input set completion.

Note that set names are sometimes inaccessible to the utility subroutines. To call `I_NEXT` from within a utility subroutine, use `*CURR*` instead of a specific set name. This enables `I_NEXT` to use the current set even if it can't access the name of the current set. If *set_name* is not passed, `I_NEXT` uses the first input set in the window.

If you don't pass *field_spec*, the context is set to the first empty enabled field, which is the equivalent of setting `*FCLR*`. (An empty field is one that the user hasn't entered anything in and that has not been displayed using `I_DSPFLD` or `I_DISPLAY`.)

Use `*NMBR*` followed by a numeric argument (*field#*) to specify the next field by index. This index is a number that represents the field's position in the input set and is limited to the valid range for the input set. (A value less than 1 or greater than the number of fields in the window will result in a fatal Toolkit error.) For example, the following sets the context to the third field in an input set named **setnam**.

```
xcall i_next(inpid, setnam, "*NMBR*", 3)
```

On Windows, use `*BUTTON*` followed by a button name or index to move context to a specific button. (On UNIX and OpenVMS, passing `*BUTTON*` will cause a fatal Toolkit error.) The *button* argument is required if you pass `*BUTTON*`.

- ▶ If *button* is an alphanumeric value, Toolkit treats it as a case-insensitive name. (If the name is not a valid name for a button in the window, you'll get a fatal Toolkit error.)

- If *button* is a numeric value, Toolkit treats it as an index, which represents the button's position in the input set and is limited to the valid range for the input set. (A value less than 1 or greater than the number of buttons in the window will result in a fatal Toolkit error.) For example, the following sets the context to the second button in the window:

```
xcall i_next(inp_id,, "*BUTTON*", 2)
```

NONE prevents I_INPUT from performing input to any field. Instead, Toolkit uses single-keystroke input. This option is rarely used.

Using **DONE** or **DFLT** is like pressing ENTER for each remaining field without entering any input. The difference is that **DONE** completes the input set while **DFLT** displays any defaults that are specified in the script file, leaves all other fields unmodified (which does not affect their empty state), and allows input to continue on the next empty field.

On Windows, **NEXTCTL**, **LASTCTL**, and **PREVCTL** set context to the next, last, or previous enabled field or button. On UNIX or OpenVMS, if you are using termination processing, you can use **NEXTCTL**, **LASTCTL**, and **PREVCTL** to set context to the termination message.



If you call I_NEXT with **FCLR** or **NCLR** in a change method, I_NEXT uses the current state of the field set to determine where to move input context. This causes I_NEXT to set input context to the current field if it's empty. To instead move context to the next enabled empty field, use %M_SIGNAL to signal I_NXTCLR. This moves context to the next empty field (unless you signal some other menu entry at the same time) because I_NXTCLR won't be processed until the change method has returned and the input has been accepted. And because a menu entry was generated, Toolkit won't move input context until the menu entry is processed.

If an application is in I_DONE (**DONE**), I_DFLT (**DFLT**), or I_OK processing, calling I_NEXT from a method will cancel that processing and move focus to the field specified by I_NEXT. If you do not want this to happen, test the value of **inp_nulinp** (which is in the **inputinfo** group defined in **inpinfo.def**) before calling I_NEXT. If **inp_nulinp** is non-zero, the application is performing one of these types of processing.

If you pass **MENU**, I_NEXT will set context based on the value of **g_entnam** at the time that I_NEXT is called. (So with **MENU**, I_NEXT recognizes only menu entries that start with I_, such as I_NEXT, I_PREV, and I_DONE.) This is useful when a menu entry has been signaled outside of an input routine (such as I_INPUT, I_INPFLD, etc.) in order to force that menu entry to be applied upon re-entry to that routine. For example, while within a method in I_INPUT, passing "I_OK" to M_SIGNAL will cause validation of all fields to occur upon exiting the method, and (presuming no validation errors result) I_INPUT will then exit. However, if you make the same call to M_SIGNAL during break processing, then that menu entry will not apply to the next call to I_INPUT unless you also call I_NEXT with **MENU**.

See also

- ▶ [I_DISABLE on page 8-11](#) for information on disabling input fields.
- ▶ [I_ENABLE on page 8-19](#) for information on enabling input fields.

Examples

In the following example, I_NEXT resets the context in input set **cusmas** by skipping all remaining fields without entering input. The next time I_INPUT is called on this set, it will complete the set as if the user pressed ENTER at each field (using the default for each) and perform termination processing.

```
xcall i_next(inpid, "cusmas", "*DONE*")
```

The next example sets the context of **maindcs** to the first empty field (the default).

```
xcall i_next(dcswnd, "maindcs")
```


I_PROMPT – Set a variable prompt



```
xcall I_PROMPT (window_id, field_spec, prompt)
```

Arguments

window_id

The ID of the input window in which you want the prompt to be displayed. **(n)**

field_spec

The specification for the field the prompt is associated with. **(a)**

prompt

The prompt to display. **(a)**

Discussion

I_PROMPT sets the prompt for a field.

- ▶ If a prompt has already been assigned to the field, I_PROMPT changes the prompt to *prompt*. Note that the length of *prompt* must be less than or equal to the length of the original prompt for the field.
- ▶ If no prompt has been assigned to the field, but a number was specified for the PROMPT qualifier for .FIELD, the D_FIELD_PROMPT qualifier for IB_FIELD, or the prompt field in Repository (if the field is imported from a repository), I_PROMPT sets the prompt to *prompt*. Note that the number of characters in *prompt* must be less than or equal to the number specified for the qualifier.
- ▶ If no prompt or number was specified for the PROMPT qualifier for .FIELD, the D_FIELD_PROMPT qualifier for IB_FIELD, or the prompt field in Repository, I_PROMPT cannot set a prompt for the field.

For more information on setting prompts, see [PROMPT, NOPROMPT](#) on page 2-60.

Examples

In the following, a number is specified for the PROMPT qualifier for .FIELD, so the field **fld1** has no prompt, but space for 20 characters has been allocated for a prompt.

```
.input inp, 5, 40
.field fld1, a10, prompt(20)
.end
```

The following call assigns a prompt to **fld1**:

```
xcall i_prompt(id, "fld1", "This is field one")
```

I_PUTFLD – Put data into an input field



```
xcall I_PUTFLD (window_id, [set_name], data_area, field_spec, new_data
&                [, a_methoddata1, ..., a_methoddata20])
```

Arguments

window_id

The ID of the input window containing the field. **(n)**

set_name

(optional) The name of the input set. **(a)**

data_area

The data area for the input set. **(a)**

field_spec

The specification for the field that I_PUTFLD will load data into. (See “[Field specifications](#)” on [page 2-35](#) for information.) **(a)**

new_data

The data to load. **(a)**

a_methoddata1 through *a_methoddata20*

(optional) Up to 20 additional data arguments. **(any)**

Discussion

The I_PUTFLD subroutine places data into an input field. It displays *new_data* in the field and puts it into *data_area*.

If you do not specify a set name, *set_name* defaults to the first input set.

Data_area must correspond to the structure associated with the input set.

Field_spec must be part of the input set, and if either *window_id* or *field_spec* is invalid, a fatal error occurs.

Data is retrieved from *new_data* according to the definition within the input window (regardless of the actual type of *new_data*) so you need to make sure the data you pass as *new_data* has the same type and precision as the input field (*field_spec*). If the input field is a dimensioned alpha field, *new_data* is assumed to be the first element of the array as specified in the field’s definition, and remaining elements are assumed to follow *new_data* in memory. (If the other elements aren’t

defined in the memory that follows *new_data*, subsequent elements of the alpha array field will be filled with garbage.) For dimensioned alpha fields then, it's best to define the data for all elements in the field in an identical dimensioned alpha variable, and then pass the first element as *new_data*.

Note that by default, Toolkit does not ensure that the size of *data_area* or the size of *new_data* matches the size of the referenced field (*field_spec*) or that they are even large enough for the field. However, bounds checking can check for the following conditions. (See [g_dtkbounds](#) on page 18-3 for more information.)

- ▶ If **g_dtkbounds** is set to 1, Toolkit generates a fatal error if *data_record* is not large enough for the field.
- ▶ If **g_dtkbounds** is set to 2 or greater, Toolkit generates a fatal error if *data_area* is not large enough (as it does if **g_dtkbounds** is set to 1). It generates a non-fatal error if *data_area* is large enough for the field, but isn't the exact size. And it generates a non-fatal error if *new_data* is smaller than the field.

Note that if **g_dtkbounds** is set to 2 or greater, Toolkit also generates a fatal error if an integer value is passed for a non-integer field or vice versa (though no error is generated for user-defined data types).

A_methoddata1 through *a_methoddata20* are up to 20 additional optional arguments that can be passed to I_PUTFLD. I_PUTFLD, in turn, passes them directly to %IDISPLAY_METHOD. This allows you to communicate additional information to your display method.

If I_PUTFLD is used with an unframed field after the input window for the field has been added to a tab set, renditions will be applied to the field that may not match the renditions for the window. See [%TS_TABSET](#) on page 13-7 for more information.

See also

- ▶ [I_DSPFLD](#) on page 8-17 for information on displaying data in a field.
- ▶ [I_FORCE](#) on page 8-41 for information on forcing data into a field.

Examples

In the following example, the contents of **total** are loaded into the field **TOTAL**, as well as into the corresponding area of **dcsmast**, as defined by the structure associated with the **maindcs** set.

```
xcall i_putfld(dcswnd, "maindcs", dcsmast, "TOTAL", total)
```

I_READONLY – Set read-only state for input fields



```
xcall I_READONLY ([operation,] [subfunction,] arguments)
```

Arguments

operation

(optional) One of the following, which determine whether specified fields will be set to read-only or read/write. (**n**)

D_OFF Set fields to read/write.

D_ON Set fields to read-only. (default)

subfunction

(optional) One of the following, which determine which fields will be affected. (**n**)

D_ALL Set the read-only state for all fields in the specified window.

D_FLDS Set the read-only state for individually specified fields. (default)

D_SET Set the read-only state for all fields in the specified input set.

arguments

One or more arguments, depending on *subfunction*.

Discussion

I_READONLY enables you to set the read-only state of one or more input fields in an input window, regardless of the current input context.

This routine differs from I_FLDMOD in that it can set the read-only state for multiple fields, and it preserves the contents of affected fields. (I_FLDMOD can set the read-only state for only one field, and it initializes the field.) If an affected field is empty when I_READONLY is called, it will remain empty. If the field has text, I_READONLY will preserve the text.

If a field specified in a call to I_READONLY can't be set to read-only (in other words, if the field is a check box, a radio button set, or a field with a selection list or selection window), a read-only setting will be accepted, but it won't affect the field. (For these field types, use I_DISABLE rather than I_READONLY.) However, if the field is subsequently changed to a regular text field, Toolkit may use this setting because when a field is changed to a text field, the most recent read-only state setting is used to determine whether the field will be read-only or read/write.

See also

[“Notes on Input Methods” on page D-3](#) for information on changing read-only settings from an input method.

D_ALL

```
xcall I_READONLY ([operation,] D_ALL, window_id)
```

Arguments

operation

(optional) One of the following, which determine whether the specified fields will be set to read-only or read/write. (**n**)

D_OFF Set fields to read/write.

D_ON Set fields to read-only. (default)

window_id

The ID of the input window whose fields you want to modify. (**n**)

Discussion

The D_ALL subfunction of I_READONLY enables you to set the read-only state for all the input fields in an input window.

Examples

The following examples set all the fields in a window (the window whose ID is stored in **win_id**) to read-only:

```
xcall i_readonly(D_ON, D_ALL, win_id)
```

```
xcall i_readonly(D_ALL, win_id)
```

The next example sets all the fields in a window to read/write:

```
xcall i_readonly(D_OFF, D_ALL, win_id)
```

D_FLDS

```
xcall I_READONLY ([operation,] [D_FLDS,] window_id, field_spec[, ...])
```

Arguments

operation

(optional) One of the following, which determine whether the specified fields will be set to read-only or read/write. **(n)**

D_OFF Set fields to read/write.

D_ON Set fields to read-only. (default)

window_id

The ID of the input window whose fields you want to modify. **(n)**

field_spec

One or more field specifications. **(a)**

Discussion

The D_FLDS subfunction of I_READONLY enables you to set the read-only status for a specified field or list of fields in an input window. If a specified field does not exist in the input window, a fatal error will occur.

You can specify up to nine field names in *field_spec* and up to nine field specifications (nine occurrences of *field_spec*), which enables you to specify a maximum of 81 fields per D_FLDS call. The maximum length of *field_spec* is 99 characters. Additional characters are ignored.

To specify multiple field names in *field_spec*, separate them with any character *except* alpha characters, numeric characters, dollar sign (\$), underscore (_), and square brackets ([or] , which are used to enclose dimension specifications for repository arrayed fields). For example, the following could be passed as a *field_spec*:

```
"name, b_day, hr_dt, soc_sec, emp, policy, phone, conditions[2,4]"
```

Examples

The first two examples set the specified fields (which are the same in both cases) to read-only. One difference is that in the second example the *operation* and *subfunction* arguments have been omitted. (These examples use the defaults for these arguments, so they are optional.) Another difference is that in the first example, the fields have been grouped into three *field_spec* arguments, while in the second example, the fields have been grouped into two *field_spec* arguments.

```
xcall i_readonly(D_ON, D_FLDS, win_id, "name, b_day, hr_date, soc_sec",  
& "employer, policy, phone, fax, email",  
& "addr[1,1], addr[1,2], addr[1,3]")
```

```
xcall i_readonly(win_id, "name, b_day, hr_date, soc_sec, employer",
```

```
&          "policy, phone, fax, email, addr[1,1], addr[1,2],  
&          addr[1,3] ")
```

The following examples set the specified fields to read/write:

```
xcall i_readonly(D_OFF, D_FLDS, win_id, "name, b_day, hr_date, soc_sec",  
&          "employer, policy, phone, fax, email",  
&          "addr[1,1], addr[1,2], addr[1,3] ")
```

```
xcall i_readonly(D_OFF, win_id, "name, b_day, hr_date, soc_sec,  
employer",  
&          "policy, phone, fax, email, addr[1,1], addr[1,2],  
&          addr[1,3] ")
```

D_SET

```
xcall I_READONLY ([operation,] D_SET, window_id, set_name)
```

Arguments

operation

(optional) One of the following, which determine whether the specified fields will be set to read-only or read/write. (**n**)

D_OFF Set fields to read/write.

D_ON Set fields to read-only. (default)

window_id

The ID of the input window whose fields you want to modify. (**n**)

set_name

(optional) The case-insensitive name of an input set. (**a**)

Discussion

The D_SET subfunction of I_READONLY enables you to set the read-only state for all the fields in an input set.

If the specified input set does not exist in the window, a fatal error will occur.

Examples

The following examples set all the fields in an input set (the input set whose name is stored in **in_set**) to read-only:

```
xcall i_readonly(D_ON, D_SET, win_id, in_set)
```

```
xcall i_readonly(D_SET, win_id, in_set)
```

The next example sets all the fields in an input set to read/write:

```
xcall i_readonly(D_OFF, D_SET, win_id, in_set)
```


I_SETDEL – Delete fields from an input set



```
xcall I_SETDEL (window_id, set_name, field_list[, ...][, status])
```

Arguments

window_id

The ID of the input window containing the fields. **(n)**

set_name

The name of the input set. **(a)**

field_list

One or more field specifications. **(a)**

status

(optional) Returned with a non-zero value if the field or fields were not found in the set, or with zero if the field or fields were found and successfully deleted from the set. *Status must be the twelfth argument.* **(n)**

Discussion

The I_SETDEL subroutine enables you to remove fields or lists of fields (specified in *field_list*) from an input set during program execution. You can specify up to nine field lists (*field_list* arguments), each of which can have up to nine field specifications. The maximum length of a *field_list* is 99 characters. Additional characters are ignored. For information on field specifications, see [“Field specifications” on page 2-35](#).

The specified fields or lists of fields are deleted from the loaded input window set. You can use I_SNAPSHOT to save or restore the set definition.

If a *field_list* has multiple field specifications, separate the field specifications with any character *except* alpha characters, numeric characters, dollar sign (\$), underscore (_), and square brackets ([or]), which are used to enclose dimension specifications for repository arrayed fields). For example:

```
"name"
```

or

```
"name, birthday, soc_sec_no, employer, phone, policy_no, conditions[2]"
```

If passed, *status* must be the twelfth argument. Therefore, if you pass *status*, but you specify fewer than nine fields or lists to be removed, you must still use commas in place of the non-specified fields.

If you pass *status*, the first field that is not found within the set causes I_SETDEL to cease processing and return with the error status true. In this case, none of the fields following the one that caused I_SETDEL to terminate are deleted from the set. If you don't pass *status* and a specified field is not found within the set, a fatal error occurs.

Examples

The following example does input to the set that gets a name and asks whether there is a co-signer. If no co-signer exists, the next set is modified to skip the fields for the name and address of the co-signer. Then the full set is re-established and the process is repeated.

```
.include "WND:tools.def"
record
    id          ,i4
    c_id        ,i4
    stat        ,d3
    snapshot    ,a20
record data
    applicant   ,a30
    joint       ,a3
    addr        ,3a30
    cosign      ,a30
    coaddr      ,3a30
proc
    xcall u_start("setdel")
    xcall i_ldinp(id, g_utlib,"setdel")
    xcall i_snapshot(D_COPY, id, snapshot)
    xcall m_ldcol(c_id, g_utlib, "input", D_NOPLC)
    do
        begin
            xcall i_input(id, "name", data, c_id)
            if (g_select)
                exitloop
            if (joint.eq."No") then
                xcall i_setdel(id, "detail", "cosign coaddr",,,,,,,,,,
stat)
            else
                xcall i_snapshot(D_LOAD, id, snapshot)
                xcall i_input(id, "detail", data, c_id)
                xcall i_init(id)
            end
        until (g_select)
    xcall u_finish
.end
```

The input window definition looks like this:

```
.input setdel, 10, 65
.field applicant, a30, pos(1,1), prompt("Applicant's name "), req
.field joint, a3, pos(1,50), prompt("Co-signer? "),
sel(0,2,2,"YES","NO")
.text applicant, pos (4,1)
\hAddress of Applicant\ -h
.field addr, 3a30, pos(5,1)
.text cosigner, pos (4,34)
\hName & Address of Co-signer\ -h
.field cosign, a30, pos(5,34)
.field coaddr, 3a30, pos(6,34)
.set name, setdel, applicant, joint
.set detail, setdel, addr, cosign, coaddr
.end
```

I_SETINF – Retrieve information about an input set



```
xcall I_SETINF (window_id, [set_name], [set_num], [next_field], [set_controls]
&               [, processing_state])
```

Arguments

window_id

The ID of the input window containing the set. **(n)**

set_name

(optional) The returned set name. **(a)**

set_num

(optional) The returned set number. **(n)**

next_field

(optional) The returned next input entry index. **(n)**

set_controls

(optional) The returned set controls. **(a)**

processing_state

(optional) The processing state for an input set. One of the following: **(n)**

D_MODE_NORMAL Normal input processing mode.

D_MODE_DONE Done (I_DONE) processing occurring.

D_MODE_DFLT Default (I_DFLT) processing occurring.

D_MODE_OK OK (I_OK) processing occurring.

Discussion

The I_SETINF subroutine retrieves information about an input set. In order to use it, however, be sure to **.INCLUDE** these two distributed **.def** files (in the following order):

- ▶ **WND:system.def**
- ▶ **WND:setinf.def**

Window_id must contain the ID of a loaded input window.

You must specify either *set_name* or *set_num*, but you don't need to specify both. If you specify both and

- ▶ *set_num* is non-zero on entry, *set_name* is returned with the name of the set specified by *set_num*. (Sets are numbered sequentially as defined by the .SET script qualifier or by the order of calls to IB_SET.)
- ▶ *set_num* is zero on entry, *set_name* is assumed to contain the name of the desired set, and *set_num* is returned with its ordinal number as defined above.

If *next_field* is passed, *next_field* is returned with the ordinal number of the next input field in the set to be processed.

If *set_controls* is passed, *set_controls* is returned with the control structure for the indicated set. The format of the control structure is defined in the distributed file **setinf.def**.

If *processing_state* is passed, *processing_state* is returned with a value used to determine whether done, default, or OK processing is in progress on the set. In this case, the processing has not yet completed.

If *processing_state* returns D_MODE_DONE, this means that I_NEXT(*DONE*) was called for the current set or the user selected I_DONE. All remaining fields are filled with their default values, blank, or zero. Validation and any break processing occurs next. When complete, if any buttons exist on the window, the first button will be focused; otherwise, termination processing will occur.

If *processing_state* returns D_MODE_DFLT, this means that I_NEXT(*DFLT*) was called for the current set or the user selected I_DFLT. All remaining empty fields that have a default value are filled with defaults. Validation and any break processing occurs next. When complete, the first empty field or first button will be focused, or termination processing will occur, in that order.

If *processing_state* returns D_MODE_OK, this means that I_OK was selected. Processing is identical to D_MODE_DONE, except that when complete I_INPUT is exited with **g_setsts = 0**.

Within the distributed **setinf.def** file, the field **gs_setcnt** contains the number of fields in the input set. The field **gs_setnam** is a 16-bit pointer and will not be useful. The array **gs_setfld** contains one 8-bit value for each field in the set. By using %DECML on each element, you can determine the UI Toolkit field number for each of the fields in the set. You can then use I_FLDINF to retrieve information about each of these fields, including the actual window field numbers (which may differ). See [I_FLDINF on page 8-22](#) for more information.

Examples

The following example returns the number of the next input field in **context** and the control structure for the **entries** set in **gs_fldset**.

```
xcall i_setinf (wndid, "entries",, context, gs_fldset)
```

The following example returns the name of the third input set in **set_name**, and the processing state for that set in **state**.

```
xcall i_setinf (wndid, set_name, 3,,, state)
```

I_SNAPSHOT – Save/restore an input window’s controls



```
xcall I_SNAPSHOT(D_COPY|D_LOAD, window_id, snapshot, [length], [record]
&                [, a_methoddata1, ..., a_methoddata20])
```

Arguments

D_COPY

Copy the current controls. (**n**)

D_LOAD

Load previously copied controls. (**n**)

window_id

The ID of the input window. (**n**)

snapshot

The snapshot information field. (**a**)

length

(optional) The returned length of the snapshot information. (**n**)

record

(optional) The structure data for redisplaying the fields on a D_LOAD operation. (**a**)

a_methoddata1 through *a_methoddata20*

(optional) Up to 20 additional data arguments. (**any**)

Discussion

The I_SNAPSHOT routine enables you to “take a picture” of the input window’s controls, then restore them at a later time. The information that is saved/restored includes the following:

- ▶ Cleared state of all input fields
- ▶ Context of all input sets
- ▶ What fields are in what sets
- ▶ Enabled/disabled state of all input fields

Additionally, on a D_LOAD (restore) operation, the fields can be redisplayed according to their cleared states. This is done by passing *record*, which will be used as the structure data for all input sets. If *record* is not passed, the display will not be affected.

This subroutine is useful in conjunction with I_SETDEL, as it can restore fields that had been previously deleted from a set.

I_SNAPSHOT has the following limitations:

- ▶ If *record* is passed, it must be usable as the structure data for all input sets in the window.
- ▶ The *snapshot* argument must be large enough to contain the snapshot information. This length is presently computed as the total number of fields in the window, plus 8 times the number of input sets, plus the number of fields used in each set, for a maximum length of 6,778. (The equation would be represented as $maxfields + [maxsets * 8] + [maxsets * maxfields]$.) The *length* argument can optionally be returned with this length.
- ▶ You must not modify the data in *snapshot* between D_COPY or D_LOAD operations. This data has a private format that will be changed in future versions of UI Toolkit.
- ▶ The D_LOAD operation always removes frames from fields. To restore field frames, call I_FRAMES.

A_methoddata1 through *a_methoddata20* are up to 20 additional optional arguments that can be passed to I_SNAPSHOT. I_SNAPSHOT, in turn, passes them directly to %IDISPLAY_METHOD. This allows you to communicate additional information to your display method.

Examples

The following example copies the current input window control information for the window identified by **inpid** into the field **inpinfo**, and returns the length of that information in **infolen**.

```
xcall i_snapshot(D_COPY, inpid, inpinfo, infolen)
```

The next example resets the same input window to its state at the time that the D_COPY operation in the first example occurred.

```
xcall i_snapshot(D_LOAD, inpid, inpinfo(1:infolen))
```

The example for I_SETDEL also shows the I_SNAPSHOT subroutine being used. (See the I_SETDEL [Examples on page 8-78](#) for more information.).

I_TXTPOS – Get or set the input position for a multi-line text field



```
xcall I_TXTPOS (window_id, field_spec, D_LAST)
```

```
xcall I_TXTPOS (window_id, field_spec, [D_CHANGE,] row, column)
```

```
xcall I_TXTPOS (window_id, field_spec, D_GETNEXT | D_GETLAST, row, column)
```

Arguments

window_id

The ID of the input window containing the field. **(n)**

field_spec

The specification for the multi-line text field. (See “[Field specifications](#)” on page 2-35 for information.) **(a)**

D_LAST

Sets the position for subsequent input to *field_spec* to the position the cursor last occupied when accepting input.

D_CHANGE

(optional) Specifies that the values passed in *row* and *column* are offsets relative to the last position occupied during input (if you pass D_CHANGE), or it specifies that *row* and *column* are absolute coordinates relative to the field origin if you omit D_CHANGE (i.e., if you pass *row* and *column* without a D_ option).

D_GETNEXT

Returns in *row* and *column* the next projected position that will be used for input (or zeros if there is no projected next position).

D_GETLAST

Returns in *row* and *column* the last position occupied by the cursor to accept input (or zeros if there has been no previous input).

row

A row within the multi-line text field. **(n)**

column

A column within the multi-line text field. **(n)**

Discussion

I_TXTPOS gets or sets the input position for a multi-line text field:

- ▶ If you pass D_LAST or D_CHANGE, or if you pass *row* and *column* without a D_ option, I_TXTPOS sets the cursor position for subsequent input. The next time *field_spec* is processed through one of Toolkit's input routines, the cursor will be placed at the specified position. (Note that you can also use **gs_inpfld** fields to set the cursor position for a multi-line text field. See [“Notes on gs_inpfld fields for multi-line text fields”](#) on page 8-23.)
- ▶ If you pass D_GETNEXT or D_GETLAST, I_TXTPOS returns (in *row* and *column*) the next projected position or the last position that the cursor occupied when an input routine processed *field_spec*.

If either *window_id* or *field_spec* is invalid, a fatal error occurs.

Note the following when setting the position for subsequent input (D_LAST, D_NEXT, or no D_ option):

- ▶ If you pass *row* and *column* without passing any of the D_ options for I_TXTPOS, do not include a comma placeholder for the omitted D_ option. For example:

```
xcall i_txtpos(inpid, "mytext", 2, 1)
```
- ▶ *Field_spec* must be set to RETAIN_POSITION (.FIELD) or D_FLD_RETPOS (IB_FIELD and I_FLDMOD). If it is *not* set to one of these, *row* and *column* are treated as zero, even if they are passed as other values.
- ▶ If *row* and *column* lie outside valid bounds for the field, Toolkit automatically limits them to valid values when *field_spec* is next processed by one of Toolkit's input routines.

If you use D_GETNEXT, and the field is not set to RETAIN_POSITION (.FIELD) or D_FLD_RETPOS (IB_FIELD or I_FLDMOD), *row* and *column* are returned as zero unless you have set them to something else (with I_TXTPOS). In that case, they'll return the values they were set to, as long as input hasn't occurred on the field since they were set.

Examples

This example sets the position for the next input of the text field **mytext** to be the same position as last occupied during input of this field:

```
xcall i_txtpos(inpid, "mytext", D_LAST)
```

This example moves the position within the text field down one row from the previous input position (if the text field is set to RETAIN_POSITION or D_FLD_RETPOS):

```
xcall i_txtpos(inpid, "mytext", D_CHANGE, 1, 0)
```

The next example moves the position to the beginning of the second row of the text field:

```
xcall i_txtpos(inpid, "mytext", 2, 1)
```

The following examples retrieve the next projected input position and the last input position:

```
xcall i_txtpos(inpid, "mytext", D_GETNEXT, next_row, next_col)
```

```
xcall i_txtpos(inpid, "mytext", D_GETLAST, last_row, last_col)
```

I_USER – Retrieve/load the user-defined string for an input field

WT	WN	U	V

```
xcall I_USER(window_id, field_spec, [data][, new_data])
```

Arguments

window_id

The ID of the input window containing the field. **(n)**

field_spec

The specification for the field whose user-defined string I_USER will retrieve or modify. (See [“Field specifications” on page 2-35](#) for information.) **(a)**

data

(optional) Returned with the user data string. **(a)**

new_data

(optional) The new user data string. **(a)**

Discussion

The I_USER subroutine reads or modifies a user-defined string (up to 80 characters) for a field.

If either *window_id* or *field_spec* is invalid, a fatal error occurs.

If no user data was defined for the field (in other words, you did not specify a **user** qualifier on the .FIELD script command), no operations will occur and no errors will be generated. Even if you specify *data*, it will not be modified. (Thus you could check for user field data by placing a known string in *data* before the call and seeing if it is still there after the I_USER call.)

Note that by default Toolkit does not check for size. It truncates the user field data if *data* or *new_data* is not large enough. Toolkit checks the size of these arguments and generates an error for a size mismatch only if **g_dtkbounds** is set to 2 or greater. See [g_dtkbounds on page 18-3](#).

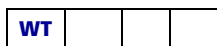
Also note that the size of the user field data is fixed by the size of the string used during the set’s compilation. Writing data out to the string won’t change the size of the string.

Examples

The following example gets the user data associated with the **password** field. In this example, the system password could be maintained in the input window for easy verification.

```
xcall i_user(inpid, "password", sec_chk)
```

IB_BUTTON – Include a Windows-style button



```
xcall IB_BUTTON (build_id, name, [type], [type_data], [method], [elb],
&                [select_char], [position][, tooltip])
```

Arguments

build_id

The ID of the input window under construction. **(n)**

name

The name of the button to add (a maximum of 30 characters). **(a)**

type

(optional) The button face type: **(n)**

DSB_TEXT Text. (default)

DSB_BITMAP Graphic.

type_data

(optional) Text if *type* is DSB_TEXT. A filename for the graphic if *type* is DSB_BITMAP. **(a)**

method

(optional) The name of the method subroutine to invoke. **(a)**

elb

(optional) The name of the library from which to draw *method*. **(a)**

select_char

(optional) The quick-select character. **(a)**

position

(optional) The position the button will be inserted in the button set. **(a or n)**

tooltip

(optional) The text of a ToolTip to be displayed when the mouse pointer rests over the button. **(a)**

Discussion

The `IB_BUTTON` subroutine creates a Windows-style button in a new input window. This subroutine is used when building a new input window at runtime.

The default location for a set of buttons is at the bottom, left-justified, with as many buttons on a row as will fit before wrapping to a new row. `IB_BUTTONSET` can be called to change these defaults.

The basic functionality for this routine is documented in the [B_BUTTON Discussion on page 5-4](#).

This routine has no effect on applications running on UNIX or OpenVMS or in Synergy .NET. It does, however, save button information on these platforms, so if the window is saved to a window library (by using `U_SAVE`) and later loaded from that library on a Windows platform in traditional Synergy, the button will be on the window (assuming ISAM-compatible platforms).

See also

- ▶ [.BUTTON on page 2-13](#) for more information on defining an input window's buttons in a script file.
- ▶ [IB_BUTTONSET on page 8-90](#) for information about specifying button placement for an input window built at runtime.

Examples

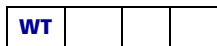
The following example defines an OK button named **I_OK**. When the button is pressed, the remaining input fields are filled with defaults and standard termination processing occurs. The OK button has the quick-select character "K." When ALT+K is pressed, a click of the OK button is simulated.

```
xcall ib_button(inpid, "I_OK", DSB_TEXT, "OK",,, "K")
```

The next example creates a button named **lookup** with the graphic **lookup.gif** on its face. Pressing this button will invoke the subroutine **lookup** located in the library **button.elb**.

```
xcall ib_button(bldid, "lookup", DSB_BITMAP, "lookup.gif", "lookup",  
& "button.elb")
```

IB_BUTTONSET – Define the placement of a set of buttons



```
xcall IB_BUTTONSET (build_id, [placement], [per_row_col], [justification][, button_name])
```

Arguments

build_id

The ID of the input window under construction. **(n)**

placement

(optional) One of the following: **(n)**

DSB_BOTTOM The set of buttons will be placed horizontally along the bottom of the window with a maximum of *per_row_col* buttons per row. (default)

DSB_RIGHT The set of buttons will be placed vertically along the right side of the window with a maximum of *per_row_col* buttons per column.

per_row_col

(optional) The maximum number of buttons to place on a row if **DSB_BOTTOM** is specified, or on a column if **DSB_RIGHT** is specified. The default behavior is to display as many buttons as will fit and wrap to the next row or column. **(n)**

justification

(optional) One of the following: **(n)**

DSB_BEGIN The buttons will be left-justified if **DSB_BOTTOM** is specified, or positioned towards the top if **DSB_RIGHT** is specified. (default)

DSB_CENTER The buttons will be centered along the designated side.

DSB_END The buttons will be right-justified if **DSB_BOTTOM** is specified, or positioned towards the bottom if **DSB_RIGHT** is specified.

button_name

(optional) The name of the default button. **(a)**

Discussion

IB_BUTTONSET defines the placement of a set of buttons within a new input window. This subroutine is used when building a new input window at runtime.

Every button is contained within a set and only one button set is allowed per input window; therefore, calling **IB_BUTTONSET** will affect all buttons in an input window.

If the E_STATE option D_RETURNBTN is set, the default button will be selected if you press RETURN, no matter what field you're on in the window. If *button_name* is not specified and D_RETURNBTN is set, the first button in the window is the default.

This routine has no effect on applications running on UNIX or OpenVMS or in Synergy .NET. It does, however, save button information on these platforms, so if the window is saved to a window library (by using U_SAVE) and later loaded from that library on a Windows platform in traditional Synergy, the button will be on the window (assuming ISAM-compatible platforms).

See also

- ▶ [.BUTTON_SET on page 2-17](#) for information on defining the placement of an input window's button set in a script file.
- ▶ [IB_BUTTON on page 8-88](#) for information about creating input window buttons at runtime.

Examples

The following example places the input window's buttons on the right side of the window, centered, with, three buttons per column (at most).

```
xcall ib_buttonset(bldid, DSB_RIGHT, 3, DSB_CENTER)
```

IB_DSP_AREA – Set the display area for a new input window



```
xcall IB_DSP_AREA (build_id, win_row, win_column, rows, columns)
```

Arguments

build_id

The ID of the input window under construction. **(n)**

win_row

The row position of the upper-left corner of the display area (relative to the upper-left corner of the window, not the screen). **(n)**

win_column

The column position of the upper-left corner of the display area (relative to the upper-left corner of the window, not the screen). **(n)**

rows

The height of the display area. **(n)**

columns

The width of the display area. **(n)**

Discussion

The IB_DSP_AREA subroutine sets the display area for a new input window. It is used when building a new input window at runtime. The default display area is the entire input window.

The view is movable if the display area is smaller than the size of the window. If the window is an input window, the input processing system adjust the view as necessary to include the entire input field being processed. If you want a movable view, note that you must use this subroutine rather than the W_AREA windowing routine.

See also

[“Notes and Tips for ActiveX Toolkit Lists” on page 9-15](#) for information on setting the display area for an input window that is associated with an ActiveX Toolkit list.

Examples

The following example defines a display area for the current input window being built. The upper-left corner of the display area is at window position row 2, column 1, and the area is 10 rows deep by 25 columns wide.

```
xcall ib_dsp_area(bldid, 2, 1, 10, 25)
```


IB_END – Define the end of a new input window



```
xcall IB_END (build_id, window_id, [channel][, no_load])
```

Arguments

build_id

The ID of the input window under construction. **(n)**

window_id

Returned with the ID of the input window that is created. **(n)**

channel

(optional) The window library channel from which to load any pre-built selection windows. **(n)**

no_load

(optional) Any non-zero value. If passed, it will specify that the input window should not be loaded at this time. **(n)**

Discussion

The IB_END subroutine ends an input window specification session. This subroutine is used when building a new input window at runtime.

Window_id is the identifier for the new input window. If *window_id* is returned zero, the input window building was unsuccessful.

After you call IB_END, no other IB_xxx subroutines can be used on this window (*build_id*). In fact, *build_id* should be discarded as it is no longer valid. However, you can still define other window features such as the border, display area, and text, just as you can with any input window.

After calling IB_END, you can place the window since you have the final ID (*window_id*).

If any of the fields in the input window use a prebuilt selection window (with the D_FLD_SELWND qualifier on IB_FIELD), and that selection window is not currently resident, *channel* is required in order to provide a window library channel from which to load the selection window, unless *no_load* is specified or the selection windows are already loaded.

No_load enables you to build an input window without using it right away. You can save it to a window library and load it later with I_LDINP. To do this, you must pass *no_load*, and then call U_SAVE for *window_id*. You will also need to specify the window name in the U_SAVE subroutine.

Examples

The following example ends the specification of an input window being built. The ID for the new input window is returned in **wndid**. Any associated selection windows will be loaded from the window library opened on **g_utlib**.

```
xcall ib_end(bldid, wndid, g_utlib)
```

IB_FIELD – Add an input field to a new input window



```
xcall IB_FIELD (build_id, field_spec[, qualifier, ...])
```

Arguments

build_id

The ID of the input window under construction. **(n)**

field_spec

The field specification for the input window field. See “Field specifications” on page 2-35 for information. **(a)**

qualifier

(optional) One or more of the following field qualifiers: **(n)**

D_FLD_TYPE, *type*

Field type. See “D_FLD_TYPE” on page 8-99 for valid values for *type*.

D_FLD_SIZE, *storage_size*

Storage size. See “D_FLD_SIZE” on page 8-99 for size limits.

D_FLD_DEC, *num_places*

Decimal places.

D_FLD_DIM, *dim_size*

Dimension. Only single dimension alpha arrays are supported.

D_ON

Turn on one or more of the following field qualifiers. See “D_ON and D_OFF” on page 8-99 for details. (default)

D_OFF

Turn off one or more of the following field qualifiers. When qualifiers are preceded by D_OFF, flags that normally require a following argument do not need one. See “D_ON and D_OFF” on page 8-99 for details.

D_FLD_ALLOW, *num_entries*, *entry_array*

Allow list.

D_FLD_ARRIVE, *arrive_method*

Arrive method.

D_FLD_ATTR, *attribute*

Attributes. See “D_FLD_ATTR” on page 8-100 for valid values for *attribute*.

D_FLD_AUTO

Auto entry.

D_FLD_BREAK

Break on change.

D_FLD_BRK_ALWAYS	Break always.
D_FLD_BRK_RTRN	Break on return only.
D_FLD_BZRO	Blank if zero.
D_FLD_CASE	Match case.
D_FLD_CHANGE , <i>change_method</i>	Change method.
D_FLD_CHECKBOX	Check box.
D_FLD_COLOR , <i>palette#</i>	Color.
D_FLD_COPY	Copy default.
D_FLD_DATE , <i>date_type</i>	Date field. See “ D_FLD_DATE ” on page 8-100 for valid values for <i>date_type</i> .
D_FLD_DECR	Decrement default.
D_FLD_DISABLED	Disabled.
D_FLD_DISPLAY , <i>display_method</i>	Display method.
D_FLD_DFLT , <i>default_string</i>	Default entry.
D_FLD_DRILL , <i>drill_method</i>	Drill method.
D_FLD_DSPLN , <i>display_len</i>	Display length. Cannot be specified for check boxes, radio buttons, selection windows, or multi-line text fields.
D_FLD_EDITFMT , <i>editfmt_method</i>	Edit format method.
D_FLD_ENUM , <i>dsp_len</i> , [<i>base</i>], [<i>step</i>]	Enumeration properties.
D_FLD_EXACT	Match exact.
D_FLD_FMT , <i>string</i>	Format string.
D_FLD_FONT , <i>font_id</i>	Field font. See “ D_FLD_FONT and D_FLD_PROMPTFONT ” on page 8-100 for valid values for <i>font_id</i> .
D_FLD_FPOS [, D_CHANGE], <i>row</i> , <i>col</i>	Field position.
D_FLD_HELP , <i>help_id</i>	Help identifier.
D_FLD_HYPERLINK , <i>hyper_method</i>	Hyperlink method.
D_FLD_INCR	Increment default.

D_FLD_INFO , <i>string</i>	Information text.
D_FLD_INPLEN , <i>input_len</i>	Maximum input length. Cannot be specified for check boxes, radio buttons, selection windows, or multi-line text fields
D_FLD_JUST , <i>just_flag</i>	Justification, where <i>just_flag</i> is D_JST_CENTER, D_JST_LEFT, or D_JST_RIGHT.
D_FLD_LEAVE , <i>leave_method</i>	Leave method.
D_FLD_NEG	Negative allowed.
D_FLD_NEGONLY	Negative allowed only.
D_FLD_NEGORZERO	Negative or zero allowed only.
D_FLD_NODEC	No decimal required.
D_FLD_NOECHO	No echo.
D_FLD_NOECHR , <i>char</i>	Fill character when D_FLD_NOECHO is specified (otherwise ignored).
D_FLD_NOTERM	No termination required.
D_FLD_PAINT , <i>char</i>	Paint character.
D_FLD_POS [, D_CHANGE], <i>row, col</i>	Prompt position.
D_FLD_PROMPT , <i>prompt_string</i>	Prompt.
D_FLD_PROMPTFONT , <i>font_id</i>	Prompt font. See “ D_FLD_FONT and D_FLD_PROMPTFONT ” on page 8-100 for valid values for <i>font_id</i> .
D_FLD_PROMPTLEN , <i>length</i>	Runtime prompt.
D_FLD_RADIO	Radio buttons. See “ D_FLD_RADIO ” on page 8-101.
D_FLD_RANGE , <i>min, max</i>	Range for numeric field. If you use variables for the <i>min</i> and <i>max</i> arguments, they must be alpha variable; decimal variables will not work.
D_FLD_READONLY	Read-only.
D_FLD_REQUIRED	Required entry.
D_FLD_RETPOS	Retain position in text field.
D_FLD_SELECT , <i>row, col, numrows, [numcols], num_entries, entry_array</i>	Selection list. On Windows, use with D_FLD_RADIO to specify radio buttons or without

to specify a combo box. For the former, see [“D_FLD_RADIO” on page 8-101](#).

D_FLD_SELWND , <i>row, col, window_name</i>	Selection window. On Windows, use with D_FLD_RADIO to specify radio buttons or without to specify a combo box. For the former, see “D_FLD_RADIO” on page 8-101 .
D_FLD_TIME , <i>time_type</i>	Time field. See “D_FLD_TIME” on page 8-101 for valid values for <i>time_type</i> .
D_FLD_TIMESTAMP , <i>format</i>	Timestamp field. See “D_FLD_TIMESTAMP” on page 8-101 for details on this option and valid values for <i>format</i> .
D_FLD_UC	Uppercase only.
D_FLD_USER , <i>string</i>	User-defined text string.
D_FLD_VIEWLEN , <i>view_length</i>	The width of the field (i.e., the width of the area on the screen that will display data for the field). Cannot be specified for check boxes or radio buttons
D_FLD_WAIT , <i>wait_time</i>	Input time-out for the field. See “D_FLD_WAIT” on page 8-102 for valid values for <i>wait_time</i> .

Discussion

The IB_FIELD subroutine adds a field to a new input window being built at runtime. The maximum number of input fields in an input window or input set is 253.



IB_FIELD's options are also available with the .FIELD script command. For more detailed information on IB_FIELD options, see the corresponding qualifier in [“FIELD qualifiers” on page 2-36](#).

Defaults are used for qualifiers that are not specified. If IB_RPS_STRUCTURE has been called prior to IB_FIELD, default values will be drawn from the repository, and the field must exist in the repository or a fatal error will occur. For notes on fields drawn from repositories, see [Using .FIELD with .REPOSITORY_STRUCTURE on page 2-33](#).

If you specify a qualifier that does not apply to the field's destination type, it is ignored. (Integer fields support all qualifiers that are valid for decimal fields without precision.)

Any D_FLD_xxx *qualifier* not listed will result in the fatal error “Invalid IB_FIELD code (*code*),” where *code* is the invalid value passed.

D_FLD_TYPE

D_FLD_TYPE supports the following options for *type*:

D_ALPHA	Alphanumeric
D_DECIMAL	Decimal
D_INTEGER	Integer
D_NUMERIC	Numeric (same as decimal)
D_USERTYPE, <i>type_string</i>	User-defined

If you use D_FLD_TYPE to change the field to D_INTEGER type, the storage size is changed to the nearest (but not larger) valid integer storage size. Supported sizes are 1, 2, 4, and 8. Integer fields support all qualifiers that are valid for decimal fields without precision.

D_FLD_SIZE

Alpha and alpha array fields can have up to 65,535 characters. Decimal fields (including implied decimal fields) have a maximum size of 28, and for implied-decimal fields, up to 28 digits can follow the decimal point. Integer fields on any operating system can have 1, 2, 4, or 8 digits, depending on size. If the size is not correct, Toolkit will ignore the field.

If IB_FIELD creates a text field that is too large for the window, a warning will display only if **g_dtkbounds** is set to a value of 2 or greater. See [g_dtkbounds on page 18-3](#).

D_ON and D_OFF

The D_ON and D_OFF codes indicate whether the field qualifiers that follow in the argument list are to be turned on or off. D_ON is the default state for any qualifiers listed.

When qualifiers are preceded by D_OFF, flags that normally require a following argument do not need one. Thus, a prompt defined with the following call to IB_FIELD

```
xcall ib_field(bldid, "myfield", D_FLD_PROMPT, "Help")
```

can be turned off with the following:

```
xcall ib_field(bldid, "myfield", D_OFF, D_FLD_PROMPT)
```

Even though the prompt defined above is a text prompt, it could also be turned off by using D_OFF with D_FLD_PROMPTLEN. This is because, when used with either D_FLD_PROMPT or D_FLD_PROMPTLEN, D_OFF always turns off the specified prompt, regardless of its type (text or runtime).

Turning off D_FLD_BRK, D_FLD_BRK_ALWAYS, or D_FLD_BRK_RTRN results in no break for the field.

D_FLD_ATTR

D_FLD_ATTR accepts the following values for *attribute* (defined in **tools.def**):

A_ALL	All attributes (same as A_BHRU)
A_NONE	No attributes
A_B A_I	Blinking (UNIX and OpenVMS) or italic (Windows). “I” can be substituted whenever “B” appears in the following flags.
A_BH	Blinking, highlighted
A_BHR	Blinking, highlighted, reverse
A_BHRU	Blinking, highlighted, reverse, underlined
A_BHU	Blinking, highlighted, underlined
A_BR	Blinking, reverse
A_BRU	Blinking, reverse, underlined
A_BU	Blinking, underlined
A_H	Highlighted
A_HR	Highlighted, reverse
A_HRU	Highlighted, reverse, underlined
A_HU	Highlighted, underlined
A_R	Reverse
A_RU	Reverse, underlined
A_U	Underlined

D_FLD_DATE

D_FLD_DATE accepts the following values for *date_type*:

D_DATE_Y2J	YYJJJ date format
D_DATE_Y2MD	YYMMDD date format
D_DATE_Y2P	YYPP date format
D_DATE_Y4J	YYYJJJ date format
D_DATE_Y4MD	YYYYMMDD date format
D_DATE_Y4P	YYYYPP date format

To make the field default to the today’s date, add D_DATE_TODAY to any of the above. For a short date, add D_DATE_SHORT to any of the above. (You can add both of these if desired,)

D_FLD_FONT and D_FLD_PROMPTFONT

The *font_id* argument to D_FLD_FONT and D_FLD_PROMPTFONT can be the name of a font palette entry, a font handle, or one of the following:

DF_ALTERNATE	Alternate font
DF_CURRENT	Current font
DF_DEBUGGER	Debugger font
DF_DEFAULT	Global font

D_FLD_RADIO

On Windows, use D_FLD_RADIO with either D_FLD_SELECT or D_FLD_SELWND to specify a field that is a group of radio buttons. (If you use D_FLD_SELECT or D_FLD_SELWND without specifying D_FLD_RADIO on Windows, the field will be a combo box.)

- ▶ If you use D_FLD_SELECT with D_FLD_RADIO, the *numrows* argument for D_FLD_SELECT determines how many radio buttons can fit in a column. If *numrows* is not passed, the radio buttons will be placed vertically in a single column.
- ▶ If you use D_FLD_SELWND with D_FLD_RADIO, the height of the associated selection window determines how many buttons can fit in a column. You can control spacing between radio button columns by changing the column width for the selection window. This spacing is calculated by adding the width of the radio button to the width of a column in the selection window (which is equal to the number of characters times the character width for the font).

For more information on radio buttons, see [RADIO, NORADIO](#) on page 2-61.

D_FLD_TIME

D_FLD_TIME accepts the following values for *time_type*:

D_TIME_HM	HHMM time format
D_TIME_HMS	HHMMSS time format

To make the field default to the current time, add D_TIME_NOW to either of the above. To add AM/PM to the display, add D_TIME_AMPM to either of the above. (You can add both of these if desired.)

D_FLD_TIMESTAMP

By default, an autotime or ctimestamp field from a repository results in a timestamp field in a Toolkit application. However, you can use D_FLD_TIMESTAMP with D_OFF to create a standard integer field from a repository autotime field or with D_ON to create a timestamp field from scratch or from a standard repository 8-byte integer field.

When D_OFF is in effect, *do not* use the *format* argument. (It will be interpreted as another qualifier.)

When D_ON is in effect, D_FLD_TIMESTAMP *must* be followed by the *format* argument, which enables you to change the formatting used for a timestamp field. The *format* argument can be one or both of the following, or 0 (for neither):

D_TIMESTAMP_AMPM	Use 12-hour notation and AM/PM designation
D_TIMESTAMP_SHORT	Use short display format (no century, seconds, or milliseconds)

To specify both D_TIMESTAMP_AMPM and D_TIMESTAMP_SHORT, use the bitwise OR operator (.BOR. or |) or add them. If a flag isn't passed, it will be reset to its default value. For example, if you don't pass D_TIMESTAMP_SHORT, the long display format will be used even if the field was previously set to use the short format.

For D_ON, D_FLD_TYPE must be set to D_INTEGER, and the field must be a read-only, 8-byte integer. It cannot be a selection field, a radio button set, or a check box. Otherwise, an error will occur.

D_FLD_WAIT

The *wait_time* argument to D_FLD_WAIT can be the number of seconds to wait or one of the following:

D_FOREVER or -1	Wait until I/O processing is completed
D_GLOBAL or -2	Use the globally specified time-out
D_IMMEDIATE or 0	Time out immediately

See also

- ▶ [.FIELD on page 2-31](#) for information about the various field options.
- ▶ [“Renditions” on page 2-34](#) for information and notes on setting field-level renditions.
- ▶ [I_FLDMOD on page 8-25](#) to modify an existing field at runtime.
- ▶ [“Basic field information”](#) in the “Working with Fields” chapter of the *Repository User’s Guide* for information on using implicit groups to combine repository structures.
- ▶ [“Using Fonts on Windows” on page 18-40](#) for information on fonts and font palettes.

Examples

The following example adds the field **company** to the current input window being built. It is an alpha field, 15 characters long.

```
xcall ib_field(bldid, "company", D_FLD_TYPE, D_ALPHA, D_FLD_SIZE, 15)
```

The next example adds the field **age** to the current input window being built. It is a three-digit decimal field, which the user is required to input. The field will prompt the user with the string “Enter your age.”

```
xcall ib_field(bldid, "age", D_FLD_TYPE, D_DECIMAL, D_FLD_SIZE, 3,  
& D_FLD_REQUIRED, D_FLD_PROMPT, "Enter your age ")
```

The following example adds the field **date[2,6]** to the current input window being built. It is an eight-digit decimal field from an array named **date** defined in a repository named **cusmas**. Note that the IB_FIELD call specifies the format (YYYYMMDD), information that Toolkit does not import from the repository. The field will default to today’s date if the user presses ENTER without entering any data.

```
xcall ib_rps_structure(bldid, "cusmas")  
xcall ib_field(bldid, "date[2,6]", D_FLD_DATE, D_DATE_Y4MD+D_DATE_TODAY)
```

IB_INPUT – Create a new input window



```
xcall IB_INPUT (build_id, name, rows, columns, [no_log], window_id)
```

Arguments

build_id

Returned with the input window build ID. **(n)**

name

The name of the input window (a maximum of 15 characters). **(a)**

rows

The number of rows in the window. **(n)**

columns

The number of columns in the window. **(n)**

no_log

(optional) The true/false flag that specifies whether UI Toolkit should skip logging the input window with the environment. **(n)**

window_id

(optional) Returned with the intermediate stage window ID. **(n)**

Discussion

The IB_INPUT subroutine is used to begin building a new input window at runtime. Like .INPUT does in a script file, IB_INPUT starts the input window building definition. The IB_END subroutine (like .END) ends the definition. All other IB_xxx subroutines associated with this window should be called in between IB_INPUT and IB_END.

The *build_id* variable should be an aligned **i4**. This *build_id* is passed as the first argument to all other IB_xxx subroutines.

If you pass a null string (“”) for *name*, Toolkit assigns a unique window name in the form *_W_nnn*, where *nnn* is the window ID.

The new input window will be logged in the current environment, unless *no_log* is passed and is true.

The intermediate *window_id* can be useful if you want to display text in the window or set other window features. The final input window (created when you call IB_END) will inherit the features of this window (such as title and border), with the exception that the window fields and controls will be added later.

The display area must only be changed using `IB_DSP_AREA` until after the window is finished (with `IB_END`). If you set it directly, the input processor will not know that the window has a movable view.

Examples

The following example initiates an input window specification session. The name of the input window will be **questions** and the working ID that will be used is returned in **bldid**. The height of the new input window is determined by **grows** and its width by **qcols**. The new input window will be logged in the current environment.

```
xcall ib_input(bldid, "questions", grows, qcols)
```

IB_LOCAL – Set the current input window structure to local



```
xcall IB_LOCAL(build_id)
```

Arguments

build_id

The ID of the input window under construction. (**n**)

Discussion

The IB_LOCAL subroutine specifies that subsequent fields are to be created, instead of drawn from the repository. This is the default state when you call IB_INPUT. This subroutine is used after you have used IB_RPS_STRUCTURE to draw fields from the repository and you want to create local fields.

This subroutine is used when building a new input window at runtime.

Examples

The following example sets the current state of the input window being built to local. Any subsequent fields specified by IB_FIELD are to be created rather than retrieved from the repository.

```
xcall ib_local(bldid)
```

IB_PAINT – Set the paint character for a new input window



```
xcall IB_PAINT (build_id, character)
```

Arguments

build_id

The ID of the input window under construction. **(n)**

character

The paint character. **(a)**

Discussion

The IB_PAINT subroutine determines which character “paints” an empty input field in the new input window. This subroutine is used when building a new input window at runtime.

Initially, the paint character is a blank. IB_PAINT resets the default paint character. Note that IB_FIELD’s D_FLD_PAINT option will override the character specified with IB_PAINT. Fields drawn from the Repository always override IB_PAINT.

A paint character can only be a printable character.

On Windows, paint characters have no meaning.

Examples

In the following example, the dot character will indicate where the user should type input in the input window being built. This paint character takes effect for all fields defined in the input window after this point using IB_FIELD.

```
xcall ib_paint (bldid, '.')
```

IB_RPS_STRUCTURE – Specify the repository structure for a new input window



```
xcall IB_RPS_STRUCTURE (build_id, structure_name, [main_file, text_file][, alias_name])
```

Arguments

build_id

The ID of the input window under construction. **(n)**

structure_name

The name of the repository structure to access. **(a)**

main_file

(optional) The name of the main repository file to use. **(a)**

text_file

(optional) The name of the repository text file to use. **(a)**

alias_name

(optional) The name by which *structure_name* will be referenced within this input window. **(a)**

Discussion

The IB_RPS_STRUCTURE subroutine specifies that subsequent calls to IB_FIELD are to draw the definitions from the given repository structure. This subroutine is used when building a new input window at runtime.

The IB_RPS_STRUCTURE subroutine automatically creates a .STRUCTURE in the input window with the specified *structure_name*, containing all the referenced fields in the appropriate positions. You do not need to call IB_STRUCTURE for this structure.

If *main_file* and *text_file* are not specified, the repository filenames specified by the RPSMFIL and RPSTFIL logicals will be used. (RPSMFIL should be set to the name of the main repository file. RPSTFIL should be set to the name of the repository text file.) If these logicals are not set, the default repository file name will be used: **RPSDAT:rpsmain.ism** and **RPSDAT:rpstext.ism**, respectively.

You should use the *alias_name* qualifier in the following circumstances:

- ▶ When you include the same repository structure more than once within an input window.
- ▶ When you include two or more fields that have the same name although they are referenced from different repository structures.

In either of the preceding situations, you must rename the second and succeeding “same” structures using the *alias_name* argument.

After you have assigned an *alias_name* to a structure, any runtime references to the structure must use the *alias_name*. If you have given the structure an *alias_name* and are referencing a field during runtime, the reference to the field will have the following syntax:

```
alias_name.field_name
```

This syntax can be used only for fields that are in a renamed structure.

The first 30 characters of the full reference above must be unique within the input window. Thus, we recommend that you use short alias names.

You can combine multiple repository structures, enabling their access as one structure, by using the implicit group feature of Repository.

See also

“[Basic field information](#)” in the “Working with Fields” chapter of the *Repository User’s Guide* for information on using implicit groups to combine repository structures.

Examples

The following example specifies that subsequent fields in the input window being built are to be drawn from the **cusmas** structure in the default repository.

```
xcall ib_rps_structure(bldid, "cusmas")
```

The next example specifies that subsequent fields in the input window being built are to be drawn from the **order** structure in the repository residing in the files **frnchmn.ism** and **frnctx.ism**.

```
xcall ib_rps_structure(bldid, "order", "frnchmn.ism", "frnctx.ism")
```


IB_SET – Add a set definition to a new input window



```
xcall IB_SET (build_id, set, [structure,] [count][, fields])
```

Arguments

build_id

The ID of the input window under construction. **(n)**

set

The name of the input set (a maximum of 30 characters). **(a)**

structure

(optional) The name of the structure to associate with the set. **(a)**

count

(optional) The number of fields in the set. **(n)**

fields

(optional) The array of field specifications for the set. (See [“Field specifications” on page 2-35](#) for information.) **(a)**

Discussion

The IB_SET subroutine creates a named set of fields, specifies their order of reference, and associates the input set with the input window being built. This subroutine is used when building a new input window at runtime. (Remember, you don’t have to put all fields in sets—you can process fields individually as well.)

If no structure has been defined (with IB_RPS_STRUCTURE or IB_STRUCTURE) when you call IB_SET, you must pass the default structure name as *structure* if you want the input set to have any fields. The default structure has the same name as the input window, and will include all the fields defined for the input set.

If you omit *structure* and *fields*, Toolkit creates an empty input set. (If you omit *structure*, you must also omit *fields*.) This is useful if you want a set of buttons that can be placed anywhere in a composite window. See [“Buttons in composite windows” on page 6-6](#) for more information.

If you use the *fields* array, include field specifications in the order in which you want them processed at runtime. The number of fields that you can specify in the array is 1 to *n*, where *n* is the total number of fields in the structure specification. All fields specified must be members of the same *structure* specification. (See [IB_STRUCTURE on page 8-111](#) and [IB_RPS_STRUCTURE on page 8-107](#).)

If you include *structure* and omit *fields*, all the fields within the specified structure will be used in the order of their definition within the structure.

If *count* is passed and non-zero, *fields* is required and must have at least *count* elements. If *count* is not passed or is zero, *fields* is ignored and all fields in the structure are included in the set.

If you don't call IB_SET, a default set with the name of the input window being built will be created. This set will contain all the fields defined in the first structure (or the default structure if *structure* is not specified), in the order they are defined.

Examples

This example passes an array of field specifications from structure **mystruct** in order to define the set.

```
bldid      ,i4
nmflds     ,i4
fldnames   ,[10]a30,
&  "field1",
&  "field2",
&  "field3[7]",
&  "field4[2,7]",
.
.
.
xcall ib_set(bldid, "myset", "mystruct", nmflds, fldnames)
```

IB_STRUCTURE – Define a local structure within a new input window



```
xcall IB_STRUCTURE (build_id, structure, count, fields, [pos][, offset_fields])
```

Arguments

build_id

The ID of the input window under construction. **(n)**

structure

The name of the local structure (a maximum of 30 characters). **(a)**

count

The number of structure specifications in the *fields* array. **(n)**

fields

An array of structure specifications. A structure specification can be either a field name, or “<value>” where *value* is the number of pad characters. You can use *value* to skip space in your structure between fields. *Value* must be enclosed in angle brackets. **(a)**

pos

(optional) An array, parallel to the *fields* array, which contains absolute record positions. **(n)**

offset_fields

(optional) An array, parallel to the *fields* array, which contains the field names from which to offset. **(a)**

Discussion

The IB_STRUCTURE subroutine defines a local structure within the input window being built.

This subroutine is used when building a new input window at runtime.

Structure definitions are used to verify that no input sets contain fields from different structures; a field can only be a member of one structure. Note, however, that a field does not have to be a member of any structure.

Structure definitions are used to transfer data to and from the field’s data position within its record buffer.

Although a given field can only be a member of one structure you can define up to 25 structures per input window.

The maximum size of a structure is 9999 characters (**d4**). Text fields take up space that equals (*size * dimension*).

You should not call IB_STRUCTURE for fields drawn from a repository structure. In fact, calling IB_STRUCTURE with the same structure name as in IB_RPS_STRUCTURE will result in a “Structure already defined” error.

IB_STRUCTURE is only needed for fields defined after calling IB_LOCAL. (IB_LOCAL is also the default state.)

If you do not call IB_STRUCTURE or IB_RPS_STRUCTURE for a given input window, a default structure is defined. Its name is the name of the input window, and it is composed of all fields in the order that they are created in the order the IB_FIELD calls were made. (If the input window has no fields, the default structure will have no fields.) Thus, if you need only one record, you don’t need to define it with IB_STRUCTURE. It will be created by default.

IB_STRUCTURE defines membership and position for each field specified. If you do not pass fields, all the fields will be used in the order they were defined.

If *pos* is zero or not passed, the next position after the previous specification will be used.

If *offset_fields* is passed and non-blank for a given *fields* entry, the corresponding *pos* entry is assumed to be the relative offset from that field.

See also

[.STRUCTURE on page 2-99](#) for information on structure field specifications.

Examples

This example passes an array of field names and their absolute record positions in order to define the structure.

```
bldid      ,i4
nmflds     ,i4
fldnames   ,10a30
fldpos     ,10d3
.
.
.
xcall ib_structure(bldid, "mystruct", nmflds, fldnames, fldpos)
```

The Input Methods

The input method subroutines are written by you, and then their names are specified as arguments to Toolkit subroutines (such as `I_FLDMOD` or `IB_FIELD`) or the `.FIELD` script command. UI Toolkit then calls them at the appropriate time.

Method subroutines are important in applications that will run in a Windows environment.

`IARRIVE_METHOD` – Perform input field arrive processing

`%ICHANGE_METHOD` – Perform input field change processing

`%IDISPLAY_METHOD` – Perform input field display processing

`IDRILL_METHOD` – Perform drilldown button processing

`%IEDITFMT_METHOD` – Perform input field “edit format” processing

`IHYPER_METHOD` – Perform hyperlink prompt processing

`ILEAVE_METHOD` – Perform input field leave processing

IARRIVE_METHOD – Perform input field arrive processing



```
subroutine IARRIVE_METHOD
  a_inpinfo          , a
  a_inprec           , a
  a_methoddata1      , any
  .
  .
  a_methoddata20     , any
```

Arguments

a_inpinfo

The structure of input information. (**a**)

a_inprec

The *data_area* argument (or *data_location* argument for I_INPFLD on Windows) passed to the calling input routine. (**a**)

a_methoddata1 through *a_methoddata20*

(optional) Up to 20 additional data arguments. (**any**)

Discussion

IARRIVE_METHOD is a subroutine you write and name when you want your program to perform extra actions before a specific input field is processed.

This subroutine is called by I_INPUT, I_INPFLD (Windows only), and L_INPUT when an arrive method is specified for the field about to be processed. You can specify an arrive method for the field by using the ARRIVE_METHOD qualifier in a .FIELD script command or the D_FLD_ARRIVE option in I_FLDMOD or IB_FIELD.

The *a_inpinfo* argument is defined in **inpinf.def**. It is defined as a group and must be included within a record or as an argument. You can define multiple copies of it within your method routine. For example:

```
record data1
.include "WND:inpinf.def"
record data2
.include "WND:inpinf.def"
```

You can also use `.INCLUDE` to define this group argument without having to move it into a local data area. For example:

```
subroutine my_arrive_method
  .include "WND:inpinf.def"
  a_inprec      ,a
  a_methoddata1 ,any
  .
  .
  a_methoddata20 ,any
```

Note the following:

- ▶ For performance, make sure that this group is aligned.
- ▶ Do not modify the data in *a_inpinfo*.
- ▶ To retrieve the strings associated with the pointers, use the function `%I_GETSTRING`. (See [%I_GETSTRING on page 8-46](#) for more information.)
- ▶ The **inp_entered** and **inp_return** fields in *a_inpinfo* will be false in the arrive method. (See [“Appendix D: Methods”](#) for information on values for these fields in leave methods.)

For information on *a_inpinfo* group members, see the comments in **inpinf.def**. Additionally, note that the **inp_navevent** field lists the event that triggered navigation (arriving or leaving a field). It can have one of the following values (defined in **tools.def**):

- ▶ **D_NAVMOUSE**, which indicates that a mouse click occurred in the field (and that no subsequent event triggered focus). Note, however, that in a change, display, or leave method, this indicates that a mouse click occurred on some other window (and that no subsequent event triggered a loss of focus).
- ▶ **D_NAVTAB**, which indicates that the user pressed TAB to arrive on this field. Note that the same is true for edit format methods. For change, display, and leave methods, however, this indicates that the user pressed TAB to leave the field.
- ▶ **D_NAVSTAB**, which indicates that the user pressed SHIFT+TAB to arrive on this field. Note that the same is true for edit format methods. For change, display, and leave methods, however, this indicates that the user pressed SHIFT+TAB to leave the field.
- ▶ **D_NAVNONE**, which indicates that navigation happened for some reason other than those listed above (RETURN, menu entry, or I_NEXT, for example). Note that on UNIX and OpenVMS, this is always the value in **inp_navevent**, so you will need to use other data members (**inp_return**, **g_select**, and **g_entnam**) to determine the cause.

A_inprec is the record passed to the *data_area* argument for `I_INPUT` and `L_INPUT` and, on Windows, the *data_location* argument for `I_INPFLD`. (On UNIX and OpenVMS, this argument is not passed if `I_INPFLD` is the calling routine.)

A_methoddata1 through *a_methoddata20* are up to 20 additional optional arguments that can be passed to I_INPUT, I_INPFID, or L_INPUT. They are passed, in turn, directly to IARRIVE_METHOD. This enables you to communicate additional information to IARRIVE_METHOD.

See also

- ▶ [“Overloading UI Toolkit Routines” on page 18-8](#) for more information on creating this subroutine.
- ▶ [“Appendix D: Methods”](#).

Examples

```
.field custno, d6, user("h_popup=h_custno"), arrive_method("popup")
```

Given the above field definition, the following arrive method, popup, displays a popup-style window for the **custno** field.

```
subroutine popup
.include "WND:inpfld.def"          ;Input window information
    a_inpfld          ,a          ;Input record

.include "WND:fldinf.def"
.include "WND:tools.def"

external common
    g_popupwnd      ,i4          ;ID of currently placed popup
.align
record
    ndx              ,i4          ;instr index
    scrch            ,i4          ;Search
    err              ,i4          ;Error
    name              ,a30        ;Doubles as field name/window name
    user              ,a80        ;User text

.define D_POPID , "hpopup="

proc
    if (g_popupwnd)              ;Any currently placed window?
        begin
            xcall u_window(D_DELETE, g_popupwnd)          ;Delete it
            clear g_popupwnd                                ;Note none placed
        end

                                ;Get field
                                ;information
    xcall i_fldinf(inp_wndid, name, inp_fldnum,, gs_inpfld)

    user = %i_getstring(inp_wndid, inp_usertext)          ;Get user text
```



```

if (ndx = %instr(1, user, D_POPID))           ;Any popup
                                              ;specified?
    name = user(ndx+^size(D_POPID),^size(user)) ;Load it
if (ndx = %instr(1, name, ','))
    clear name(ndx,^size(name))               ;Allow comma
                                              ;delimiter

if (.not. name)                               ;Just "hpopup="
    xreturn
xcall u_ldwnd(g_popupwnd, g_utlib, name,, srch, err);Load the
                                              ;window

if (err)
    xreturn                                   ;No such window

;Place the window one row below and one column beyond the field being
;processed.

    xcall u_window(D_PLACE, g_popupwnd,
&          %w_info(WIF_PLCROW, inp_wndid) + gs_frow,
&          %w_info(WIF_PLCCOL, inp_wndid) + gs_fcol + gs_dsplen)
    xreturn
endsubroutine

```

%ICHANGE_METHOD – Perform input field change processing



```
function ICHANGE_METHOD, ^val
    a_data_entered      , a
    a_data_stored       , a
    a_pending_status    , n
    a_inpinfo           , a
    a_inprec             , a
    a_methoddata1       , any
    .
    .
    a_methoddata20      , any
```

Return value

%ICHANGE_METHOD must return one of the values listed under *a_pending_status* below. (n)

Arguments

a_data_entered

A buffer containing the field data as entered by the user. (a)

a_data_stored

A buffer for the final storage of the data. (a)

a_pending_status

The result of Toolkit's field validations; one of the following (defined in **tools.def**): (n)

D_OK	Input is valid, proceed.
D_RNGERR	Value not in range.
D_VALERR	Invalid numeric digit.
D_LSTERR	Not an allowed entry.
D_REQERR	Non-zero or non-blank value required.
D_DATERR	Invalid date.
D_NEGERR	Negative value not allowed.
D_OVFERR	Too many digits.
D_TIMERR	Invalid time.

D_NEGOERR Negative value only allowed.

D_EMITTEDERR Invalid entry, but error message already emitted.

a_inpinfo

The structure of input information. **(a)**

a_inprec

(optional) The *data_area* argument (or the *data_location* argument for I_INPFLD on Windows) passed to the calling input routine. This is *not* passed if the calling routine is L_INPFLD. And, on UNIX and OpenVMS, this is *not* passed if the calling routine is I_INPFLD. **(a)**

a_methoddata1 through *a_methoddata20*

(optional) Up to 20 additional data arguments. **(any)**

Discussion

%ICHANGE_METHOD is a function you write and name when you want your program to perform extra actions after a specific input field is validated. By default, a field is not validated until focus is lost. Immediate validation can be specified for check boxes, radio buttons, and combo boxes by using the D_VALSTCHG option in E_STATE. %ICHANGE_METHOD must be declared as a ^VAL function.

This function is called by I_INPUT, I_INPFLD, L_INPUT, and L_INPFLD when a change method is specified for the field being processed. You can specify a change method for the field by using the CHANGE_METHOD qualifier in a .FIELD script command or the D_FLD_CHANGE option in I_FLDMOD or IB_FIELD.

%ICHANGE_METHOD is called only after the new field contents have been subjected to all the specified Toolkit validation tests. %ICHANGE_METHOD has the “last chance” opportunity to override or add to the Toolkit’s functionality.

When %ICHANGE_METHOD is called, *a_data_stored* contains the storage form of the new data as validated by the Toolkit. If you wish to override the storage form of valid input, you may store the desired information in this argument prior to returning from the method. The following table shows the expected contents of *a_data_entered* based on field type.

Field type	Contents of <i>a_data_entered</i>
Editable (including enumerated allow list)	Data as entered
Selection list/window (including enumerated and radio buttons)	Text of selected entry
Check box	Blank (for non-checked) or the contents of g_chk_char (for checked). See g_chk_char on page 18-2 for more information.

Note that the length of the text passed in *a_data_entered* is defined by the input length for the field. It is not limited by either the display length or the view length.

If *a_pending_status* is one of the D_XXXERR values when %ICHANGE_METHOD is called, the contents of *a_data_stored* are undefined.

A_pending_status contains the result of Toolkit's validations. %ICHANGE_METHOD can simply return this value or return a different value (as the return value of the function) if you want to override Toolkit's validation of the field.

If %CHANGE_METHOD returns D_OK, the entry is accepted. Any other return value will revert the entry and storage to its original contents and force re-entry. (You do *not* need to revert the data in *a_data_entered*.) If %ICHANGE_METHOD returns any of the values D_RNGERR through D_NEGOERR, the appropriate error message is also displayed. D_EMITTEDERR should be returned when %ICHANGE_METHOD displays its own error message. (This implies that, on Windows, a change of focus within the change method is acceptable and is actually expected when returning D_EMITTEDERR.)

The include file **inpinf.def** defines *a_inpinf*. See [IARRIVE_METHOD on page 8-114](#) for more information, but note that you must include **inpinf.def** as the *fourth* argument within your change method.

A_inprec is the same record passed to the *data_area* argument for I_INPUT and L_INPUT and, on Windows, the *data_location* argument for I_INPFLD. (On UNIX and OpenVMS, this argument is not passed if I_INPFLD is the calling routine.) Note that the area of the record occupied by the current field does *not* yet contain the data being validated.

A_methoddata1 through *a_methoddata20* are up to 20 additional optional arguments that can be passed to I_INPUT, I_INPFLD, or L_INPUT. They are passed, in turn, directly to %ICHANGE_METHOD. This enables you to communicate additional information to %ICHANGE_METHOD.

See also

- ▶ [“Overloading UI Toolkit Routines” on page 18-8](#) for more information on creating this function.
- ▶ [“Appendix D: Methods”](#) for important information about method limitations.

Examples

In this example, if the user enters a non-zero value in the field, all fields in the set named “rest” are cleared and disabled. Otherwise, these fields are enabled.

```
function check_full      , ^val
    a_data_entered      ,n
    a_data_stored       ,n
    a_pending_status    ,n
    .include 'WND:inpinf.def'
    a_record            ,a
```

```
.include 'WND:tools.def'

proc
  if (g_select)
    freturn D_OK      ;Menu entry signaled, don't do anything special.

  if (a_data_entered) then ;The value entered is non-zero.
    begin
      xcall i_init(inp_wndid, 'rest', a_record) ;Clear fields in set
      xcall i_disable(D_SET, inp_wndid, 'rest') ; 'rest' and disable
                                              ; them.
    end
  else
    ;A non-zero value
    xcall i_enable(D_SET, inp_wndid, 'rest') ;Enable the fields
                                              ; in 'rest'.

  freturn D_OK      ;Accept the input for this field
end
```

%IDISPLAY_METHOD – Perform input field display processing

WT	WN	U	V

```
function IDISPLAY_METHOD, ^val
    a_data_stored      , a
    a_data_displayed   , a
    a_inpinfo          , a
    a_inprec           , a
    a_attributes       , n
    a_color            , n
    a_methoddata1      , any
    .
    .
    a_methoddata20     , any
```

Return value

%IDISPLAY_METHOD returns true to display and false to not display the contents of *a_data_displayed* into the field. (**a**)

Arguments

a_data_stored

A buffer for the final storage of the data that's about to be displayed. (**a**)

a_data_displayed

A buffer for the display form of the data. (**a**)

a_inpinfo

The structure of input information. (**a**)

a_inprec

(optional) The *data_area* argument (or the *data_location* argument for I_INPFLD on Windows) passed to the calling input routine. This is *not* passed if the calling routine is I_DSPFLD. And on UNIX and OpenVMS, it is *not* passed if the calling routine is I_INPFLD. (**a**)

a_attributes

The attribute to apply to the data displayed. One of the following flags defined in **tools.def**: (**n**)

A_NONE No attributes.

A_U Underlined.

A_R Reverse.

A_RU	Reverse, underlined.
A_B A_I	Blinking (on UNIX and OpenVMS) or italic (on Windows). “I” can be substituted whenever “B” appears in the following flags.
A_BU	Blinking, underlined.
A_BR	Blinking, reverse.
A_BRU	Blinking, reverse, underlined.
A_H	Highlighted.
A_HU	Highlighted, underlined.
A_HR	Highlighted, reverse.
A_HRU	Highlighted, reverse, underlined.
A_BH	Blinking, highlighted.
A_BHU	Blinking, highlighted, underlined.
A_BHR	Blinking, highlighted, reverse.
A_BHRU	Blinking, highlighted, reverse, underlined.
A_ALL	All attributes (same as A_BHRU).

a_color

The color to apply to the data displayed, where *a_color* is a palette number in the range 1 to 16. **(n)**

a_methoddata1 through *a_methoddata20*

(optional) Up to 20 additional data arguments. **(any)**

Discussion

%IDISPLAY_METHOD is a function you write and name when you want your program to display data differently from how Toolkit will display it. This method is called after Toolkit has formatted the display according to its rules, thus overriding or adding to Toolkit’s functionality.

This function is called by I_INPUT, I_INPFLD, I_DISPLAY, and I_DSPFLD when a display method is specified for the field being processed. You can specify a display method for the field by using the DISPLAY_METHOD qualifier in a .FIELD script command or the D_FLD_DISPLAY option in I_FLDMOD or IB_FIELD.

%IDISPLAY_METHOD is called only after the new field contents have been subjected to all the specified Toolkit display logic. %IDISPLAY_METHOD has the “last chance” opportunity to override or add to the Toolkit’s functionality.



If a view length has been specified for a field, make sure your display method allows Toolkit to perform the final display—rather than displaying the data within your display method and returning false to prevent Toolkit from displaying it. The display method should only format the data for Toolkit to display and then return true.

The **inpinf.def** file defines *a_inpinfo*. See [IARRIVE_METHOD](#) on page 8-114 for information.

Toolkit ignores any changes made to the *a_data_stored* argument. If the current field is a multi-line text field, the *a_data_stored* argument is the first element in the array containing the storage data for each line of the field.

Changes made to *a_data_displayed* are reflected in the input field if %IDISPLAY_METHOD returns true. If the current field is a multi-line text field, the *a_data_displayed* argument is the first element in the array containing the display data for each line of the field. Note that the text returned in *a_data_display* is not limited by the display length or the view length.

If the calling routine is either I_DISPLAY, I_INPUT, or I_INPFLD (on Windows), the *a_inprec* argument contains the data area that was passed to that routine. (On UNIX and OpenVMS, this argument is not passed if I_INPFLD is the calling routine.)

Changes made to the *a_attributes* and *a_color* arguments are applied to the input field’s display if %IDISPLAY_METHOD returns true. These renditions will override those otherwise specified, specifically the default renditions (bold, 3), renditions set by U_REND and U_RSTREND, and field level renditions.

A_methoddata1 through *a_methoddata20* are up to 20 additional optional arguments passed from I_INPUT, I_INPFLD, I_DISPLAY, or I_DSPFLD.

Any field with a user-defined data type will not invoke the %IDISPLAY_METHOD. Instead, the EDSPFLD_METHOD routine should be used to display directly into the field.

Toolkit will format *a_data_display* prior to calling %IDISPLAY_METHOD, but the data is not actually displayed until after the method returns.

See also

- ▶ “Renditions” on page 2-34 for information and notes on setting field-level renditions.
- ▶ “Overloading UI Toolkit Routines” on page 18-8 for more information on creating this function.
- ▶ “Appendix D: Methods” for important information on using methods.

IDRILL_METHOD – Perform drilldown button processing



```
subroutine IDRILL_METHOD
  a_inpinfo      , a
  a_inprec       , a
  a_methoddata1  , any
  .
  .
  a_methoddata20 , any
```

Arguments

a_inpinfo

The structure of input information. (**a**)

a_inprec

The *data_area* argument (or *data_location* argument for I_INPFLD on Windows) passed to the calling input routine. (**a**)

a_methoddata1 through *a_methoddata20*

(optional) Up to 20 additional data arguments. (**any**)

Discussion

IDRILL_METHOD is a subroutine you write and name when you want your program to perform extra actions when a drilldown button is pressed (on Windows) or the I_DRILL menu entry is signaled for a specific input field.

This subroutine is called by I_INPUT, I_INPFLD, and L_INPUT when a drill method is specified for the current field being processed. You can specify a drill method for the current field by using the DRILL_METHOD qualifier in a .FIELD script command or the D_FLD_DRILL option in I_FLDMOD or IB_FIELD.

On Windows, if a drill method is specified for a field, a drilldown (“...”) button is placed to the right of the field. If the user left-mouse clicks on the drilldown button, the associated drill method is invoked.

If a menu entry I_DRILL is placed on a menu column and the user selects the menu entry while processing a field that has an associated drill method, the drill method is invoked.

Your drill method can be used for a number of things. Primarily, it is used for some form of lookup or presentation of additional information. Your subroutine can use I_FORCE or I_DSPFLD to return data back into the field being processed. If the default type for the field is COPY, any change you make to the data area will affect the default value for the field.

The **inpinf.def** file defines *a_inpinfo*. See [IARRIVE_METHOD](#) on page 8-114 for information.

A_inprec is the record passed to the *data_area* argument for I_INPUT and L_INPUT and, on Windows, the *data_location* argument for I_INPFLD. (On UNIX and OpenVMS, this argument is not passed if I_INPFLD is the calling routine.)

A_methoddata1 through *a_methoddata20* are up to 20 additional optional arguments that can be passed to I_INPUT, I_INPFLD, or L_INPUT. They are passed, in turn, directly to IDRILL_METHOD. This enables you to communicate additional information to IDRILL_METHOD.

Invocation of a drill method behaves just like selecting a menu entry. Therefore, any leave method for the current field will be invoked first. To determine if a leave method is being invoked due to a drilldown, test **g_select** for true (non-zero) and **g_entnam** for "I_DRILL" in the leave method.

See also

- ▶ [“Overloading UI Toolkit Routines”](#) on page 18-8 for more information on creating this subroutine.
- ▶ [“Appendix D: Methods”](#) for important information on using methods.

Examples

```
.field name, a30, pos(1, 2), prompt("Name"), fpos(1, 15), -
    drill_method("mydrill")
```

Given the above field definition, the following drill method, mydrill, displays data to the **name** field.

```
subroutine mydrill
;
; Description: Display data to a field when the drilldown button is
;             pressed. This drill method will display the field name for
;             the field that had its drilldown button pressed.
;
; Arguments:
.include "WND:inpinf.def"          ;Input information
    a_inprec      ,a              ;Input record

.include "WND:tools.def"
record
    fldnam      ,a30
proc
    fldnam = %i_getstring(inp_wndid, inp_fldnam)
    xcall i_next(inp_wndid, "*CURR*", fldnam)
    xcall i_force(fldnam)
    xreturn
endsubroutine
```

%IEDITFMT_METHOD – Perform input field “edit format” processing



```
function IEDITFMT_METHOD, ^val
    a_data_stored      , a
    a_data_edit        , a
    a_inpinfo          , a
    a_inprec           , a
    a_attributes       , n
    a_color            , n
    a_methoddata1      , any
    .
    .
    a_methoddata20     , any
```

Return value

%IEDITFMT_METHOD should always return true. (n)

Arguments

a_data_stored

A buffer for the final storage of the data that’s about to be displayed. (a)

a_data_edit

A buffer for the edit display form of the data. (a)

a_inpinfo

The structure of input information. (a)

a_inprec

(optional) The *data_area* argument (or the *data_location* argument for I_INPFLD on Windows) passed to the calling input routine. (a)

a_attributes

The attribute to apply to the data displayed. One of the following flags defined in **tools.def**: (n)

A_NONE	No attributes.
A_U	Underlined.
A_R	Reverse.
A_RU	Reverse, underlined.

A_B A_I	Blinking (on UNIX and OpenVMS) or italic (on Windows). “I” can be substituted whenever “B” appears in the following flags.
A_BU	Blinking, underlined.
A_BR	Blinking, reverse.
A_BRU	Blinking, reverse, underlined.
A_H	Highlighted.
A_HU	Highlighted, underlined.
A_HR	Highlighted, reverse.
A_HRU	Highlighted, reverse, underlined.
A_BH	Blinking, highlighted.
A_BHU	Blinking, highlighted, underlined.
A_BHR	Blinking, highlighted, reverse.
A_BHRU	Blinking, highlighted, reverse, underlined.
A_ALL	All attributes (same as A_BHRU).

a_color

The color to apply to the data displayed, where *a_color* is a palette number in the range 1 to 16. (**n**)

a_methoddata1 through *a_methoddata20*

(optional) Up to 20 additional data arguments. (**any**)

Discussion

%IEDITFMT_METHOD is a function you write and name when you want your program to format editable data differently from how Toolkit will format it. This method is called after the Toolkit has formatted the data according to its rules, thus overriding or adding to the Toolkit’s functionality.

This function is called by I_INPUT and I_INPFLD when an edit format method is specified for the field being processed. You can specify an edit format method for the field by using the EDITFMT_METHOD qualifier in a .FIELD script command or the D_FLD_ EDITFMT option in I_FLDMOD or IB_FIELD.

%IEDITFMT_METHOD has the “last chance” opportunity to override or add to the Toolkit’s functionality. %IEDITFMT_METHOD is called only after the new field contents have been subjected to all the specified Toolkit display logic. In addition, this method isn’t called until Toolkit is in edit mode. On Windows, Toolkit enters edit mode as soon as a field gains focus. On UNIX and OpenVMS, however, Toolkit doesn’t enter edit mode until the user starts typing or selects a reserved menu entry that starts with E_. (See [“Appendix B: Reserved Menu Entries”](#) for a list of these menu entries.)

Toolkit ignores any changes made to the *a_data_stored* argument. If the current field is a multi-line text field, the *a_data_stored* argument is the first element in the array containing the storage data for each line of the field.

Changes made to the *a_data_edit* argument are reflected in the input field. If the current field is a multi-line text field, the *a_data_edit* argument is the first element in the array containing the edit display data for each line of the field. Note that the text returned in *a_data_edit* is not limited by the display length or the view length; its size is determined by the input length for the field.

The **inpinf.def** file defines *a_inpinfo*. See [IARRIVE_METHOD](#) on page 8-114 for information.

If the calling routine is I_INPUT or I_INPFLD (on Windows), the *a_inprec* argument contains the data area that was passed to that routine. (On UNIX and OpenVMS, this argument is not passed if I_INPFLD is the calling routine.)

Changes made to the *a_attributes* and *a_color* arguments are applied to the input field's display after %IEDITFMT_METHOD returns. These renditions will override those otherwise specified, specifically the default renditions (bold, 3), renditions set by U_REND and U_RSTREND, and field level renditions.

A_methoddata1 through *a_methoddata20* are up to 20 additional optional arguments passed from I_INPUT or I_INPFLD.

Toolkit will invoke the EEDTDSP_METHOD prior to calling the %IEDITFMT_METHOD for any field with a user-defined data type.

Toolkit will format *a_data_edit* prior to calling %IEDITFMT_METHOD, but the data is not actually displayed until after the method returns.

See also

- ▶ [“Renditions”](#) on page 2-34 for information and notes on setting field-level renditions.
- ▶ [“Overloading UI Toolkit Routines”](#) on page 18-8 for more information on creating this function.
- ▶ [“Appendix D: Methods”](#) for important information on using methods.

IHYPER_METHOD – Perform hyperlink prompt processing



```
subroutine IHYPER_METHOD
  a_inpinfo          , a
  a_inprec           , a
  a_methoddata1      , any
  .
  .
  a_methoddata20     , any
```

Arguments

a_inpinfo

The structure of input information. (**a**)

a_inprec

The *data_area* argument (or *data_location* argument for I_INPFLD on Windows) passed to the calling input routine. (**a**)

a_methoddata1 through *a_methoddata20*

(optional) Up to 20 additional data arguments. (**any**)

Discussion

IHYPER_METHOD is a subroutine you write and name when you want your program to perform extra actions when a hyperlink prompt is clicked (on Windows) or the I_HYPER menu entry is signaled for a specific input field.

This subroutine is called by I_INPUT, I_INPFLD, and L_INPUT when a hyperlink method is specified for the current field being processed. You can specify a hyperlink method for the current field by using the HYPERLINK_METHOD qualifier in a .FIELD script command or the D_FLD_HYPERLINK option in I_FLDMOD or IB_FIELD.

On Windows, if a hyperlink method is specified for a field, and the user left-mouse clicks on the field's prompt, the associated hyperlink method is invoked.

If a menu entry I_HYPER is placed on a menu column and the user selects the menu entry while processing a field that has an associated hyperlink method, the hyperlink method is involved.

Your hyperlink method can be used for a number of things. Primarily, it is used for some form of lookup or presentation of additional information. Your subroutine can use I_FORCE or I_DSPFLD to return data back into the field being processed. If the default type for the field is COPY, any change you make to the data area will affect the default value for the field.

The **inpinf.def** file defines *a_inpinfo*. See [IARRIVE_METHOD](#) on page 8-114 for information.

A_inprec is the record passed to the *data_area* argument for I_INPUT and L_INPUT and, on Windows, the *data_location* argument for I_INPFLD. (On UNIX and OpenVMS, this argument is not passed if I_INPFLD is the calling routine.)

A_methoddata1 through *a_methoddata20* are up to 20 additional optional arguments that can be passed to I_INPUT, I_INPFLD, or L_INPUT. They are passed, in turn, directly to IHYPER_METHOD. This enables you to communicate additional information to IHYPER_METHOD.

See also

- ▶ [“Overloading UI Toolkit Routines” on page 18-8](#) for more information on creating this subroutine.
- ▶ [“Appendix D: Methods”](#) for important information on using methods.

Examples

```
.field department, a30, pos(2, 2), prompt("Department"), fpos(2, 15), -
    help("h_department"), hyperlink_method("myhyper")
```

Given the above field definition, the following hyperlink method, “myhyper,” displays additional help for the **department** field.

```
subroutine myhyper
;
; Description: Hyperlink prompt method for additional help
;
; Arguments
.include "WND:inpinf.def"                ;Input information
    a_inprec      ,a                    ;Input record
;
;Notes:
; This routine assumes that a window exists in the default window
; library with the name "dh_" + field_name. If it does not exist,
; display an error message.
.include "WND:tools.def"
.align quad
record
    err      ,i4
    fldnam   ,a30
proc
    fldnam = %i_getstring(inp_wndid, inp_fldnam)
    xcall u_popup (g_utlib, "dh_" + fldnam,, D_NOBELL, D_ALERT, err)
    if (err)
        xcall u_message("More detailed help not available for this
field.")
    xreturn
endsubroutine
```

ILEAVE_METHOD – Perform input field leave processing



```

subroutine ILEAVE_METHOD
  a_inpinfo          , a
  a_inprec           , a
  a_methoddata1      , any
  .
  .
  a_methoddata20     , any

```

Arguments

a_inpinfo

The structure of input information. (**a**)

a_inprec

The *data_area* argument (or *data_location* argument for I_INPFLD on Windows) passed to the calling input routine. (**a**)

a_methoddata1 through *a_methoddata20*

(optional) Up to 20 additional data arguments. (**any**)

Discussion

ILEAVE_METHOD is a subroutine you write and name when you want your program to perform extra actions after a specific input field is processed.

This subroutine is called by I_INPUT, I_INPFLD (Windows only), and L_INPUT when a leave method is specified for the current field being processed. You can specify a leave method for the current field by using the LEAVE_METHOD qualifier in a .FIELD script command or the D_FLD_LEAVE option in I_FLDMOD or IB_FIELD.

The ILEAVE_METHOD is called whenever a field is deaccessed by

- ▶ focus change to a different field.
- ▶ TAB/SHIFT+TAB keys (Windows).
- ▶ mouse action (Windows).
- ▶ ENTER key.
- ▶ menu entry selection.

Note that the leave method is called prior to any break processing or any menu entry dispatching.

When the leave method is called, **g_select** is true if a menu entry caused the leave. In this case, **g_entnam** contains the name of that menu entry. Note that the menu entry has not yet been dispatched at this point, and that its action is still pending.

The **inpinf.def** file defines *a_inpinf*. See [IARRIVE_METHOD](#) on page 8-114 for information.

A_inprec is the record passed to the *data_area* argument for **I_INPUT** and **L_INPUT** and, on Windows, the *data_location* argument for **I_INPFLD**. (On UNIX and OpenVMS, this argument is not passed if **I_INPFLD** is the calling routine.)

A_methoddata1 through *a_methoddata20* are up to 20 additional optional arguments that can be passed to **I_INPUT**, **I_INPFLD**, or **L_INPUT**. They are passed, in turn, directly to **ILEAVE_METHOD**. This enables you to communicate additional information to **ILEAVE_METHOD**.

See also

- ▶ [“Overloading UI Toolkit Routines”](#) on page 18-8 for more information on creating this subroutine.
- ▶ [“Appendix D: Methods”](#) for important information on using methods.

Examples

```
.field access_code, a30, pos(3, 2), prompt ("Access code"), fpos(3, 15),
-
    leave_method("myleave")
```

Given the above field definition, the following leave method, “myleave,” validates the **access_code** field.

```
subroutine myleave
;
;Description: Field leave method for validation access code.
;
; Arguments
    .include "WND:inpinf.def"          ;Input information
    a_inprec          ,a              ;Input record
;
; Note:
;   There are up to 20 additional optional arguments that you can pass
;   into i_input, which will be passed into the field leave
;   method.
;
;   The field arrive method has the same arguments as the field leave
;   method and can be used to set up anything that needs to be set up
;   when a field is arrived on.
;
;   This field leave method validates that the access code does
;   not have any spaces and is at least 7 characters long.
```

```
;
.include "WND:tools.def"
.align quad
record
    length,i4      ;Length of input field
    fldnam ,a30    ;Field name
record data
    ,a30
    ,a30
    acc      ,a30
proc
    data = a_inprec
    length = %trim(acc)
    if ((length .lt. 7) .or. (%instr(1, acc(1:length), " ")))
        begin
            xcall u_message("Invalid access code")

            ; We get the field name from the input string associated to
            ; the field (its pointer is inp_fldnam). We need to do this
            ; because g_fldnam is NOT set at this point.

            fldnam = %i_getstring(inp_wndid, inp_fldnam)

            ; Initialize the field in the current data set

            xcall i_init(inp_wndid,, data, fldnam)
            a_inprec = data      ;Return the updated data area

            ; Reset the context back to the field
            xcall i_next(inp_wndid, "*CURR*", fldnam)
        end
    xreturn
endsubroutine
```

9

List Routines

Using Lists in Toolkit Applications 9-4

Explains the methods you can use to include a list in your Toolkit application, including using a Toolkit list and using our ActiveX implementation of a Toolkit list.

Defining Lists 9-5

Describes how to specify the general appearance of a list, the appearance of individual list items, how to associate methods with lists and use them to define list behavior, and how to use non-window data.

Processing and Maintaining Lists 9-10

Explains how to support processing functions that are common to all lists.

Notes and Tips for ActiveX Toolkit Lists 9-15

Lists issues you should keep in mind when using ActiveX Toolkit list.

Published Properties for ActiveX Toolkit Lists 9-17

Lists and discusses published properties for the ActiveX Toolkit list control.

Published Methods for ActiveX Toolkit Lists 9-22

GetCellColor – Retrieve the color of an individual cell.....	9-23
GetCellFont – Retrieve the font of an individual cell.....	9-25
GetColumnWidth – Retrieve the width of a column.....	9-27
GetFooterCell – Retrieve the text in an individual cell of a footer	9-28
GetFooterCellAlignment – Retrieve the alignment of a footer cell	9-29
GetFooterCellFont – Retrieve the font of an individual footer row	9-30
GetHeaderCell – Retrieve the text in an individual cell of a header	9-32
GetHeaderCellAlignment – Retrieve the alignment of a header cell	9-33
GetHeaderCellFont – Retrieve the font of an individual header row	9-34
GridRepaint – Force a full repaint.....	9-36
SetCellColor – Change the color of an individual cell.....	9-37
SetCellFont – Change the font of an individual cell.....	9-39

SetColumnWidth – Change the width of a column	9-41
SetFooterCell – Change the text in an individual cell of a footer.....	9-42
SetFooterCellAlignment – Change the alignment of a footer cell.....	9-43
SetFooterCellFont – Change the font of an individual footer row	9-44
SetHeaderCell – Change the text in an individual cell of a header.....	9-46
SetHeaderCellAlignment – Change the alignment of a header cell.....	9-47
SetHeaderCellFont – Change the font of an individual header row	9-48

Published Event for ActiveX Toolkit Lists 9-50

OnOriginChange – The origin (upper left cell) has changed.....	9-50

The List Subroutines and Functions 9-52

L_BORDER – Add or remove the border and drag bar on a list.....	9-54
L_BUTTON – Add, delete, or modify a button on a list.....	9-55
L_BUTTONSET – Define the placement of a set of buttons on a list	9-62
%L_BUTTONSTATE – Enable or disable a button on a list.....	9-64
L_CHR – Perform single-keystroke input to a list	9-66
L_CLASS – Create a list class at runtime.....	9-68
%L_CLASSINFO – Return information about a list class	9-72
L_CREATE – Create a list.....	9-75
L_CTRCONTEXT – Establish focus within a child list window.....	9-80
L_DATA – Manipulate the data in a list.....	9-82
L_DELETE – Delete a list	9-85
L_DISABLE – Disable a list	9-86
L_ENABLE – Enable a list.....	9-87
L_FINDSPEC – Specify parameters for a find operation.....	9-88
%L_FINDWND – Construct a find window for L_SELECT.....	9-92
L_GLOBAL – Promote a list to global status.....	9-95
%L_ICON – Set a list icon	9-96
L_INPFLD – Perform single-field input within a list.....	9-98
L_INPUT – Perform input to a set within a list.....	9-101
L_METHOD – Register a list method	9-105
L_NEXT – Set selected item by number	9-107
L_PLACE – Place a list	9-109

L_PROCESS – Process a list.....	9-111
L_QUEUE – Queue commands for a list	9-118
L_REMOVE – Remove a list	9-120
L_RESIZE – Resize a list	9-121
L_RESTART – Restart a list	9-122
L_RETURNKEY – Set the return state of privatized keys	9-123
L_SECT – Place text in a list header, footer, or title	9-125
L_SECTDRAW – Draw a line in a list header or footer	9-127
L_SELECT – Do list selection processing	9-128
%L_SELSTYLE – Retrieve/change the selection style for a list	9-136
L_STATE – Modify a list’s state.....	9-138
L_STATUS – Retrieve the status of a list	9-139
%L_USER – Set/retrieve user data associated with a list	9-144
L_VIEW – Shift the horizontal display area of a list	9-146

The List Methods 9-147

LARRIVE_METHOD/LLEAVE_METHOD – Perform list item arrive or leave processing ..	9-148
LDBLCLICK_METHOD – Perform list item double-click processing.....	9-151
LLOAD_METHOD – Perform list item “load” processing	9-154

Using Lists in Toolkit Applications

A list is a collection of similar data elements that can be used for selection processing, input processing, or just paging through large amounts of data. Lists can be static or dynamic.

A single list element or item can consist of one field or an entire record. For example, a simple, one-field list might be a one-column selection window. A more complex list might be one that contains multi-line, multi-field transaction entries.

There are several ways to include lists in your Toolkit application.

- ▶ Add a Toolkit list.
- ▶ Add our ActiveX implementation of a Toolkit list (an ActiveX Toolkit list).
- ▶ Add an ActiveX list control of your choosing.
- ▶ Add the Synergy™-supplied Spread control (an ActiveX control).

This chapter discusses the first two options listed above (using a Toolkit list and using an ActiveX Toolkit list). For information on the second two options, see [“Options for ActiveX” on page 4-3](#).

To create a list that’s portable between UNIX, OpenVMS, and all supported Windows platforms, use a Toolkit list. For a more Windows-like list, use an ActiveX Toolkit list (see [“Specifying and determining the list type” on page 9-5](#)). ActiveX Toolkit lists wrap some of the functionality of the Spread control, creating ActiveX lists that you can control with Toolkit list routines. These work as ActiveX Toolkit lists on Windows platforms and as standard Toolkit lists on UNIX and OpenVMS. The following are some of the additional features available with ActiveX Toolkit lists:

- ▶ Grid lines drawn between rows and columns
- ▶ Multi-line list entries (available with Toolkit lists on UNIX and OpenVMS, but not Windows)
- ▶ Resizable columns
- ▶ Ability to focus on any cell
- ▶ True proportional font support (columns line up when proportional fonts are used)
- ▶ Ability to move between columns with TAB and SHIFT+TAB
- ▶ Ability to specify column widths
- ▶ Ability to specify alternating row colors
- ▶ Ability to specify font and color for all cells or on a per-cell basis
- ▶ Access to published properties and methods

See [“Notes and Tips for ActiveX Toolkit Lists” on page 9-15](#) for more information on ActiveX Toolkit lists.

See [“Appendix C: Keyboard Navigation for Lists on Windows”](#) for information on keyboard navigation in Windows environments (what keys and key combinations do in their default state, which keys are privatized, and which keys can never be used as shortcuts).

Defining Lists

To set up a list, you must first define what kind of list it will be (Toolkit list or ActiveX Toolkit list), define how the list will look, and specify any methods you will write for the list. To do this, you must define a list class (or accept default list characteristics), and you must create a special input window for the list.

- ▶ The *list class* defines the general appearance of the list, enables you to associate the list with any methods you'll write for the list, and can define the list type. For information, see [“The list class: specifying general list characteristics”](#) below.
- ▶ The *input window* enables you to define the appearance of individual list items. For information, see [“The input window: specifying the list title and the appearance of individual list items”](#) on page 9-6.

In addition to the data that's displayed in a list (display data), you can add buttons to a list (using L_BUTTON and L_BUTTONSET), and you can associate a data buffer with each item in a list. The data in such a buffer is called *non-window data*. Non-window data doesn't appear in the list, but it's stored and retrieved with the display data.

Specifying and determining the list type

By default, lists are created as Toolkit lists. If you want an ActiveX Toolkit list, do one of the following:

- ▶ Set the ACTIVEX_LIST environment variable. (See [ACTIVEX_LIST](#) in the “Environment Variables” chapter of *Environment Variables & System Options*.)
- ▶ Specify the ACTIVEX option when you define the list class. (See [.LISTCLASS](#) on page 2-82.)
- ▶ Set **g_ActiveX_list** to true. (You will need to include **tkctl.def** from the synergyde/toolkit directory.)

To find out if a list is actually instantiated as an ActiveX Toolkit list, call L_STATUS with D_LACTIVEX or D_LAXCTRL. Both options return a value of 0 unless the list is currently instantiated as an ActiveX list. Based on this information, you can then alter column headings, avoid invoking methods and properties when they don't apply to the list type, and so forth.

The list class: specifying general list characteristics

By defining a list class, you can specify the following characteristics of your list:

- ▶ Row and column position
- ▶ Number of visible lines
- ▶ Header and footer sections
- ▶ Scroll bars
- ▶ Whether a null list is allowed

- ▶ Whether the list will maintain input window control information for each item in the list
- ▶ Whether the list will be a Toolkit list or an ActiveX Toolkit list

You can also associate methods with a list. You can specify

- ▶ a load method. This is a subroutine that supplies the items for the list.
- ▶ an arrive method and a leave method. These are subroutines that perform additional processing when a list item is accessed or deaccessed.
- ▶ a double-click method. This is a subroutine that performs additional processing when the user double-clicks on a list item.

Creating a list class

There are two ways you can define a list class: in a script file or at runtime. To define a list in a script file, use the `.LISTCLASS` command. To define a list class at runtime, use the `L_CLASS` subroutine.

When you create the list, you'll pass the name of the list class to `L_CREATE`. (If you don't pass the list class name, which is optional, the list will have the default characteristics documented in [.LISTCLASS on page 2-82](#) and [L_CLASS on page 9-68](#).) Once you've defined a list class, you can use it as the basis for one or more lists in your application. List classes can help standardize your user interface and simplify global changes to your lists.

Note the following:

- ▶ Although the list class defines the characteristics of the header and footer, you must use the `L_SECT` subroutine to place text in the header or footer, and you must use the `L_SECTDRAW` subroutine to draw lines in the header or footer.
- ▶ You can overload (replace) methods specified in the list class. To do this, use the `L_METHOD` subroutine.

The input window: specifying the list title and the appearance of individual list items

To specify the appearance and layout of individual list items, you must create an input window that defines the layout of one list item. For example, the following input window definition specifies three fields, which will result in a list with three columns:

```
.input detlstimp, 1, 49
.repository_structure ORDER_DETAIL
.field OD_PRODUCT, noprompt, fpos(1, 5), nodrill_method
.field OD_ORD_QTY, noprompt, fpos(1, 20)
.field OD_PRICE, noprompt, fpos(1, 35)
.end
```


Note that the example above specifies only one row. This example would result in a list in which each list item takes up only one row, which is the most common scenario. List items can, however, take up more than one row. (On Windows, multi-row list items are supported only for ActiveX lists.) To create a list with multi-row items, define an input window with one line for each list item row. For example, if you want each list item to take up three rows (to hold an entire record, for example), define three lines in the input window.

Note that there's no need to specify borders, and column headings usually replace individual field prompts. The input window title is also optional; if you specify it, it will become the list title.



Never explicitly remove the input window associated with a list. Doing so will corrupt the placement order. If you want to use the same input window for other input, you will need to load a separate copy of the window.

Creating the input window

You can create the input window in a script file or at runtime. To create it in a script file, use the `.INPUT` command. To create the input window at runtime, use the input build (`IB_`) subroutines. Then, once you've created the list, pass the input window ID to `L_CREATE`. Note that an input window can be associated with only one list.

The input window's other role: processing input

You'll use the input window to pass display data between the list processor and your application. When the list processor asks your application for data for a list item, you simply place the data in the input window. The list processor uses that data to display the current item in the list. When list processing is complete, the input window is returned to your application with the data for the new current item along with any non-window data that has been associated with this item. Your application can also use this input window to edit list items.

Using non-window data

In addition to the data that's displayed in a list (display data), you can associate a data buffer with a list. The data in this buffer is called *non-window data*. Non-window data doesn't appear in the list, but it's stored and retrieved with the display data and may duplicate the display data.

To use non-window data, pass the initial non-window data item as the *data* argument to `L_CREATE` when you create the list. When you call a list processing routine, place non-window data for the current item by passing it as the *data* argument. When processing is complete, the list processing routine will return the non-window data for the new current item as the *data* argument.

Unlike display data, which is stored and retrieved as displayed, non-window data is stored and retrieved in the form passed to `L_CREATE` and the list processing routines. For more information on non-window data, see the `L_CREATE` [Discussion on page 9-76](#).

Defining behavior for lists

List processing methods enable you to define the behavior of a list. Methods are routines you write; they are a modularized way to determine how lists are loaded and any additional processing that you want to happen when an item is accessed, de-accessed, or double-clicked.

The `.LISTCLASS` script command and `L_CLASS` subroutine have arguments that enable you to specify the method names when you define the list class. You can use the `L_METHOD` subroutine to overload methods that you've previously specified with the list class or to register methods not specified in the list class.

On Windows, applications *must* use method subroutines (as opposed to inline code).

See [“Appendix D: Methods”](#) for information on issues that are common to different types of methods.

List arrive and leave methods

Arrive and leave methods enable your program to perform additional processing when arriving at or leaving a list item. Arrive and leave methods are supported on all platforms, but are especially useful on Windows, where users can change the current list item during a call to a Toolkit subroutine, such as `L_PROCESS`. In this situation your program may not be in control (the user may use the mouse, for instance), so you must create an arrive or leave method if you want additional processing.

In addition to writing an arrive or leave method, you must also associate the arrive or leave method with the list. To do this, do one of the following:

- ▶ Pass the name of the arrive or leave method as the *arrive_method* or *leave_method* argument to the `L_CLASS` subroutine.
- ▶ Specify the name of the arrive or leave method after the `ARRIVE_METHOD` or `LEAVE_METHOD` option in a `.LISTCLASS` script command.
- ▶ Pass the name of the arrive or leave method after the `D_LARRIVE` or `D_LLEAVE` option in a call to the `L_METHOD` subroutine.

See [LARRIVE_METHOD/LLEAVE_METHOD on page 9-148](#) for more information.

List double-click method

By default, Toolkit simulates an `ENTER` when the user double-clicks on a list item. If you want something different to occur for a double-click, you can write an `LDBLCLICK_METHOD` subroutine and either pass the *dblclick_method* argument to `L_CLASS` or specify `DOUBLE_CLICK_METHOD` in your `.LISTCLASS` script.

In addition to writing a double-click method, you must also associate the double-click method with the list. To do this, do one of the following:

- ▶ Pass the name of the double-click method as the *dblclick_method* argument to the `L_CLASS` subroutine.

- ▶ Specify the name of the double-click method after the `DOUBLE_CLICK_METHOD` option in a `.LISTCLASS` script command.
- ▶ Pass the name of the double-click method after the `D_LDBLCLCK` option in a call to the `L_METHOD` subroutine.

See [LDBLCLICK_METHOD on page 9-151](#) for more information.

List load method

The list load method is an external subroutine you write. This subroutine is called by the list processing routines (`L_PROCESS`, `L_SELECT`, or `L_INPUT`) when an item needs to be loaded into the list (when the user scrolls down the list, for example). The load method must be written to load a single list item at a time. In other words, it should retrieve a single item from your data file (using `READS`, for example), and then, using `I_DISPLAY`, display that item in the input window associated with the list.

If you are developing an application for Windows, you must use a load method. If you are writing code for UNIX or OpenVMS only, it isn't required, but is recommended. If you don't use a load method for UNIX or OpenVMS systems, the list processing routines will use their default processing, which is to return to the calling routine when new items are needed. Load methods free your application from having to satisfy load requests whenever a list processing routine is called, and they encapsulate your item-loading code.

In addition to writing the load method subroutine, you must also associate the load method with the list. To do this, do one of the following:

- ▶ Pass the name of the load method as the *load_method* argument to the `L_CLASS` subroutine.
- ▶ Specify the name of the load method after the `LOAD_METHOD` option in a `.LISTCLASS` script command.
- ▶ Pass the name of the load method after the `D_LLOAD` option in a call to the `L_METHOD` subroutine.

See [LLOAD_METHOD on page 9-154](#) for more information on creating a load method.

Processing and Maintaining Lists

These are the basic steps you'll follow to process a Toolkit or ActiveX Toolkit list:

1. In your program, use `L_LDINP` to load the input window you created for the list. But *don't* place the window. (Be sure to pass `D_NOPLC` as the *no_placement* argument.) See [L_LDINP on page 8-62](#) for information.
2. Use `L_CREATE` to create the list. In this step, you'll pass the name of the list class and input window, and you'll determine how the list will be loaded (top to bottom, bottom to top, or from the middle). See [L_CREATE on page 9-75](#) for more information.
3. (optional) Use `L_SECT` to add text to the header, footer, or title.
4. (optional) Place the list with `L_PLACE`, and enable the list with `L_ENABLE` if necessary.
5. Process the list with `L_SELECT`, `L_PROCESS`, or one of the list input routines. These routines should be called from a loop. See “[Performing list processing](#)” below.
6. Delete the list either explicitly, with `L_DELETE` (see [L_DELETE on page 9-85](#)), or implicitly by exiting the environment in which the list was created.

Note that there are other routines that enable you to maintain lists. There are routines that enable you to remove a list, enable a list, disable a list, queue requests for a list, promote a list to global status, and so forth. See “[The List Subroutines and Functions](#)” on [page 9-52](#) for information.

Performing list processing

The three primary list processing routines are `L_INPUT`, `L_PROCESS`, and `L_SELECT`. Both `L_PROCESS` and `L_SELECT` support the processing functions that are generic to all Toolkit and ActiveX Toolkit lists:

- ▶ Load initial set of items into the list
- ▶ Insert in and delete items from the list
- ▶ Update the current item
- ▶ Cancel changes to the current item
- ▶ Move through the list one item or page at a time
- ▶ Move to the top or bottom of the list
- ▶ Move an item in the list one row at a time
- ▶ Move the display area of the list to the left or to the right
- ▶ Call list load method when new list items are needed for the display

Neither `L_PROCESS` nor `L_SELECT` performs any input. (Use one of the list input routines, such as `L_INPUT`, to do this.) `L_PROCESS` performs any requests to the list (such as to insert an item, delete an item, and so forth) and returns to the calling program. `L_SELECT` also performs requests,

but it returns only when the user has selected an item from the list or has selected an unknown menu entry. See [L_PROCESS on page 9-111](#), [L_SELECT on page 9-128](#), and [L_INPUT on page 9-101](#) for more information.

Call `L_INPUT` or `L_SELECT` from a loop in your application, as you would `I_INPUT` or `M_PROCESS`. Use an `L_INPUT` loop when you want to perform input into the list. Use an `L_SELECT` loop for making a selection from a list of existing items. One of the arguments to `L_PROCESS` and `L_SELECT` is a request flag, which is used by the calling routine to request action from `L_PROCESS` or `L_SELECT` and by these routines to request action from the calling routine. Note that `L_INPUT` does not have a request flag argument. Use calls to `L_PROCESS` within your `L_INPUT` loop to handle requests.

How list items are loaded

Toolkit lists and ActiveX Toolkit lists are empty when they are first created, but the first call to a list processing routine loads the list with the list items needed for the initial display. List items are loaded only as they are needed for display. The first time an item is needed, the list processing routine requests the item from the list load method (or the calling routine if you don't have a load method) and loads the item into the list. Once an item is loaded, UI Toolkit continues to use it when necessary. It doesn't need to be loaded again. Because the list processor maintains loaded items in a linked list on the file stack, the calling routine does not need to supply loaded items a second time, which improves processing speed. (Refer to [“Appendix E: File-Stack Processing”](#) and [“Appendix F: Linked-List Processing”](#) for more information about the file-stack and linked-list subroutines.)



We strongly recommend that you use a load method to load items into a list. For Windows applications, you *must* write and use a list load method to enable scrolling.

For information on list load methods, see [“List load method” on page 9-9](#).

List processing examples

A selection list

The following example creates and processes a simple look-up list. The attributes of the list are defined by the list class `lst_lookup`. The items for the list are provided by the load method `listload_method`, which is automatically called by `L_SELECT`. (See [“Load method for the examples” on page 9-13](#).) The list enables a selected item to be deleted. Choosing delete sets the request flag to `D_LDELITEM`, which instructs the next call to `L_SELECT` in the loop to process the request. Note that an item is deleted only from the visible list, not from the file from which the list was loaded.

```
xcall i_ldinp(inpid, g_utlib, "inp_lookup")
xcall l_create(listid, inpid, data, g_utlib, "lst_lookup")
xcall l_button(listid, "O_EXIT", DSB_TEXT, "Exit")
xcall l_button(listid, "O_DELETE", DSB_TEXT, "Delete")
xcall l_method(listid, D_LLOAD, "listload_method")
```

List Routines

Processing and Maintaining Lists

```
request = D_LNOP
repeat
  begin
    xcall l_select(listid, request, data)
    if (g_select) then                                ;Unknown menu entry
      begin
        using g_entnam select
          ('O_EXIT'), exitloop
          ('DELETE'), request = D_LDELITEM
        endusing
      end
    else
      xcall u_msgbox (data)                            ;Display chosen entry
    end
  end
```

An input list

The following example creates and processes a simple input list. The attributes of the list are defined by the list class **lst_process**. The items for the list are provided by the list load method **listload_method**, which is automatically called by L_INPUT. (See [“Load method for the examples” on page 9-13.](#)) The list enables existing items to be deleted and new ones to be inserted. Choosing insert or delete calls L_PROCESS with the request flag set to the appropriate value. Note that an item is deleted only from the visible list, not from the file from which the list was loaded.



Direct input into an ActiveX Toolkit list may cause repainting problems. See [“Notes and Tips for ActiveX Toolkit Lists” on page 9-15](#) for information.

```
xcall i_ldinp(inpid, g_utlib, "inp_process")
xcall l_create(listid, inpid, data, g_utlib, "lst_process")
xcall l_button(listid, "O_EXIT", DSB_TEXT, "Exit")
xcall l_button(listid, "O_DELETE", DSB_TEXT, "Delete")
xcall l_button(listid, "O_INSERT", DSB_TEXT, "Insert")
xcall l_method(listid, D_LLOAD, "listload_method")
request = D_LNOP
repeat
  begin
    xcall i_next(inpid,, "*FRST*")
    xcall i_input(listid,, data, data)
    if (g_select) then                                ;Unknown menu entry
      begin
        using g_entnam select
          ('O_EXIT'), exitloop
          ('O_DELETE'), xcall l_process(listid, request=D_LDELITEM,
&                                data)
          ('O_INSERT'), xcall l_process(listid, request=D_LINSERT,
&                                data)
        endusing
      end
    end
  end
```

```

        end
    else
        ;process new entry
    end
end

```

Load method for the examples

The load method for the above list examples would look like this:

```

subroutine listload_method
;
; Description: Load method for lookup and input lists
; Arguments:
    a_listid      ,n      ;ID of the list
    a_req         ,n      ;Request
    a_data        ,a      ;Data
    a_inpid       ,n      ;Input window ID
    a_disabled    ,n      ;(Optional) disabled item flag
    a_itemindex   ,n      ;Request item number

static record
    chn           ,i4      ;Channel for loading

proc
    if (a_itemindex .eq. 1) ;First item?
        xcall u_open(chn, "I:I", "customer") ;Open file
        reads(chn, a_data) [eof=nomore] ;Read next record
        xcall i_display(a_inpid, "customer_set", a_data) ;Display the data
        xreturn

nomore,
    a_req = D_LEOF ;Signal no more
    xreturn
endsubroutine

```

Enabling users to search a list

Depending on the subroutine you use to process your list (L_INPUT or L_SELECT), there are two ways to provide a search for your list.

- ▶ If you use L_INPUT, use L_FINDSPEC to specify the parameters the search, and then use the D_LFIND request with L_PROCESS to execute the search. For more information, see [L_FINDSPEC on page 9-88](#) and [L_PROCESS on page 9-111](#).
- ▶ If you use L_SELECT, use the find options that are part of the L_SELECT subroutine. For information, see [L_SELECT on page 9-128](#). You can also use the L_FINDSPEC routine to provide more search functionality.

You can search the display data or the non-window data. See [L_FINDSPEC on page 9-88](#) for more information on how to specify which type of data will be searched.

IDs for list windows

There are two types of window ID for a list: the ID for the input window used to define the list and the ID for the list window, which is the window that contains the list when the list is placed. When you create a list with `L_CREATE`, you supply the first type of window ID. If, however, you want to use other Toolkit routines that require a window ID (for example, `%U_WINMETRICS`), you must supply the ID for the list window. To get the list window ID, use the `D_LCTRID` subfunction for `L_STATUS`.

Notes and Tips for ActiveX Toolkit Lists

The following is a list of issues you should keep in mind when using the ActiveX implementation of a Toolkit list. For additional notes, see [“Using Lists in Toolkit Applications” on page 9-4](#). For information on setting options that cause lists to be ActiveX lists, see [“Specifying and determining the list type” on page 9-5](#).

- ▶ To retrieve the ActiveX control ID for an ActiveX Toolkit list, use `L_STATUS`.
- ▶ Prompts and fields create columns in an ActiveX Toolkit list. (Other text residing in the window does not.) When using multi-line list entries, the widest entry in a column determines the width of the column. For selection fields, the width of the column is set to the width of the widest selection list entry, regardless of the size of the field, unless the selection field is enumerated, in which case the enumerated display length is used instead.
- ▶ Prompts and fields are aligned within their corresponding columns without changing their content. Although this preserves leading spaces within a prompt, it does not preserve leading spaces before a prompt.
- ▶ To create an empty cell for alignment purposes, create a field that is not part of the input set.
- ▶ If the input window used for an ActiveX list contains fields that have display or view lengths that are different than their field lengths, don't use windowing display routines (e.g., `W_DISP`) to populate these fields. The data won't display correctly. Use `I_DISPLAY`, `I_DSPFLD`, or `I_PUTFLD`.
- ▶ Unlike Toolkit lists, ActiveX Toolkit lists enable users to scroll the current item out of view without causing focus to move to another item. This is mostly for performance, but it is also a common behavior for Windows applications. There are times, however, when the internal list control snaps the currently focused item into view when `L_INPUT` or `L_INPFLD` is being used (as opposed to `L_SELECT`). This usually occurs when the user expects it (when focus moves to a new cell that is also not currently in view, for example), but it can occur at unexpected times. For example, this can happen when the initial and/or last item parameters passed to `L_CREATE` (or the default estimates if these parameters are not passed) prove incorrect by the responses from the load method. At that point, the internal list must be reorganized for the new list dimensions, which often causes the focused cell to snap back into the viewable area. For instance, if you have a list that's longer than twice the viewable area, you don't pass the *last* argument to `L_CREATE`, and the user uses the scroll bar to scroll all the way down, the list may suddenly snap back to the focused item when the item loaded exceeds the initially estimated last item. Note that this only occurs in `L_INPUT` or `L_INPFLD` (not in `L_SELECT`), and does not occur if a correct estimate of the dimensions of the list is given to `L_CREATE`.
- ▶ When entering input into the list, the first field in the list will be selected even if it is not in the current input set.
- ▶ Dragging the scroll bar doesn't update the list until the mouse button is released.
- ▶ Dragging the scroll bar all the way down may or may not go to the bottom of the list. For the scroll bar to work properly, you need to provide an accurate number (or overestimate the number) of list items for `L_CREATE`.

- ▶ If the width of a column is narrower than its header or footer text, that text will be truncated unless vertical bars (|) are used within the text to designate the initial width of the column. See [L_SECT on page 9-125](#) for more information.
- ▶ The order of items loaded is different when two-way loading is enabled. The specific order we have chosen to implement achieves the best performance for each type of list: Toolkit lists load as needed for display, and ActiveX Toolkit lists load ahead further in each direction to minimize self-correction when the initial estimates are incorrect. (See [“L_CREATE” on page 9-75](#) for information on loading methods for lists.)
- ▶ There may be repainting issues if users input directly into an ActiveX Toolkit list. As an alternative, you can set up the list as a selection list (using `L_SELECT`) and then create a separate input window for adding and updating list items. To see an example of this method, open *S/DE Repository* and select *Modify > Structures*. This opens the *Structures Definitions* list (which isn't an ActiveX Toolkit list, but illustrates the method). Note that you can highlight list entries, but you can't make any changes in this window. To make a change, you must highlight a list entry and press `ENTER`. This opens a separate input window, which enables you to add and update list items.
- ▶ By default, the size of a list container is based on the display area of the associated input window. However, if you don't specify this display area for the input window, the runtime calculates the size by adding the columns using the larger of the input or display format to determine the size of each column. If any column has a header or footer that's larger than the column's input or display format, however, the container won't be large enough to show all the columns. To correct this, set the display area for the input window by doing the following:
 1. For each column, find the largest of the following: the header, the footer, the display format, or the input format.
 2. Add the numbers from step 1. (There should be a number for each column.) Then add an additional one or two characters for each column.
 3. Test the window by compiling the window and looking at the list. If the column header, footer, or data is truncated, increase the display area until everything displays properly.

For example, if your list has two columns (one whose largest element is the header, which has 13 characters, and a second whose largest element is the footer, which has 15 characters), add these two numbers. The sum is 28. Then add four to the total, which makes 32. (In this case, we're adding two characters for each column in the list.) The `display_area` line of the list definition is

```
display_area 1, 1, 1, 32
```

Set the window width larger than the sum calculated in [step 2](#) above. For example, if you calculate that the display area needs to be 32 characters, set the window size to at least 33:

```
.input table_list, 1, 33
```

Published Properties for ActiveX Toolkit Lists

A *published property* is a named attribute that is associated with a given ActiveX control. The following published properties are supported for the ActiveX Toolkit list control in Traditional Synergy. (They are not available in Synergy .NET.) All these can be read (using AX_GET or %AX_GETINT), and most can be modified (using AX_SET). (Note that changing a property using AX_SET will change the object's visual representation if necessary.)

Property names are case sensitive. Settings that specify invalid values are ignored.

Published Properties for ActiveX Toolkit Lists			
Name	Type	Write?	Description
ActiveCellHighlightStyle	n	Read/write	A value that determines whether the focus rectangle is visible. By default, the ActiveCellHighlightStyle property is set to 0, which makes the focus rectangle invisible. If you want the cell focus rectangle to be visible (whether or not the list has focus), set ActiveCellHighlightStyle to 1. Note that Toolkit sets this property to 1 when starting L_INPUT or L_SELECT processing, and that it sets it to 0 when exiting L_INPUT or L_SELECT processing. You will need to reset it to 1 if you want the focus rectangle to remain visible after L_INPUT or L_SELECT processing.
ArrowKeyAction	n	Read/write	A value that determines how UP ARROW, DOWN ARROW, and other navigation keys work. For example, when set to 0 (the default), UP ARROW and DOWN ARROW move highlighting up and down one row at a time. When set to 1, UP ARROW and DOWN ARROW move highlighting up and down one item at a time. (An item can have more than one row.) See “Appendix C: Keyboard Navigation for Lists on Windows” for details.
BackColor	n	Read/write	The RGB value of the background color for all cells. See the GetCellColor Discussion on page 9-23 for information on RGB values.
Border	n	Read/write	A value that determines whether the border for an ActiveX Toolkit list is visible. By default, this property is set to 1, which makes the border visible. If you want the border to be invisible, set this to 0.
ColumnsPerRow	n	Read-only	The number of columns in the list.

List Routines

Published Properties for ActiveX Toolkit Lists

Published Properties for ActiveX Toolkit Lists (Continued)			
Name	Type	Write?	Description
CurrentColumn	n	Read/ write	The horizontal column number of the currently highlighted cell. Note that modifying CurrentColumn from a method that is called from L_INPUT can have unpredictable results. If this happens, L_INPUT both modifies and depends on the value of CurrentColumn. Also note that setting RowMode won't affect the CurrentColumn setting. But if RowMode is on, a CurrentColumn setting won't affect the way highlighting works in a table.
DarkItemColor	n	Read/ write	The RGB value for the background color of the second alternating item. See the GetCellColor Discussion on page 9-23 for information on RGB values. If DarkItemColor is modified, L_RESTART must be called before the color change will take effect.
Enabled	n	Read/ write	A value that determines if the control is enabled for input. This should be set and is read as a true or false value. Note that if Enabled is set to false (0) and the window subsequently receives focus (by calling L_SELECT, L_INPUT, etc.), a fatal error will occur.
Ext_hWnd	n	Read- only	The window handle of the control.
Ext_Visible	n	Read/ write	A value that determines whether the control is visible. This property should be set and is read as true or false value. Note that if Ext_Visible is set to false (0) and the window subsequently receives focus (by calling L_SELECT, L_INPUT, etc.), a fatal error will occur.
FontName	a	Read/ write	The typeface of the font for all cells.
FontSize	n	Read/ write	The point size of the font for all cells.
FooterFontName	a	Read/ write	The typeface of the font for all footers.
FooterFontSize	n	Read/ write	The point size of the font for all footers.

Published Properties for ActiveX Toolkit Lists (Continued)			
Name	Type	Write?	Description
HeaderFontName	a	Read/ write	The typeface of the font for all headers.
HeaderFontSize	n	Read/ write	The point size of the font for all headers.
HorizontalScrollBar	n	Read/ write	A value that determines whether the horizontal scroll bar will be shown when the list is too wide to be fully visible. This should be set and is read as a true or false value.
LightItemColor	n	Read/ write	The RGB value for the background color of the first alternating item. See the GetCellColor Discussion on page 9-23 for information on RGB values. If LightItemColor is modified, L_RESTART must be called before the color change will take effect.
OriginColumn	n	Read/ write	The leftmost column that is currently visible.
NonGridBackColor	n	Read/ write	The RGB value for the color of the background area that is outside the bounds of data columns but inside the ActiveX control. See the GetCellColor Discussion on page 9-23 for information on RGB values.
RowMode	n	Read/ write	A value that determines whether the list will have cell highlighting or row highlighting when used with L_SELECT. If RowMode is set to a non-zero value, an entire row is highlighted when a user selects a cell in the row. If RowMode is set to zero, just the selected cell is highlighted (which is the default). Note that RowMode has no effect with L_INPUT or L_INPFLD.
SelBackColor	n	Read/ write	The RGB value for the background color for cells in a selected row. This is applicable only when RowMode is set to a non-zero value. See the GetCellColor Discussion on page 9-23 for information on RGB values.
SelForeColor	n	Read/ write	The RGB value for the text color for cells in a selected row. This is applicable only when RowMode is set to a non-zero value. See the GetCellColor Discussion on page 9-23 for information on RGB values.

List Routines

Published Properties for ActiveX Toolkit Lists

Published Properties for ActiveX Toolkit Lists (Continued)			
Name	Type	Write?	Description
TextColor	n	Read/ write	The RGB value of the text color for all cells. See the GetCellColor Discussion on page 9-23 for information on RGB values.
VerticalScrollBar	n	Read/ write	A value that determines if the vertical scroll bar is shown when the list is too long to be fully visible. This property should be set and read as a true or false value.

ActiveX Toolkit lists and color

Color for ActiveX Toolkit lists is applied in the following order:

1. Color specified for the cell (if selected and RowMode is set) by the SelBackColor and the SelForeColor properties
2. Color specified for the list by the BackColor and TextColor properties
3. Color specified for the list by the DarkItemColor and LightItemColor properties
4. Color specified for the field in a script file or by IB_FIELD
5. Color specified for the field by I_FLDMOD
6. Color specified for the cell by the SetCellColor method

Setting a color that is higher in the hierarchy list above will not override a color that is at a lower level. For instance, setting the BackColor property will not have any effect on cells that have had the SetCellColor method applied.

ActiveX Toolkit lists and color

Fonts for ActiveX Toolkit lists are applied in the following order:

1. Font specified for the list's input window
2. Font specified for this list by the FontName and FontSize properties
3. Font specified for this cell by the SetCellFont method

Similar to color, setting a font that is higher in the hierarchy list above will not override a font that is at a lower level. For instance, setting the FontName property will not have any effect on cells that have had the SetCellFont method applied.

Examples

The following example sets the background color of a list to a light teal.

```
xcall l_status(listid, D_LAXCTRL, axctrl)
xcall ax_set(axctrl, "BackColor", RGB_VALUE(204,255,255))
```

This example sets the default text color for a list to dark blue, specifies row highlighting when a cell is selected, and sets the background and text colors for selected cells to different shades of green.

```
xcall l_status(listid, D_LAXCTRL, axctrl)
xcall ax_set(axctrl, "TextColor", RGB_VALUE(0,0,102))
xcall ax_set(axctrl, "RowMode", 1)
xcall ax_set(axctrl, "SelBackColor", RGB_VALUE(153, 204, 153))
xcall ax_set(axctrl, "SelForeColor", RGB_VALUE(102, 102, 0))
```

This example simulates the look of green-bar paper by changing colors for alternating items.

```
xcall l_status(listid, D_LAXCTRL, axctrl)
xcall ax_set(axctrl, "DarkItemColor", RGB_VALUE(153,204,153))
xcall ax_set(axctrl, "LightItemColor", RGB_VALUE(227,255,227))
xcall l_restart(listid) ;Restart required to paint the colors.
```

This example sets the font typeface and point size to match a Synergy font named ActiveXListFont.

```
xcall l_status(listid, D_LAXCTRL, axctrl)
xcall u_wndfont(D_GETFONT, "ActiveXListFont", fontname, ptsize)
xcall ax_set(axctrl, "FontName", fontname)
xcall ax_set(axctrl, "FontSize", ptsize)
```

This example creates a Synergy font named “HeaderFont” that matches the font used by the list.

```
xcall l_status(listid, D_LAXCTRL, axctrl)
xcall ax_get(axctrl, "HeaderFontName", fontname)
xcall u_wndfont(D_SETFONT, "HeaderFont", fontname,
& %ax_getint(axctrl, "HeaderFontSize"))
```

The following example changes the footer font to Times New Roman, 12 point.

```
xcall l_status(listid, D_LAXCTRL, axctrl)
xcall ax_set(axctrl, "FooterFontName", "Times New Roman")
xcall ax_set(axctrl, "FooterFontSize", 12)
```

The following example prevents scroll bars from being displayed.

```
xcall l_status(listid, D_LAXCTRL, axctrl)
xcall ax_set(axctrl, "HorizontalScrollBar", FALSE)
xcall ax_set(axctrl, "VerticalScrollBar", FALSE)
```

This example scrolls the list horizontally so that the third column is to the left of the visible area.

```
xcall l_status(listid, D_LAXCTRL, axctrl)
xcall ax_set(axctrl, "OriginColumn", 3)
```

Published Methods for ActiveX Toolkit Lists

A *published method* is a named routine that can be called for a given ActiveX control. Method names are case sensitive. Methods are manipulated using the %AX_CALL function.

The following published methods are supported:

GetCellColor – Retrieve the color of an individual cell.

GetCellFont – Retrieve the font of an individual cell.

GetColumnWidth – Retrieve the width of a column.

GetFooterCell – Retrieve the text in an individual cell of a footer.

GetFooterCellAlignment – Retrieve the alignment of a footer cell.

GetFooterCellFont – Retrieve the font of an individual footer row.

GetHeaderCell – Retrieve the text in an individual cell of a header.

GetHeaderCellAlignment – Retrieve the alignment of a header cell.

GetHeaderCellFont – Retrieve the font of an individual header row.

GridRepaint – Force a full repaint.

SetCellColor – Change the color of an individual cell.

SetCellFont – Change the font of an individual cell.

SetColumnWidth – Change the width of a column.

SetFooterCell – Change the text in an individual cell of a footer.

SetFooterCellAlignment – Change the alignment of a footer cell.

SetFooterCellFont – Change the font of an individual footer row.

SetHeaderCell – Change the text in an individual cell of a header.

SetHeaderCellAlignment – Change the alignment of a header cell.

SetHeaderCellFont – Change the font of an individual header row.

You can retrieve the ActiveX control ID for an ActiveX Toolkit list by using L_STATUS.

GetCellColor – Retrieve the color of an individual cell

WT			

```
status = %AX_CALL (axlist_id, "GetCellColor", row, column, foreground_color,
&                background_color)
```

Return value

status

Returns true if successful, otherwise false. **(n)**

Arguments

axlist_id

The ID for the ActiveX Toolkit list. This ID can be retrieved with L_STATUS. **(n)**

row

The row number of the cell whose color is being retrieved. **(n)**

column

The column number of the cell whose color is being retrieved. **(n)**

foreground_color

The returned text color (RGB value) of the cell. **(n)**

background_color

The returned background color (RGB value) of the cell. **(n)**

Discussion

The GetCellColor method retrieves the foreground (text) and background color for a specified cell as RGB values.

An RGB value is a four-byte integer value that includes the red, green, and blue components of a color:

`^X(00bbggrr)`

where *bb*, *gg*, and *rr* are intensity values in the decimal range 0 through 255 (00 through FF in hexadecimal) for the blue, green, and red color components. This value can be stored in other numeric types (decimal, for instance), but the variable must be large enough to hold at least a value of `^X(FFFFFF)` or truncation may occur.

Toolkit includes macros (**defined in DBLDIR:activex.def**) that make it easier to use these values:

- ▶ `RGB_VALUE(rr,gg,bb)` enables you to construct an RGB value from decimal values for red, green, and blue.
- ▶ `RED_PART(rgb_value)`, `GREEN_PART(rgb_value)`, and `BLUE_PART(rgb_value)` enable you to extract the decimal value for blue, green, or red from an RGB value.

There's no reason to use the `RGB_VALUE` macro with this method, but the `RED_PART`, `GREEN_PART`, and `BLUE_PART` macros are convenient for extracting red, green, and blue values from *foreground_color* and *background_color*.

See also

- ▶ [SetCellColor on page 9-37](#).
- ▶ [%AX_CALL](#) in the "Synergy ActiveX API" chapter of the *Synergy DBL Language Reference Manual*.

Examples

The following example brightens the background color of the fourth column in the currently selected row by approximately 50%.

```
xcall l_status(listid, D_LAXCTRL, axctrl, D_LCURITM, curr)
xcall ax_call(axctrl, "GetCellColor", curr, 4, fgcolor, bgcolor)
red = RED_PART(bgcolor)
green = GREEN_PART(bgcolor)
blue = BLUE_PART(bgcolor)

;Compute average distance from brightest possible
avgdelta = ^x(FF) - ((red + green + blue) / 3)
avgdelta /= 2 ;Multiply by 50%
red += avgdelta ;Brighten each color by that amount
if (red .gt. ^x(FF) ; or as bright as possible
    red = ^x(FF)
green += avgdelta
if (green .gt. ^x(FF))
    green = ^x(FF)
blue += avgdelta
if (blue .gt. ^x(FF))
    blue = ^x(FF)
xcall ax_call(axctrl, "SetCellColor", curr, 4,
& RGB_VALUE(red,green,blue), 1)
```

GetCellFont – Retrieve the font of an individual cell

WT			

```
status = %AX_CALL (axlist_id, "GetCellFont", row, column, face_name, point_size,
&          rendition)
```

Return value

status

Returns true if successful, otherwise false. (n)

Arguments

axlist_id

The ID for the ActiveX Toolkit list. This ID can be retrieved with L_STATUS. (n)

row

The row number of the cell whose font is being retrieved. (n)

column

The column number of the cell whose font is being retrieved. (n)

face_name

The returned name of the Windows font. (a)

point_size

The returned size of the Windows font in points. (n)

rendition

The returned rendition. This is zero if no renditions are applied, or one or more of the following options, which can be combined using .BOR. or the vertical bar (|) operator: (n)

D_CELL_BOLD	Bold.
D_CELL_ITALIC	Italic.
D_CELL_UNDER	Underlined.

Discussion

The GetCellFont method retrieves the font and rendition for a specified cell.

Examples

In the following example, the text in the fourth column of the current item is set to underline and bold. This does not affect any other font settings.

```
xcall l_status(listid, D_LAXCTRL, axctrl, D_LCURITM, curr)
xcall ax_call(axctrl, "GetCellFont", curr, 4, fontname, ptsize, style)
xcall ax_call(axctrl, "SetCellFont", curr, 4, fontname, ptsize,
&                (style .bor. D_CELL_BOLD .bor. D_CELL_UNDER))
```

The following example turns off italic for the text in the second column of the current item without affecting any other font settings.

```
xcall l_status(listid, D_LAXCTRL, axctrl, D_LCURITM, curr)
xcall ax_call(axctrl, "GetCellFont", curr, 4, fontname, ptsize, style)
xcall ax_call(axctrl, "SetCellFont", curr, 4, fontname, ptsize,
&                (style .band. (.bnot. D_CELL_ITALIC)))
```

See also

- ▶ [SetCellFont on page 9-39](#).
- ▶ [%AX_CALL](#) in the “Synergy ActiveX API” chapter of the *Synergy DBL Language Reference Manual*.

GetColumnWidth – Retrieve the width of a column

WT			

```
status = %AX_CALL(axlist_id, "GetColumnWidth", column, width)
```

Return value

status

Returns true if successful, otherwise false. **(n)**

Arguments

axlist_id

The ID for the ActiveX Toolkit list. This ID can be retrieved with L_STATUS. **(n)**

column

The column number of the cell whose width is being retrieved. **(n)**

width

The returned width of the column (in pixels). **(n)**

Discussion

The GetColumnWidth method retrieves the width, in pixels, for a specified column.

All cells in the same physical column must have the same width, so *column* is limited to the number of columns across, even for multi-line list items.

See also

- ▶ [SetColumnWidth on page 9-41.](#)
- ▶ [%AX_CALL](#) in the “Synergy ActiveX API” chapter of the *Synergy DBL Language Reference Manual*.

Examples

The following example increases the width of the third column by 7%.

```
xcall l_status(listid, D_LAXCTRL, axctrl) ;Get control ID
xcall ax_call(axctrl, "GetColumnWidth", 3, width)
xcall ax_call(axctrl, "SetColumnWidth", 3, (width * 107)#2)
```

GetFooterCell – Retrieve the text in an individual cell of a footer

WT			

```
status = %AX_CALL(axlist_id, "GetFooterCell", column, text)
```

Return value

status

Returns true if successful, otherwise false. **(n)**

Arguments

axlist_id

The ID for the ActiveX Toolkit list. This ID can be retrieved with L_STATUS. **(n)**

column

The column number of the footer whose text is being retrieved. **(n)**

text

The returned footer text. **(a)**

Discussion

The GetFooterCell method retrieves the text in the footer for a specified column.

See also

- ▶ [SetFooterCell on page 9-42.](#)
- ▶ [%AX_CALL](#) in the “Synergy ActiveX API” chapter of the *Synergy DBL Language Reference Manual*.

Examples

The following increments a right-justified numeric value displayed in the second footer cell.

```
xcall l_status(listid, D_LAXCTRL, axctrl)
xcall ax_call(axctrl, "GetFooterCell", 2, text)
onerror ($ERR_DIGIT) not_a_number
number = text
if (FALSE)
  begin
    not_a_number,
      clear number
    end
offerror
xcall ax_call(axctrl, "SetFooterCell", 2, %string(number+1))
xcall ax_call(axctrl, "SetFooterCellAlignment", 2, 3)
```

GetFooterCellAlignment – Retrieve the alignment of a footer cell

WT			

status = %AX_CALL(*axlist_id*, "GetFooterCellAlignment", *column*, *justification*)

Return value

status

Returns true if successful, otherwise false. **(n)**

Arguments

axlist_id

The ID for the ActiveX Toolkit list. This ID can be retrieved with L_STATUS. **(n)**

column

The column number of the footer whose justification is being retrieved. **(n)**

justification

One of the following: **(n)**

- 1** Left-justified.
- 2** Centered.
- 3** Right-justified.

Discussion

The GetFooterCellAlignment method retrieves the justification of the footer specified by *column*.

See also

- ▶ [SetFooterCellAlignment on page 9-43.](#)
- ▶ [%AX_CALL](#) in the “Synergy ActiveX API” chapter of the *Synergy DBL Language Reference Manual*.

GetFooterCellFont – Retrieve the font of an individual footer row

WT			

```
status = %AX_CALL (axlist_id, "GetFooterCellFont", row, face_name, point_size,
&          rendition)
```

Return value

status

Returns true if successful, otherwise false. **(n)**

Arguments

axlist_id

The ID for the ActiveX Toolkit list. This ID can be retrieved with L_STATUS. **(n)**

row

The row number of the footer whose font is being retrieved. **(n)**

face_name

The returned name of the Windows font. **(a)**

point_size

The returned size of the Windows font in points. **(n)**

rendition

The returned rendition. This is zero if no renditions are applied, or one or more of the following options, which can be combined using .BOR. or the vertical bar (|) operator: **(n)**

D_CELL_BOLD	Bold.
D_CELL_ITALIC	Italic.
D_CELL_UNDER	Underlined.

Discussion

The GetFooterCellFont method retrieves the font and rendition for a specified row of the footer.

See also

- ▶ [SetFooterCellFont](#) on page 9-44.
- ▶ [%AX_CALL](#) in the “Synergy ActiveX API” chapter of the *Synergy DBL Language Reference Manual*.

Examples

The following example sets the font for the third row of footers to bold. This does not affect other font parameters.

```
xcall l_status(listid, D_LAXCTRL, axctrl)
xcall ax_call(axctrl, "GetFooterCellFont", 3, face, ptsize, style)
xcall ax_call(axctrl, "SetFooterCellFont", 3, %atrim(face), ptsize,
&                style .bor. D_CELL_BOLD)
```

GetHeaderCell – Retrieve the text in an individual cell of a header

WT			

```
status = %AX_CALL(axlist_id, "GetHeaderCell", column, text)
```

Return value

status

Returns true if successful, otherwise false. **(n)**

Arguments

axlist_id

The ID for the ActiveX Toolkit list. This ID can be retrieved with L_STATUS. **(n)**

column

The column number of the header whose text is being retrieved. **(n)**

text

The returned header text. **(a)**

Discussion

The GetHeaderCell method retrieves the text in the header for a specified column.

See also

- ▶ [SetHeaderCell](#) on page 9-46.
- ▶ [%AX_CALL](#) in the “Synergy ActiveX API” chapter of the *Synergy DBL Language Reference Manual*.

Examples

The following example appends the string “ - Audited” to the text of the third header cell and ensures that the text is left justified.

```
xcall l_status(listid, D_LAXCTRL, axctrl)
xcall ax_call(axctrl, "GetHeaderCell", 3, text)
text = %atrim(text) + " - Audited"
xcall ax_call(axctrl, "SetHeaderCell", 3, %atrim(text))
xcall ax_call(axctrl, "SetHeaderCellAlignment", 3, 1)
```

GetHeaderCellAlignment – Retrieve the alignment of a header cell

WT			

```
status = %AX_CALL(axlist_id, "GetHeaderCellAlignment", column, justification)
```

Return value

status

Returns true if successful, otherwise false. **(n)**

Arguments

axlist_id

The ID for the ActiveX Toolkit list. This ID can be retrieved with L_STATUS. **(n)**

column

The column number of the header whose justification is being retrieved. **(n)**

justification

One of the following: **(n)**

- 1** Left-justified.
- 2** Centered.
- 3** Right-justified.

Discussion

The GetHeaderCellAlignment method retrieves the justification of the header specified by *column*.

See also

- ▶ [SetHeaderCellAlignment on page 9-47.](#)
- ▶ [%AX_CALL](#) in the “Synergy ActiveX API” chapter of the *Synergy DBL Language Reference Manual*.

GetHeaderCellFont – Retrieve the font of an individual header row

WT			

```
status = %AX_CALL (axlist_id, "GetHeaderCellFont", row, face_name, point_size,
&          rendition)
```

Return value

status

Returns true if successful, otherwise false. **(n)**

Arguments

axlist_id

The ID for the ActiveX Toolkit list. This ID can be retrieved with L_STATUS. **(n)**

row

The row number of the header whose font is being retrieved. **(n)**

face_name

The returned name of the Windows font. **(a)**

point_size

The returned size of the Windows font in points. **(n)**

rendition

The returned rendition. This is zero if no renditions are applied, or one or more of the following options, which can be combined using .BOR. or the vertical bar (|) operator: **(n)**

D_CELL_BOLD	Bold.
D_CELL_ITALIC	Italic.
D_CELL_UNDER	Underlined.

Discussion

The GetHeaderCellFont method retrieves the font and rendition for a specified row of the header.

See also

- ▶ [SetHeaderCellFont](#) on page 9-48.
- ▶ [%AX_CALL](#) in the “Synergy ActiveX API” chapter of the *Synergy DBL Language Reference Manual*.

Examples

The following example sets the font of the second row of headers to Times New Roman, increases the current font size by 1, and leaves the style unchanged.

```
xcall l_status(listid, D_LAXCTRL, axctrl)
xcall ax_call(axctrl, "GetHeaderCellFont", 2, face, ptsize, style)
xcall ax_call(axctrl, "SetHeaderCellFont", 2,
&          "Times New Roman", ptsize+1, style)
```

GridRepaint – Force a full repaint

WT			

```
xcall AX_CALL(axlist_id, "GridRepaint")
```

Return value

status

Always returns zero. (n)

Arguments

axlist_id

The ID for the ActiveX Toolkit list. This ID can be retrieved with L_STATUS. (n)

Discussion

The GridRepaint method forces a full repaint of the ActiveX Toolkit list.

See also

[%AX_CALL](#) in the “Synergy ActiveX API” chapter of the *Synergy DBL Language Reference Manual*.

Examples

The following example forces the list to be repainted.

```
xcall l_status(listid, D_LAXCTRL, axctrl)
xcall ax_call(axctrl, "GridRepaint")
```

SetCellColor – Change the color of an individual cell

WT			

```
status = %AX_CALL (axlist_id, "SetCellColor", row, column, color, background_flag)
```

Return value

status

Returns true if successful, otherwise false. **(n)**

Arguments

axlist_id

The ID for the ActiveX Toolkit list. This ID can be retrieved with L_STATUS. **(n)**

row

The row number of the cell whose color is being modified. **(n)**

column

The column number of the cell whose color is being modified. **(n)**

color

The color (RGB value) to apply to the cell. **(n)**

background_flag

One of the following: **(n)**

0 Change text color.

1 Change background color.

Discussion

The SetCellColor method sets either the text or background color for a specified cell. The specified color overrides any color settings specified by the BackColor, DarkItemColor, or LightItemColor properties, or any field color specified in the window script or by the I_FLDMOD subroutine. See the GetCellColor [Discussion on page 9-23](#) for information on RGB values.

See also

- ▶ [GetCellColor on page 9-23](#).
- ▶ [%AX_CALL](#) in the “Synergy ActiveX API” chapter of the *Synergy DBL Language Reference Manual*.

List Routines

SetCellColor

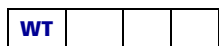
Examples

The following example sets the text color of the third column in the currently selected row to red.

```
xcall l_status(listid, D_LAXCTRL, axctrl, D_LCURITM, curr)
xcall ax_call(axctrl, "SetCellColor", curr, 3, RGB_VALUE(255,0,0), 0)
```

See also the GetCellColor [Examples on page 9-24](#).

SetCellFont – Change the font of an individual cell



```
status = %AX_CALL (axlist_id, "SetCellFont", row, column, face_name, point_size,
&                rendition)
```

Return value

status

Returns true if successful, otherwise false. **(n)**

Arguments

axlist_id

The ID for the ActiveX Toolkit list. This ID can be retrieved with L_STATUS. **(n)**

row

The row number of the cell whose font is being modified. **(n)**

column

The column number of the cell whose font is being modified. **(n)**

face_name

The name of the Windows font. **(a)**

point_size

The size of the Windows font in points. **(n)**

rendition

Zero to turn off all renditions, or one or more of the following options, which can be combined using .BOR. or the vertical bar (|) operator: **(n)**

D_CELL_BOLD	Bold.
D_CELL_ITALIC	Italic.
D_CELL_UNDER	Underlined.

Discussion

The SetCellFont method sets the font and rendition for a specified cell. The specified font overrides any font and point size specified by the FontName and FontSize properties, and any font and renditions specified in the window script. It does not override font and renditions specified by I_FLDMOD.

See also

- ▶ [GetCellFont](#) on page 9-25.
- ▶ [%AX_CALL](#) in the “Synergy ActiveX API” chapter of the *Synergy DBL Language Reference Manual*.

Examples

In the following example, the text in the fourth column of the current item is set to underline and bold. This does not affect any other font settings.

```
xcall l_status(listid, D_LAXCTRL, axctrl, D_LCURITM, curr)
xcall ax_call(axctrl, "GetCellFont", curr, 4, fontname, ptsize, style)
xcall ax_call(axctrl, "SetCellFont", curr, 4, %atrim(fontname), ptsize,
&                (style .bor. D_CELL_BOLD .bor. D_CELL_UNDER))
```

The following example turns off italic for the text in the second column of the current item without affecting any other font settings.

```
xcall l_status(listid, D_LAXCTRL, axctrl, D_LCURITM, curr)
xcall ax_call(axctrl, "GetCellFont", curr, 2, fontname, ptsize, style)
xcall ax_call(axctrl, "SetCellFont", curr, 2, %atrim(fontname), ptsize,
&                (style .band. (.bnot. D_CELL_ITALIC)))
```

SetColumnWidth – Change the width of a column

WT			

```
status = %AX_CALL(axlist_id, "SetColumnWidth", column, width)
```

Return value

status

Returns true if successful, otherwise false. **(n)**

Arguments

axlist_id

The ID for the ActiveX Toolkit list. This ID can be retrieved with L_STATUS. **(n)**

column

The column number of the cell whose width is being modified. **(n)**

width

The new width of the column (in pixels). **(n)**

Discussion

The SetColumnWidth method sets the width, in pixels, for a specified column.

All cells in the same physical column must have the same width, so *column* is limited to the number of columns across, even for lists with multi-line items.

See also

- ▶ [GetColumnWidth on page 9-27.](#)
- ▶ [%AX_CALL](#) in the “Synergy ActiveX API” chapter of the *Synergy DBL Language Reference Manual*.

Examples

The following example increases the width of the third column by 7%.

```
xcall l_status(listid, D_LAXCTRL, axctrl) ;Get control ID
xcall ax_call(axctrl, "GetColumnWidth", 3, width)
xcall ax_call(axctrl, "SetColumnWidth", 3, (width * 107)#2)
```

SetFooterCell – Change the text in an individual cell of a footer

WT			

```
status = %AX_CALL(axlist_id, "SetFooterCell", column, text)
```

Return value

status

Returns true if successful, otherwise false. **(n)**

Arguments

axlist_id

The ID for the ActiveX Toolkit list. This ID can be retrieved with L_STATUS. **(n)**

column

The column number of the footer whose text is being modified. **(n)**

text

The new footer text. **(a)**

Discussion

The SetFooterCell method sets the text in the footer for a specified column.

See also

- ▶ [GetFooterCell on page 9-28.](#)
- ▶ [%AX_CALL](#) in the “Synergy ActiveX API” chapter of the *Synergy DBL Language Reference Manual*.

Examples

See the GetFooterCell [Examples on page 9-28.](#)

SetFooterCellAlignment – Change the alignment of a footer cell

WT			

```
status = %AX_CALL(axlist_id, "SetFooterCellAlignment", column, justification)
```

Return value

status

Returns true if successful, otherwise false. **(n)**

Arguments

axlist_id

The ID for the ActiveX Toolkit list. This ID can be retrieved with L_STATUS. **(n)**

column

The column number of the footer whose justification is being modified. **(n)**

justification

One of the following: **(n)**

- 1** Left-justified.
- 2** Centered.
- 3** Right-justified.

Discussion

The SetFooterCellAlignment method sets the justification of the footer specified by column.

See also

- ▶ [GetFooterCellAlignment](#) on page 9-29.
- ▶ [%AX_CALL](#) in the “Synergy ActiveX API” chapter of the *Synergy DBL Language Reference Manual*.

Examples

See the GetFooterCell [Examples](#) on page 9-28.

SetFooterCellFont – Change the font of an individual footer row

WT			

```
status = %AX_CALL (axlist_id, "SetFooterCellFont", row, face_name, point_size,
&          rendition)
```

Return value

status

Returns true if successful, otherwise false. **(n)**

Arguments

axlist_id

The ID for the ActiveX Toolkit list. This ID can be retrieved with L_STATUS. **(n)**

row

The row number of the footer whose font is being modified. **(n)**

face_name

The name of the Windows font. **(a)**

point_size

The size of the Windows font in points. **(n)**

rendition

Zero to turn off all renditions, or one or more of the following options, which can be combined using .BOR. or the vertical bar (|) operator: **(n)**

D_CELL_BOLD	Bold.
D_CELL_ITALIC	Italic.
D_CELL_UNDER	Underlined.

Discussion

The SetFooterCellFont method sets the font and rendition for a specified row of the footer. The specified font overrides any font and point size specified by the FooterFontName and FooterFontSize properties.

See also

- ▶ [GetFooterCellFont](#) on page 9-30.
- ▶ [%AX_CALL](#) in the “Synergy ActiveX API” chapter of the *Synergy DBL Language Reference Manual*.

Examples

The following example sets the font for the third row of footers to bold. This does not affect other font parameters.

```
xcall l_status(listid, D_LAXCTRL, axctrl)
xcall ax_call(axctrl, "GetFooterCellFont", 3, face, ptsize, style)
xcall ax_call(axctrl, "SetFooterCellFont", 3, %atrim(face), ptsize,
&                style .bor. D_CELL_BOLD)
```

SetHeaderCell – Change the text in an individual cell of a header

WT			

```
status = %AX_CALL(axlist_id, "SetHeaderCell", column, text)
```

Return value

status

Returns true if successful, otherwise false. **(n)**

Arguments

axlist_id

The ID for the ActiveX Toolkit list. This ID can be retrieved with L_STATUS. **(n)**

column

The column number of the header whose text is being modified. **(n)**

text

The new header text. **(a)**

Discussion

The SetHeaderCell method sets the text in the header for a specified column.

See also

- ▶ [GetHeaderCell on page 9-32.](#)
- ▶ [%AX_CALL](#) in the “Synergy ActiveX API” chapter of the *Synergy DBL Language Reference Manual*.

Examples

The following example appends the string “ - Audited” to the text of the third header cell and ensures that the text is left justified.

```
xcall l_status(listid, D_LAXCTRL, axctrl)
xcall ax_call(axctrl, "GetHeaderCell", 3, text)
text = %atrim(text) + " - Audited"
xcall ax_call(axctrl, "SetHeaderCell", 3, %atrim(text))
xcall ax_call(axctrl, "SetHeaderCellAlignment", 3, 1)
```


SetHeaderCellAlignment – Change the alignment of a header cell

WT			

status = %AX_CALL(*axlist_id*, "SetHeaderCellAlignment", *column*, *justification*)

Return value

status

Returns true if successful, otherwise false. **(n)**

Arguments

axlist_id

The ID for the ActiveX Toolkit list. This ID can be retrieved with L_STATUS. **(n)**

column

The column number of the header whose justification is being modified. **(n)**

justification

One of the following: **(n)**

- 1** Left-justified.
- 2** Centered.
- 3** Right-justified.

Discussion

The SetHeaderCellAlignment method sets the justification of the header specified by column.

See also

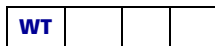
- ▶ [GetHeaderCellAlignment on page 9-33.](#)
- ▶ [%AX_CALL](#) in the “Synergy ActiveX API” chapter of the *Synergy DBL Language Reference Manual*.

Examples

The following example appends the string “ - Audited” to the text of the third header cell and ensures that the text is left justified.

```
xcall l_status(listid, D_LAXCTRL, axctrl)
xcall ax_call(axctrl, "GetHeaderCell", 3, text)
text = %atrim(text) + " - Audited"
xcall ax_call(axctrl, "SetHeaderCell", 3, %atrim(text))
xcall ax_call(axctrl, "SetHeaderCellAlignment", 3, 1)
```

SetHeaderCellFont – Change the font of an individual header row



```
status = %AX_CALL (axlist_id, "SetHeaderCellFont", row, face_name, point_size,
&          rendition)
```

Return value

status

Returns true if successful, otherwise false. **(n)**

Arguments

axlist_id

The ID for the ActiveX Toolkit list. This ID can be retrieved with L_STATUS. **(n)**

row

The row number of the header whose font is being modified. **(n)**

face_name

The name of the Windows font. **(a)**

point_size

The size of the Windows font in points. **(n)**

rendition

Zero to turn off all renditions, or one or more of the following options, which can be combined using .BOR. or the vertical bar (|) operator: **(n)**

D_CELL_BOLD	Bold.
D_CELL_ITALIC	Italic.
D_CELL_UNDER	Underlined.

Discussion

The SetHeaderCellFont method sets the font and rendition for a specified row of the header. The specified font overrides any font and point size specified by the HeaderFontName and HeaderFontSize properties.

See also

- ▶ [GetHeaderCellFont](#) on page 9-34.
- ▶ [%AX_CALL](#) in the “Synergy ActiveX API” chapter of the *Synergy DBL Language Reference Manual*.

Examples

The following example sets the font of the second row of headers to Times New Roman, increases the current font size by 1, and leaves the style unchanged.

```
xcall l_status(listid, D_LAXCTRL, axctrl)
xcall ax_call(axctrl, "GetHeaderCellFont", 2, face, ptsize, style)
face = "Times New Roman"
xcall ax_call(axctrl, "SetHeaderCellFont", 2,
&                %atrim(face), ptsize+1, style)
```

Published Event for ActiveX Toolkit Lists

A published event is a named event that can be signaled from the list. Event names are case sensitive. All events are bound to Synergy subroutines using the AX_BIND subroutine and the list ID retrieved from L_STATUS. The OnOriginChange published event is supported.

OnOriginChange – The origin (upper left cell) has changed

WT			

```
xcall AX_BIND(axlist_id, "OnOriginChange"[, function])
```

Arguments

axlist_id

The ID for the ActiveX Toolkit list. This ID can be retrieved with L_STATUS. **(n)**

function

(optional) A string that contains the name of the function to call when the event occurs. **(a)**

Discussion

The OnOriginChange event is signaled when the origin (upper left cell) of the list is changed. When this happens, the specified Synergy function is called with five arguments passed: the ActiveX control ID, the previous top row, the previous left column, the new top row, and the new left column. See the function prototype example below:

```
function OnOriginChange, ^val, reentrant
    a_ctlid      ,n          ;ActiveX control ID
    a_oldrow     ,n          ;Previous top row
    a_oldcol     ,n          ;Previous left column
    a_newrow     ,n          ;New top row
    a_newcol     ,n          ;New left column
```

If the *function* argument is not passed, the bind to OnOriginChange is cleared.

Function is a function you name and write. The origin of the list can be modified by calling the AX_NEWVAL function and passing the argument you want to modify and its value, either by the new top or the new left. For example, the following modifies the declared **a_newrow** argument to the value 10.

```
xcall ax_newval(a_newrow, 10)
```

Any value returned from *function* is ignored.

See also

[AX_BIND](#) in the “Synergy ActiveX API” chapter of the *Synergy DBL Language Reference Manual* for information on AX_BIND.

Examples

The following example prevents the first column in the list from being scrolled into view.

```

xcall l_status(listid, D_LAXCTRL, axctrl)
xcall ax_set(axctrl, "OriginColumn", 2)
xcall ax_bind(axctrl, "OnOriginChange", "my_origin_change")
.
.
.
function my_origin_change ,^val, reentrant
;
; Description:  Origin change event routine to prevent visibility
;               of column 1.
; Arguments:
;
;       a_ctlid      ,n      ;ID of the control
;       a_prevrow    ,n      ;Previous origin row
;       a_prevcol    ,n      ;Previous origin column
;       a_newrow     ,n      ;New origin row
;       a_newcol     ,n      ;New origin column
;
; Return value:      TRUE
;
proc
    if (a_newcol .le. 1)
        xcall ax_newval(a_newcol, 2)
    freturn TRUE
endfunction

```

The List Subroutines and Functions

The following subroutines and functions enable you to create, process, and maintain lists. For general information on these tasks, see [“Defining Lists” on page 9-5](#) and [“Processing and Maintaining Lists” on page 9-10](#). For specific on defining list behavior, see [“Defining behavior for lists” on page 9-8](#).

`L_BORDER` – Add or remove the border and drag bar on a list

`L_BUTTON` – Add, delete, or modify a button on a list

`L_BUTTONSET` – Define the placement of a set of buttons on a list

`%L_BUTTONSTATE` – Enable or disable a button on a list

`L_CHR` – Perform single-keystroke input to a list

`L_CLASS` – Create a list class at runtime

`%L_CLASSINFO` – Return information about a list class

`L_CREATE` – Create a list

`L_CTRCONTEXT` – Establish focus within a child list window

`L_DATA` – Manipulate the data in a list

`L_DELETE` – Delete a list

`L_DISABLE` – Disable a list

`L_ENABLE` – Enable a list

`L_FINDSPEC` – Specify parameters for a find operation

`%L_FINDWND` – Construct a find window for `L_SELECT`

`L_GLOBAL` – Promote a list to global status

`%L_ICON` – Set a list icon

`L_INPFLD` – Perform single-field input within a list

`L_INPUT` – Perform input to a set within a list

`L_METHOD` – Register a list method

`L_NEXT` – Set selected item by number

`L_PLACE` – Place a list

`L_PROCESS` – Process a list

`L_QUEUE` – Queue commands for a list

`L_REMOVE` – Remove a list

L_RESIZE – Resize a list

L_RESTART – Restart a list

L_RETURNKEY – Set the return state of privatized keys

L_SECT – Place text in a list header, footer, or title

L_SECTDRAW – Draw a line in a list header or footer

L_SELECT – Do list selection processing

%L_SELSTYLE – Retrieve/change the selection style for a list

L_STATE – Modify a list's state

L_STATUS – Retrieve the status of a list

%L_USER – Set/retrieve user data associated with a list

L_VIEW – Shift the horizontal display area of a list

L_BORDER – Add or remove the border and drag bar on a list



```
xcall L_BORDER (list_id[, option, ...])
```

Arguments

list_id

The ID of the list to modify. (**n**)

option

(optional) One or more of the following list options: (**n**)

D_ON	Turn border on.
D_OFF	Turn border off.
D_DRAGON	Turn drag bar on.
D_DRAGOFF	Turn drag bar off.

Discussion

The L_BORDER subroutine allows the border and drag bar to be turned on or off for an ActiveX or Toolkit list.

List_id must contain the ID of a valid list, otherwise a fatal error will occur.

If a list's border is off, its drag bar (caption bar) will not be visible, even if it is turned on.

D_DRAGON and D_DRAGOFF have no visible effect on UNIX or OpenVMS.

If multiple, conflicting options are specified, the option specified last has precedence.

See also

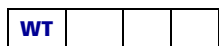
[.LISTCLASS on page 2-82](#) for defining a list class.

Examples

The following example turns off the list's border and drag bar.

```
xcall l_border(cust_lst, D_OFF, D_DRAGOFF)
```


L_BUTTON – Add, delete, or modify a button on a list



`xcall L_BUTTON (list_id, [subfunction,] args)`

Arguments

list_id

The ID of the list that contains or will contain the button. **(n)**

subfunction

(optional) One of the following: **(n)**

DSB_ADD Add a button to the list. (default)

DSB_DELETE Delete a button from the list.

DSB_MODIFY Modify a button on the list.

args

One or more arguments, depending on *subfunction*.

Discussion

The L_BUTTON subroutine creates, deletes, or modifies a Windows-style button on a list. See the individual subfunctions for details.

On UNIX and OpenVMS, this routine is ignored.

See also

[“Using the Button Subroutines” on page 5-2](#) for general information on buttons and button sets on windows, tab sets, and lists.

DSB_ADD

```
xcall L_BUTTON (list_id, [DSB_ADD,] name, [type], [type_data], [method], [elb],  
&           [select_char], [position][, tooltip])
```

Arguments

list_id

The ID of the list you want to add the button to. **(n)**

name

The name of the button to add (a maximum of 30 characters). **(a)**

type

(optional) The button face type: **(n)**

DSB_TEXT Text. (default)

DSB_BITMAP Graphic.

type_data

(optional) Text for the button face if *type* is DSB_TEXT. The filename for a graphic if *type* is DSB_BITMAP. **(a)**

method

(optional) The name of the method subroutine to invoke. **(a)**

elb

(optional) The name of the executable library (**.elb**) that contains *method*. **(a)**

select_char

(optional) The quick-select character. **(a)**

position

(optional) The position the button will be inserted in the button set. **(a or n)**

tooltip

(optional) The text of a ToolTip to be displayed when the mouse pointer rests over the button. **(a)**

Discussion

The DSB_ADD subfunction of L_BUTTON enables you to add a button to a list. If there are no existing buttons, a default button set is created. By default, button sets are placed at the bottom of the list, are left-justified, and can have as many buttons on a row as will fit before the row wraps. You can change these defaults with L_BUTTONSET.

Buttons are *not* included in the list's tabbing order. In order to select a button, it must either be clicked or invoked using its quick-select character.

For information on the arguments for this subroutine, see the [DSB_ADD Discussion on page 5-6](#). All the comments in that section (which discuss arguments for windows and tab sets), apply to lists as well.

Examples

The following example adds an OK button named "I_OK." When the button is pressed, standard termination processing occurs. The OK button has the quick-select character "K", so when ALT+K is pressed, a click of the OK button is simulated. If there are other buttons on the list, this button will be added to the end of the button set. Note that we didn't include "DSB_ADD". You can omit DSB_ADD when you add a button because DSB_ADD is the default.

```
xcall l_button(listid, "I_OK", DSB_TEXT, "OK",,, "K")
```

The next example creates a button named "lookup" with a graphic, **lookup.gif**, on its face. Clicking this button will invoke the **lookup** subroutine located in the **button.elb** library. This button will be added to the beginning of the button set and will have the ToolTip "Enter a customer number."

```
xcall l_button(listid, DSB_ADD, "lookup", DSB_BITMAP, "lookup.gif",  
& "lookup", "button.elb", 1, "Enter a customer number")
```

List Routines

L_BUTTON

DSB_DELETE

```
xcall L_BUTTON(list_id, DSB_DELETE, name)
```

Arguments

list_id

The ID of the list that contains the button you want to delete. **(n)**

name

The name of the button to delete. **(a)**

Discussion

The DSB_DELETE subfunction of L_BUTTON deletes a button from a list. Remaining buttons are repositioned as if the deleted button had never existed. For example, if the button is the second button of three in a button set, the third button will be moved to the second button position.

Note that *name* is not case sensitive and that if no button with that name exists, Toolkit generates a fatal error.

Examples

The following example deletes a button named **lookup**.

```
xcall l_button(wndid, DSB_DELETE, "lookup")
```

DSB_MODIFY

```
xcall L_BUTTON(list_id, DSB_MODIFY, name, [type], [type_data], [method], [elb],
&           [select_char], [position][, tooltip])
```

Arguments

list_id

The ID of the list that contains the button you want to modify. **(n)**

name

The name of the button to modify. **(a)**

type

(optional) The new type of button face. (If you don't pass this argument, the previous value is the default.) **(n)**

DSB_TEXT Text.

DSB_BITMAP Graphic.

type_data

(optional) New text for the button if *type* is DSB_TEXT. If *type* is DSB_BITMAP, this is the filename for the graphic. **(a)**

method

(optional) The name of a new method subroutine to invoke. **(a)**

elb

(optional) The name of the library to find *method* in. **(a)**

select_char

(optional) A new quick-select character. **(a)**

position

(optional) A position you want the button moved to in the button set. **(a or n)**

tooltip

(optional) The text of a ToolTip to be displayed when the mouse pointer rests over the button. **(a)**

Discussion

The DSB_MODIFY subfunction of L_BUTTON modifies a button in a list.

For information on the arguments for this subroutine, see the DSB_MODIFY [Discussion on page 5-11](#). All the comments in that section (which apply to windows and tab sets), apply to lists as well.

Examples

The following example specifies that the button face for the **lookup** button will display a new graphic named **lookup2.jpg**. (We're assuming that the button previously had a graphic displayed on the button face. If it instead had text on the button face, this example would need to pass **DSB_BITMAP** as the *type* argument to prevent "lookup2.jpg" from being interpreted as a text string rather than a filename.)

```
xcall l_button(listid, DSB_MODIFY, "lookup",, "lookup2.jpg")
```

The following example changes the button face for the **lookup** button to the text "Search All".

```
xcall l_button(listid, DSB_MODIFY, "lookup", DSB_TEXT, "Search All")
```

The following example changes the method name and the ELB for the **lookup** button.

```
xcall l_button(listid, DSB_MODIFY, "lookup",,, "search_all",
&                "button2.elb")
```

The following example changes the button so that it no longer calls a method when clicked. Instead, Toolkit will use the first ten characters of the button name to signal a menu entry.

```
xcall l_button(listid, DSB_MODIFY, "lookup",,, "")
```

The following example changes the button so that Toolkit no longer looks in a specific ELB for the method.

```
xcall l_button(listid, DSB_MODIFY, "lookup",,,, "")
```

The following example changes the quick-select character for the button to "S". When the user presses ALT+S, the button will be activated.

```
xcall l_button(listid, DSB_MODIFY, "lookup",,,,, "S")
```

The following example moves the **lookup** button to the beginning of the button set. (This will be the far left of top depending the orientation of the button set.)

```
xcall l_button(listid, DSB_MODIFY, "lookup",,,,,, 1)
```

The following example moves the **lookup** button to the position that precedes the **I_CANCEL** button. Depending on the button set orientation, it will be above or to the left of the **I_CANCEL** button.

```
xcall l_button(listid, DSB_MODIFY, "lookup",,,,,, "I_CANCEL")
```

The following example moves the **lookup** button to the third position in the button set. Depending on the orientation of the button set, it will be either the third button from the top or the third button from the left.

```
xcall l_button(listid, DSB_MODIFY, "lookup",,,,,, 3)
```

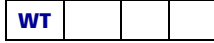
The following example moves the **lookup** button to the end of the button set. Depending on the orientation of the button set, it will either be at the far right or the bottom of the button set.

```
xcall l_button(listid, DSB_MODIFY, "lookup",,,,,, "")
```

The following example sets the ToolTip text to “Enter a vendor number”:

```
xcall l_button(listid, DSB_MODIFY, "lookup",,,,,,  
& "Enter a vendor number")
```

L_BUTTONSET – Define the placement of a set of buttons on a list



```
xcall L_BUTTONSET (list_id, [placement], [per_row_col], [justification][, button_name])
```

Arguments

list_id

The ID of the list. **(n)**

placement

(optional) One of the following: **(n)**

DSB_BOTTOM The set of buttons will be placed horizontally along the bottom of the container with a maximum of *per_row_col* buttons per row. (default)

DSB_RIGHT The set of buttons will be placed vertically along the right side of the container with a maximum of *per_row_col* buttons per column.

per_row_col

(optional) The maximum number of buttons to place on a row if DSB_BOTTOM is specified, or on a column if DSB_RIGHT is specified. The default behavior is to display as many buttons as will fit and wrap to the next row or column. **(n)**

justification

(optional) One of the following: **(n)**

DSB_BEGIN The buttons will be left-justified if DSB_BOTTOM is specified, or positioned towards the top if DSB_RIGHT is specified. (default)

DSB_CENTER The buttons will be centered along the designated side.

DSB_END The buttons will be right-justified if DSB_BOTTOM is specified, or positioned towards the bottom if DSB_RIGHT is specified.

button_name

(optional) The name of the default button. **(a)**

Discussion

L_BUTTONSET defines the placement of a set of buttons for a list. Every button is contained within a set and only one button set is allowed per list; therefore, calling L_BUTTONSET will affect all buttons for the list.

The background rendition of the button area is defined by the background rendition of the input window at the time that L_CREATE is called.

This routine is ignored on UNIX and OpenVMS.

See also

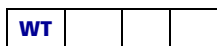
[“Using the Button Subroutines” on page 5-2](#) for general information on buttons and button sets on windows, tab sets, and lists.

Examples

The following example places the list's buttons on the right side, centered, with three buttons per column (at most).

```
xcall l_buttonset(list_id, DSB_RIGHT, 3, DSB_CENTER)
```

%L_BUTTONSTATE – Enable or disable a button on a list



```
prev_state = %L_BUTTONSTATE (list_id, name[, new_state])
```

or

```
xcall L_BUTTONSTATE (list_id, name[, new_state])
```

Return value

prev_state

On Windows, returns the previous state of the button: (^VAL)

D_DISABLED Disabled.

D_ENABLED Enabled.

D_INPUTONLY Enabled only within input.

On UNIX, returns the previously specified state (though it has no meaning on UNIX platforms). On OpenVMS, returns true.

Arguments

list_id

The ID of the list. (**n**)

name

The name of the button. (**a**)

new_state

(optional) One of the following: (**n**)

D_DISABLED Disabled.

D_ENABLED Enabled.

D_INPUTONLY Enabled only within input.

Discussion

L_BUTTONSTATE enables or disables a button within a list.

If *new_state* is not specified, the previous state remains unchanged.

You can XCALL this function if you don't need to know the previous state of the button.



If a button is enabled and the user clicks on it while not in a Toolkit input processing routine, one of two things occurs. Either the click is ignored, or the event will be signaled at the invocation of the next Toolkit input processing routine, depending upon timing. To ensure that the button cannot be pressed when input is not being done to the list, set *new_state* to D_INPUTONLY.

On UNIX and OpenVMS, this routine is ignored.

See also

[“Using the Button Subroutines” on page 5-2](#) for general information on buttons and button sets on windows, tab sets, and lists.

Examples

The following example enables the I_OK button whenever we're performing input on the list.

```
xcall l_buttonstate(listid, I_OK, D_INPUTONLY)
```

L_CHR – Perform single-keystroke input to a list



```
xcall L_CHR (character, list_id, [data], [disabled], [row], [column], [help_id], [wait_time]
&          [, a_methoddata1, ..., a_methoddata20])
```

Arguments

character

Returned with the character typed. **(a)**

list_id

The ID of the list. **(n)**

data

(optional) The non-window data for the current item. **(a)**

disabled

(optional) The disabled item status. **(n)**

row

(optional) The row within the input window (within an item). **(n)**

column

(optional) The column within an item. **(n)**

help_id

(optional) The help identifier. **(a)**

wait_time

(optional) The time-out limit for I/O processing to be completed before returning to the calling routine. **(n)**

D_FOREVER or **-1** Never time out.

D_GLOBAL or **-2** Use the global value (see [g_wait_time](#) on page 18-6). (default)

D_IMMEDIATE or **0** Time out immediately.

n Wait up to *n* seconds (where *n* is a positive value) for I/O processing to be complete.

a_methoddata1 through *a_methoddata20*

(optional) Up to 20 additional data arguments. **(any)**

Discussion

This subroutine performs single-keystroke input on a list. It performs essentially the same function in a list as U_CHR does in a window.

This subroutine does not return until a character is typed by the user or a menu entry (other than those listed below for Windows) is selected.

On Windows, L_CHR recognizes the following reserved menu entries. Note that E_PASTE pastes a single character, and the others do nothing when L_CHR is being processed.

E_CLEAR
E_COPY
E_CUT
E_MARK
E_PASTE

Whether or not the *data* argument is passed must correspond to whether it was passed in the L_CREATE subroutine when the list was created. *Data* is required here if it was also passed to L_CREATE. If *data* is passed here but was *not* passed to L_CREATE, a fatal error occurs.

Disabled is a true/false flag that indicates whether the current item is disabled. Whether or not the *disabled* argument is passed must correspond to whether it was passed to other list routines such as L_DATA, L_INPFLD, L_INPUT, L_PROCESS, and L_SELECT. See [L_PROCESS on page 9-111](#) for information about the disabled item status.

If the operation times out, **g_select** is set to true and **g_entnam** set to the entry name specified in **g_time_entry**, which is defined in **tkctl.def**. See [g_time_entry on page 18-6](#).

A_methoddata1 through *a_methoddata20* are up to 20 additional optional arguments that can be passed to L_CHR. L_CHR, in turn, passes them directly to each field's methods. This allows you to communicate additional information to your input field methods.

On Windows, the user can change the current item in the list with the mouse or the PAGE UP, PAGE DOWN, HOME, END, and arrow keys before this subroutine returns. Therefore, if you want any special processing to be done when arriving at or leaving a list item, you will have to write and use an arrive or leave method. (In Windows environments, these keys are “privatized” and unavailable to calling routines unless you use L_RETURNKEY to return the key out of the list. “Privatized” means that they are reserved for the private use of the graphical control, which in this case would be the list.)

If you are using L_SELECT you do not need to call this subroutine because L_SELECT handles list input.

See also

- ▶ [U_CHR on page 16-16](#) for more information on single-keystroke input processing.
- ▶ [L_RETURNKEY on page 9-123](#) for more information on returning keys out of the list.

L_CLASS – Create a list class at runtime



```
xcall L_CLASS (class_id, class_name, [placemt_row], [placemt_col], [num_items],
&          [num_headers], [header_static], [num_footers], [footer_static], [no_null], [snapshot],
&          [load_method], [options], [error], [arrive_method], [leave_method]
&          [, dblclick_method])
```

Arguments

class_id

The returned class ID. **(n)**

class_name

The name of the class to create (a maximum of 15 characters). **(a)**

placemt_row

(optional) The placement row for the list. **(n)**

placemt_col

(optional) The placement column for the list. **(n)**

num_items

(optional) The number of items to be visible at one time. **(n)**

num_headers

(optional) The number of header lines to reserve. **(n)**

header_static

(optional) The true/false flag that indicates whether the header is to remain static. **(n)**

num_footers

(optional) The number of footer lines to reserve. **(n)**

footer_static

(optional) The true/false flag that indicates whether the footer is to remain static. **(n)**

no_null

(optional) The true/false flag that indicates whether null lists are prohibited. **(n)**

snapshot

(optional) The true/false flag that indicates whether to take input set information snapshots. **(n)**

load_method

(optional) The name of an LLOAD_METHOD subroutine you have written. (a)

options

(optional) One or more of the *option_string* options for .LISTCLASS. See [.LISTCLASS on page 2-82](#). (a)

error

(optional) The returned Toolkit error number, or 0 if the list class is successfully created. (n)

arrive_method

(optional) The name of an LARRIVE_METHOD subroutine you have written. (a)

leave_method

(optional) The name of an LLEAVE_METHOD subroutine you have written. (a)

dblclick_method

(optional) The name of an LDBLCLICK_METHOD subroutine you have written. (a)

Discussion

The L_CLASS subroutine defines a list class at runtime, as opposed to defining one through the .LISTCLASS script command. The class name is then used when creating a list with L_CREATE.

The returned *class_id* is not needed for subsequent list creation or processing (since L_CREATE identifies the class by name), but you can pass the ID to the U_SAVE subroutine to store the list class in a window library.

Class_name must be unique within the set of windows.

If *placemt_row* and *placemt_col* are passed, but the list is not placed when created, these coordinates will be retained and used when the list is placed.

For more information about the *snapshot* argument, see the .LISTCLASS [Discussion on page 2-85](#).

If passed, *load_method* is the name of an LLOAD_METHOD subroutine that you have written. L_PROCESS or L_SELECT will call this subroutine to load new items into the list. (See [LLOAD_METHOD on page 9-154](#).) Using a load method avoids the necessity of satisfying load requests (D_LLOADBOT, D_LLOADTOP) whenever L_PROCESS is called, and also encapsulates your item-loading code.

If any method subroutines are specified, they must be available. In other words, they must either be contained within the current program image, or in an executable library (ELB) that was either linked to or opened with OPENELB. The default behavior, if no *load_method* is specified for the list, is for L_PROCESS and L_SELECT to return to the caller when new items are needed.

If *error* is passed, it is returned as 0 if the list class is successfully created. If the list class cannot be created, *error* is returned with the Toolkit error number. If *error* is not passed, all errors are fatal.

The following are the errors that can be returned in *error*. You can use %U_GETWNDERR to get the text of *error* (see [%U_GETWNDERR on page 16-69](#)).

- ▶ 519, DTK_DUPCLASS, “List class *class* already defined”—A list class with the name passed in *class_name* already exists.
- ▶ 520, DTK_CLSCREAT, “Cannot create list class *class*”—The list class could not be created (e.g., because the system ran out of available resources).

WIN

When using the L_INPUT subroutine, be aware that the user can change the current list item without control returning to your program first. This means that the state of the input fields and the input set context can be propagated across list items. For example, if your program uses L_INIT to initialize fields in the input set, and the user moves to a different item in the list, the fields are still considered empty. This results in a different redisplay of the data, and validation occurs unexpectedly. There are two ways to solve this problem. Use the INPUT_SNAPSHOT option to preserve input context and field states. If, however, you don’t want the context preserved for each item, you can use a list item arrive method to either initialize or display the data in the associated input window (depending on the content of the list item), and then use L_NEXT to establish the correct context.

In Windows environments, the current list item can be interactively changed within a call to a Toolkit subroutine (such as L_SELECT or L_INPUT). To enable you to support change “events,” you can pass the *arrive_method* and *leave_method* arguments so the specified subroutines can be called to perform any additional processing required. These method subroutines are also supported in UNIX and OpenVMS environments.

Dblclick_method specifies the name of a double-click subroutine that you have written. If specified, the list processor will call the double-click method subroutine whenever the user double-clicks on an item within the list. You can overload this method at runtime using the L_METHOD subroutine. (See [L_METHOD on page 9-105](#) for more information.) If *dblclick_method* is not specified and the user double-clicks on the current item, the list processor will assume that the item is being selected.

See also

- ▶ [LLOAD_METHOD on page 9-154](#) for more information on writing your load method.
- ▶ [LARRIVE_METHOD/LLEAVE_METHOD on page 9-148](#) for more information on writing your arrive and leave methods.
- ▶ [LDBLCLICK_METHOD on page 9-151](#) for more information on writing your double-click method.
- ▶ [.LISTCLASS on page 2-82](#) for more information on the usage, meaning, and defaults of L_CLASS’s optional arguments. (All the optional arguments correspond to qualifiers of the .LISTCLASS script command.)

Examples

The following example creates a list class named “lookup,” whose placement will be determined when L_CREATE is called. The list will have one header and one footer line, will display 10 items at a time, and will invoke the “lookup_load_method” subroutine when a new item is needed for the list.

```
xcall l_class(lstid, "lookup",,,10,1,,1,,, "lookup_load_method",,error)
```

%L_CLASSINFO – Return information about a list class

WT	WN	U	V

result = %L_CLASSINFO (*listclass_id*, *info_type*)

or

xcall L_CLASSINFO (*listclass_id*, *info_type*)

Return value

result

A value that is specific to *info_type*. (^VAL)

Arguments

listclass_id

The ID of a list class window. (n)

info_type

One of the following to indicate the type of information to be returned (defined in **tools.def**): (n)

D_LNAME[, *name*]

Returns the trimmed length of the list class name in *result* and the name of the list class in *name*. (a)

D_LPROW

Returns the placement row specified for the list class. Returns zero if no placement row was specified.

D_LPCOL

Returns the placement column specified for the list class. Returns zero if no placement column was specified.

D_LHEIGHT

Returns the list class setting for the number of items to be displayed at one time. Returns zero if this was not specified.

D_LHDLNS

Returns the number of header lines specified for the list class. Returns zero if the number of headers was not specified.

D_LFTRLNS

Returns the number of footer lines specified for the list class. Returns zero if the number of footers was not specified.

D_LHDRSTATIC

Returns true if headers for the list class are set to static. Otherwise, returns false.

D_LFTRSTATIC

Returns true if footers for the list class are set to static. Otherwise, returns false.

D_LNONULL

Returns true if the list class is set to prevent an empty list. Otherwise, returns false.

D_LSNAPSHOT

Returns true if the list class is set to maintain input window control information for list items. Otherwise, returns false.

D_LMETHOD, *methodtype*[, *method*]

Returns information about the method specified by *methodtype*. The trimmed length of the method name is returned in *result*. Zero is returned if no method was specified for the list class when it was created.

Optionally, the method name or address can be returned in *method*. If method is passed as **a**, the name of the method is returned; if it is passed as **n**, the address is returned. (If *method* is **n** and the method is not available—e.g., the routine doesn't exist or the ELB hasn't been opened yet—Toolkit will generate a trappable runtime error.) (**a** or **n**)

methodtype is one of the following (defined in **tools.def**): (**n**)

D_LLOAD	Load method.
D_LARRIVE	Arrive method for list items.
D_LLEAVE	Leave method for list items.
D_LDBLCLK	Double-click method for list items.

D_LOPTIONS[, *options*]

Returns information about the options string defined for the list class. The trimmed length of the string is returned in *result* (or zero if no option string was defined for the list class). The options string itself can optionally be returned in *options*. If no options string was defined for the list class, *options* is cleared to all spaces. (**a**)

Discussion

The %L_CLASSINFO function returns information about a list class (a window created by the .LISTCLASS script command or the L_CLASS routine).

To return most of the most of the *info_type* options, you must call %L_CLASSINFO as a function because information is returned in the return value (*result*). A few *info_type* options (such as D_LNAME and D_LMETHOD) include arguments that return additional information. So, for example, if you call %L_CLASSINFO as a function and pass D_LNAME, it will return the name of the list class in *name* and the length of the list class name in *result*. However, if you make this same call but as a subroutine, it will return only the name of the list class in *name*. Calling D_LPROW, for example, as a subroutine is pointless because nothing will be returned.

See also

[.LISTCLASS on page 2-82](#) and [L_CLASS on page 9-68](#) for details on information returned by this function.

Examples

The following example returns the length of the list class's name in **length**, and it returns the name of the list class in **lc_name**:

```
length = %L_CLASSINFO(lc_id, D_LNAME, lc_name)
```

The next example returns the length of the name of the load method for the list class, and it returns the method name in **meth_name**:

```
length = %L_CLASSINFO(lc_id, D_LMETHOD, D_LLOAD, meth_name)
```

L_CREATE – Create a list



```
xcall L_CREATE (list_id, window_id, [data], [channel], [class], [row], [column],  
&             [D_NOPLC], [disabled], [initial], [last][, error])
```

Arguments

list_id

Returned with the ID of the list created. **(n)**

window_id

The ID of the input window to associate with the list. **(n)**

data

(optional) The initial contents of the associated non-window data to store or retrieve for each list item. See the [Discussion](#) below for more information. **(a)**

channel

(optional) The window library channel to load the list class from if it's not already loaded. **(n)**

class

(optional) The name of the list class. **(a)**

row

(optional) The row position of the list. **(n)**

column

(optional) The column position of the list. **(n)**

D_NOPLC

(optional) The no-placement flag; if passed, the list is not placed. **(n)**

disabled

(optional) The true/false flag that indicates whether the list is disabled. **(n)**

initial

(optional) The item number of the initial item to load. **(n)**

last

(optional) The estimated number of items that will be loaded. **(n)**

error

(optional) The returned Toolkit error number, or 0 if the list is successfully created. **(n)**

Discussion

The `L_CREATE` subroutine creates a new list, associating it with the specified input window. Only one list can be associated with the input window (*window_id*).

Since UI Toolkit appends the “_LST” to the end of the input window name, the maximum size of this name is 11.

The title of the list, if any, and its position (if not passed in the *row* and *column* arguments, nor specified in the list class) are inherited from the input window. If the input window has a border, the list also inherits that. (When the list is deleted, the input window’s border is reset to its state prior to list creation.)

You should never explicitly remove the input window associated with a list, because doing so will corrupt the placement order.

Non-window *data* is additional information that you don’t want displayed, but that you want to be maintained with the list. There are two pieces of data associated with every item in a list: an image of the input window’s display (sometimes referred to as “window data,” and the contents of the “a_itemdata” record (often referred to as “non-window data.”) The latter can be confusing, because it almost always includes a storage representation of the former in order to be useful. Non-window data is not required at all, but lists are most often useful if the data for each item can be stored or retrieved in a record form (as opposed to merely a display form).

The data record passed to `L_CREATE` establishes the default value for non-window data for each item. This is the value that is assigned to the record before the load method is called. It is also assigned to any new item created with `D_LAPPEND` or `D_LINSERT`. The same thing happens for window data; in this case, the image of the input window when `L_CREATE` was called is used.

Both window and non-window data can

- ▶ be modified by the load method when loading an item.
- ▶ be modified by passing a new value into `L_PROCESS` or `L_SELECT` for the current item, as long as the request flag is not `D_LRESTORE` (or `D_LCANCEL`, a synonym).
- ▶ be modified by calling `L_DATA` with `D_LWRITE`.
- ▶ be modified in the list item arrive, leave, or double-click method.
- ▶ be searched with a `D_LFIND` request in the `L_PROCESS` subroutine—`L_FINDSPEC` specifies which will be searched.

The “initial data” is stored only once. When new items are created, they are initially set to this value. It is expected that the item will be modified by the load method or subsequent processing. It is really only intended to ensure that the initial state of the data is valid. For instance, you might want integer fields in the record initialized to integer 0, as opposed to blanks.

To change the data for each item in the list, you would need to use `L_DATA` or `L_PROCESS` to access each item, or you would need to restart the list (`L_RESTART`) and provide the modified data in the load method.

If *class* is passed and it is loaded from *channel*, it will remain loaded when this subroutine returns. If *channel* is not passed and the class specified in *class* is not already loaded, an error will occur. (L_CREATE will *not* use **g_utilib** by default.) If *class* is not passed or if it is blank, the default class will be used. The default list class has the following characteristics:

- ▶ The row and column placement of the list is inherited from the input window.
- ▶ The number of items visible at once is the maximum number that will fit without causing the list to occlude the information bar.
- ▶ The list is allowed to be null.
- ▶ The list contains vertical (and horizontal, if appropriate) scroll bars. (On Windows, these scroll bars take up slightly more screen area.)
- ▶ The input window set context and field display is not maintained for each list item.
- ▶ No methods are specified for the class.

You can override the row and column placement, whether from the specified class or the default class, by specifying *row* and *column*. You can override the default placement by passing a non-zero *no_placement* value. The coordinates that you set when *no_placement* is passed are now retained and used when the list is placed. You can also create the list in a disabled state by passing a non-zero *disabled* value.

By default, lists are loaded by the L_PROCESS subroutine beginning with the first, or topmost, item displayed, and proceeding down the list either until no more items exist or until the display area of the list is full. When more display items are required (for example, when the user moves down from the last item in the list), additional items are requested from the calling routine (or are obtained by calling the specified LLOAD_METHOD) until no more are available. Because each new requested item is loaded at the bottom of the known list and follows the last item requested, we refer to this method as “loading at the bottom of the list, forward.”

Sometimes, however, you’ll want to load the last window of items first, as for a transaction entry subroutine that displays a history of previous transactions but allows new transactions at the bottom. If several previous transactions exist, a significant pause would occur if the first set of items had to be loaded first, followed by all the other items as the list was traversed to get to the end.

An alternative method of loading items begins with the last item in the list, loads items in reverse order until the display is full, and then requests additional items as needed as the user moves up through the list. Because each new requested item is loaded at the top of the known list and precedes the last item requested, we refer to this method as “loading at the top of the list, backward.”

A third method of loading combines the previous two. This method (sometimes known as two-way or bi-directional loading) begins anywhere in the list, loads the first window of items in a forward direction, and requests additional items as needed at either end of the list. Items requested at the top of the known list are expected to be retrieved backwards, while items at the bottom of the list are

retrieved forwards. You might use this combination method for lists with a known starting point that is preceded and followed by a potentially large number of items (for example, a “search” function in a transaction history file).

The *initial* argument determines how list items will be loaded into the list on subsequent calls to `L_PROCESS`:

- ▶ If *initial* has a value of 1, existing items will only be loaded at the bottom of the list, forward.
- ▶ If *initial* is greater than 1 and equal to *last*, existing items will only be loaded at the top of the list, backward.
- ▶ If *initial* is greater than 1 but less than *last*, existing items may be loaded at either end, but loading will begin at the bottom, forward.
- ▶ If *initial* is greater than 1 but inaccurate, the list allows items to be loaded before the logical item 1. On Windows, you should never allow a “Page Up” operation to load an entire page previous to logical item 1.

If *initial* is not passed, the default initial item is 1.

If *last* is not passed or if it has a value of zero, the calling routine is unable to determine the number of list items. This number will be determined internally by UI Toolkit. In such a case, Toolkit assumes that *last* is greater than *initial*.

If *error* is passed, it is returned as 0 if the list is successfully created. If the list cannot be created, *error* is returned with the number for the Toolkit error. If *error* is not passed, all errors are fatal.

Error can be one of the Synergy runtime errors, Synergy window errors, or Toolkit errors listed below. You can use `%U_GETWNDERR` to retrieve error text. (See [%U_GETWNDERR on page 16-69](#).)

- ▶ Synergy runtime errors are defined in `rdeferr.h` and documented in [“Runtime error messages”](#) in the “Error Messages” chapter of *Synergy Tools*.
 - ▶ 421, `E_AXERR`, “Error while processing an ActiveX control”—An ActiveX error occurred as the ActiveX list was loaded. This is often caused by dependencies on a missing file.
 - ▶ 422, `E_AXNOLOAD`, “Could not load ActiveX control”—The runtime was not able to load the ActiveX list because the ActiveX list control was not registered and could not be automatically registered. This is often caused by a missing file.
- ▶ Synergy window errors are documented in [“Window error messages”](#) in the “Error Messages” chapter of *Synergy Tools*:
 - ▶ 4, `WE_OUTOFMEM`, “Not enough memory (needed *n*, had *n*)”—There isn’t enough memory to create the supporting windows for the list.
 - ▶ 9, `WE_DUPWNDNAME`, “Window name already defined”—A window already exists with the name of one of the generated supporting windows (which are `inputWindowName_LST` on all platforms and on UNIX and OpenVMS, `inputWindowName_HDR` for headers and `inputWindowName_FTR` for footers).

- ▶ 10, WE_TOOMANYWND, “Too many windows (max of 1024)”—The supporting windows created for the list exceed the number allowed by Toolkit, which is 1024.
- ▶ 18, WE_OUTOFRES, “Windows resource error”—There aren’t enough system resources to create the list.
- ▶ UI Toolkit errors:
 - ▶ 521, DTK_NOCLASS, “List class *class* not found”—The name passed as *class* is not a valid list class.
 - ▶ 522, DTK_NOCLSLIB, “List class *class* not loaded - no library supplied”—The number passed as *channel* is not the channel number for the window library that contains the list class *class*.
 - ▶ 523, DTK_NOCLSWND, “Id *n* is not a list class”—The name passed as *class* is not a valid list class.
 - ▶ 530, DTK_INPINUSE, “Input window *n* is already associated with a list”—The ID passed in *window_id* is for an input window that is already associated with a list. An input window can be associated with only one list.

See also

“List load method” on page 9-9 and [LLOAD_METHOD](#) on page 9-154 for more information about using load methods.

Examples

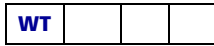
The following example creates a list from the class “lst_lookup” defined in the library opened on **g_utilib**. **Inpid** is the associated input window, and **lookup_data** contains the initial contents of the associated data to store or retrieve for each list item.

```
xcall l_create(lstid, inpid, lookup_data, g_utilib, "lst_lookup")
```

The next example creates a list from the list class “lst_trans” defined in the library opened on **g_utilib**. The ID of the associated input window is contained in **item_id**, and **trans_data** contains the initial contents of the associated data to store or retrieve for each list item. The list is initially disabled and not placed on the screen. However, its placement is specified as row 5, column 20, which is where it will be placed if **L_PLACE** is called without re-specifying these parameters. Presuming that **nmitems** is greater than 1, loading of items into the list will be in reverse order, starting with the last item. Any error that occurs while attempting to create the list will be returned in **error**.

```
xcall l_create(trans_lst, item_id, trans_data, g_utilib, "lst_trans", 5,
&                20, D_NOPLC, TRUE, nmitems, nmitems, error)
```

L_CTRCONTEXT – Establish focus within a child list window



```
xcall L_CTRCONTEXT (list_id, reason)
```

Arguments

list_id

The ID of the child list. (**n**)

reason

The reason the list window was activated (which determines what L_CTRCONTEXT will move focus to). This argument can be passed as blank (which indicates that the list was already activated), a program-supplied reason (because it was activated programmatically), or one of the following: (**a**)

C_FIRST Focus moved in attempt to reach the first child in the composite window. This instructs L_CTRCONTEXT to move focus to the list (rather than a button on the list window).

C_LAST Focus moved in attempt to reach the last child in the composite window. This instructs L_CTRCONTEXT to move focus to the last active button on the tab set (if the list is on a tab). If there is no such button, it moves focus to the last active button for the list window. If there's no active button for the window, it moves focus to the list itself.

C_NEXT Focus moved from the previous child in the composite window—e.g., by pressing TAB. This instructs L_CTRCONTEXT to move focus to the list (rather than a button on the list window).

C_PREV Focus moved from the next child in the composite window—e.g., by pressing SHIFT+TAB. This instructs L_CTRCONTEXT to move focus to the last active button on the tab set (if the list is on a tab). If there is no such button, it moves focus to the last active button for the list window. If there's no active button for the list window, it moves focus to the list itself.

C_SET List or button explicitly activated. For example, this could be the result of a mouse click, or it could be that the list or button was activated programmatically without a specified reason. With this reason, L_CTRCONTEXT does not change focus.

If *reason* is blank or any value other than the above, L_CTRCONTEXT does not change focus.

Discussion

The `L_CTRCONTEXT` subroutine sets focus for a child list window to either the list or a button on the list window. (On UNIX and OpenVMS and in Synergy .NET, this routine is ignored.)

`L_CTRCONTEXT` uses *reason* to determine where to establish focus, and is designed for use in child list methods (`C_METHLST` by default), where it's likely that you want focus to initially reflect the reason the list window was activated. For example, if you have a composite window with a list and an input window, and the user presses `SHIFT+TAB` to move from the input window back to the list, you may want the last button in the list to receive focus.

To use in a child method, pass the reason passed to the child method (by `C_PROCESS`) as the *reason* argument for `L_CTRCONTEXT`. See the `ctrmeths.dbl` file included in your UI Toolkit distribution for an example. This file includes code for the `C_METHLST` method, which uses `L_CTRCONTEXT`.

See also

- ▶ [C_METHLST](#) on page 6-54.
- ▶ [“Controlling tabbing, focus, and input context”](#) on page 6-4 for information on tabbing indices.
- ▶ [DC_CTRCONTEXT](#) on page 6-33 for information on moving focus to a specific child window.

L_DATA – Manipulate the data in a list



```
xcall L_DATA (list_id, request, window_id, [data], [disabled], no_find
&          [, a_methoddata1, ..., a_methoddata20])
```

Arguments

list_id

The ID of the list to manipulate. **(n)**

request

The requested operation: **(n)**

D_LCURRENT Retrieve L_PROCESS's current item.

D_LFIRST Retrieve the first loaded item.

D_LLAST Retrieve the last loaded item.

D_LNEXT Retrieve the next loaded item.

D_LPREV Retrieve the previous loaded item.

D_LWRITE Rewrite an item.

window_id

The ID of the associated input window. **(n)**

data

(optional) The associated non-window data. **(a)**

disabled

(optional) The disabled item status. **(n)**

no_find

A true/false flag that indicates whether the operation failed. **(n)**

a_methoddata1 through *a_methoddata20*

(optional) Up to 20 additional arguments for field methods. **(any)**

Discussion

The `L_DATA` subroutine enables you to read or write items that have already been loaded into the list without modifying the context for the next call to `L_PROCESS`.

`L_DATA` retrieves and rewrites the input window display, reading and writing data for items in the list that the `L_PROCESS` subroutine has already loaded. It can also retrieve and rewrite the associated non-window data, the disabled flag, and the input set controls.

Disabled is a true/false flag that indicates whether the current item is disabled. Whether or not the *disabled* argument is passed must correspond to whether it was passed to other list routines such as `L_CHR`, `L_INPFLD`, `L_INPUT`, `L_PROCESS`, and `L_SELECT`. See [L_PROCESS on page 9-111](#) for more information about the disabled item status. It is both passed (with `D_LWRITE`) and returned.

If the display of any visible item is modified by `D_LWRITE`, that change is reflected in the list. The context of `L_PROCESS`, however, is not modified. The purpose of this subroutine is to enable the calling routine to peruse and/or modify data in other items of the list without changing the list's context.

We recommend that you use a different input window and data area for this subroutine than for `L_PROCESS`. Another option is to issue the request `D_LRESTORE` before attempting any other operation when you are ready to proceed with `L_PROCESS`.



For ActiveX Toolkit lists, the display of each field in the input window is mapped to the associated column in the list. So if the input window whose ID you pass to `L_DATA` has different field definitions than the input window used to create the list, you may get unexpected field/column mappings, and some fields may not be mapped at all. (Each column in the list maps to the Toolkit field number used to create that column initially, and if that field number is greater than the number of fields in the input window passed to `L_DATA`, the field won't be mapped.)

Unlike `L_PROCESS`, the `L_DATA` subroutine only records information in the list when the `D_LWRITE` flag is specified.

The `D_LCURRENT` flag retrieves the current item being used by `L_PROCESS`. This item is not necessarily the same as the item retrieved by the most recent call to `L_DATA`. If the current item has not been loaded (as in the case of a null list), *no_find* will be returned with a value of true.

The `D_LFIRST` flag retrieves the first item currently loaded into the list. If the list permits item loading at the top and a `D_LEOF` has not yet been signaled, `D_LFIRST` retrieves the uppermost item loaded and does not cause a load request to be issued. If the list doesn't contain any items, the *no_find* argument is returned with a value of true.

The `D_LLAST` flag retrieves the last item currently loaded into the list. If the list permits item loading at the bottom and a `D_LEOF` has not yet been signaled, `D_LLAST` will retrieve the lowermost item loaded and will not cause a load request to be issued. If the list doesn't contain any items, the *no_find* argument will be returned with a value of true.

List Routines

L_DATA

The `D_LNEXT` flag retrieves the item that follows the last item retrieved by this subroutine. If `L_DATA` has not retrieved any items, or if no item follows the last item retrieved, the *no_find* argument will be returned with a value of true, and no item will be retrieved. `D_LNEXT` will only retrieve an item that has already been loaded.

The `D_LPREV` flag retrieves the item that precedes the last item retrieved by this subroutine. If `L_DATA` has not retrieved any items, or if no item precedes the last item retrieved, the *no_find* argument will be returned with a value of true, and no item will be retrieved. `D_LPREV` will only retrieve an item that has already been loaded.

The `D_LWRITE` flag rewrites the last item that was retrieved by `L_DATA`. (If `L_DATA` has not retrieved an item, *no_find* is returned with the value of true, and no rewrite occurs.) `D_LWRITE` updates both the display and non-window data for the list, but the display will reflect changes only if `I_DSPFLD` or `I_DISPLAY` is called with the window that's passed to `L_DATA` and if that call occurs before the call to `L_DATA`. Non-window data is updated with changes in any case.

If `L_PROCESS` deletes the last item that was retrieved, no context exists for `D_LNEXT`, `D_LPREV`, or `D_LWRITE`. You can use `D_LCURRENT`, `D_LFIRST`, or `D_LLAST` to re-establish a context for this subroutine.

A_methoddata1 through *a_methoddata20* are up to 20 additional (and optional) arguments. These arguments are used only if the `INPUT_SNAPSHOT` qualifier was specified in the list class when the list was created and you specify a request that updates the input window (`D_LCURRENT`, `D_LFIRST`, `D_LNEXT`, or `D_LPREV`—not `D_LWRITE`). When `L_DATA` updates the input window by calling `I_SNAPSHOT`, it redisplay the data into the input window, which requires it to use any display methods for the fields. The method data arguments are passed to those display methods.

Examples

In this example, the only associated data in the list is a number, which is renumbered beginning with the current item in the list and starting with the current item's value for the number plus one. For example, starting with the current item, if the items in the list were numbered 5 through 20, the items would be renumbered 6 through 21 after the code below was executed. Any visible item would reflect that change.

```
xcall l_data(mylist, D_LCURRENT, inpid, number,, eof)
while (.not. eof)
  begin
    incr number
    xcall i_dspfld(inpid, "number", number)
    xcall l_data(mylist, D_LWRITE, inpid, number,, eof)
    xcall l_data(mylist, D_LNEXT, inpid, number,, eof)
  end
```

L_DELETE – Delete a list



```
xcall L_DELETE(list_id)
```

Arguments

list_id

The ID of the list to delete. (**n**)

Discussion

The L_DELETE subroutine deletes a list.

A list is automatically deleted when the user exits the environment in which the list was created (unless the list is promoted to global status). The L_DELETE subroutine enables you to delete a list explicitly. If you try to delete a list that is placed at a prior environment level, a fatal error occurs.

Examples

The following example deletes the list whose ID is stored in **mylst**.

```
xcall l_delete(mylst)
```

L_DISABLE – Disable a list



```
xcall L_DISABLE(list_id)
```

Arguments

list_id

The ID of the list to disable. (**n**)

Discussion

The L_DISABLE subroutine disables a list.

The current rendition for the scroll bars remains in effect whether the list is enabled or disabled.

On Windows, there is no visual difference between enabled and disabled lists. Both use the same Windows Control Panel setting for background and foreground colors. List renditions are not used on Windows except for the header (D_RND_LST_HDR) and footer (D_RND_LST_FTR) on Toolkit (non-ActiveX) lists.

On UNIX and OpenVMS, a disabled list is distinguished from an enabled list by a change in renditions. When an enabled list is disabled, its body, border, title, and selected item take on new renditions. The following renditions apply:

D_RND_LST_DSA	Disabled list body, border, and title
D_RND_LST_DSA_S	Selected item in disabled list
D_RND_LST_FTR	Footer
D_RND_LST_HDR	Header

Examples

The following example disables the list whose ID is stored in **order_lst**.

```
xcall l_disable(order_lst)
```


L_ENABLE – Enable a list



```
xcall L_ENABLE(list_id)
```

Arguments

list_id

The ID of the list to enable. (**n**)

Discussion

The L_ENABLE subroutine enables a list.

The current rendition for the scroll bars remains in effect whether the list is enabled or disabled.

On Windows, there is no visual difference between enabled and disabled lists. Both use the same Windows Control Panel setting for background and foreground colors. List renditions are not used on Windows except for the header (D_RND_LST_HDR) and footer (D_RND_LST_FTR) on Toolkit (non-ActiveX) lists.

On UNIX and OpenVMS, an enabled list is distinguished from a disabled list by a change in renditions. When a disabled list is enabled, its body, border, title, and selected item take on new renditions. The following renditions apply:

D_RND_LST_ENA	Enabled list's body
D_RND_LST_ENA_B	Border
D_RND_LST_ENA_T	Title
D_RND_LST_FTR	Footer
D_RND_LST_HDR	Header
D_RND_LST_SEL	Selected item in an enabled list

Examples

The following example enables the list whose ID is stored in **file_lst**.

```
xcall l_enable(file_lst)
```

L_FINDSPEC – Specify parameters for a find operation



```
xcall L_FINDSPEC (list_id, [string], [column], [length], [direction], [case], [type], [current],
&                [wrap][, find_source])
```

Arguments

list_id

The ID of the list to search. **(n)**

string

(optional) The string for which to search. **(a)**

column

(optional) The first column to search. **(n)**

length

(optional) The length of the search area (a maximum of 256 characters). **(n)**

direction

(optional) The direction of the search: **(n)**

D_LFND_FRWD Forward. (default)

D_LFND_RVRS Reverse.

case

(optional) The case-sensitivity flag: **(n)**

D_LFND_CASE Case sensitive. (default)

D_LFND_NOCASE Case insensitive.

type

(optional) The flag that indicates what type of search will occur: **(n)**

D_LFND_CONTAIN Search for a literal string. (default)

D_LFND_PATTERN Search for a wild card match.

D_LFND_EQ Search for equality.

D_LFND_NE Search for inequality.

D_LFND_GT Search for a string greater than.

D_LFND_LT Search for a string less than.

D_LFND_GE Search for a string greater than or equal to.

D_LFND_LE Search for a string less than or equal to.

current

(optional) The flag that indicates whether the current item is checked as part of the search: **(n)**

D_LFND_NOCURR Do not test current item. (default)

D_LFND_CURR Test current item.

wrap

(optional) The wrap-around find flag: **(n)**

D_LFND_WRAP Search to end of list and then wrap around to start at beginning of list until search string is found or original item is re-encountered.

D_LFND_NOWRAP Search only from the original item to the end of the list. (default)

find_source

(optional) If passed and non-zero, the flag that indicates whether the non-window data or the associated input window (display data) will be searched: **(n)**

D_LFND_DATA Search non-window data. (default)

D_LFND_DISPLAY Search display data.

Discussion

The L_FINDSPEC subroutine specifies the parameters for subsequent find operations performed by a D_LFIND request in the L_PROCESS subroutine. This subroutine does not perform a search; it merely sets the parameters for subsequent find operations that will occur. If L_FINDSPEC is called more than once, the parameters that it sets are combined. If any of these parameters contradict one another, the parameter that is set last overrides the others.

For example, if you make the following calls to L_FINDSPEC:

```
xcall l_findspec(trans_lst, find_string,,D_LFND_RVRS,,D_LFND_NE,,,
&      D_LFND_DATA)
```

```
xcall l_findspec(trans_lst, find_string,,,D_LFND_NOCASE,D_LFND_EQ,,,
&      D_LFND_DISPLAY)
```

the following search parameters are defined: reverse direction, case insensitivity, search for equality, search display data.

If *string* is not passed, the previous find string is used. If none exists, a blank string is used.

If *find_source* is D_LFND_DATA,

- ▶ the *column* argument specifies the position to start the search in the non-window data. If *column* is not passed or is zero, the previously specified value for *column* is used—even if this was specified in conjunction with D_LFND_DISPLAY. If there is no previous value for *column*, it is assumed to be 1. The value of the *column* argument is limited to the length of the non-window data. So, for example, if the non-window data has 60 columns, but *column* is passed as 61, only 60 are used.
- ▶ the *length* argument specifies the length of the area to search. This area begins at the position specified by *column*. If *length* is not passed or is zero, the previously specified value for *length* is used—even if this was specified in conjunction with D_LFND_DISPLAY. If there is no previous value for *length* and
 - ▶ a *data* argument was passed to L_CREATE, the length of that argument is used as the initial value for *length*.
 - ▶ no *data* argument was passed to L_CREATE, the total number of characters in the associated input window is used as the initial value for *length*.

The *length* argument is limited to the number of characters from *column* to the end of the non-window data. So, for example, if *length* is passed as 17 and there are 15 characters from *column* to the end of the non-window data, only 15 characters are used.

If *find_source* is D_LFND_DISPLAY,

- ▶ the *column* argument specifies the position to start the search in the associated input window. A value of 1 specifies the first column in the first row. A value of 2 specifies the second column in the first row, and so forth. If *column* is greater than the number of columns in the input window, the position wraps to subsequent rows. If *column* is not passed or is zero, the previously specified value for *column* is used—even if this was specified in conjunction with D_LFND_DATA. If there is no previous value for *column*, it is assumed to be 1. The *column* argument cannot be greater than the number of rows multiplied by the number of columns in the input window. So, for example, if the number of rows multiplied by the number of columns equals 60, but *column* is passed as 61, only 60 are used.
- ▶ the *length* argument specifies the length of the area to search. This area begins at the position specified by *column* and continues by column, wrapping to subsequent rows if necessary. If *length* is not passed or is zero, the previously specified value for *length* is used—even if this was specified in conjunction with D_LFND_DATA. If there is no previous value for *length* and
 - ▶ a *data* argument was passed to L_CREATE, the length of that argument is used as the initial value for *length*.
 - ▶ no *data* argument was passed to L_CREATE, the total number of characters in the associated input window is used as the initial value for *length*.

The *length* argument is limited to the number of characters from *column* to the last column and row of the input window. So, for example, if *length* is passed as 17 and there are 15 characters from *column* to the last column and row of the input window, only 15 characters are used.

If *find_source* is not passed or is zero, the previously specified value for *find_source* is used. If there is no previous value for *find_source*, the value is assumed to be

- ▶ D_LFND_DATA if a *data* argument was passed to L_CREATE.
- ▶ D_LFND_DISPLAY if no data argument was passed to L_CREATE.

If *direction* is not passed, the previous direction value will be used. If no previous direction exists, the search will go forward.

You can pass the %I_FLDSTRPOS, %I_FLDSIZE, and/or %I_FLDDIM field information functions in place of passing *column* and *length*. An example of these functions used with the L_FINDSPEC subroutine is in the %I_FLDSTRPOS [Examples on page 8-40](#).

If *case* is not passed, the previous case value will be used. If no previous case value exists, the search will be case sensitive.

If *type* is not passed, the previous find type is used. If no previous find type exists, D_LFIND searches for a literal string (D_LFND_CONTAIN). The following rules apply to the *type* flags:

- ▶ If D_LFND_PATTERN is passed, the Synergy DBL S_MATCH subroutine is used to compare the search string with the designated area of the *data* argument passed to L_PROCESS.
- ▶ If D_LFND_CONTAIN is passed, the intrinsic function %INSTR is used instead of S_MATCH.
- ▶ If D_LFND_EQ, D_LFND_NE, D_LFND_GT, D_LFND_LT, D_LFND_GE, or D_LFND_LE is passed, the corresponding Synergy DBL comparison operator is used.

If *current* is not passed, the previous testing value will be used. If no previous value exists, D_LFIND won't test the current item. The following rules apply to the *current* flags:

- ▶ If D_LFND_CURR is passed, the search for a matching item during the next find begins with the current item.
- ▶ If D_LFND_NOCURR is passed, the search begins with the next item in the specified direction.

Examples

The following example defines search parameters for the list identified by **order_lst**. If this is the first and only call to L_FINDSPEC, the L_PROCESS subroutine will search the list for a case-sensitive match for **find_string** (a literal string). The search will start with the item that follows the current one and search forward through the list. The entire data area passed to L_PROCESS will be searched. If needed, L_PROCESS will call the load method for this list to retrieve additional items.

```
xcall l_findspec(order_lst, find_string,,,,,D_LFIND_DATA)
req = D_LFIND
xcall l_process(order_lst, req, order_rec)
```

%L_FINDWND – Construct a find window for L_SELECT

WT	WN	U	V

```

window_id = %L_FINDWND (list_id, window_name, field_name, [prompt], [search]
&                    [, nofindspec])

```

Return value

window_id

The ID of the created window or zero if an error occurred. (^VAL)

Arguments

list_id

The ID of the list. (n)

window_name

A unique name for the window to be constructed (a maximum of 15 characters). (a)

field_name

The name of the field on which to base the find. (a)

prompt

(optional) The prompt for input. (a)

search

(optional) The true/false flag that, when returned, will indicate whether an existing *window_name* was found. (n)

nofindspec

(optional) The true/false flag that indicates whether this function should perform without calling L_FINDSPEC. (n)

Discussion

The %L_FINDWND function creates an input window for a find string, based on a field within the input window associated with a list. The resulting input window is designed to be passed to L_SELECT as the *find_window* argument.

By using this function, you assure that the data storage format of the input find string will match that of the target field, including length, type, and enumeration. This function also optionally calls L_FINDSPEC to establish the position and length of the target field for the find. The assumption here is that the optional data passed to L_SELECT matches the input window data structure containing the target field. Even if this is not a correct assumption, however, you can call L_FINDSPEC subsequently to adjust.

Window_name must be a unique window name. If *search* is passed and a window with *window_name* already exists, *search* is returned true and no construction of the window occurs (L_FINDSPEC can still be called). *Search* is returned false if the window did not already exist. If *search* is not passed and the window already exists, a fatal error occurs.

Field_name must be the name of an input field within the input window associated with the list indicated by *list_id*. If not, a fatal error occurs.

If *prompt* is passed, it will be used as the prompt for the input field in the constructed window. Otherwise, the default find prompt will be used, which is defined in the message file (**syntxt.ism**), facility DTK, mnemonic FINDPRMPT. *Prompt* is positioned at row 2, column 2 of the constructed window, and the field immediately follows (one space is added to the trimmed value of the default prompt, if used—if *prompt* is passed, it must contain any trailing spaces desired).

The window is sized to have one empty row and one empty column before and after the input field (including prompt).

The input field inherits all attributes of the target field, except for the following:

- ▶ Prompt position
- ▶ Prompt
- ▶ Field position
- ▶ Break (always false)
- ▶ Auto (always false)
- ▶ Allow (unless enumerated)
- ▶ Range

The constructed window only contains an input field for the find string itself. It does not provide access to the other find parameters. (See the information on the *find_window*, *find_init*, and *find_column* arguments in [L_SELECT on page 9-128](#).)

By default, this function calls L_FINDSPEC to establish the position and length of the target field for the find. If *nofindspec* is passed and true, this will not occur.

You do not need to declare this function to use it. It is defined in **tools.def**.

Examples

In the following example, %L_FINDWND is called to construct an input window named “i_findacct” for entering an account number to find. If a window by that name already exists, the existing window will be used instead and **srch** will be returned true.

```
findid = %l_findwnd(acct_list, "i_findacct", "account_num",
    & "Find account ", srch)
xcall u_window(D_PLACE, findid, 12, 40, D_REMOVE, findid)
xcall l_select(acct_list, req, acct_rec,,,,,,,, findid)
```

List Routines

%L_FINDWND

The account number field will inherit attributes from the “account_num” field in the input window associated with the list whose ID is **acct_lst**. The prompt for the field will be “Find account.”

L_FINDSPEC will be called by %L_FINDWND to establish the position and length of the find data as the position and length of the “account_num” field within a structure in the list’s input window.

Next, U_WINDOW is called to establish the placement of the resulting window. Then the window is removed so that it will not be visible until needed. The ID is passed to L_SELECT so that the window can be “popped up” when the user selects the S_FIND menu entry.

L_GLOBAL – Promote a list to global status



```
xcall L_GLOBAL(list_id)
```

Arguments

list_id

The ID of the list to be promoted. (**n**)

Discussion

The L_GLOBAL subroutine promotes the list's associated input window to global status, along with any other window (header, footer, and so forth) that may be associated with the list.

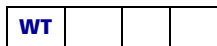
If *list_id* is already global, this subroutine will be ignored. Otherwise, if *list_id* is not local to the current environment, a fatal error occurs.

Examples

In the following example, L_CREATE returns the variable **lstid** with the list ID, and the L_GLOBAL subroutine promotes that list to global status.

```
xcall l_create(lstid, inpid)  
xcall l_global(lstid)
```

%L_ICON – Set a list icon



```
status = %L_ICON(list_id, filename[, index])
```

or

```
xcall L_ICON(list_id, filename[, index])
```

Return value

status

On Windows, returns true if the icon was loaded, and false otherwise. On UNIX, always returns false. On OpenVMS, always returns true. (^VAL)

Arguments

list_id

The ID of the list. (n)

filename

The name of the icon file to use. (a)

index

(optional) The index within *filename* of the icon to load (1-based). (n)

Discussion

%L_ICON sets an icon for a Toolkit list or ActiveX Toolkit list control.

The icon is used for the system menu, which is displayed if a close, move, size, minimize, or maximize method is associated with the window. See [%U_WNDEVENTS on page 16-203](#) for the circumstances under which a system menu is displayed. The L_METHOD subroutine must also be called to associate a %U_WNDEVENTS method set with the list.

You must either include **tools.def** or use the **-X** compiler option. Otherwise, a compilation error will occur.

If *list_id* is not the ID of an active list, an error is generated.

Filename can be an **.ico** file, a DLL, an executable program (**.exe**), or a bitmap image (**.bmp**). You can use environment variables in *filename*. If *filename* is invalid, no error is generated, however the return value will be false and the list's icon will remain unmodified.

A file can contain multiple icons. Setting *index* specifies which icon to use. The default is 1, and a value of 0 sets the icon to the default Windows system icon. Note that for an **.ico** file, *index* must be set to 1.

See also

[%U_ICON on page 16-87](#) for more information on setting a window icon.

Examples

The following example sets the list icon to the third icon in the **iconfil.dll** file. If the icon could not be found, an error is displayed.

```
if (.not.(%l_icon(list, "iconfil.dll", 3)))  
    xcall u_message("Could not set icon")
```

L_INPFLD – Perform single-field input within a list

WT	WN	U	V

```
xcall L_INPFLD (list_id, field_name, data_location, [data], [disabled], [input_column],
&             [select_column], [text_column], [init_char] [, a_methoddata1, ..., a_methoddata20]
&             {reserved})
```

Arguments

list_id

The ID of the list. **(n)**

field_name

The field to process. **(a)**

data_location

Where to place the input data. **(a or n)**

data

(optional) The non-window list item data. **(a)**

disabled

(optional) The disabled item status. **(n)**

input_column

(optional) The input column ID. **(n)**

select_column

(optional) The selection column ID. **(n)**

text_column

(optional) The text column ID. **(n)**

init_char

(optional) The first character to input (specified as a decimal value). **(n)**

a_methoddata1 through *a_methoddata20*

(optional) Up to 20 additional data arguments. **(any)**

{reserved}

Reserved for internal use. **(n)**

Discussion

This subroutine performs the same function as the input subroutine `I_INPFLD`, except `L_INPFLD` performs the function for a field contained within a list. See [I_INPFLD on page 8-51](#) for information on the arguments you can pass to `L_INPFLD` and the menu entries recognized by `L_INPFLD`.

Whether or not the *data* argument is passed must correspond to whether it was passed in the `L_CREATE` subroutine when the list was created. *Data* is required here if it was also passed to `L_CREATE`. If *data* is passed here but was *not* passed to `L_CREATE`, a fatal error occurs.

Note that by default, Toolkit does not ensure that the size of *data_location* matches the size of *field_name* or that it is even large enough for *field_name*. However, bounds checking can check for the following conditions. (See [g_dtkbounds on page 18-3](#) for more information.)

- ▶ If **g_dtkbounds** is set to 1, Toolkit generates a fatal error if *data_location* is not large enough for *field_name*.
- ▶ If **g_dtkbounds** is set to 2 or greater, Toolkit generates a fatal error if *data_location* is not large enough for *field_name* (as it does if **g_dtkbounds** is set to 1), and it generates a non-fatal error if *data_location* is large enough, but isn't the exact size of *field_name*.

Disabled is a true/false flag that indicates whether the current item is disabled. Whether or not the *disabled* argument is passed must correspond to whether it was passed to other list routines such as `L_CHR`, `L_DATA`, `L_INPUT`, `L_PROCESS` and `L_SELECT`. See [L_PROCESS on page 9-111](#) for more information about the disabled item status.

On UNIX and OpenVMS *init_char*, if passed, is converted to its alpha (ASCII) equivalent and placed into the field as if it were the first character entered by the user. *Init_char* is ignored on Windows.

A_methoddata1 through *a_methoddata20* are up to 20 additional optional arguments that can be passed to `L_INPFLD`. `L_INPFLD`, in turn, passes them directly to each field's methods. This allows you to communicate additional information to your input field methods.

When this subroutine returns, either field entry has been completed or a menu entry has been selected.

If you are using `L_SELECT` you do not need to call this subroutine because `L_SELECT` handles list input.

On Windows, the user can change the current item in the list with the mouse or the PAGE UP, PAGE DOWN, HOME, END, and arrow keys before this subroutine returns. Therefore, if you want any special processing to be done when arriving at or leaving a list item, you must write and use an arrive or leave method. (In Windows environments, these keys are "privatized" and unavailable to calling routines unless you use `L_RETURNKEY` to return the key out of the list. "Privatized" means that they are reserved for the private use of the graphical control, which in this case would be the list.)

See also

- ▶ [L_INPFLD on page 8-51](#) for more information about the arguments you can pass and for information about handling the input processing.
- ▶ [LARRIVE_METHOD/LLEAVE_METHOD on page 9-148](#) for more information on writing an arrive or leave method subroutine.
- ▶ [“Using Lists in Toolkit Applications” on page 9-4](#) for information on the differences between Toolkit lists and ActiveX Toolkit lists.

L_INPUT – Perform input to a set within a list



```
xcall L_INPUT (list_id, [set_name], record, [data], [disabled], [input_column],
&             [select_column], [edit_column], [D_NOTERM], [init_char], {reserved}
&             [, a_methoddata1, ..., a_methoddata20])
```

Arguments

list_id

The ID of the list. **(n)**

set_name

(optional) The name of the input set. **(a)**

record

The input set's record. **(a)**

data

(optional) The non-window list item data. **(a)**

disabled

(optional) The disabled item status. **(n)**

input_column

(optional) The input column ID. **(n)**

select_column

(optional) The selection column ID. **(n)**

edit_column

(optional) The edit column ID. **(n)**

D_NOTERM

(optional) The flag that tells L_INPUT to skip set termination processing. **(n)**

init_char

(optional) The first character to input (specified as a decimal value). **(n)**

{reserved}

A reserved placeholder for a future UI Toolkit argument. You do not specify any argument here, but *do* add the additional comma.

a_methoddata1 through *a_methoddata20*

(optional) Up to 20 additional data arguments. **(any)**

Discussion

L_INPUT performs essentially the same function as the input subroutine L_INPUT, but targeted to an input set contained within a list. See [L_INPUT on page 8-55](#) for information on the arguments you can pass to L_INPUT and the menu entries recognized by L_INPUT, and note that if you are using L_SELECT, you do not need to call this subroutine because L_SELECT handles list input.

When this subroutine returns, one of the following has occurred:

- ▶ Input to the set has been completed.
- ▶ A break field has been entered.
- ▶ A menu entry has been selected.
- ▶ TAB from the last field of an item in L_INPUT when the list is part of a composite window. This signals “C_NEXT”.
- ▶ SHIFT+TAB from the first field of an item in L_INPUT when the list is part of a composite window. This signals “C_PREV”.

L_INPUT calls L_PROCESS to update the current item before performing input. This means that you should include the fourth argument (*data*) in order to preserve the list item data.

Whether or not the *data* argument is passed must correspond to whether it was passed in the L_CREATE subroutine when the list was created. Note the following:

- ▶ *Data* is required here if it was also passed to L_CREATE. If *data* is passed here but was *not* passed to L_CREATE, a fatal error occurs.
- ▶ We highly recommended that the list *data* also contain the input data for the set. Otherwise, confusion may result over the contents of the input set data as list items are changed interactively.
- ▶ By default, Toolkit does not ensure that *data* is large enough. If it isn't, Toolkit generates an error only if **g_dtkbounds** is set to 1 or greater. See [g_dtkbounds on page 18-3](#).

Disabled is a true/false flag that indicates whether the current item is disabled. Whether or not the *disabled* argument is passed depends on whether it was passed to L_CREATE. See [L_PROCESS on page 9-111](#) for information about the disabled item status.

WIN

The D_NOTERM flag is ignored on Windows. The input window is always treated as if D_NOTERM were specified. This is necessary to support item-to-item movement within the list during input.

In Windows environments, the user can change the current item in the list with the mouse or the PAGE UP, PAGE DOWN, HOME, END, and arrow keys before this subroutine returns. Therefore, if you want any special processing to be done when arriving at or leaving a list item, you will have to write and use an arrive or leave method. (In Windows environments, these keys are “privatized” and

unavailable to calling routines unless you use `L_RETURNKEY` to return the key out of the list. “Privatized” means that they are reserved for the private use of the graphical control, which in this case would be the list.)

`L_INPUT` has the following limitations:

- ▶ Multi-line text fields are not supported within Toolkit lists or ActiveX Toolkit lists.
- ▶ Selections fields (`SELECT` or `SELWND`) are not supported within Toolkit lists.
- ▶ Radio button fields are not supported within Toolkit lists. Within ActiveX Toolkit lists, radio buttons are rendered as if they were selection fields.
- ▶ Check box fields are not supported within Toolkit lists.
- ▶ Drill methods are supported by the menu entry `I_DRILL`, but no drilldown button is displayed.
- ▶ Hyperlink methods are supported by the menu entry `I_HYPER`, but the prompt cannot be clicked on to launch `I_HYPER`, nor is the prompt rendered green.
- ▶ Noecho fields do not work properly in Toolkit lists. For ActiveX Toolkit lists, asterisks (*) are displayed.
- ▶ When using ActiveX Toolkit lists, the first field in the list will be selected regardless of whether or not that field is in the current input set.
- ▶ We recommend that you use a fixed font for Toolkit lists; columns may not line up as expected when a proportional font is used.
- ▶ If you do use a proportional font for Toolkit lists, note that bold proportional fonts use an actual bold font, rather than rendering bold as a color change the way fixed fonts do. Consequently, bold characters are more likely to be truncated because Toolkit determines the width of the field or prompt using a non-bold (normal) font.

UNIX, VMS

If passed, *init_char* is converted to its alpha (ASCII) equivalent and placed into the field as if it were the first character entered by the user. *Init_char* is ignored on Windows.

A_methoddata1 through *a_methoddata20* are up to 20 additional optional arguments that can be passed to `L_INPUT`. `L_INPUT`, in turn, passes them directly to each field’s methods. This allows you to communicate additional information to your input field methods.



Toolkit passes the method data arguments to the current list processing routine (`L_PROCESS`, `L_INPUT`, or `L_SELECT`) and to the list methods. Thus, if you pass data arguments to one list processing routine, to ensure these arguments are always available, you must pass them to *all* the list processing routines.

See also

- ▶ [L_INPUT on page 8-55](#) for more information about the arguments you can pass and for information about handling the input processing.
- ▶ [L_METHOD on page 9-105](#) for more information on registering a method with the list processor.
- ▶ [“Using Lists in Toolkit Applications” on page 9-4](#) for information on the differences between Toolkit lists and ActiveX Toolkit lists.

L_METHOD – Register a list method



```
xcall L_METHOD (list_id, event_method_pair[, ...])
```

Arguments

list_id

The ID of the list. (n)

event_method_pair

The method routine name to register. This argument has the following format:

method_option[, method_routine | method_set_id]

where *method_option* is one of the following:

- D_LARRIVE** Set or clear the arrive method. To specify an arrive method, pass this with *method_routine*. To clear any arrive method associated with the list, pass D_LARRIVE, but omit *method_routine*. (Do not use *method_set_id* with this option.)
- D_LDBLCLK** Set or clear the double-click method. To specify a double-click method, pass this with *method_routine*. To clear any double-click method associated with the list, pass D_LDBLCLK, but omit *method_routine*. (Do not use *method_set_id* with this option.)
- D_LLEAVE** Set or clear the leave method. To specify a leave method, pass this with *method_routine*. To clear any leave method associated with the list, pass D_LLEAVE, but omit *method_routine*. (Do not use *method_set_id* with this option.)
- D_LLOAD** Set or clear the load method. To specify a load method, pass this with *method_routine*. To clear any load method associated with the list, pass D_LLOAD, but omit *method_routine*. (Do not use *method_set_id* with this option.)
- D_LWNDEVENT** Set or clear an event method set. To specify an event method set for the list, pass this with *method_set_id*. To clear any events method set associated with the list, pass this and omit *method_set_id*. (Do not use *method_routine* with this option.)

method_routine

The name of a method that L_PROCESS or L_SELECT will call to load new items onto the list, perform arrive or leave processing for a list item, or process double-click events. *Method_routine* must either be contained within the current program image or in an executable library (ELB) that was either linked to or opened with OPENELB. (a)

method_set_id

The ID for a method set that will process mouse events for the list. Valid only with D_LWNDEVENT. This must be an ID returned from a D_CREATE call to %U_WNDEVENTS. (**n**)

Discussion

The L_METHOD subroutine registers method routine names with the list processor. Use L_METHOD if you want to do additional processing when the user initiates the specified event. When that event occurs, UI Toolkit will call the method you specify. (You write your own *method_routines*.) L_METHOD settings override L_CLASS and .LISTCLASS method settings.

You may specify more than one method in a single L_METHOD call. However, if you omit a *method_routine* or *method_set_id* argument, you must include a comma placeholder if further method specifications follow. For example, the following specifies a load method and an arrive method and clears any leave method associated with the list. (Note the comma placeholder after D_LLEAVE.)

```
xcall l_method(lstid, D_LLOAD, "loadmethod", D_LLEAVE,, D_LARRIVE,  
& "arrivemethod")
```

On Windows, if you don't populate the list with the L_PROCESS requests D_LAPPEND or D_LINSERT, you must write load method (and use D_LLOAD to specify that method) if you want scrolling enabled. Additionally, if you are changing the load method and restarting the list (using L_RESTART), you must call L_METHOD first. Unlike UNIX and OpenVMS environments, your load method will be called immediately, rather than on the next call to L_PROCESS, and calling L_METHOD first will not affect the way applications on UNIX or OpenVMS perform.

See also

- ▶ [LLOAD_METHOD on page 9-154](#) for more information on writing load methods.
- ▶ [LARRIVE_METHOD/LLEAVE_METHOD on page 9-148](#) for more information on writing arrive and leave processing methods.
- ▶ [LDBLCLICK_METHOD on page 9-151](#) for more information on writing double-click processing methods.
- ▶ [%UWNDEVENTS_METHOD on page 16-237](#) for more information on writing window events methods.
- ▶ [“Appendix D: Methods”](#).

L_NEXT – Set selected item by number



```
xcall L_NEXT(list_id, itemno)
```

Return value

status

Always returns true. (i)

Arguments

list_id

The ID of the list. (n)

itemno

The row number of the list item you want selected. (n)

Discussion

The L_NEXT subroutine enables you to specify which list item will be selected when a list is next processed. It queues a request to set the selected item for a list to the *n*th item where *n* is the value passed as *itemno*. As with L_QUEUE, the selection is queued so that actions queued before the call to L_NEXT occur before the selection, and actions queued after the call to L_NEXT occur after the selection. Note the following:

- ▶ Call L_NEXT before calling L_PROCESS or a routine that calls L_PROCESS (such as L_SELECT or L_INPUT). L_NEXT queues a request, but an L_PROCESS call is required to actually set the specified item.
- ▶ If the selection is successful, the found status of the list (the L_STATUS flag D_LFOUND) is set to 1.
- ▶ If the requested item cannot be accessed (in other words, if *itemno* is less than 1 or greater than the number of items in the list), the list selection remains unchanged, and the found status of the list is set to zero.
- ▶ If item numbering within the list is changed (by loading more or fewer items at the top than expected), the item is selected according to the most recent numbering scheme. This means that item 1 always selects the top item in the list. However, this also means that using the L_STATUS subfunction D_LCURITM to retrieve the current item number and then using L_NEXT in an attempt to return to that item after loading items may cause L_NEXT to access a different item.

List Routines

L_NEXT

Examples

In the following example, the `D_LCURITM` subfunction of `L_STATUS` is used to save the current item number. The example then navigates to the end of the list to ensure that all items have been loaded, enabling `D_LLASTITM` to get the total number of items. Finally, `L_NEXT` returns the list to the previously selected item.

```
xcall l_status(listid, D_LCURITM, itmno)           ;Fetch current item.
xcall l_process(listid, request=D_LBOTTOM, data)    ;Go to the end to load
                                                    ; all items.
xcall l_status(listid, D_LLASTITM, numitems)        ;Get number of items.
xcall l_next(listid, itmno)                         ;Specify where to go.
xcall l_process(listid, request=D_LNOP, data)       ;This actually does it.
```

L_PLACE – Place a list



```
xcall L_PLACE (list_id, [row][, column])
```

Arguments

list_id

The ID of the list to place. **(n)**

row

(optional) The row position for the list. **(n)**

column

(optional) The column position for the list. **(n)**

Discussion

The L_PLACE subroutine places a list. After you have placed the list, if you remove it at a subsequent environment level, the list will automatically be placed again when you call E_EXIT to exit that later environment.

If *row* and/or *column* is not passed, the default position is the previous row and/or column placement position. If no previous position exists, the list will be placed at the row and/or column position that was specified when the list was created.

See also

[L_REMOVE on page 9-120](#) for information on removing a list.

Examples

The following example moves the list, whose ID is stored in **order_lst**, vertically to row 5. The horizontal position will be the same as the previous placement, if it's valid, or the original placement.

```
xcall l_place(order_lst, 5)
```

The next example removes the list whose ID is stored in **trans_lst**, and then processes the menu. If the user selects the menu entry whose name is "PRCTRNS," the list will be placed at its previous location.

```
xcall l_remove(trans_lst)
xcall m_process
case g_entnam of
```

List Routines

L_PLACE

```
begincase
.
.
.
"PRCTRNS": begin                ;Place the list at
    xcall l_place(trans_lst)    ; previous position
    call process                ;Process the list
end
endcase
```


L_PROCESS – Process a list



```
xcall L_PROCESS (list_id, request, [data], [disabled]
&               [, a_methoddata1, ..., a_methoddata20])
```

Arguments

list_id

The ID of the list to process. (**n**)

request

One of the following requests (which must be passed as a variable). (**n**)

D_LNOP	No request, but update the current item.
D_LLOADBOT	Request to load at the bottom has been satisfied.
D_LLOADTOP	Request to load at the top has been satisfied.
D_LEOF	No more items to load at this end.
D_LUP	Position up one item.
D_LDOWN	Position down one item.
D_LLEFT	Scroll left.
D_LRIGHT	Scroll right.
D_LTOP	Position to the first item.
D_LBOTTOM	Position to the last item.
D_LPGUP	Position up one page.
D_LPGDN	Position down one page.
D_LPGTOP	Position to the top of the page.
D_LPGBOT	Position to the bottom of the page.
D_LINSERT	Insert a new item before the current item.
D_LAPPEND	Insert a new item after the current item.

D_LDELITEM	Delete the current item.
D_LMOVEUP	Move the current item up one place.
D_LMOVEDN	Move the current item down one place.
D_LFIND	Find an item between 1 and 256 characters.
D_LCANCEL or D_LRESTORE	Don't update the current item; instead, restore the previous contents of the window and data area.
D_LEXTENDBOT	Reopen the bottom of the list.
D_LEXTENDTOP	Reopen the top of the list.
Returned requests that L_PROCESS can generate are	
D_LNOP	No request.
D_LLOADBOT	Load a new item at the bottom.
D_LLOADTOP	Load a new item at the top.
D_LABORT	Fatal list error.

data

(optional) The associated non-window data. (**a**)

disabled

(optional) The disabled item status. (**n**)

a_methoddata1 through *a_methoddata20*

(optional) Up to 20 additional data arguments. (**any**)

Discussion

The L_PROCESS subroutine processes a list.

Request must be passed as a variable. See “Requests” on page 9-114 for more information on the values that can be passed and values that are returned.

Whether or not the *data* argument is passed must correspond to whether it was passed in the L_CREATE subroutine when the list was created. *Data* is required here if it was also passed to L_CREATE. If *data* is passed here but was *not* passed to L_CREATE, a fatal error occurs. (Non-window data is additional information that you don't want displayed, but that you want to be maintained with the list.)

Disabled is a true/false flag that indicates whether the current item is disabled. Whether or not the *disabled* argument is passed must correspond to whether it was passed to other list routines such as L_CHR, L_DATA, L_INPFLD, L_INPUT and L_SELECT.

If *disabled* is present and non-zero, the current item will be disabled. Otherwise, the current item will be disabled only if the entire list is disabled.

The *disabled* argument will be returned with the enabled/disabled status of the item. On return, *disabled* will be false (0) if the item inherited the entire list's enabled/disabled status, regardless of whether the list is actually enabled or disabled.

A disabled list item is designated by the "Disabled list" rendition. (See the **tools.def** file.) While a list item's appearance may be affected by the disabled flag, Toolkit processing is *not* affected by the fact that an item is disabled. Your calling routine is solely responsible for assigning significance to the disabled state of an item: you can leave the item as is, make it uneditable, or process it some other way. For example, the L_SELECT subroutine (which calls L_PROCESS) does not allow the user to select a disabled item, and you can make it skip over disabled items completely by passing a non-zero value in the *skip_disabled* argument.

On Windows, disabled list entries have no meaning.

A_methoddata1 through *a_methoddata20* are up to 20 additional optional arguments that can be passed to L_PROCESS. L_PROCESS, in turn, passes them directly to each field's methods. This allows you to communicate additional information to your input field methods.



Toolkit passes the method data arguments to the current list processing routine (L_PROCESS, L_INPUT, or L_SELECT) and to the list methods. Thus, if you pass data arguments to one list processing routine, to ensure these arguments are always available, you must pass them to *all* the list processing routines.

The first time this subroutine is called after the list is created, it initiates the loading of the first window full of items.

Unlike many of the processing routines in UI Toolkit, L_PROCESS doesn't perform input. The calling routine must perform any input and interpret any menu entry selections.

A list does not have to be placed to be processed.

If you are currently processing more than one list, you must call L_PROCESS or L_SELECT with *request* set to D_LRESTORE before calling any list processing routine when changing context from one list to the other.

Requests

D_LNOP, D_LLOADBOT, and D_LLOADTOP are the only requests that can be both passed to and returned from the L_PROCESS subroutine. The D_LABORT flag should *not* be passed to L_PROCESS.

If an LLOAD_METHOD subroutine has been specified, L_PROCESS will call it every time a new list item is needed. LLOAD_METHOD can be specified with the .LISTCLASS script command, or with the L_CLASS or L_METHOD subroutines.

On Windows, you must use a load method to support scrolling.

If no load method is specified for the list, L_PROCESS returns to the calling routine with *request* set to D_LLOADBOT or D_LLOADTOP, depending on which kind of loading is to be done initially. (See [L_CREATE](#) on page 9-75.)

Always set *request* to D_LEOF for a load request that you don't support. For example, if you support D_LLOADBOT but not D_LLOADTOP, you'd set *request* to D_LEOF if D_LLOADTOP was returned.

Before requesting a load, L_PROCESS sets both the data in the input window and the optional *data* argument to whatever they contained when the list was created. This state of the data is called the "initial" state for an item, and it can be useful when you're loading prompts or other static or default data.

The load method (or calling routine, if no load method exists) should honor requests from L_PROCESS immediately. When loading is requested, you must do the following:

- ▶ Obtain the necessary data.
- ▶ Display any appropriate information in the associated input window.
- ▶ Load the optional *data* argument with the associated non-window data.
- ▶ Set the optional *disabled* flag to indicate the disabled state of the item.
- ▶ Return to L_PROCESS again with *request* still set to the load request (which acknowledges the request).

Between the time you get a load bottom or load top request and the time you return to L_PROCESS again, do not do any screen I/O (for example, M_PROCESS or U_UPDATE).

If no more items are available for the requested load, the load method (or calling routine) must return to L_PROCESS with *request* set to D_LEOF, to signal this state to the list processor. If no items have been loaded into the list, the processor will leave only one null item in the list, unless the NONULL option was specified for this list's class. In the latter case, the D_LABORT flag will be returned to indicate a fatal error.

This loop continues until you either send a D_LEOF flag or load the first window full of items. When one of these events occurs, the list processor repositions the list to the first item loaded.

If D_LABORT is returned, a fatal list error has occurred and you will need to delete the list. You can call L_STATUS to get the reason for the error.

The D_LEOF request tells the processor that no more items are available for loading at the requested end of the list.

D_LUP and D_LDOWN tell the processor to move the list's positional indicator to the previous or next item in the list and to make that item the current one. You cannot move to a previously deleted item. If the indicator is already positioned at the end of the list, the keyboard will beep. If the indicator is already positioned at the end of a page, the list will be scrolled one item in the desired direction.

The horizontal display can change in one of three ways:

- ▶ You call L_VIEW.
- ▶ UI Toolkit adjusts the input window display automatically if the user moves to a field that was previously off the screen.
- ▶ You send a D_LLEFT or D_LRIGHT request to the L_PROCESS subroutine.

The D_LLEFT and D_LRIGHT requests only have an effect if the display area is narrower than the full window. They tell the processor to move the display area of the list to the left or to the right. The display area of the associated input window moves as well. The number of characters that move at one time is half the width of the display area.

Movement can only occur within the display area boundaries. If the current position is on the left or right boundary, no movement occurs in that direction.

The D_LTOP and D_LBOTTOM requests tell the processor to position to the first or last item in the list, respectively. If the indicator is already positioned at the specified end of the list, nothing will happen.

The D_LPGUP and D_LPGDN requests tell the processor to position up or down one page, respectively. A page is defined as the number of visible items. If the top or bottom of the list is encountered before the appropriate number of items has been reached, the first or last item in the list will become the current item.

The D_LPGTOP and D_LPGBOT requests tell the processor to position to the first or last item that is currently visible. If the indicator is already on that item, nothing will happen.

The D_LINSERT and D_LAPPEND requests tell the processor to open a new item before or after the current item, respectively, and make the new item the current item. If you want to insert or append an item somewhere other than adjacent to the current item, you must first move to that location and then use D_LAPPEND or D_LINSERT.

The D_LDELITEM flag tells the processor to delete the current item from the list. The next item, if one exists, becomes the current item. If no next item exists, the previous item becomes the current item. If the only remaining item in a list is deleted, the processor will leave only one null item in the list, unless the NONULL option was specified for this list's class. In the latter case, the D_LABORT flag will be returned to request that the list be deleted.

List Routines

L_PROCESS

The D_LMOVEUP and D_LMOVEDN requests switch the position of the current item with the previous or next item in the list, respectively. If either the beginning or end of a page is encountered, the current item remains at the page boundary, and the new, previously concealed item with which it is being swapped appears below or above that item, with all other visible items scrolled appropriately.

The D_LFIND flag tells the processor to find the next item that matches the criteria set up by any previous calls to the L_FINDSPEC subroutine. (See [L_FINDSPEC on page 9-88](#) for more information.) If a matching item is not found, the current item remains the same. You can find out whether D_LFIND was successful by calling the L_STATUS subroutine.

The D_LCANCEL (or D_LRESTORE) flag tells the processor not to update the list from the input window and the *data* argument. Instead, the input window and *data* are loaded with the previous contents of the current item. (The list will display the same current item that was there before, instead of the data that was just passed.) Any optional disabled flag and input window controls will be updated if you specified the INPUT_SNAPSHOT option when you created the list class. (See [.LISTCLASS on page 2-82](#) for information about INPUT_SNAPSHOT.)

D_LEXTENDBOT and D_LEXTENDTOP tell the processor to reopen the previously closed bottom or top of the list, respectively. Even if D_LEOF was passed back to L_PROCESS on a previous load request for the bottom or top of the list, the processor can now request that more items be loaded at that end of the list.

See also

[L_CHR on page 9-66](#), [L_INPFLD on page 9-98](#), [L_INPUT on page 9-101](#), and [L_SELECT on page 9-128](#) for more information on processing input within a list.

Examples

The following example processes the list identified by **lstid**. On entry, **req** contains the request to be processed by L_PROCESS. On return, it contains any request to be processed by the calling routine. By default, it is enabled.

```
xcall l_process(lstid, req, lst_data)
```

The next example processes the list identified by **lstid**. On entry, **req** contains the request to be processed by L_PROCESS. In this example, no initial action has been requested. The paragraph **satisfy_request** will perform the request returned from L_PROCESS. This could be either to load data into the list (if no load method was specified) or to abort the list. **Satisfy_request** will set **req** to indicate the result of the performed action. The paragraph **do_input** is called only when all requests from the list processor are satisfied. It may generate requests to the list processor based on menu entries selected by the user.

```
do
  begin
    xcall l_process(lstid, req=D_LNOP, journal_rec)
    if (req) then
      call satisfy_request ;List processing requests
```

```
                                ;must be handled first
    else
        call do_input           ;Then we can do input
    end
until (g_select)               ;Unrecognized menu entry
```

L_QUEUE – Queue commands for a list



```
xcall L_QUEUE (list_id[, request, ...])
```

Arguments

list_id

The ID of the list. (n)

request

(optional) One or more of the possible requests passed to L_PROCESS. (See the list of requests in [L_PROCESS on page 9-111.](#)) (n)

Discussion

The L_QUEUE subroutine queues requests for the L_PROCESS subroutine. When queued requests are executed, note the following rules:

- ▶ Requests are executed in the sequence in which they are passed. This rule applies not only to the sequence of calls to this subroutine, but also to the request flag arguments that are passed to L_PROCESS.
- ▶ If L_PROCESS cannot execute one of the queued requests, all subsequent pending requests are canceled. For example, if you request D_LUP when positioned on the first item in the list (and no more items are available to load at the top), any request that was queued after the D_LUP request will be canceled.
- ▶ A maximum of 20 requests may be pending at one time. If a 21st request is queued, a fatal error occurs.
- ▶ While requests are being executed, L_PROCESS may require that more items be loaded. Such requests must be honored exactly as they are during normal list processing.
- ▶ Except for loading new items when no load method is supplied, L_PROCESS won't return to the calling routine until either all queued requests have been performed or one of the requests could not be performed. For this reason, no "queue cancel" functionality is provided.

Examples

The following example will move to the bottom of the list before processing the append request passed directly to L_PROCESS. This example does not use a load method, so it must honor load requests.

```
xcall l_queue(list_id, D_LBOTTOM) ;Queue a bottom request
req = D_LAPPEND                  ;Insert at the end
call process                     ;Process the requests
```



```

process,
do
  begin
    xcall l_process(list_id, req, list_rec)
    case req of
      begincase
;D_LLOADBOT:
          call load      ;Load requested item
;D_LLOADTOP:
          req = D_LEOF   ;Tell L_PROCESS "none available"
;D_LABORT:
          req = D_LNOP   ;Force exit from this loop
        endcase
      end
    until (req.eq.D_LNOP)
  return

```

The routine would be simplified by using a load method:

```

xcall l_queue(list_id, D_LBOTTOM)
req = D_LAPPEND
xcall l_process(list_id, req, list_rec)
return

```

L_REMOVE – Remove a list

WT	WN	U	V

```
xcall L_REMOVE(list_id)
```

Arguments

list_id

The ID of the list to remove. (**n**)

Discussion

The L_REMOVE subroutine removes a list.

If you remove a list that was placed at a prior environment level, a call to E_EXIT will automatically place the list again.

See also

[L_PLACE](#) on page 9-109 for more information on placing a list.

Examples

This example removes the list whose ID is stored in **file_lst**.

```
xcall l_remove(file_lst)
```

L_RESIZE – Resize a list



```
xcall L_RESIZE (list_id, [items][, columns], {reserved})
```

Arguments

list_id

The ID of the list to resize. **(n)**

items

(optional) The number of items visible at a time. **(n)**

columns

(optional) The number of columns visible at a time. **(n)**

{reserved}

Reserved for internal use. **(n)**

Discussion

The L_RESIZE subroutine resizes a list, even if the list is on a tab.

If *items* and/or *columns* is not passed, that current value for the list is not affected.

If *columns* exceeds the number of columns in the input window associated with the list, *columns* is automatically limited.

Examples

The following example resizes the list, whose ID is stored in **order_lst**, to have 5 items and the first 7 columns visible.

```
xcall l_resize(order_lst, 5, 7)
```

L_RESTART – Restart a list

WT	WN	U	V

```
xcall L_RESTART (list_id, [initial][, last])
```

Arguments

list_id

The ID of the list. (**n**)

initial

(optional) The number of the initial item to load. (**n**)

last

(optional) The estimated number of items that will be loaded. (**n**)

Discussion

The L_RESTART subroutine “restarts” a list by erasing all existing items and setting up the list to start loading the first window of items. It also cancels any pending requests for L_PROCESS and for ActiveX controls, resets individual cell overrides for color and font.

The *initial* argument determines how list items are loaded into the list. If *initial* has a value of 1, existing items will only be loaded at the bottom of the list, forward. If *initial* is greater than 1 and equal to *last*, existing items will only be loaded at the top of the list, backward. If *initial* is greater than 1 but less than *last*, existing items can be loaded at either end, but loading will begin at the bottom, forward.

If *initial* is not passed, the item-loading method defaults to the method that was specified on the previous call to L_CREATE or L_RESTART.

If *last* is not passed, or if it has a value of zero, the calling routine cannot determine the number of list items. Instead, Toolkit assumes *last* is greater than *initial* and determines this number internally.

No other features of the list are modified by L_RESTART; the title, headers, footers, enabled state, find specification, and definition of the initial state of an item all remain the same.

Examples

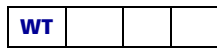
The following restarts the list whose ID is **mylist**. The item-loading method will be the one used when the list was last restarted or created. The number of items to be loaded is unknown.

```
xcall l_restart(mylist)
```

The next example restarts the list whose ID is **mylist**. The items will be loaded in reverse order. The approximate number of items to be loaded is contained in **nmitms**.

```
xcall l_restart(mylist, nmitms, nmitms)
```

L_RETURNKEY – Set the return state of privatized keys



```
xcall L_RETURNKEY (list_id, [key_code, return][, ...])
```

Arguments

list_id

The ID of the list. (n)

key_code(optional) One or more of the virtual key codes (defined in **tools.def**): (n)**D_VKUP** UP arrow key.**D_VKDOWN** DOWN arrow key.**D_VKLEFT** LEFT arrow key.**D_VKRIGHT** RIGHT arrow key.**D_VKPRIOR** PAGE UP key.**D_VKNEXT** PAGE DOWN key.**D_VKHOME** HOME key.**D_VKEND** END key.*return*

(optional) One of the following flags: (n)

TRUE The key should be returned out of the list.**FALSE** The key should be private to the list. (default)

Discussion

The L_RETURNKEY subroutine enables you to use keys that are privatized for lists as menu entry shortcuts. (On UNIX and OpenVMS, this routine is ignored because all keys are returned and can potentially be used as shortcuts to menu entries.)

By default, privatized keys, which correspond to the key codes listed above, will not signal menu entries when in a list on Windows, but will instead perform their privatized functions for the list. (See [“Appendix C: Keyboard Navigation for Lists on Windows”](#) for details.) To circumvent the privatized behavior of a key and make the key available as a menu entry shortcut, set *return* to TRUE for the key.

List Routines

L_RETURNKEY

To use one of the reserved privatized keys as a menu entry shortcut, call `L_RETURNKEY` before testing `g_entnam` for the menu. In the `L_RETURNKEY` call, pass the corresponding key code followed by `TRUE`. For example:

```
xcall l_returnkey(listid, D_VKUP, TRUE)
```

To reset a key's behavior to its privatized function (and let the list handle the key), pass the key code followed by `FALSE`. For example:

```
xcall l_returnkey(listid, D_VKUP, FALSE)
```

You can call this subroutine multiple times for the same list and/or key, contextually altering the behavior of a key within your application.

Examples

The following example specifies that for `listid`, the `HOME` and `END` keys are available to the application and may be mapped to menu entries. The up arrow and down arrow keys, on the other hand, have their usual behavior of being interpreted immediately by the list control. The other unspecified virtual keys will have their behavior unchanged.

```
xcall l_returnkey(listid, D_VKHOME, TRUE, D_VKEND, TRUE, D_VKUP, FALSE,  
&                      D_VKDOWN, FALSE)
```

L_SECT – Place text in a list header, footer, or title



```
xcall L_SECT(list_id, [text][, text_flag, ...])
```

Arguments

list_id

The ID of the list in which to place text. (**n**)

text

(optional) The text to display. (**a**)

text_flag

(optional) One or more of the following text flags: (**n**)

D_HEADER	Display text in the header section. (default)
D_FOOTER	Display text in the footer section.
D_TITLE	Display text in the title.
D_POSITION , <i>row</i> , <i>col</i>	The text's position, where <i>row</i> is the row position and <i>column</i> is the column position.
D_CLEAR	Clear the entire line of text. (default)
D_CLRBOL	Clear to the beginning of the line.
D_CLREOL	Clear to the end of the line.
D_NOCLEAR	Don't clear the line of text.
D_COLRESIZE	Resize ActiveX list columns if the user hasn't manually resized them. (default)
D_COLNORESIZE	Don't resize ActiveX list columns.
D_COLFORCESIZE	Resize ActiveX list columns even if the user has manually resized them.
D_DISPLAY	Display text at the current position. (default)
D_LEFT	Left-justify the text.
D_CENTER	Center the text.
D_RIGHT	Right-justify the text.

D_VLINE

Convert “|” characters to vertical lines for ActiveX lists or vertical bar line-drawing characters on UNIX or OpenVMS. (default)

D_NOVLINE

Don’t convert “|” characters to vertical lines for ActiveX lists or vertical bar line-drawing characters on UNIX or OpenVMS.

Discussion

The L_SECT subroutine clears text from and/or places text into a list header, footer, or title. Note that by default,

- ▶ text is displayed at the current position (D_DISPLAY), which is generally left-justified, unless a previous call to L_SECT has changed the current position.
- ▶ L_SECT clears the entire affected row (D_CLEAR) unless you specify one of the other clearing flags (D_CLRBOL, D_CLREOL, D_NOCLEAR).

The D_POSITION flag positions the text by specifying the row and column on which the text should begin. Note that Toolkit positions the text before it clears or justifies text, so justification nullifies a *col* setting. Additionally, if you pass D_POSITION, you must pass both *row* and *col*.

On Windows, the Windows theme in use determines the location of the title; D_TITLE, has no effect.

When using D_HEADER or D_FOOTER with an ActiveX Toolkit list,

- ▶ use vertical bars (|) to explicitly divide column headings or footers and to set the minimum column width. If you don’t use vertical bars (or if you pass D_NOVLINE), Toolkit will divide the text in ways that may not make sense.
- ▶ don’t use D_CENTER or D_RIGHT, which aren’t designed to be used in this situation.

By default, ActiveX Toolkit list columns resize automatically when a column header or footer is changed, unless the user has manually resized a column. You can change this behavior by passing D_COLNORESIZE to prevent resizing or D_COLFORCESIZE to force resizing even when a user has resized a column.

On UNIX and OpenVMS, only the D_LEFT, D_CENTER, and D_RIGHT flags have any effect for D_TITLE, and the text is centered by default.

Examples

The following clears the list header by specified by **mylst** and then centers the message within it.

```
xcall l_sect(mylst, "No available items", D_HEADER, D_CLEAR, D_CENTER)
```


L_SECTDRAW – Draw a line in a list header or footer



```
xcall L_SECTDRAW (list_id, section, [direction], row, column, length)
```

Arguments

list_id

The ID of the list. (**n**)

section

The section of the list in which to draw the line: (**n**)

D_HEADER Header section.

D_FOOTER Footer section.

direction

(optional) The direction of the line: (**n**)

D_VERTICAL Vertical line. (default)

D_HORIZONTAL Horizontal line.

row

The starting row position. (**n**)

column

The starting column position. (**n**)

length

The length of the line to draw. (**n**)

Discussion

On UNIX and OpenVMS, the L_SECTDRAW subroutine draws a horizontal or vertical line in a list header or footer. On Windows, L_SECTDRAW either adds unexpected characters to the header or footer or is ignored. For standard (non-ActiveX) lists, L_SECTDRAW may add unexpected characters if it is followed by an L_SECT call that doesn't erase existing text. L_SECTDRAW is ignored for ActiveX Toolkit lists.

Examples

The following example draws a vertical line in the footer of the list identified by **lstid**. The line starts at row 1, column 5 of the footer and is two characters long.

```
xcall l_sectdraw(lstid, D_FOOTER,, 1,5,2)
```

L_SELECT – Do list selection processing



```
xcall L_SELECT (list_id, request, [data], [disabled], [skip_disabled], [find_prompt],
&      [find_length], [character], [select_column], [help_id], [pre_input_routine],
&      [find_window], [find_init], [find_column], [wait_time]
&      [, a_methoddata1, ..., a_methoddata20])
```

Arguments

list_id

The ID of the list. **(n)**

request

The request flag. **(n)**

Passed requests that L_SELECT can accept are the same requests honored by L_PROCESS. (See [L_PROCESS on page 9-111](#) for more information on these requests.) These are the returned requests that L_SELECT can generate:

D_LNOP Item was selected, or unknown menu entry was selected.

D_LLOADBOT Need an item at the bottom.

D_LLOADTOP Need an item at the top.

D_LABORT Cannot proceed; delete the list.

data

(optional) The associated non-window data. **(a)**

disabled

(optional) The disabled item status. **(n)**

skip_disabled

(optional) The true/false flag that indicates whether disabled items will be skipped. **(n)**

find_prompt

(optional) The prompt to the user to enter a string for which to search. **(a)**

find_length

(optional) The maximum length (between 1 and 256 characters) allowed for a search string. **(n)**

character

(optional) The ASCII value of the character the user typed. **(n)**

select_column(optional) The selection column ID. **(n)***help_id*(optional) The help identifier. **(a)***pre_input_routine*(optional) The name of a subroutine to call before character input is performed. **(a)***find_window*(optional) The ID of the input window for find. **(n)***find_init*(optional) The true/false flag that indicates whether L_SELECT will initialize the find window. **(n)***find_column*(optional) The input column for find. **(n)***wait_time*(optional) The time-out limit for I/O processing to be completed before returning to the calling routine. **(n)****D_FOREVER** or **-1**

Never time out.

D_GLOBAL or **-2**Use the global value (see [g_wait_time](#) on page 18-6). (default)**D_IMMEDIATE** or **0**

Time out immediately.

*n*Wait up to *n* seconds (where *n* is a positive value) for I/O processing to be complete.*a_methoddata1* through *a_methoddata20*(optional) Up to 20 additional data arguments that can be passed to L_SELECT. L_SELECT, in turn, passes them directly to each field's methods. This allows you to communicate additional information to your input field methods. **(any)**

Discussion

The L_SELECT subroutine provides selection processing similar to S_SELECT, except using a list. The list (or some part of it) is presented to the user, and the user can navigate the list, find specific values, and finally, select an item. The advantage to using a list is that it can be virtually any length, but only the necessary items are loaded.

If the list is an ActiveX list, when a user selects a cell in the list, by default only the cell is highlighted. You can, however, instruct Toolkit to highlight the entire row by setting the RowMode property to any non-zero value. For more information, see [“Published Properties for ActiveX Toolkit Lists”](#) on page 9-17.

List Routines

L_SELECT

List_id is the ID of a list that was previously created with the L_CREATE subroutine. (Refer to [L_CREATE on page 9-75](#) for a description of the options available.)

Request is the request flag for the L_PROCESS subroutine. The L_SELECT subroutine translates menu entries into request flag values, but the calling routine must either specify the name of the LLOAD_METHOD subroutine for loading list items or provide the items to load into the window itself. LLOAD_METHOD can be specified using the .LISTCLASS script command, or the L_CLASS or L_METHOD subroutines.

Request should be set to D_LNOP (0) the first time L_SELECT is called. L_SELECT will call the load method, or return with *request* set to either D_LLOADBOT or D_LLOADTOP (depending on the options set when the list was created) until no more items are currently needed. The load method or the calling routine must do either one of the following:

- ▶ Retrieve another item, display the data in the associated input window, fill in data appropriately, set the optional disabled flag, and call L_SELECT again with *request* still set to the load request value.
- ▶ If no more items are available at the requested end of the list, set *request* to D_LEOF and call L_SELECT again.

Between the time you get a load bottom or load top request and the time you return to L_SELECT, do not do any screen I/O (for example, M_PROCESS or U_UPDATE).

If a load method has been specified, L_SELECT calls it every time a new list item is needed, and does *not* return to the calling routine.

If L_SELECT returns *request* set to D_LABORT, an unsupportable situation has occurred, and the calling routine should delete the list.

L_SELECT honors the following menu entries and translates them into list requests:

L_SELECT Menu Translations	
Menu entry	List request
S_UP	D_LUP
S_DOWN	D_LDOWN
S_LEFT ^a	D_LLEFT
S_RIGHT ^a	D_LRIGHT
S_TOP	D_LTOP
S_BOTTOM	D_LBOTTOM
S_PGUP	D_LPGUP
S_PGDN	D_LPGDN

L_SELECT Menu Translations (Continued)	
Menu entry	List request
S_PGTOP	D_LPGTOP
S_PGBOT	D_LPGBOT
S_FIND	D_LFIND (after prompting for a find string and calling L_FINDSPEC)
S_FINDNEXT	D_LFIND
S_FINDDIR	(Toggle find direction)

- a. Must first be activated using L_RETURNKEY.

You can also translate menu entries yourself. If the user selects a menu entry that L_SELECT does not recognize, the calling routine can map it to any request code recognized by L_PROCESS.

As shown in the above table, both the S_FIND and S_FINDNEXT menu entries use D_LFIND, but S_FIND's operation does some additional processing. S_FIND first pops up an input window for the user to input specifications for the find operation, then it calls L_FINDSPEC to set the characteristics, and lastly it calls L_PROCESS. See [L_PROCESS on page 9-111](#) and [L_FINDSPEC on page 9-88](#) for more details. You can call L_FINDSPEC before calling L_SELECT to establish "find" defaults.

When either an S_FIND or an S_FINDNEXT is being performed, L_SELECT will display a message on the information line. This message is contained in the **syntxt** file under the facility DTK, mnemonic FIND_MSG.

If L_SELECT will be displaying a find window (honoring S_FIND), you have three options:

- ▶ Use L_SELECT's default find window, which has the prompt "Find:", accepts 80 characters or the number of characters passed as *data* (whichever is less), and on Windows has OK and Cancel buttons. This is The default find window will display centered within the current display area.
- ▶ Create your own find window in a script file (or with the IB_XXX routines) and pass that window's ID in *find_window*.
- ▶ Create your own find window by using the %L_FINDWND function as the *find_window* argument.

If you would prefer either of the last two options (creating your own find window), see the information on *find_window* below.

Whether or not the *data* argument is passed must correspond to whether it was passed in the L_CREATE subroutine when the list was created. *Data* is required here if it was also passed to L_CREATE. If *data* is passed here but was *not* passed to L_CREATE, a fatal error occurs.

Data is the optional associated non-window data for the list. (Non-window data is additional information that you don't want displayed, but that you want to be maintained with the list.) Whenever this subroutine returns, the data for the current item is loaded in *data*, and the list's associated input window contains the display data for that item.

Disabled is a true/false flag that indicates whether the current item is disabled. Whether or not the *disabled* argument is passed must correspond to whether it was passed to other list routines such as L_CHR, L_DATA, L_INPFLD, L_INPUT and L_PROCESS. See [L_PROCESS](#) on page 9-111 for more information about the disabled item status.

If *disabled* is passed and has a value of true, the current item is disabled and cannot be selected. If *skip_disabled* is passed and has a value of true, the positional indicator cannot move to a disabled item after it is loaded; the indicator will move past the item automatically. The *disabled* argument is returned with the disabled status for the current item.

If the user presses enter, TAB, or SHIFT+TAB, or if an unknown menu entry is selected, L_SELECT returns, setting *request* to D_LNOP. The *data* argument will contain the associated data for the item, and the list's input window will contain the display data for the item. Note the following:

- ▶ On Windows, if the selection list is part of a composite window, TAB signals "C_NEXT" and SHIFT+TAB signals "C_PREV".
- ▶ If an unknown menu entry is selected, **g_select** is returned with a value of true and **g_entnam** will contain the menu entry name. Note that you must clear **g_select** after you satisfy the event.

Character helps you determine whether the user typed a printable character while positioned on an enabled item. If *character* is not passed and the user enters any key other than ENTER or a menu shortcut, the terminal bell will ring and processing will continue.

If *character* is passed, one of the following may occur:

- ▶ If the user types a printable character while positioned on an enabled item, the ASCII value of that character will be returned.
- ▶ If the user selects a menu entry or presses ENTER while positioned on an enabled item, *character* will be returned with a value of 0.
- ▶ If the user is positioned on a disabled item (which means *skip_disabled* either is not passed or has a value of false) and types a printable character or presses ENTER, the terminal bell will ring and processing will continue.

The user can select a menu entry while on a disabled item if *skip_disabled* is not passed or has a value of false, in which case the optional *character* argument will be returned with a value of zero.

On Windows, *select_column* and *find_column* are, by default, ignored if passed. These columns usually contain reserved menu entries that are automatically supported by the Windows environments.

On UNIX and OpenVMS, if *select_column* is passed, the specified selection column will be placed before selection processing occurs, and it will be removed on exit. If *find_column* is passed, the menu column with that ID is placed before performing input to *find_window*, and removed afterwards.

The default behavior for each environment's handling of columns is controlled by the **g_plc_col_args** flag, which is defined in the distributed **tkctl.def** file. See [g_plc_col_args on page 18-4](#) for information.

If the operation times out, **g_select** is set to true and **g_entnam** set to the entry name specified in **g_time_entry**, which is defined in **tkctl.def**. See [g_time_entry on page 18-6](#).

If *pre_input_routine* is passed, L_SELECT will pass the specified pre-input routine the list ID. If a menu entry is selected while in the pre-input routine, L_SELECT will honor the menu name if L_SELECT recognizes it. If L_SELECT doesn't recognize the name, L_SELECT will return the name to the calling routine.

If specified, the *pre_input_routine* must be available. In other words, it must either be contained within the current program image, or in an executable library (ELB) that was either linked to or opened with OPENELB. The default state, if no *pre_input_routine* is specified anywhere, is for L_SELECT to return to the caller when input is ready to be performed.

The L_SELECT subroutine turns both echo and cursor off. You may want to call E_ENTER and E_EXIT around the call to this subroutine, because L_SELECT doesn't call E_ENTER, nor does it reset the echo and cursor parameters when it's done.

You can pass *find_window* to replace L_SELECT's default find window. *Find_window* must be either the ID of an input window you have created in a script file, with the IB_XXX subroutines, or with %L_FINDWND. (See [%L_FINDWND on page 9-92](#) for more information.)



If D_RETURNBTN will be enabled and the list will be part of a composite window that on Windows has buttons, you will need to add a default button to the find window. This will enable users to close the find window normally. The default find window, for example, has OK and Cancel buttons, with OK as the default.

L_SELECT places and displays the find window within the current display area when the user selects the menu entry "S_FIND". If you create your own find window with a script, placement of the find window can be specified within your script. If you create the window with the IB_XXX subroutines, you can use U_WINDOW to specify placement. If you use the %L_FINDWND function, the find window will be centered within the current display area by default. To change the placement of the %L_FINDWND window, place and remove the window (without updating the screen) *before* calling L_SELECT. When L_SELECT calls the %L_FINDWND window, it will place it at the last placement position.

List Routines

L_SELECT

When *find_window* is specified, L_SELECT calls a UI Toolkit input routine to perform the input in the *find_window*. L_SELECT passes the input routine the D_NOTERM option (to suppress the input termination message) and passes the **finddata** data structure:

```
record finddata
    finddir      ,i4      ;Find direction
    findcase     ,i4      ;Find case
    findtype     ,i4      ;Find type
    findcurr     ,i4      ;Find current item
    findpos      ,i4      ;Search area position
    findlen      ,i4      ;Search area length
    findstrng    ,a80     ;Find string
```

See the **tools.def** file for the values associated with **finddata**'s first four variables (**finddir**, **findcase**, **findtype**, and **findcurr**). **Finddata** is preloaded with the current values, as previously specified by a call to L_FINDSPEC (or the default values).

Note that even though **findstrng** is declared as **a80**, only positions **1** through **findlen** will be considered, and only the trimmed input within these bounds will be used for find matching.

Find_window does not have to provide input for or display all of **finddata**'s options. For example, if you only wanted to allow the user to input the find string, the .STRUCTURE script command for the input window would look something like:

```
.structure find, <24>, findstrng
```

If *find_init* is passed and true, the default set for the window is initialized with I_INIT. Otherwise, the previous find data is displayed into the default set using I_DISPLAY, and I_NEXT is used to position the context of that set to the first field.

See also

[WAIT on page 2-72](#) for more information on input time-outs. (WAIT is a qualifier for the .FIELD command.)

Examples

The following example processes the selection list identified by **lstid**. On entry, **req** contains the request to be processed by L_SELECT. In this example, no initial action has been requested. The paragraph **satisfy_request** will perform the request returned from L_SELECT. This could be either to load data into the list (assuming no load method is defined for the list) or to abort the list.

Satisfy_request will set **req** to indicate the result of the performed action. The paragraph **process_selection** is called only when all requests from the list processor are satisfied, and in this case either a menu entry (unrecognized by L_SELECT) has been selected or the user has pressed ENTER on an item in the list.

```
req = D_LNOP
do
    begin
        xcall l_select(lstid, req, lst_data)
```



```

        if (req) then
            call satisfy_request      ;Process list requests first
        else
            call process_selection    ;Process a selection
        end
until (g_select)          ;Unrecognized menu entry
satisfy_request,
    case req of
        begincase
;D_LLOADBOT
            call load                ;Load requested item
;D_LLOADTOP
            req = D_LEOF             ;Tell L_SELECT "none available"
;D_LABORT
            g_select = TRUE          ;Force exit from L_SELECT loop
        endcase
    return

process_selection,
    if (g_select) then
        case g_entnam of           ;Process a menu entry
            begincase
                .
                .
                .
            endcase
        else
            ;An item was selected
            g_select = TRUE          ;Force exit from L_SELECT loop
        return
    
```

The following call to L_SELECT passes the data associated with the selection list in **lst_data** and the current disabled status of the item in **disabled**. Disabled selections will be skipped and the new disabled status returned in **disabled**.

```
xcall l_select(lstid, req, lst_data, disabled, TRUE)
```

%L_SELSTYLE – Retrieve/change the selection style for a list



```
oldstyle = %L_SELSTYLE (list_id [, newstyle])
```

or

```
xcall L_SELSTYLE (list_id [, newstyle])
```

Return value

oldstyle

The previous setting for the selection style. (^VAL)

Arguments

list_id

The ID of the list whose selection style you want to change. (n)

newstyle

(optional) The new selection style to apply to the list. One of the following (defined in **tools.def**): (n)

D_OFF Do not change appearance of selected item.

D_ON Highlight selected item. (default)

D_OUTLINE Outline selected item.

Discussion

The %L_SELSTYLE function retrieves and/or changes the style for the selected item in a standard Toolkit list. (ActiveX lists ignore %L_SELSTYLE settings, but automatically have a bold frame when active.) Unless the list is part of a composite window, the default selection style, which is highlighting, is generally sufficient. However, if the list is part of a composite window, it's difficult for users to tell when the list has focus. There's no visual indication unless you change the selection style. The default method for child lists (C_METHLIST), for example, sets the selection style to D_ON (highlighting) when the method is first invoked, and then sets it to D_OUTLINE right before returning.

Note that changes made with %L_SELSTYLE remain in effect, regardless of changes in focus, until a subsequent %L_SELSTYLE call changes the selection style again.

The *oldstyle* argument enables you to retrieve the current style for the list. If you also pass *newstyle*, the *oldstyle* argument retrieves the style that the list had before this function changed it.

If *newstyle* is omitted, no change is made to the selection style.

On UNIX and OpenVMS, D_OUTLINE has the same effect as D_ON (because outlining an item can occlude other items, headers, footers, and possibly other windows). Consider using L_ENABLE and L_DISABLE on these platforms instead of %L_SELSTYLE.

See also

- ▶ [C_METHLST](#) – Process a child list on page 6-54.
- ▶ [L_DISABLE](#) – Disable a list on page 9-86.
- ▶ [L_ENABLE](#) – Enable a list on page 9-87.

Examples

See the **ctrmeths.dbl** file included in your UI Toolkit distribution. It includes code for the C_METHLST method, which uses %L_SELSTYLE.

L_STATE – Modify a list’s state



```
xcall L_STATE(list_id, state)
```

Arguments

list_id

The ID of the list to resize. (**n**)

state

One of the following list states: (**n**)

D_LNORMAL	Restored.
D_LMAXIMIZED	Maximized.
D_LMINIMIZED	Minimized.
D_LHIDDEN	Hidden.

Discussion

The L_STATE subroutine modifies the state of a list.

See also

[L_STATUS on page 9-139](#) for retrieving the state of a list.

Examples

The following example maximizes the list.

```
xcall l_state(order_lst, D_LMAXIMIZED)
```

L_STATUS – Retrieve the status of a list



```
xcall L_STATUS (list_id[, flags, ...])
```

Arguments

list_id

The ID of the list to retrieve information for. **(n)**

flags

(optional) One or more information request flags and return values: **(n)**

D_LABTRSN , <i>reason</i>	Returns the code that indicates the reason for aborting. See the tools.def file. (n)
D_LACTIVEX , <i>list_typ</i>	Returns true (1) if the list is an ActiveX Toolkit list; otherwise, returns false (0). (n)
D_LATBOTTOM , <i>bottom_flag</i>	Returns true (1) if the user is at the bottom of the list (with no more to load at the bottom); otherwise, returns false (0). (n)
D_LATPGBOT , <i>pgbot_flag</i>	Returns true (1) if the user is at the bottom of the page; otherwise, returns false (0). (n)
D_LATPGTOP , <i>pgtop_flag</i>	Returns true (1) if the user is at the top of the page; otherwise, returns false (0). (n)
D_LATTOP , <i>top_flag</i>	Returns true (1) if the user is at the top of the list (with no more to load at the top); otherwise, returns false (0). (n)
D_LAXCTRL , <i>axlist_id</i>	Returns the ID of the ActiveX control used for a list if the list is an ActiveX Toolkit list; otherwise, returns false (0). (n)
D_LCOLSIZED , <i>resized</i>	Returns true (1) if one or more columns in the list have been resized by the user and the list is an ActiveX list; otherwise returns false (0). (n)
D_LCTRID , <i>container_id</i>	Returns the list container window ID. (n)
D_LCURITM , <i>current_item</i>	Returns the current item number. (n)
D_LDATALEN , <i>data_len</i>	Returns the length of the data area passed to L_CREATE, or 0 if no data area was passed. (n)

D_LDISITEM , <i>disable_flag</i>	Returns true (1) if the current item in the list is disabled; otherwise, returns false (0). (n)
D_LENABLED , <i>enabled_flag</i>	Returns true (1) if the list is enabled; otherwise, returns false (0). (n)
D_LFINDCASE , <i>case</i>	Indicates whether the search is case sensitive or case insensitive. See the tools.def file. (n)
D_LFINDCURR , <i>test_current</i>	Indicates whether the current item will be tested as part of the search. See the tools.def file. (n)
D_LFINDDIR , <i>direction</i>	Returns the current direction in which to search. See the tools.def file. (n)
D_LFINDLEN , <i>data_length</i>	Returns the length of the area to search. (n)
D_LFINDPOS , <i>data_position</i>	Returns the current position of the area to search. (n)
D_LFINDSOURCE , <i>find_source</i>	Indicates area to search, where <i>find_source</i> is D_LFND_DATA to search non-window data or D_LFND_DISPLAY to search display data (associated input window). (n)
D_LFINDSTR , <i>data_string</i>	Returns search string last supplied to L_FINDSPEC. (a)
D_LFINDSTRLEN , <i>string_len</i>	Returns length of last search string supplied to L_FINDSPEC. (n)
D_LFINDTYPE , <i>find_type</i>	Returns current type of search for a find operation. See the tools.def file. (n)
D_LFOUND , <i>found_flag</i>	Returns true (1) if the last D_LFIND request was successful; otherwise, false (0). (n)
D_LFTRID , <i>ftr_win_id</i>	Returns the ID of the window used internally by Toolkit for the list footer. (n)
D_LFTRLNS , <i>#ftr_lns</i>	Returns the number of footer lines in the list. (n)
D_LFTRSTATIC , <i>ftr_static</i>	Returns true (1) if footers are static; otherwise returns false (0). (n)
D_LGETEVENTS , <i>method_set</i>	Returns the method set ID for a list or a value of 0 if there is no method set for the list. (n)
D_LHDRID , <i>hdr_win_id</i>	Returns the ID of the window used internally by Toolkit for the list header. (n)
D_LHDRLNS , <i>#hdr_lns</i>	Returns the number of header lines in the list. (n)

D_LHDRSTATIC , <i>hdr_static</i>	Returns true (1) if headers are static; otherwise returns false (0). (n)
D_LHEIGHT , <i>height</i>	Returns the height of the list as the number of visible items. See the tools.def file. (n)
D_LINPID , <i>input_id</i>	Returns the ID of the input window. (n)
D_LLASTITM , <i>last_item</i>	Returns the highest item number loaded. (n)
D_LLOADSTATE , <i>end, state</i>	Returns the load state for an end (top or bottom) of the list.

end is one of the following (defined in **tools.def**): (n)

D_LLOADBOT	Return the load state for the bottom of the list.
D_LLOADTOP	Return the load state for the top of the list.

state is returned as one of the following (defined in **tools.def**): (n)

DLS_BEGIN	Loading has not begun.
DLS_DONE	Received D_LEOF at the specified end.
DLS_FULL	No more items are needed at this time.
DLS_LOAD	Loading is in progress.

D_LMETHOD , <i>methodtype, method</i>	Returns the name or address for a list method, or the ID for the %UWNDEVENTS_METHOD set.

methodtype is one of the following (defined in **tools.def**): (n)

D_ARRIVE	Return the arrive method name or address.
D_LDBLCLK	Return the double-click method name or address.
D_LLEAVE	Return the leave method name or address.
D_LLOAD	Return the load method name or address.
D_LWNDEVENT	Return the %UWNDEVENTS_METHOD set ID.

For D_LWNDEVENT, *method* returns the ID of the %UWNDEVENTS_METHOD set. For other *methodtypes*, *method* returns the name of the method if the argument is **a** or the address of the method if it is **n**. (*Method* is treated as **n** for D_LWNDEVENT, so if it is declared **a**, the ID is returned as a decimal value.) (**a** or **n**)

D_LNULL , <i>null_flag</i>	Returns true (1) if the current item is null; otherwise, returns false (0). (n)
D_LONENABLED , <i>onenable_flag</i>	Indicates whether the current item as well as the list as a whole, are enabled. See the tools.def file. (n)

List Routines

L_STATUS

D_LPLACED , <i>placed_flag</i>	Returns true (1) if the list is placed; otherwise, returns false (0). (n)
D_LPLACEMENT , <i>row, col</i>	Returns the list row (relative to the screen body) where the list is placed, and the column where the list is placed. (n)
D_LPRISTINE , <i>data, pris_flag</i>	Indicates whether the input window and data buffer are in their initial state (the state they were in when L_CREATE was called).
<i>data</i> is the data buffer, supplied by the calling routine. (See the Discussion below.) (a)	
<i>pris_flag</i> returns true (1) if the data buffer and input window are in their initial state; otherwise, it returns false (0). (n)	
D_LQAVAIL , <i>requests</i>	Returns the number of requests that can be added to the request queue via L_QUEUE before the limit of 20 requests is exceeded. Note that <i>requests</i> will be one less than the number of requests that can be submitted (because L_PROCESS/L_SELECT use one entry). In other words, if there are three requests in the queue, <i>requests</i> will be returned as 16, though you can actually submit 17 requests. (n)
D_LROW , <i>item_row</i>	Returns the current item's row number, which specifies the item's position in the currently visible set of rows. (n)
D_LSTATE , <i>state</i>	Returns one of the following, indicating the state of the list (defined in tools.def): (n)
D_LNORMAL	List is in normal state.
D_LMAXIMIZED	List is maximized.
D_LMINIMIZED	List is minimized.
D_LHIDDEN	List is hidden.

Discussion

The L_STATUS subroutine retrieves information about the current status of a list.

Each information request flag must be followed by its argument field(s). You can specify a maximum of 40 information request flags and return value fields in one call to L_STATUS.

The *data* argument required by the D_LPRISTINE flag is supplied by the calling routine rather than returned to it. *Data* is required here if the data argument was also passed to L_CREATE when the list was created. If *data* is passed here but was not passed to L_CREATE, a fatal error occurs.

The L_STATUS subroutine provides the only reliable means of obtaining information about the current state of a list. Attempts to retrieve or infer such information by other means may have unpredictable results. For instance, you might think that the placement of an associated input window could help you obtain the current item's location within the visible list. However, this assumption might not always be true. (To keep the input window's movement to a minimum, the window may not always be synchronized with the current item.)

Examples

The following example retrieves the placement values and placement status of the list whose ID is stored in **order_lst**.

```
xcall l_status(order_lst, D_LPLACEMENT, row, column, D_LPLACED, placed)
```

The next example returns the reason for the abort flag, returned from a previous call to L_PROCESS, on the list whose ID is stored in **mylst**.

```
xcall l_status(mylst, D_LABTRSN, abort_code)
```

The next example submits a FIND request to the list processor and then calls L_STATUS to determine whether the FIND was successful. The find status is returned in **found**.

```
xcall l_findspec(lstid, fname)
req = D_LFIND                               ;Submit a FIND request
xcall l_process(lstid, req, data)             ;Process the request
xcall l_status(lstid, D_LFOUND, found)        ;Was it found?
if(.not.found)
    xcall u_message("Item not found")
```

%L_USER – Set/retrieve user data associated with a list

WT	WN	U	V

```
status = %L_USER (list_id, D_LOAD | D_COPY, data)
```

or

```
xcall L_USER (list_id, D_LOAD | D_COPY, data)
```

Return value

status

Returns true if successful, otherwise false. (^VAL)

Arguments

list_id

The ID of the list for which to set/retrieve user data. (n)

D_LOAD

Set the user data; overrides any previous user data. (n)

D_COPY

Get the user data; returns false if there is no data associated with the list. (n)

data

The user data to store with the list on D_LOAD, or the user data retrieved from the list on D_COPY. *Data* can be from 1 to 65,535 characters. (a)

Discussion

The %L_USER function enables you to set or retrieve user data associated with a list. This data could be used, for example, by the list load method.

Examples

The following example associates the user data in **lstdata** with the list stored in **lstid**.

```
record lstdata
    itemnum      ,d3
    nmflds       ,d4

proc
    clear itemnum, nmflds
    xcall l_user(lstid, D_LOAD, lstdata)
```

The next example shows code from the list load method, which retrieves the user data associated with the list **a_lstid** (passed as an argument to the load method) and puts it into **lstdata**.

```
record lstdata
  itemnum      ,d3
  nmflds       ,d4
record
  sts          ,i4

proc
  sts = %L_USER(a_lstid, D_COPY, lstdata)
  if (sts) .or. (itemnum.eq.nmflds)
    a_req = D_LEOF
```

L_VIEW – Shift the horizontal display area of a list



```
xcall L_VIEW(list_id, column)
```

Arguments

list_id

The ID of the list. (n)

column

The column position of the new display area. (n)

Discussion

The L_VIEW subroutine shifts the horizontal display area of a list.

The horizontal display can change in one of three ways:

- ▶ You call L_VIEW.
- ▶ UI Toolkit adjusts the input window display automatically if the user moves to a field that was previously off the screen.
- ▶ You send a D_LLEFT or D_LRIGHT request to the L_PROCESS subroutine.

L_VIEW is only used to shift the contents of the display area to the left or to the right, not to change its size. The display width of a list is fixed once the list is created. If *column* is less than 1, a column position of 1 will be used. If *column* is greater than the result of the following equation, the result of the equation will be used instead:

$$\text{window\_width} - \text{display\_area\_width} + 1$$

In other words, the column in which the display area begins is automatically limited to boundaries that do not require changing the display area's width.

Examples

The following example adjusts the column position of the list whose ID is stored in **desc_lst** to 20.

```
xcall l_view(desc_lst, 20)
```

The List Methods

This section contains specification pages for list method subroutines. These method subroutines are written by you and their names specified as arguments to UI Toolkit subroutines (such as `L_CLASS` or `L_METHOD`) or as qualifiers to the `.LISTCLASS` script command. Toolkit will then call them at the appropriate time.

Method subroutines are important in applications that will run in Windows environments, and they are also supported in UNIX and OpenVMS environments.

The following list methods are supported:

`LARRIVE_METHOD/LLEAVE_METHOD` – Perform list item arrive or leave processing

`LDBLCLICK_METHOD` – Perform list item double-click processing

`LLOAD_METHOD` – Perform list item “load” processing

LARRIVE_METHOD/LLEAVE_METHOD – Perform list item arrive or leave processing



```

subroutine LARRIVE_METHOD | LLEAVE_METHOD
  a_status                , n
  a_listid                , n
  a_win_event             , n
  a_itemdata              , a
  a_inpid                 , n
  a_disabled              , n
  a_index                 , n
  a_row                   , n
  a_methoddata1           , any
  .
  .
  a_methoddata20          , any

```

Arguments

a_status

The returned status of the item. This argument is ignored and reserved for future use. **(n)**

a_listid

The ID of the list. **(n)**

a_win_event

The windows-event flag, with the following possible values: **(n)**

TRUE If called in response to a Windows list box control event.

FALSE If called in response to a program request to L_PROCESS.

a_itemdata

(optional) The item data for the list. **(a)**

a_inpid

The ID of the list's associated input window. **(n)**

a_disabled

(optional) The true/false flag that indicates whether to disable the list item. **(n)**

a_index

The index of the list item (1-based). **(n)**

a_row

The item's row number within the currently visible set of rows. **(n)**

a_methoddata1 through *a_methoddata20*

(optional) Up to 20 additional data arguments. **(any)**

Discussion

LARRIVE_METHOD and LLEAVE_METHOD are subroutines that you write and name when you want your program to perform extra actions before a specific list item is highlighted or before the user leaves a specific list item, respectively.

You register the LARRIVE_METHOD and/or LLEAVE_METHOD subroutines using the .LISTCLASS script command, or the L_CLASS or L_METHOD subroutines.

If registered, LARRIVE_METHOD and/or LLEAVE_METHOD must either be contained within the current program image or reside in an executable library (ELB) that was either linked to or opened with OPENELB.

When you specify LARRIVE_METHOD or LLEAVE_METHOD qualifiers, the above arguments will be passed to the LARRIVE_METHOD and/or LLEAVE_METHOD subroutines.

For the arrive method, *a_itemdata*, *a_disabled*, *a_index*, *a_row*, and the display in the input window correspond to the item being accessed. For the leave method, they correspond to the item being left.

The *a_index* argument starts at one (1), rather than zero (0).

A_methoddata1 through *a_methoddata20* are up to 20 additional optional arguments that can be passed to L_INPUT or L_SELECT. They are passed, in turn, directly to LARRIVE_METHOD and/or LLEAVE_METHOD. This enables you to communicate additional information to LARRIVE_METHOD and LLEAVE_METHOD.



Toolkit passes the method data arguments to the current list processing routine (L_PROCESS, L_INPUT, or L_SELECT) and to the list methods. Thus, if you pass data arguments to one list processing routine, to ensure these arguments are always available, you must pass them to *all* the list processing routines.

The arrive and leave methods for a list are called only once, when the selected item changes. If no change is effected, neither method is called. Special cases include D_INSERT (which leaves and arrives, even though the new item is at the same logical index), and D_LDELITEM (which does not leave the item being deleted, but arrives at the new item that took its place). D_LMOVEUP and D_LMOVEDN do not call the arrive or leave methods, because the item is the same although its index has changed.

An attempt to move to a nonexistent item will generate a leave and arrive at the same time. Requests that are queued with L_QUEUE and processed together in a single call to L_PROCESS or L_SELECT are treated as one movement. For instance, if you queue a D_LFIND and process a

D_LDOWN, there will only be one leave (from the original item) and one arrive (to the item resulting from the D_LDOWN). If you need to arrive at the intervening points, call L_PROCESS for each operation.

WIN

The arrive and leave method subroutines are required for applications that will run in Windows environments. Windows applications can interactively change the current item in the list *within* a call to a UI Toolkit subroutine. Therefore, you must write these subroutines to support event procedures for arriving at and leaving from a list item. These method subroutines (also called “event procedures”) enable your program to perform any additional processing required.

If no arrive or leave methods are specified, the default behavior is to do nothing.

See also

- ▶ [“Overloading UI Toolkit Routines” on page 18-8](#) for more information on creating these routines.
- ▶ [“Appendix D: Methods”](#).

LDBLCLICK_METHOD – Perform list item double-click processing

WT			

```
subroutine LDBLCLICK_METHOD
  a_status                , n
  a_listid               , n
  a_data                 , a
  a_inpid                , n
  a_disabled            , n
  a_itemindex           , n
  a_row                  , n
  a_methoddata1         , any
      .
      .
  a_methoddata20        , any
```

Arguments

a_status

Returned with one of the following status flags: **(n)**

TRUE Condition handled; proceed.

FALSE Simulate an ENTER.

a_listid

The ID of the list. **(n)**

a_data

(optional) The data associated with the list item. **(a)**

a_inpid

The ID of the associated input window. **(n)**

a_disabled

(optional) The true/false flag that indicates whether to disable the list item. **(n)**

a_itemindex

The ordinal index of the list item. **(n)**

List Routines

LDBLCLICK_METHOD

a_row

The item's row number within the currently visible set of rows. (**n**)

a_methoddata1 through *a_methoddata20*

(optional) Up to 20 additional data arguments. (**any**)

Discussion

LDBLCLICK_METHOD is a subroutine that you write and name. L_PROCESS calls it to handle the event of double-clicking on a list item.

You can register the LDBLCLICK_METHOD using the .LISTCLASS script command, or the L_METHOD or L_CLASS subroutines.

On UNIX and OpenVMS, the LDBLCLICK_METHOD is allowed to be registered, but it will never be called.

If this subroutine returns *a_status* as false, an ENTER is simulated as if no method were registered.

On entry to this subroutine, **g_select** will be false. If **g_select** is true when it returns to the current routine, the menu entry contained in **g_entnam** will be resigned and if the current routine (L_SELECT or L_PROCESS) recognizes that menu entry, the menu entry will be honored by that routine. Otherwise, it will be returned to the caller. Therefore, you can allow your LDBLCLICK_METHOD to perform any standard list function (such as S_UP or S_DOWN) or cause a full return to the caller.

A_methoddata1 through *a_methoddata20* are up to 20 additional optional arguments that can be passed to L_INPUT or L_INPUT. They are passed, in turn, directly to LDBLCLICK_METHOD. This enables you to communicate additional information to LDBLCLICK_METHOD.



Toolkit passes the method data arguments to the current list processing routine (L_PROCESS, L_INPUT, or L_SELECT) and to the list methods. Thus, if you pass data arguments to one list processing routine, to ensure these arguments are always available, you must pass them to *all* the list processing routines.

See also

- ▶ [.LISTCLASS](#) on page 2-82 for specifying the LDBLCLICK_METHOD in your script file.
- ▶ [L_CLASS](#) on page 9-68 and [L_METHOD](#) on page 9-105, for specifying the LDBLCLICK_METHOD at runtime.
- ▶ [“Overloading UI Toolkit Routines”](#) on page 18-8 for more information on creating this subroutine.
- ▶ [“Appendix D: Methods”](#).

Examples

```

subroutine mydblclk
;
; Description: Double-click method for a list
;
; Arguments:
;
;         a_sts          ,n          ;Return status
;         a_listid       ,n          ;List ID
;         a_data         ,n          ;Data of item
;         a_inpid        ,n          ;Input window ID
;         a_disabled     ,n          ;Disabled?
;         a_itmindx      ,n          ;Item's index
;         a_row          ,n          ;Item's visible row

; Notes:
;
; This double-click method enables the user to select the
; desired action. It is assumed that the list is being
; processed with L_SELECT.
;
; If the user double-clicks an item in the list, a
; message box is popped up giving the user the option of:
;
; 1. Going to the end of the list (press "Yes")
; 2. Simulating a RETURN          (press "No")
; 3. Doing nothing                (press "Cancel")

.include "WND:tools.def"

proc
    a_sts = TRUE                                ;Success by default

    case %u_msgbox("Double-click encountered -- select:\n"+
&                "      Yes      = goto last\n"+
&                "      No       = simulate RETURN\n"+
&                "      Cancel = do nothing", D_MYESNOCANCEL) of
        begincase
            D_MIDYES: begin                    ;"Yes" selected, go to last item
                        g_select = TRUE
                        g_entnam = "S_BOTTOM"
                    end
            D_MIDNO: a_sts = FALSE ;"No" select, simulate RETURN
            D_MIDCANCEL: nop              ;"Cancel" just returns TRUE
        endcase
    xreturn
endsubroutine

```

LLOAD_METHOD – Perform list item “load” processing



```

subroutine LLOAD_METHOD
  a_listid          , n
  a_request         , n
  a_data            , a
  a_inpid           , n
  a_disabled        , n
  a_itemindex       , n
  a_methoddata1     , any
  .
  .
  a_methoddata20    , any

```

Arguments

a_listid

The ID of the list. **(n)**

a_request

The load request: **(n)**

Flags that LLOAD_METHOD can accept are

D_LLOADBOT Load next item at end.

D_LLOADTOP Load next item at beginning.

Flags that LLOAD_METHOD can return are

(unchanged) Load request satisfied.

D_LEOF No more items.

a_data

(optional) The data associated with the item. **(a)**

a_inpid

The ID of the associated input window. **(n)**

a_disabled

(optional) The true/false flag that indicates whether to disable the item. **(n)**

a_itemindex

The index of the item desired. **(n)**

a_methoddata1 through *a_methoddata20*

(optional) Up to 20 additional data arguments. (**any**)

Discussion

LLOAD_METHOD is a subroutine that you write and name. L_PROCESS calls it when it needs to load new items into the list.

You can register the LLOAD_METHOD using the .LISTCLASS script command, or the L_CLASS or L_METHOD subroutines.

If registered, LLOAD_METHOD must either be contained within the current program image or reside in an executable library (ELB) that was either linked to or opened with OPENELB.

Your LLOAD_METHOD subroutine cannot contain any calls, direct or indirect, to L_PROCESS, L_RESTART, or L_DATA.

If a load method is not specified, L_PROCESS will use the default method, which is to return to the caller when new items are needed.

At minimum, your LLOAD_METHOD subroutine should do the following:

1. If there are any more items to be retrieved at the indicated end of the known list, do all the following:
 - ▶ Display the item into the input window whose ID is *a_inpid*.
 - ▶ If passed, load *a_data* with the corresponding list item data.
 - ▶ If passed, set the *a_disabled* flag to true if the item should be disabled. It is always false upon entry.
 - ▶ Leave *a_request* unchanged.
2. If there are no more items to be loaded, set *a_request* to D_LEOF.
3. Return.

Note that the *a_data* and *a_disabled* arguments are not necessarily the same variables originally passed to L_PROCESS or L_SELECT. These arguments may be a copy. Thus, always set the argument, rather than any global you may be using.

The argument *a_itemindex* is provided as a reference. It contains the ordinal index of the next item to be loaded for a list. If your program loads the list from the bottom only, this index will be accurate. If your program allows loading from the top, the accuracy of *a_itemindex* depends on the accuracy of the value passed as the *last* argument for L_CREATE. Note the following:

- ▶ If you are using an ActiveX Toolkit list, Toolkit must pre-fetch list items for the Spread control (which is the ActiveX control used for ActiveX Toolkit lists).

- ▶ Your program should never depend on the timing of load method invocations for specific items. If that level of control is needed, change the load method to load no items, and use `D_LAPPEND` option for `L_PROCESS` to add items as needed.
- ▶ Do not change the value of `a_itemindex`.

`A_methoddata1` through `a_methoddata20` are up to 20 additional optional arguments that can be passed to `L_INPUT` or `L_SELECT`. They are passed, in turn, directly to `LLOAD_METHOD`. This enables you to communicate additional information to `LLOAD_METHOD`.



Toolkit passes the method data arguments to the current list processing routine (`L_PROCESS`, `L_INPUT`, or `L_SELECT`) and to the list methods. Thus, if you pass data arguments to one list processing routine, to ensure these arguments are always available, you must pass them to *all* the list processing routines.

We don't recommend using `%M_SIGNAL` in list load methods. Because of the way Toolkit synchronizes menu entry signaling, Toolkit may not execute `%M_SIGNAL` calls in list load methods.

When using an ActiveX Toolkit list, it's possible for the list load method to be called without the 20 additional data arguments being passed (`a_methoddata1` to `a_methoddata20`). This can happen immediately after the list is created if the next update occurs prior to entering a list input routine. The update causes the load method to be called, and because Toolkit doesn't have access to the method data arguments, it can't pass them.

See also

- ▶ [“Overloading UI Toolkit Routines” on page 18-8](#) for more information on creating this subroutine.
- ▶ [“Appendix D: Methods”](#).

Examples

The following load method filters out unwanted records before displaying one into the list.

```
subroutine load_method
    a_listid      ,n
    a_request     ,n
    a_data        ,a
    a_inpid       ,n
    a_disabled    ,n
    a_itemindex   ,n
    .align
static record
    filchn        ,i4           ;Channel of file being loaded into list

record data
    group all_fields      ,a           ;Input data record
```

```
        tag_field      ,a5
        other_fields   ,a100
    endgroup

.define TAGVAL      , "ABC"      ;Example tag value

proc
    if (.not. filchn)
        xcall u_open(filchn, "i:i", "filename")

        ;Perform tag processing
        ;Read until we get a record we want to display

        while (tag_field .ne. TAGVAL) do
            reads(filchn, data, eof) [err=eof]

            a_data = data
            xcall i_display(a_inpid, "*CURR*", data)
            xreturn

eof,
        ;This is the last piece of data.

        a_request = D_LEOF

        ;We want to re-load the list from scratch next time.

        clear filchn
        xreturn
endsubroutine
```


10

Menu Routines

Introduction to Menus 10-2

Covers the appearance of menus, as well as how to make menu selections, process menus, and generate menu columns at runtime.

M_COLUMN – Place, remove, and delete menu columns	10-10
M_DEFCOL – Set default drop-down menu column or entry	10-12
M_DISABLE – Disable an item.....	10-13
M_ENABLE – Enable an item.....	10-15
%M_INFO – Get menu system information	10-17
M_KEYTXT – Get shortcut text	10-36
M_LDCOL – Load a menu column.....	10-37
%M_POPUP – Process a pop-up menu.....	10-39
M_PROCESS – Process a menu	10-43
M_RESET – Reset to a disabled menu bar	10-47
M_RPLENT – Replace menu entry names with their shortcuts.....	10-48
M_SCINFO – Get menu and shortcut entry information	10-50
%M_SIGNAL – Signal a menu entry.....	10-52
%M_TEXT – Modify and retrieve menu entry text	10-56
M_USER – Retrieve a user text string for a menu entry	10-58
MB_BLANK – Add a blank row to a new column.....	10-59
MB_COLUMN – Create a new column.....	10-61
MB_END – Define the end of a new column.....	10-65
MB_ENTRY – Add a menu entry to a new column	10-66
MB_LINE – Add a line to a new column.....	10-68
MB_LIST – Create a list entry in a new column.....	10-69
MB_TEXT – Add a row of text to a new column	10-70

Introduction to Menus

Toolkit menus are composed of primary and submenu columns, which are either maintained as windows in a window library file (if you define them with a script file) or are generated at runtime (if you define them with the MB_ subroutines).

The menu column name, entry names, column and entry text, and shortcut associations (key sequences that automatically select particular menu entries) are stored with the menu column. If you define them in a script, this information won't be hard-coded into your program, so it can be modified easily at any time.

The menu processing utilities in UI Toolkit enable you to load, remove, and delete columns; enable and disable menu entries and sets of entries (menu lists); and process a menu directly.

General appearance of menus

Primary menus and submenus appear either as standard drop-down menus or pop-up menus. Pop-up menus can appear at any location on the screen (see [%M_POPUP on page 10-39](#) for information). Standard drop-down menus drop down from the menu bar.

For standard drop-down menus, a single menu bar is present on the screen when a menu is not being processed. This menu bar contains the headings for each column in the menu. When the menu is processed, the menu bar is highlighted, and the user can drop down a column and select an entry from the column.

Selections in a column are aligned on the left side, aligned on the right side, or centered within the column. And drop-down primary menu columns are either left-justified, right-justified, or centered beneath the header for that column

If an entry in a menu column is associated with a submenu column, a right arrow (→) is displayed as the column's shortcut. (You can modify the character used to indicate a submenu. It is shortcut function 99 on your key map.) The user can select that entry and drop down a submenu column. Any entry in a primary menu or submenu can have a submenu column associated with it. Descending into submenu columns gives the screen a "cascading" menu appearance. (See [figure 10-1](#).)



Figure 10-1. Cascading menus.

Menu columns and entries

A menu column can be global (available for deletion from any display level) or local (only the display level that created the column can delete it). Global columns are loaded to the left side of the menu bar, while local columns are loaded to the right of the global columns. See [“Using Environments” on page 7-2](#) for more information about global and local and [“Customizing the way environments place and remove columns” on page 10-9](#) for information on setting an option that changes the way local menu columns are placed and removed.

If a column entry has a shortcut, the display consists of two parts: the text of the entry and the shortcut. Shortcuts are displayed in a “column” to the right of the longest text entry. The text of a menu’s shortcut can be retrieved with M_KEYTXT.

You can link one or more column entries into a list of entries. This allows you to enable or disable multiple menu entries at once. (See [.LIST on page 2-81](#) for more information on the window building command.)

Display renditions

Five rendition states are associated with a menu:

- ▶ Enabled
- ▶ Disabled
- ▶ Selected
- ▶ Column quick-select
- ▶ Entry quick-select

When a menu is being processed, all accessible column headers and entries are displayed in the enabled rendition. The column headers and entries that you cannot access are displayed in the disabled rendition. When the menu is not being processed, the menu bar is displayed in the disabled rendition.

As you move among the columns and the entries within the columns, the currently accessed column or entry is displayed in the selected rendition.

By default, an enabled item appears in bold and reverse type, a disabled item appears in reverse type, and a selected item appears in bold type. All are color palette 2 by default.

If you want to change any of the menu renditions, use the `U_REND` subroutine, the `U_EDITREND` subroutine, or the “Renditions” function in Proto. (The Synergy UI Toolkit Control Panel also has a “Renditions” function that you can use. See [“Customizing the Look of Your Application” on page 18-24.](#))

Disabled selections

You can disable (and re-enable) entries, lists of entries, and even entire columns within a given menu. When you do so, the disabled entries or columns have a different display rendition (attribute and color) and cannot be selected.

The subroutines used to disable and enable menu items are `M_ENABLE` and `M_DISABLE`.

On Windows, disabled menu entries can be highlighted, but the user will not be able to select them.

Making a menu selection

The user has three ways to make a menu selection: arrow keys, quick-select characters, and shortcuts. (The first two options only work when the menu bar is enabled or being processed.) A menu selection can also be simulated using the `%M_SIGNAL` subroutine.

Enabling the menu bar

To initiate menu processing during terminal input, the user must press a special “process-menu” key. When this occurs, the menu is activated, the first column or the column set by the `M_DEFCOL` subroutine is automatically dropped down (unless you specified an input string on the `M_PROCESS` subroutine), and the user can select an entry. As distributed, the process-menu key

for most platforms is CTRL+P; for Windows it is the ALT key. (On Windows, the user can also use the mouse to select or drop down menus and select menu entries.) You can modify this key by supplying an EFKEY_METHOD subroutine.

The menu bar is disabled when an entry is selected or if the user presses the process-menu key a second time, without making a selection.

In UNIX and OpenVMS environments, you can remove the menu bar from the display screen using the U_BAR subroutine. If you call U_BAR, the menu bar will only be placed when the process-menu key is pressed, and it will be removed again when an entry is selected.

Selection by arrow keys

The most common way to make a menu selection is to use the arrow keys. Once the menu bar is enabled, the left and right arrow keys enable the user to move through the menu, between the various menu columns. When a column is selected (which means that the “cursor” passes over it), the possible selections for that column are dropped down in a vertical list of entries below the heading. The first entry in the column is automatically selected. The user can either press ENTER to enter this selection or use the up and down arrow keys to select a different entry within the column.

The arrow keys can be used in any combination, and they will wrap around within the menu and columns. Wrapping means that when the user is on the last entry in a column and presses the DOWN arrow key, the next entry highlighted will be the top one. Likewise, if you are viewing the rightmost menu column and you press the RIGHT arrow key, it will wrap to the leftmost menu column. Once the user has chosen an entry with the arrow keys, the user must press the ENTER key to enter the selection.

If the user selects an entry that has a submenu associated with it, the possible selections for that submenu column are dropped down in a vertical list of entries that are adjacent to the parent menu entry. The first entry in the submenu column is automatically selected. The user can press the up and down arrow keys to move among the entries of a submenu column just as in a primary menu column. After descending into submenu columns, the user can press either the RIGHT or LEFT arrow key to go to the previous menu. The user can also press the process-menu key to go to the primary menu column. (See the previous section for information on the process-menu keys available in various environments.) After returning to the parent (primary) column, the submenu column is removed from the screen.

Selection by quick-select characters

Each menu column and menu entry is associated with a quick-select character. A quick-select character is a single character that will access that entry while the menu is being processed. Usually it is related to the name or text of the entry (for example, the first letter or number of the entry). Quick-select characters are not case sensitive.

When a primary menu or submenu column is dropped down, the user can type the quick-select character to not only highlight the menu entry but also select it. If the quick-select character is not unique in the column, it simply highlights it.

If the menu bar is enabled (being processed), but no column is currently dropped down, the user can drop down a menu by typing the column's quick-select character.

Quick-select character associations are made in your menu column scripts or with the `MB_ENTRY` subroutine. See [MB_ENTRY on page 10-66](#) for more information.

In UNIX and OpenVMS environments, if a user presses an invalid quick-select character, a beep is generated.

Re-using quick-select characters

The quick-select character for a menu entry need not be unique.

If a character is duplicated within the entries of a column, the system accesses the next entry that matches, going from top to bottom and wrapping to the top after reaching the bottom.

Applying renditions to quick-select characters

The quick-select characters have their own renditions, which can be modified using `U_REND`, `U_EDITREND`, or the “Renditions” function in Proto.

There is one rendition for column quick-select characters and another for entry quick-select characters. Both are set to reverse, bold, underline and color 10 by default.

On Windows, quick-select characters are always designated by an underscore (“_”).

Adopting conventions

Although the quick-select character has no real constraints, we recommend that you adopt some conventions regarding the character and its associated text. For example, you can

- ▶ require that the quick-select character be contained within the text, and then uppercase that character (for example, “o” for “Option,” “x” for “neXt,” and so forth).
- ▶ label your entries with numbers or letters and use those labels as quick-select characters (for example, “1. Calculate,” “E - Exit”).
- ▶ flag the quick-select character by applying the same rendition (such as underlining) to it.
- ▶ always use the first character of the text as the quick-select character.

Regardless of the method you choose, it is important that your convention be logical, easy to use, and consistent across all of your applications.

For consistency with other Synergy applications, we specifically recommend that you

- ▶ use the default rendition (underline, color 10).
- ▶ make sure the quick-select character is contained in the entry or header text.
- ▶ select quick-select characters that are unique among entries in a column or among headers of placed columns.

Following these conventions for quick-select characters will help ensure that all of your Synergy applications look the same.

Shortcuts

A shortcut is a key or key sequence that automatically accesses a particular menu entry. You can use a shortcut even when the menu is not being processed.

To associate a shortcut such as a control character, function key, or arrow key with a given option, you must make an entry for that option in the current menu and assign the desired key to that entry in the menu column script.

The display codes ^, v, <, and > represent the up, down, left, and right arrow keys, respectively, in the menu entry. Arrow keys are disabled as shortcuts during menu processing and are temporarily redefined for movement within the menu.

When the user presses a shortcut, the associated menu entry is immediately selected. Pressing a shortcut is exactly equivalent to pressing the process-menu key, selecting the corresponding column, selecting the corresponding entry, and pressing the ENTER key.

The following constraints apply to shortcuts:

- ▶ Shortcuts can be used in primary menus only.
- ▶ Duplicate shortcuts are allowed, but if more than one column has the same shortcut, the entry in the rightmost column gets processed when the shortcut is pressed.
- ▶ On Windows, some keys and key combinations are *privatized* for lists. That is, they are reserved for the list and cannot be assigned as menu shortcuts without calling `L_RETURNKEY` and setting their return states to true. See “[Appendix C: Keyboard Navigation for Lists on Windows](#)” and `L_RETURNKEY` on page 9-123 for information.
- ▶ CTRL+M is not valid as a shortcut because it is a carriage return. We recommend that you also do not define any other control characters that already have a function on the system as shortcuts (for example, CTRL+S is a screen hold on many systems). Refer to [chapter 2, “Script,”](#) for more information about building menu columns.

On Windows, the PAGE UP, PAGE DOWN, HOME, END, and arrow keys are privatized for lists. That is, they are available for functions internal to the list, but aren’t available as menu shortcuts unless you call the `L_RETURNKEY` subroutine and set their return states. See `L_RETURNKEY` on page 9-123 for more information.

Simulating a menu entry or canceling a signaled menu entry

`%M_SIGNAL` enables you to either signal a menu entry programmatically or cancel a menu entry that’s been signaled programmatically or by the user. Note the following:

- ▶ If you want to invoke a menu entry from a method subroutine, we recommend using the `D_SIGNAL` operation for `%M_SIGNAL`—especially on Windows, where method subroutines are direct callbacks.

- ▶ If you signal a menu entry from a callback routine and process the menu entry in the same callback routine, use the `D_REMOVE` operation for `%M_SIGNAL` to cancel the menu entry *after* you’ve processed it. If you don’t, the menu entry will be resignaled when processing returns to the calling routine.
- ▶ We *don’t* recommend using `%M_SIGNAL` to signal a menu entry from a list load method. Because of the way Toolkit synchronizes menu entry signaling, Toolkit may not execute the menu entry.

For more information, see [%M_SIGNAL on page 10-52](#).

Processing menus

Before processing a menu column, you must load it from the window library with `M_LDCOL` or create it with the “`MB_xxx`” subroutines. If you want to have a default menu entry or have this new column be the one that is dropped down when the menu is processed, call `M_DEFCOL`.

To process a pop-up menu, use `%M_POPUP` (see [%M_POPUP on page 10-39](#)). For standard drop-down menus, you can explicitly process the menu by calling `M_PROCESS` (see [M_PROCESS on page 10-43](#)). However, you will find that you rarely need to do this because menu processing happens implicitly during input. Either way, the following global variables enable your application to determine whether a menu entry was selected, what the selection was, and if there is any user text associated with the selection.

These variables are defined in the **tools.def** file.

Check this variable...	To determine...
<code>g_select</code>	Was a menu entry selected?
<code>g_entnam</code>	If <code>g_select</code> is true, which menu entry was selected?
<code>g_entuser</code>	If <code>g_select</code> is true, what user text is associated with the menu entry?

Additional menu support subroutines include `M_COLUMN`, which places, removes, or deletes a column; `M_RESET`, which disables the menu bar; `M_SCINFO`, which returns menu and shortcut entry information; `M_TEXT`, which modifies and retrieves menu entry text; and `M_USER`, which retrieves the user text string associated with a particular menu entry.

Runtime generation of menu columns

If your application needs to generate a menu column at runtime, use the “MB_xxx” subroutines. (MB stands for “menu build”.) They provide an alternative to defining your menu columns in script files.

We recommend that you use these subroutines only when you cannot determine the contents of the menu until runtime. Loading columns from window libraries is faster and more modular than generating columns at runtime.

There is an almost one-to-one correspondence between the “MB_xxx” subroutines and the window script commands that build a column: MB_COLUMN provides essentially the same functionality as the .COLUMN window building command, MB_ENTRY provides the same functionality as the .ENTRY window building command, and so forth. (The only functionality difference is that you cannot specify user text with the “MB_xxx” routines.)

These subroutines are also called in the same order as the column window building commands. For example:

```
xcall mb_column
xcall mb_entry
.
.
.
xcall mb_end
```

Each routine has a *control* argument, which is the data area required by the menu build subroutines. *Control* must be an alpha field and may have to vary in size, depending on how many menu entries you put in your column.

If you run out of space in the *control* argument, you will get the following fatal error: “Overflow of column build control array.” This error indicates that you must increase the data area of the *control* argument.

Customizing the way environments place and remove columns

The default behavior for each environment’s handling of columns is controlled by the **g_plc_col_args** flag, which is defined in the distributed **tkctl.def** file. In UNIX and OpenVMS environments, U_START sets **g_plc_col_args** to true which tells the subroutine to place and remove the columns as needed. In Windows environments, U_START sets this flag to false which tells the subroutine to ignore any passed column arguments typically needed for input. In any environment, you can alter the value of **g_plc_col_args** after U_START is called. (Altering **g_plc_col_args** is not advisable in UNIX or OpenVMS environments because columns containing functions must be placed to make those functions available to the user.)

M_COLUMN – Place, remove, and delete menu columns



```
xcall M_COLUMN ([col_operation,] col_id[, ...])
```

Arguments

col_operation

(optional) One of the following column operations to perform: **(n)**

D_PLACE	Place menu column. (default)
D_REMOVE	Remove menu column.
D_DELETE	Delete menu column.

col_id

One or more column IDs on which to operate. It can also be one of the following: **(n)**

D_ALL	All placed columns.
D_GLOBAL	All placed global columns.
D_LOCAL	All placed local columns.

Discussion

The M_COLUMN subroutine adds, removes, or deletes a menu column.

You can specify more than one *col_operation* in the same M_COLUMN call by specifying multiple *col_operation*, *col_id* pairs. See Examples below.

The maximum number of placed primary menu columns is nine.

Col_id can either be a column ID (returned by M_LDCOL or MB_END) or one of the qualifiers listed above. Note the following:

- ▶ When using D_PLACE, you must specify an actual column ID. You cannot use D_ALL, D_GLOBAL, or D_LOCAL with D_PLACE.
- ▶ *Col_id* must be the ID of a loaded column—no matter what the *col_operation*—or a fatal Toolkit error will occur.

D_REMOVE cannot remove an unplaced column. If you use D_REMOVE, you must make sure the columns you want to remove have been placed. If the column isn't placed, Toolkit will do nothing, and there will be no indication that the remove operation failed for this reason.

D_REMOVE used with D_LOCAL removes all columns that are not global (including those belonging to any previous environment level).

D_DELETE can delete any local columns or it can delete any global columns that are unplaced in a prior environment level. If the column is placed at a prior environment level and you use D_DELETE to try to delete it, UI Toolkit will issue a fatal error: “Cannot delete window; window placed at environment level *n*.”



The maximum available memory for all loaded columns (including columns created with MB_COLUMN) is 65,565 bytes. If your application exceeds this, Toolkit will generate an error. Note that this limit includes all information about the columns: information you've specified and internal information. If you get this error, you must reduce the amount of information in the columns. For example, you might be able to reduce user text or shortcuts associated with the menu entries. (If you've defined your menu columns in a script file and there's a lot of column information, test your program by accessing *all* the column menus, including subcolumns.)

Examples

The following example places the column whose ID is stored in **util**.

```
xcall m_column(D_PLACE, util)
```

The next example removes all local columns and deletes the column whose ID is stored in **main**.

```
xcall m_column(D_REMOVE, D_LOCAL, D_DELETE, main)
```

M_DEFCOL – Set default drop-down menu column or entry



```
xcall M_DEFCOL ([default][, current])
```

Arguments

default

(optional) The default new column number or menu entry. (**a** or **n**)

current

(optional) Returned with the current default column number or menu entry. (**a** or **n**)

Discussion

The M_DEFCOL subroutine specifies which menu column will be dropped down automatically or which menu entry will be enabled when the menu is processed. You should call M_DEFCOL immediately after loading the columns. If not called, the first menu column is dropped down by default.

This subroutine determines the number of the column that will be dropped down or the menu entry that will be activated when the user presses the “process-menu” key or your program calls M_PROCESS with no arguments. (See [“Enabling the menu bar” on page 10-4](#) for available process-menu keys.)

If numeric arguments are passed, a default column is dropped down when the menu is processed. If alpha arguments are passed, a default entry is selected when the menu is processed. You can set a default menu column or menu entry but not both; setting one disables the other. To set the default menu entry, simply pass the entry name in *default*.

To set the default column, use the number 1 for the leftmost column, and so on, up to the number 9 for the rightmost column. (Specifying 9 always references the rightmost column even if fewer than 9 columns are placed.) If you specify a *default* value of 0, no column will be dropped down automatically by M_PROCESS or the process-menu key. If you specify 0, the quick-select feature will be enabled for columns as well as menu entries. Specifying a *default* value of 0 has the same effect as setting the environment variable DTK_MENU_UP.

See also

[DTK_MENU_UP](#) in the “Environment Variables” chapter of *Environment Variables & System Options* for disabling automatic drop-down menus.

Examples

In the following example, the second column from the left is the default column that is dropped down when the menu is processed.

```
xcall m_defcol(2)
```

M_DISABLE – Disable an item



```
xcall M_DISABLE ([D_ENTRY,] col_id, ent_name[, ...])
```

```
xcall M_DISABLE (D_LIST, col_id, list_name[, ...])
```

```
xcall M_DISABLE (D_COLUMN, col_id[, ...])
```

```
xcall M_DISABLE (D_SUB, sub_name, ent_name[, ...])
```

Arguments

D_ENTRY

(optional) Disables a menu entry on a placed menu column. (default)

D_LIST

Disables a list of menu entries on a placed menu column.

D_COLUMN

Disables a placed primary column.

D_SUB

Disables a submenu entry, pop-up menu entry, or entry in a submenu to a pop-up menu on a loaded menu column.

col_id

The ID of the column to be disabled or the column in which the entry or list is located. *Col_id* is returned by the M_LDCOL and MB_END subroutines. **(n)**

ent_name

The name of the entry to disable. *Ent_name* is an entry name from the .ENTRY script command or MB_ENTRY. **(a)**

list_name

The name of the list of entries to disable. *List_name* is a list name from the .LIST script command or MB_LIST. **(a)**

sub_name

The name of the submenu, pop-up menu column, or submenu to a pop-up menu that contains the menu entries or submenu columns to disable. **(a)**

Discussion

The `M_DISABLE` subroutine disables menu entries, lists of menu entries, menu columns or subcolumns, entries in pop-up menu columns, or entries in submenus to pop-up menu columns. It has four forms, which are listed above.

You can specify more than one *ent_name* or more than one *list_name*. You can also specify more than one *col_id* if the qualifier is `D_COLUMN`. But `M_DISABLE` can have no more than nine arguments.

Note the following:

- ▶ *Col_id* must be the ID of a loaded and placed column. An unloaded column will cause a fatal Toolkit error.
- ▶ *Sub_name* must be a valid menu column name. (*Sub_name* can be the name of an unplaced column or an unloaded column created using `MB_COLUMN` with the `D_NOLOAD` option.)
- ▶ If you're using `D_SUB` to enable a pop-up menu entry or a submenu to a pop-up menu, placing the column on the menu bar will cause the entry or submenu to be reset to its default state. An entry in a submenu to a pop-up menu, however, will remain enabled when the column that references the submenu is placed on the menu bar.

While your application is running, if the user selects a disabled entry in a primary column or submenu column, or if the user selects a submenu column, `M_PROCESS` will stay active but will return to the menu bar. If the user selects a disabled pop-up menu entry, `M_PROCESS` will no longer be active; control will return to the calling routine. In this case, `g_select` will be false and `g_entnam` will be blank.

On Windows, disabled menu entries can be highlighted, but the user will not be able to select them.

Examples

In the following example, the entries “ar” and “ap,” located in the column whose ID is stored in `colid(i)`, are disabled.

```
xcall m_disable(colid(i), "ar", "ap")
```

In the next example, the entries in the list `sysmgr` in the column stored in `colid(i)` are disabled.

```
xcall m_disable(D_LIST, colid(i), "sysmgr")
```

In the following example, the column whose ID is stored in `colid(i)` is disabled.

```
xcall m_disable(D_COLUMN, colid(i))
```

In the following example, the submenu entry “o_owner” is disabled in the submenu named “admin.”

```
xcall m_disable(D_SUB, "admin", "o_owner")
```

M_ENABLE – Enable an item



```
xcall M_ENABLE ([D_ENTRY,] col_id, ent_name[, ...])
```

```
xcall M_ENABLE (D_LIST, col_id, list_name[, ...])
```

```
xcall M_ENABLE (D_COLUMN, col_id[, ...])
```

```
xcall M_ENABLE (D_SUB, sub_name, ent_name[, ...])
```

Arguments

D_ENTRY

(optional) Enables a menu entry on a placed menu column. (default)

D_LIST

Enables a list of menu entries on a placed menu column.

D_COLUMN

Enables a placed primary column.

D_SUB

Enables a submenu entry, pop-up menu entry, or entry in a submenu to a pop-up menu on a loaded menu column.

col_id

The ID of the column to be enabled or the column in which the entry or list is located. *Col_id* is returned by the M_LDCOL and MB_END subroutines. **(n)**

ent_name

The name of the entry to enable. *Ent_name* is an entry name from the .ENTRY script command or MB_ENTRY. **(a)**

list_name

The name of the list of entries to enable. *List_name* is a list name from the .LIST script command or MB_LIST. **(a)**

sub_name

The name of the submenu, pop-up menu column, or submenu to a pop-up menu that contains the menu entries or submenu columns to enable. **(a)**

Discussion

The `M_ENABLE` subroutine enables menu entries, lists of menu entries, menu columns or subcolumns, entries in pop-up menu columns, or entries in submenus to pop-up menu columns. It has the four forms listed above.

You can specify more than one *ent_name* or more than one *list_name*. You can also specify more than one *col_id* if the qualifier is `D_COLUMN`. But `M_ENABLE` can have no more than nine arguments.

Note the following:

- ▶ *Col_id* must be the ID of a loaded and placed column. An unloaded column will cause a fatal Toolkit error.
- ▶ *Sub_name* must be a valid menu column name. (*Sub_name* can be the name of an unplaced column or an unloaded column created using `MB_COLUMN` with the `D_NOLOAD` option.)
- ▶ If you're using `D_SUB` to enable a pop-up menu entry or a submenu to a pop-up menu, placing the column on the menu bar will cause the entry or submenu to be reset to its default state. An entry in a submenu to a pop-up menu, however, will remain enabled when the column that references the submenu is placed on the menu bar.

Examples

The first example enables the entry “gl” located in the column whose ID is stored in **colid(i)**.

```
xcall m_enable(colid(i), "gl")
```

In the second example, the entries in the list **sysmgr** in the column stored in **colid(i)** are enabled.

```
xcall m_enable(D_LIST, colid(i), "sysmgr")
```

In the third example, the columns whose IDs are stored in **col1** and **col2** are enabled.

```
xcall m_enable(D_COLUMN, col1, col2)
```

In the last example, the submenu entry “o_owner” is enabled in the submenu named “admin.”

```
xcall m_enable(D_SUB, "admin", "o_owner")
```


%M_INFO – Get menu system information



```
status = %M_INFO (subfunction, args)
```

or

```
xcall M_INFO (subfunction, args)
```

Return value

status

An integer value specific to *subfunction*. (^VAL)

Arguments

subfunction

One of the following: (n)

D_MENU_COLUMNS	Retrieve loaded columns.
D_MENU_ENABLED	Determine whether a column or entry is enabled.
D_MENU_ENTRIES	Retrieve the entries in a column.
D_MENU_ENTRY	Retrieve information about a menu entry.
D_MENU_HEADER	Retrieve information about a column's header.
D_MENU_LISTS	Retrieve the entry lists in a column.
D_MENU_LIST	Retrieve the entries in a list.
D_MENU_SHORTCUT	Determine the entry that corresponds to a shortcut.

args

Arguments specific to *subfunction*.

Discussion

The %M_INFO routine enables you to get information about the menu system. See the individual subfunctions for details.

See also

- ▶ [M_KEYTXT](#) on page 10-36 for information on getting shortcut text.
- ▶ [M_SCINFO](#) on page 10-50 for information on getting menu and shortcut entry information.
- ▶ [%M_TEXT](#) on page 10-56 for information on modifying and retrieving menu entry text.

D_MENU_COLUMNS

numcols = %M_INFO(D_MENU_COLUMNS, [*max*], [*array*], [*env*][, *placed*])

or

xcall M_INFO(D_MENU_COLUMNS, [*max*], [*array*], [*env*][, *placed*])

Return value

numcols

The number of columns that match the default criteria or the criteria specified by the *env* or *placed* arguments. (^VAL)

Arguments

max

(optional) The maximum number of column IDs to be returned in the array specified by *array*. (n)

array

(optional) The first element of an array that will accept the returned column IDs. (n)

env

(optional) One of the following: (n)

D_ALL Include loaded columns for all environment levels.

D_GLOBAL Include loaded columns for the global environment level.

D_LOCAL Include loaded columns for the current environment level. (default)

env_level Include loaded columns for the specified environment level (1 through the number for the current environment level).

placed

(optional) One of the following: (n)

TRUE Return placed columns.

FALSE Return placed and unplaced columns. (default)

Discussion

The `D_MENU_COLUMNS` subfunction of `%M_INFO` returns the following information about loaded columns:

- ▶ The number of columns that match the default criteria or the criteria you specify. This is returned in the *numcols* return value.
- ▶ Column IDs for columns that match the criteria. This is returned in the array specified by the *array* argument. Note that you must pass both *array* and *max* to get this information. (Passing one without the other, however, does not generate an error.) The *array* argument specifies the first element in an array that will accept the column IDs. The *max* argument limits the number of column IDs that will be written to the array.

By default, `D_MENU_COLUMNS` reports the number of loaded columns (placed or not placed) at the local environment level. To change this, do one of the following:

- ▶ Pass the *env* argument to instruct the subroutine to return information on loaded columns for the global level, for the local level, for a specific level, or for all environment levels. To specify an environment level by number, pass the environment level number as *env_level*.
- ▶ Pass the *placed* argument to instruct the subroutine to return information on all loaded columns or just on placed columns. If you don't pass *placed*, this defaults to false.



Submenu columns are included only if they have been separately loaded with `M_LDCOL`. In addition, if *placed* is passed as true, submenus won't be included unless they are also placed as primary columns.

Examples

The following example returns the number of loaded columns (placed and not placed) at the local level. (It doesn't return any column IDs because we didn't pass the *array* argument.)

```
columns = %m_info(D_MENU_COLUMNS)
```

The following example returns the number of loaded columns (placed or not placed) at the local level. It returns this number in the **columns** variable. Because we passed *array* (**col_ids**) and *max* (**max_cols**), it also returns the column IDs. (**Max_cols** limits the number of returned IDs.)

```
columns = %m_info(D_MENU_COLUMNS, max_cols, col_ids)
```

The following example returns the number of loaded columns and the IDs for the loaded columns (placed or not placed) at the previous environment level.

```
prev_env = %e_info(D_ENV_LEVEL) ;Get level number for current
environment
prev_env = environment - 1 ;Decrement to get previous environment
columns = %m_info(D_MENU_COLUMNS, max_cols, col_ids, prev_env)
```

The following example returns the number of loaded columns (in **columns**) and the IDs for loaded columns that are placed at the local environment level. It returns the IDs in the array specified by **col_ids**.

```
columns = %m_info(D_MENU_COLUMNS, max_cols, col_ids,, TRUE)
```

The following example returns the number of loaded columns (in **columns**) and the IDs for loaded columns that are placed at all environment levels. It returns the IDs in the array specified by **col_ids**.

```
columns = %m_info(D_MENU_COLUMNS, max_cols, col_ids, D_ALL, TRUE)
```

D_MENU_ENABLED

enabled = %M_INFO (D_MENU_ENABLED, *colid*, [*entry*][, *env*])

or

xcall M_INFO (D_MENU_ENABLED, *colid*, [*entry*][, *env*])

Return value

enabled

Returns TRUE if the column or entry is enabled or FALSE if the column or entry is disabled. (^VAL)

Arguments

colid

The ID of a currently loaded column. (**n**)

entry

(optional) The name of the menu entry to report on. (**a**)

env

(optional) One of the following: (**n**)

D_GLOBAL Return the initial state of the column or entry.

D_LOCAL Return the state of the column or entry in the current environment. (default)

env_level Return the state of the column or entry at a specified environment level (1 through the number of the current environment level).

Discussion

The D_MENU_ENABLED subfunction of %M_INFO enables you to get the state (enabled or disabled) of a menu column or menu entry at a given environment level.

- ▶ For a menu entry, pass *entry* with the name of the menu entry. (If there is no menu entry with the name passed as *entry*, a fatal Toolkit error is generated.)
- ▶ For a menu column, omit *entry* or pass it as blank.

In either case, you must pass a valid column ID as *colid*. (Otherwise a fatal Toolkit error is generated.) The column doesn't have to be a primary column. It can be a submenu column. But if it's a submenu column from a window library, it must first be loaded using M_LDCOL with the D_NOPLC parameter. (Submenus are not loaded as menu columns during normal processing.) Note that you can pass the ID of any column created using MB_COLUMN with the D_NOLOAD option—even if it is a subcolumn.

You can also select which environment level you want the state returned for. You can use the *env* argument to instruct `D_MENU_ENABLED` to return the state at the global level (`D_GLOBAL`), the local level (`D_LOCAL`, which is the default), or a specific level (by passing the number of the level as *env_level*). Note, however, that if the column isn't placed in the specified environment (or default environment if you don't pass *env*), this function returns the initial state, the state specified when the column or entry was created.

Examples

The following example returns the current state of the column whose ID is passed as **col_id**—if the column has been placed, that is. (If the column hasn't been placed, it returns the initial state, the state specified when the column was created.) Because we don't pass the *env* argument, this example returns the column's state for the local environment.

```
state = %m_info(D_MENU_ENABLED, col_id)
```

The following example returns the current state of the `N_FINDCUST` menu entry if the **col_id** column has been placed. Note that it returns the state for the local environment because we didn't pass *env*.

```
state = %m_info(D_MENU_ENABLED, col_id, "N_FINDCUST")
```

The next example returns the initial state of the **col_id** column. It returns the initial state because we passed `D_GLOBAL` as the *env* argument.

```
state = %m_info(D_MENU_ENABLED, col_id,, D_GLOBAL)
```

The following example returns the state of the `N_FINDCUST` menu entry in the previous environment. (Assume that **prev_env** is set to the number of the previous environment level.) If **col_id** hasn't been placed in the previous environment level, however, this example returns the initial state of `N_FINDCUST`, the state specified when the column was created.

```
state = %m_info(D_MENU_ENABLED, col_id, "N_FINDCUST", prev_env)
```

D_MENU_ENTRIES

```
num_entries = %M_INFO(D_MENU_ENTRIES, colid, [max][, array])
```

or

```
xcall M_INFO(D_MENU_ENTRIES, colid, [max][, array])
```

Return value

num_entries

The number of entries in the menu column specified by *colid*. (^VAL)

Arguments

colid

The ID of a currently loaded column. (n)

max

(optional) The maximum number of entries to return in *array*. (n)

array

(optional) The first element of an array that will receive the menu entry names (up to ten characters each). (n)

Discussion

The D_MENU_ENTRIES subfunction of %M_INFO returns the number of menu entries in the menu column or submenu column specified by *colid*. If you pass *max* and *array*, it also returns the names of the menu entries in the column. (To get this information, you must pass both of these arguments. However, if you pass one without the other, you won't get an error.)

- ▶ The *array* argument specifies the first element in an array that will accept the menu entry names.
- ▶ The *max* argument specifies the maximum number of menu entry names that will be written to the array.

If *colid* is not a valid column ID, a fatal Toolkit error is generated. The column doesn't have to be a primary column. It can be a submenu column. But if it's a submenu column from a window library, it must first be loaded using M_LDCOL with the D_NOPLC parameter. (Submenus are not loaded as menu columns during normal processing.) Note that you can pass the ID of any column created using MB_COLUMN with the D_NOLOAD option—even if it is a subcolumn.

Examples

The following example returns the number of menu entries in the column whose ID was passed as **col_id**. This number is returned in **menu_entries**.

```
menu_entries = %m_info(D_MENU_ENTRIES, col_id)
```

Menu Routines

%M_INFO

The following example returns the number of menu entries in the column specified by **col_id**. It also returns the names of the menu entries in the column. These names are returned in the array whose first element is **entry_array**. The **max_entries** variable sets a limit to the number of names that can be returned.

```
menu_entries = %m_info(D_MENU_ENTRIES, col_id, max_entries, entry_array)
```


D_MENU_ENTRY

found = %M_INFO(D_MENU_ENTRY, *colid*, [*entry*], [*index*], [*text*], [*quick_select*], [*shortcut*],
& [*user*][, *noreset*])

or

xcall M_INFO(D_MENU_ENTRY, *colid*, [*entry*], [*index*], [*text*], [*quick_select*], [*shortcut*],
& [*user*][, *noreset*])

Return value

found

Returns TRUE if the menu entry is found or FALSE if the menu entry does not exist in the column. (^VAL)

Arguments

colid

The ID of a currently loaded menu column. (n)

entry

(optional) The name of the menu entry. (a)

index

(optional) The numeric position (base 1) of the menu entry. (n)

text

(optional) Returned with the text of the menu entry. (a)

quick_select

(optional) Returned with the uppercased quick-select character for the entry. (a)

shortcut

(optional) Returned with the function code (as defined in **inpctl.def**) associated with the keyboard shortcut for the menu entry, or 0 if the entry has no shortcut. (n)

user

(optional) Returned with the user text associated with the menu entry, or blank if no user text is associated with the entry. (a)

noreset

(optional) One of the following: (n)

TRUE The menu entry is set to NORESET.

FALSE The menu entry is *not* set to NORESET.

Discussion

The `D_MENU_ENTRY` subfunction of `%M_INFO` returns information about a menu entry. You must specify the menu column by passing the column ID as *colid*. And you must specify the menu entry by passing the *entry* or *index* argument.

- ▶ To use the menu entry name to specify the menu entry, pass the name as the *entry* argument (case-insensitive). If you also pass *index*, it is returned with a numeric position indicator (base 1) of the specified menu entry.
- ▶ To use a column position to specify the menu entry, pass the position number (base-1) in the *index* argument. (For example, pass 1 for the top entry in the column, pass 2 for the next entry, etc.) If you want the name of the menu entry returned in *entry*, pass *entry* as a variable that's set to blank.

In addition to retrieving the menu entry name by passing the entry's position and vice versa, you can also get the following:

- ▶ The text for the entry by passing the *text* argument.
- ▶ The quick-select character for the entry by passing the *quick_select* argument. (The quick-select character is returned as an uppercased character.)
- ▶ The numeric identifier for the entry's keyboard shortcut (as defined in **inpctl.def**) by passing the *shortcut* argument. This argument is returned as 0 if there is no shortcut for the entry or `D_SUBMENU` if a submenu is attached to the entry.
- ▶ The user text associated with the menu entry by passing the *user* argument. This argument is returned as 0 if there is no user text for the menu entry. (The `M_USER` subroutine also returns the user text for a menu entry, but `M_USER` works only with primary columns.)
- ▶ A value that indicates whether or not the menu column will be reset if the entry is selected. You can get this information by passing the *noreset* argument. (See [.ENTRY on page 2-27](#) for information on the `NORESET` qualifier that specifies this.)

If *colid* is not a valid column ID, a fatal Toolkit error is generated. The column may be a primary column or a submenu column. But if it's a submenu column from a window library, it must first be loaded using `M_LDCOL` with the `D_NOPLC` parameter. (Submenus are not loaded as menu columns during normal processing.) You can pass the ID of any column created using `MB_COLUMN` with the `D_NOLOAD` option—even if it is a subcolumn.

Examples

The following example returns a value in **col_found** that indicates whether the `O_CANCEL` menu entry exists on the menu column specified by **col_id**.

```
col_found = %m_info(D_MENU_ENTRY, col_id, "O_CANCEL")
```

As with the previous example, the following example returns a value in **col_found** that indicates whether the O_CANCEL menu entry exists on the menu column specified by **col_id**. However, it also returns the position of the entry in the column. Because *entry* is passed (“O_CANCEL”) and isn’t blank, the position is returned in **entry_num**.

```
col_found = %m_info(D_MENU_ENTRY, col_id, "O_CANCEL", entry_num)
```

Assuming **entry_name** is blank, the following example returns a value in **col_found** that indicates whether a menu entry exists at the position specified by **entry_num**. If an entry does exist at the specified position, it also returns the name of the menu entry (in **entry_name**) in the position specified by **entry_num**.

```
entry_name = ""  
col_found = %m_info(D_MENU_ENTRY, col_id, entry_name, entry_num)
```

The next example also returns a value in **col_found** that indicates whether a menu entry exists at the position specified by **entry_num**. In addition it returns the text of the entry, the uppercased quick-select character for the entry, the function number for the keyboard shortcut, any user text associated with the entry, and a value that indicates whether the column will be reset when the entry is selected. Note, however, that it doesn’t return the name of the entry because we didn’t pass the *entry* argument.

```
col_found = %m_info(D_MENU_ENTRY, col_id,, entry_num, entry_text,  
& q_select, short_num, user_text, col_reset)
```

D_MENU_HEADER

```
header_length = %M_INFO(D_MENU_HEADER, colid, [header], [quick_select], [justification]  
&                [, column_width])
```

or

```
xcall M_INFO(D_MENU_HEADER, colid, [header], [quick_select], [justification]  
&        [, column_width])
```

Return value

header_length

The number of characters in the header text for the menu column specified by *colid*. (^VAL)

Arguments

colid

The ID of a currently loaded column. (n)

header

(optional) Returned with the text of the column's header. (a)

quick_select

(optional) Returned with the uppercased quick-select character for the column header. (a)

justification

(optional) Returned with the justification for the column on UNIX and OpenVMS.
(On Windows, the column is positioned automatically.) (n)

D_JST_LEFT Menu column is left-aligned with the header.

D_JST_RIGHT Menu column is right-aligned with the header.

D_JST_CENTER Menu column is centered beneath the header.

column_width

(optional) Returned with the overall width (in characters) of the column. (n)

Discussion

The D_MENU_HEADER subroutine of %M_INFO returns the following information about a column's header:

- ▶ The number of characters in the header text (including the leading and trailing space in the header) for the column. This is returned in the *header_length* return value.
- ▶ The text of the column's header (including the leading and trailing space). This is returned in the optional *header* argument.

- ▶ The quick-select character for the column header. This is returned as an uppercased character in the optional *quick_select* argument.
- ▶ The justification of the column on UNIX and OpenVMS. This is returned in the optional *justification* argument. (On Windows, this has no effect; the column is automatically left-justified under the header.)
- ▶ The overall width, in characters, of the column. This is returned in the optional *column_width* argument. If an entry has a shortcut, five characters of this width are reserved for the shortcut text.

If *colid* is not a valid column ID, a fatal Toolkit error is generated. The column may be a primary column or a submenu column. But if it's a submenu column from a window library, it must first be loaded using `M_LDCOL` with the `D_NOPLC` parameter. (Submenus are not loaded as menu columns during normal processing.) You can pass the ID of any column created using `MB_COLUMN` with the `D_NOLOAD` option—even if it is a subcolumn.

Examples

The following example returns the number of characters in the header text for the **col_id** column.

```
h_length = %m_info(D_MENU_HEADER, col_id)
```

The next example returns the following information about the header for the **col_id** column: the number of characters in the header text (**h_length**), the text for the column's header (**h_text**), the uppercased quick-select character for the column (**q_select**), the justification of the column (**just**), and the overall width, in characters, of the column (**col_width**).

```
h_length = %m_info(D_MENU_HEADER, col_id, h_text, q_select, just,
&
                    col_width)
```

D_MENU_LISTS

num_entries = %M_INFO (D_MENU_LISTS, *colid*, [*max*][, *array*])

or

xcall M_INFO (D_MENU_LISTS, *colid*, [*max*][, *array*])

Return value

num_entries

The number of menu lists defined in the menu column specified by *colid*. (^VAL)

Arguments

colid

The ID of a currently loaded column. (n)

max

(optional) The maximum number of list names returned in *array*. (n)

array

(optional) The first element of an array that will receive the list names (up to ten characters each). (n)

Discussion

The D_MENU_LISTS subfunction of %M_INFO returns the number of menu lists defined in a menu column or submenu column specified by *colid*. If you pass the *max* and *array* arguments, it also returns the names of the lists in the column. (To get this information, you must pass both of these arguments. However, if you pass one without the other, you won't get an error.)

- ▶ The *array* argument specifies the first element in an array that will accept the list names.
- ▶ The *max* argument specifies the maximum number of list names that will be written to the array.

If *colid* is not a valid column ID, a fatal Toolkit error is generated. The column may be a primary column or a submenu column. But if it's a submenu column from a window library, it must first be loaded using M_LDCOL with the D_NOPLC parameter. (Submenus are not loaded as menu columns during normal processing.) You can pass the ID of any column created using MB_COLUMN with the D_NOLOAD option—even if it is a subcolumn.

See also

[.LIST on page 2-81](#) for information on menu lists.

Examples

The following example returns the number of menu lists defined in the column specified by **col_id**. Because we didn't pass the *max* and *array* arguments, this example won't return the names of the lists.

```
num_lists = %m_info(D_MENU_LISTS, col_id)
```

The following example returns the number of lists and the names of the lists in the column specified by **col_id**. **List_name_array** points to the first element of the array that will receive the list names, and **max_lists** specifies the maximum number of lists names that will be returned.

```
num_lists = %m_info(D_MENU_LISTS, col_id, max_lists, list_name_array)
```

D_MENU_LIST

num_entries = %M_INFO (D_MENU_LIST, *colid*, *list*, [*max*][, *array*])

or

xcall M_INFO (D_MENU_LIST, *colid*, *list*, [*max*][, *array*])

Return value

num_entries

The number of menu entries in the specified menu list, or 0 if the list doesn't exist in the menu column specified by *colid*. (^**VAL**)

Arguments

colid

The ID of a currently loaded column. (**n**)

list

The name of a menu list (case-insensitive). (**a**)

max

(optional) The maximum number of menu entry names returned in *array*. (**n**)

array

(optional) The first element of an array that will receive the names of menu entries in the list (up to ten characters each). (**n**)

Discussion

The D_MENU_LIST subfunction of %M_INFO returns the number of menu entries in a list or 0 if the menu list specified by the *list* argument is not in the column specified by *colid*. If you pass *max* and *array*, this function also returns the names of the menu entries in the list. (To get this information, you must pass both *max* and *array*. However, if you pass one without the other, you won't get an error.)

- ▶ The *array* argument specifies the first element in an array that will accept the entry names.
- ▶ The *max* argument specifies the maximum number of entry names that will be written to the array.

If *colid* is not a valid column ID, a fatal Toolkit error is generated. The column may be a primary column or a submenu column. But if it's a submenu column from a window library, it must first be loaded using M_LDCOL with the D_NOPLC parameter. (Submenus are not loaded as menu columns during normal processing.) Note that you can pass the ID of any column created using MB_COLUMN with the D_NOLOAD option—even if it is a subcolumn.

Examples

The following example returns the number of menu entries that belong to the list **list_name**. If **list_name** isn't part of the column **col_id**, the example will return 0. (Because we didn't pass *max* and *array*, this example won't return the names of the entries.)

```
num_lists = %m_info(D_MENU_LIST, col_id, list_name)
```

The following example returns the number of entries and the names of the entries that belong to the list specified by **list_name**. **List_name_array** is the first element of the array that will receive the menu entry names, and **max_lists** specifies the maximum number of entry names that will be returned.

```
num_lists = %m_info(D_MENU_LIST, col_id, list_name, max_lists,  
&list_name_array)
```

D_MENU_SHORTCUT

```
colid = %M_INFO (D_MENU_SHORTCUT, keycode, [entry][, env])
```

or

```
xcall M_INFO (D_MENU_SHORTCUT, keycode, [entry][, env])
```

Return value

colid

The ID of the menu column that contains the menu entry for the specified shortcut code, or 0 if the shortcut code isn't assigned to an entry. (^VAL)

Arguments

keycode

The shortcut code to look for. This must be one of the `_KEY` values defined in **inpctl.def**. (n)

entry

(optional) Returned with the name of the menu entry that the shortcut code is assigned to. (a)

env

(optional) One of the following: (n)

D_LOCAL The current environment level. (default)

env_level The number of an environment level (1 through the number of the current environment level).

Discussion

The `D_MENU_SHORTCUT` subfunction of `%M_INFO` returns the ID of the menu column that contains an entry for the specified shortcut code as defined in **inpctl.def**, or it returns 0 if the shortcut code isn't assigned to an entry. If you pass the *entry* argument, it also returns the name of the entry or blank if there's no entry for the shortcut.



On Windows, for F11 and F12, you can't use the shortcut codes (`F11_KEY` and `F12_KEY`) defined in **inpctl.def**. To get information for F11, pass *keycode* as `F33_KEY(68)`. To get information for F12, pass *keycode* as `F34_KEY(69)`. See the distributed **keymap.msw** file and **MSWINDOWS** keymap.

By default, `D_MENU_SHORTCUT` searches the local environment for the entry, but you can specify another environment level by passing the number for that environment level as the *env* argument. If entries on more than one placed column have the same shortcut, the entry from the column that was placed last will be returned. This corresponds to the entry that would be signaled if the user pressed the key.

Examples

The following example returns the ID of the menu column that contains an entry whose shortcut code is mapped to the F18_KEY entry in **inpctl.def**. If there is no menu entry assigned to this shortcut code, the example will return 0.

```
col_id = %m_info(D_MENU_SHORTCUT, F18_KEY)
```

In the following example, the D_MENU_SHORTCUT subfunction returns not only a column ID but also the name of the menu entry whose shortcut code is mapped to the F18_KEY entry in **inpctl.def**. This information is returned for the previous environment level.

```
env_num = %e_info(D_ENV_LEVEL)      ;Get level number for current  
environment  
env_num = environment - 1           ;Decrement to get previous environment  
col_id = %m_info(D_MENU_SHORTCUT, F18_KEY, menu_entry, env_num)
```

M_KEYTXT – Get shortcut text



```
xcall M_KEYTXT (function_code, text[, length])
```

Arguments

function_code

The function code number for the shortcut key. (**n**)

text

Returned with the shortcut text. (**a**)

length

(optional) Returned with the length of the shortcut text. (**n**)

Discussion

The M_KEYTXT subroutine returns the shortcut text associated with a function code number for the current terminal key map. The **inpctl.def** file defines the associated function code numbers. Key function codes are in the range 1 to 98 (CTRL_A_KEY through F63_KEY).

Text is loaded left-justified with trailing blanks. If *text* is too small for the shortcut text, only the leftmost portion is loaded and no warning is given. The shortcut text may be from one to five characters in length.

You may want to create an include file to define mnemonics for the function codes that you use in your key maps (F1_KEY through F63_KEY) that indicate their function (for example, HELP_KEY, EXIT_KEY), and use these mnemonics when calling this routine directly.

Examples

The following subroutine obtains the shortcut text for the “Help” shortcut and then displays it on the information line.

```
subroutine info_bar
record msg
    ,a*      , "Press "
    key      ,a5
    ,a*      , " for help"
.include "codes.def"      ;User written file

proc
    xcall m_keytxt(HELP_KEY, key)
    xcall e_sect(D_INFO, msg)
    xreturn
endsubroutine
```

M_LDCOL – Load a menu column



```
xcall M_LDCOL(col_id, [channel], column_name, [D_NOPLC], [search], [error],
&           [D_GLOBAL][, saved_name])
```

Arguments

col_id

Returned with the created column ID. **(n)**

channel

(optional) The channel of the window library. **(n)**

column_name

The name of the column to load. **(a)**

D_NOPLC

(optional) The no-placement flag; the column will not be placed on the menu. **(n)**

search

(optional) The true/false flag that indicates whether a search for a loaded column with the name *column_name* should be performed. **(n)**

error

(optional) Returned with the status of the operation. **(n)**

D_GLOBAL

(optional) The global column flag. **(n)**

saved_name

(optional) The name of the column when it was originally saved. **(a)**

Discussion

The M_LDCOL subroutine loads a menu column into an internal menu structure, which enables the column to be used with M_COLUMN for menu processing. You can use M_LDCOL to load a column from a menu library, or you can use it to load a column created using MB_COLUMN with the D_NOLOAD flag (in which case you must also pass the *search* argument).

If *channel* is not passed, **g_utlib** is used, unless the column was created with MB_COLUMN.

If you pass a null string (“”) for *column_name*, Toolkit assigns a unique column name in the form *_W_nnn*, where *nnn* is the column ID. You must pass *saved_name* in this case.

If you want to create multiple instances of the same column, you can call `M_LDCOL` as many times as you need using *column_name* to give each succeeding column a new unique name. You must also pass *saved_name* if different.

The maximum number of placed menu columns is nine.

The maximum available memory for all loaded columns (including columns created with `MB_COLUMN`) is 65,565 bytes. If your application exceeds this, Toolkit will generate an error. Note that this limit includes all information about the columns: information you've specified and internal information. If you get this error, you must reduce the amount of information in the columns. For example, you might be able to reduce user text or shortcuts associated with the menu entries. (If you've defined your menu columns in a script file and there's a lot of column information, test your program by accessing *all* the column menus, including subcolumns.)

If you pass *search*, `M_LDCOL` searches for a loaded column with the specified *column_name* before attempting to load the column. If the column is found, a successful load is simulated, *col_id* is set to the ID of the column that was found, and the *search* variable is set to true. If the column is not loaded, the *search* variable is cleared and the load continues normally.

If *error* is present, it will be returned with a value of false (0) if the requested column was found, or a value of true (1) if the column was not found. If *error* is not present and the window cannot be found, a fatal error occurs.

If `D_GLOBAL` is present and the column is loaded from the library (rather than through the search and find operation), the column is classified as global rather than local.

Examples

The following example loads the global menu column "util" from the window library that is opened on **chnl**. The loaded column will be placed on the menu, its ID will be stored in **utilid**, and the result of the operation will be placed in **sts**. If "util" is already loaded, an error will occur.

```
xcall m_ldcol(utilid, chnl, "util",, sts, D_GLOBAL)
```

Because the next example omits the *channel* argument, it loads a menu column named "util" that was either created with `MB_COLUMN` or that is in the **g_utilib** library. If it was created with `MB_COLUMN` and the `D_NOLOAD` flag was passed, the column will be loaded. If `D_NOLOAD` was not passed, `M_LDCOL` just returns the column ID.

```
xcall m_ldcol(utilid,, "util",, search, sts)
```

The next example loads the menu column whose name is stored in **billing** from the library opened on **chnl** and returns its ID in **colid(i)**. The column will not be placed on the menu. The subroutine looks for an existing column of that name before loading. Again, the result of the operation will be returned in **sts**.

```
xcall m_ldcol(colid(i), chnl, billing, D_NOPLC, search, sts)
```

%M_POPUP – Process a pop-up menu



```
status = %M_POPUP (col_id, [coordinate_flags], [coordinate_1, coordinate_2], [window_id]
&               [, input_string])
```

or

```
xcall M_POPUP (col_id, [coordinate_flags], [coordinate_1, coordinate_2], [window_id]
&               [, input_string])
```

Return value

status

True if a pop-up menu entry was selected, otherwise false. (^VAL)

Arguments

col_id

The ID of the menu column to pop up. (n)

coordinate_flags

(optional) One or more of the following, which can be combined using .BOR. or the vertical bar (|) operator: (n)

Coordinate units:

D_POPUP_PIXEL *Coordinate_1* and *coordinate_2* are pixels. (default)

D_POPUP_ROWCOL *Coordinate_1* and *coordinate_2* are the ordinal row and column.

Horizontal alignment:

D_POPUP_LEFT Align the left side of the pop-up menu with the coordinates specified or with a default location. (default)

D_POPUP_HCENTER Center the pop-up menu horizontally over the coordinates specified or the default location.

D_POPUP_RIGHT Align the right side of the pop-up menu with the coordinates specified or the default location.

Vertical alignment:

D_POPUP_TOP Align the top of the pop-up menu with the coordinates specified or with a default location. (default)

D_POPUP_VCENTER	Center the pop-up menu vertically over the coordinates specified or with a default location.
D_POPUP_BOTTOM	Align the bottom of the pop-up menu with the coordinates specified or with a default location.

coordinate_1

(optional) The coordinate that determines the row or X position of the pop-up menu. If the D_POPUP_PIXEL flag is specified, the value of this argument is interpreted in pixels. Otherwise, the value is interpreted to be the row number. **(n)**

coordinate_2

(optional) The coordinate that determines the column of Y position of the pop-up menu. If the D_POPUP_PIXEL flag is specified, the value of this argument is interpreted in pixels. Otherwise, the value is interpreted to be the column number. **(n)**

window_id

(optional) The window ID used in the positioning of the pop-up menu, or one of the following: **(n)**

WID_APP The application window (excluding the border and caption).

WID_CONTAINER The container window for the Synergy application (including the border and caption).

input_string

(optional) The “path” to follow to get to a specific pop-up menu entry. **(a)**

Discussion

The %M_POPUP function enables applications to display and process pop-up menus. The menu column you specify with *col_id* does not need to be loaded or placed.

The optional set of bit flags used by the *coordinate_flags* argument is grouped into three categories: coordinate units, and horizontal and vertical alignment. The bit values within each of these categories are mutually exclusive. If no bit flag is set or more than one bit flag is set in a category, then the default bit flag for that category is presumed.

If D_POPUP_PIXEL is specified, the *coordinate_1* argument is the horizontal (X) coordinate, and the *coordinate_2* argument is the vertical (Y) coordinate. If *window_id* is not passed, the pixel coordinates are used to determine the location of the top left corner of the pop-up menu. On Windows platforms, if both the pixel coordinates are not passed, the current position of the mouse pointer is used to determine the position of the pop-up menu. On UNIX and OpenVMS platforms, if the pixel coordinates are not passed, the center of the screen is used to determine the position of the pop-up menu.

If `D_POPUP_ROWCOL` is specified, the *coordinate_1* argument is the vertical (row) coordinate, and the *coordinate_2* argument is the horizontal (column) coordinate. If *window_id* is not passed, the row and column coordinates are used to determine the location of the top, left corner of the pop-up menu. If both row and column coordinates are not passed, the pop-up menu is positioned in the center of the screen. When the row and column coordinate system is used on a Windows platform, the actual pixel position used for the pop-up menu is the center of the specified character cell.

If *window_id* is specified, the coordinates passed are interpreted to be relative to *window_id*. If *window_id* is not passed, the screen is the presumed value. If *window_id* refers to a Synergy window that is not placed, a fatal error occurs.

The format of *input_string* is identical to the *input_string* optional parameter used by `M_PROCESS`. See [M_PROCESS on page 10-43](#) for more information on the *input_string* argument. For cross-platform compatibility, an initial “<D>” in the string will always highlight the first item in the pop-up menu.

The pop-up menu is able to extend beyond the boundaries of the application window; however, it is limited to the boundaries of the screen.

On Windows, shortcuts are not available, and `%M_POPUP` recognizes the following reserved menu entries (which are documented in [“Appendix B: Reserved Menu Entries”](#)).

- `E_CLEAR`
- `E_COPY`
- `E_CUT`
- `E_MARK`
- `E_PASTE`

On UNIX and OpenVMS, shortcuts are available while the pop-up menu is displayed.

If a menu entry (other than one of those listed above for Windows) is selected from the pop-up menu, **`g_select`** will be true and **`g_entnam`** will be the name of the menu entry selected. A menu entry can be selected by pressing `ENTER` when the entry is highlighted; pressing the entry’s quick-select character (if unique); on Windows platforms, clicking the left or right mouse buttons; or on UNIX and OpenVMS platforms, pressing the entry’s shortcut.

If the user cancels the pop-up menu, **`g_select`** will be false. On Windows platforms, the pop-up menu can be canceled by clicking the mouse outside the pop-up menu or by pressing `ESC` or `ALT`. On UNIX and OpenVMS platforms, pressing the key mapped to the drop-down menu function (usually `CTRL+P`) will cancel the pop-up menu.

If `%M_POPUP` is used in a direct callback function (such as a right-click method for a window) and the user wants to return the menu entry out of that function, `M_SIGNAL` should be used to resignal the menu entry so that the current input loop will be interrupted. For example:

```
if (%m_popup(colid))
    xcall m_signal(g_entnam)
```

Menu Routines

%M_POPUP

You can disable or enable an entry in a pop-up menu by using the D_SUB qualifier for M_DISABLE or M_ENABLE. See [M_DISABLE on page 10-13](#) and [M_ENABLE on page 10-15](#) for information.

Examples

The following example pops up a menu centered over the application space.

```
xcall m_popup(colid, D_POPUP_ROWCOL .bor. D_POPUP_HCENTER .bor.  
&          D_POPUP_VCENTER, , , WID_APP)
```

The following example pops up a menu outside (just to the right of) a Synergy window, centered vertically relative to the window's center.

```
xcall m_popup(colid, D_POPUP_ROWCOL .bor. D_POPUP_VCENTER, ,  
&          %w_info(WIF_DCOLSIZ, wndid) + 2, wndid)
```

M_PROCESS – Process a menu



```
xcall M_PROCESS ([input_string], [help_id][, wait_time])
```

Arguments

input_string

(optional) The “path” to follow to get to a specific menu column entry. **(a)**

help_id

(optional) The help identifier. **(a)**

wait_time

(optional) The time-out limit for I/O processing to be completed before returning to the calling routine. **(n)**

D_FOREVER or **-1**

Never time out.

D_GLOBAL or **-2**

Use the global value (see [g_wait_time on page 18-6](#)). (default)

D_IMMEDIATE or **0**

Time out immediately.

n

Wait up to *n* seconds (where *n* is a positive value) for I/O processing to be complete.

Discussion

The M_PROCESS subroutine explicitly processes a menu.

You rarely need to call M_PROCESS, because menu processing is normally done implicitly during input. However, you may need to use this subroutine at initial program entry to explicitly process the menu when you’re not doing any input.

If you call M_PROCESS while no columns are loaded, UI Toolkit issues a “No columns placed” error (using U_MESSAGE, error-style) and return to the calling routine with **g_select** false.

Calling M_PROCESS with no arguments is the same as typing the process-menu key when doing input. You will return from M_PROCESS when an entry is selected or when the user presses the process-menu key to deactivate the menu bar.

The *input_string* simulates what you would type in order to move to a specific position on the menu (for example, what would happen after you pressed the process-menu key). The following characters are valid (not case sensitive):

[<i>entry_name</i>]	Name of the entry—the brackets are required
<L>	LEFT arrow key
<R>	RIGHT arrow key
<U>	UP arrow key
<D>	DOWN arrow key
<E>	“Enter” key—this is needed to activate a submenu
quick-select_character	Any valid quick-select character

Assuming that *input_string* does not begin with an entry name, the sequence of characters begins as if no menu column is dropped down (in other words, as if M_DEFCOL had been called with a 0 argument). Thus, an initial <R> drops down the first column.

WIN

The number of arrow movements required does not match the actual keystrokes pressed after the ALT key in Windows environments. (Specifically, one more <R> and one less <D> are required.) For compatibility with other platforms, *input_string* must match the keystrokes for a UNIX or OpenVMS environment.

On Windows, *input_string* can only specify a submenu entry if it follows a valid menu entry. For example, the following is allowed:

```
xcall m_process (" [menu_entry] <E> [submenu_entry] ")
```

The following is not allowed:

```
xcall m_process (" <R><D><D><E> [submenu_entry] ")
```

Input_string overrides the effect of any previous call to M_DEFCOL, though the default column or entry remains in effect for future calls to M_PROCESS without *input_string*.

The following example is equivalent to pressing the RIGHT arrow key twice and typing the “m” key. In other words, it selects the entry in the second column whose quick-select character is **m**. (The user must press ENTER to complete the selection if the quick-select character is non-unique within the column.)

```
" <r><r>m "
```

On Windows, M_PROCESS recognizes the following menu entries (though they generally don’t have a purpose): E_CLEAR, E_COPY, E_CUT, E_MARK, E_PASTE. If a menu entry other than one of these is selected from the menu, **g_select** is returned with a value of true. **G_entnam** will

contain the uppercased name of the entry the user was on when the user pressed ENTER. If the selected menu entry begins with “U_” the EUTILS_METHOD subroutine is called and the menu is reprocessed. If the selected menu entry is “O_HELP” the EHELP_METHOD subroutine is called, with *help_id* as its argument.

You will need to clear **g_select** after you satisfy the event.

Additionally, when a menu entry is selected, **g_mnustring** will contain the *input_string* which, in turn, could be passed to the next call to M_PROCESS to get back to the same menu entry.

If the operation times out, **g_select** is set to true and **g_entnam** set to the entry name specified in **g_time_entry**, which is defined in **tkctl.def**. See [g_time_entry on page 18-6](#). Note that the time-out is reset after each keystroke. Additionally, in Windows environments, if a menu column is dropped down no time-out will occur.

See also

- ▶ [M_LDCOL on page 10-37](#) to load a menu from a library.
- ▶ [MB_COLUMN on page 10-61](#) to create (and load) a column at runtime.

Examples

The following example enables menu processing (exactly as if you had pressed the process-menu key at an input).

```
xcall m_process
```

The next example processes a menu and selects an entry that is in the third column, one entry down from the top. It passes the argument “menuhelp” to the EHELP_METHOD subroutine if the selected menu entry is “O_HELP”

```
xcall m_process("<r><r><r><d>", "menuhelp")
```

The following example initializes to the entry below the entry whose name is **ap**, no matter which column contains **ap**.

```
xcall m_process("[ap]<d>")
```

The following example selects the entry “o_lock” on the submenu accessed by the entry “o_admin.”

```
xcall m_process("[o_admin]<e>[o_lock]")
```

The following code fragment illustrates one way to use M_PROCESS.

```
do
  begin
    xcall m_process
    if(g_select)
      case g_entnam of
        begin
          case
```

Menu Routines

M_PROCESS

```
"ENTRY1" : xcall entry1
"ENTRY2" : xcall entry2
"ENTRY3" : xcall entry3
"EXIT"   : done=TRUE
          endcase
        end
until (done)
```

M_RESET – Reset to a disabled menu bar



```
xcall M_RESET
```

Discussion

The M_RESET subroutine resets the menu to a disabled bar (with no menu columns dropped down).

If column entries are defined with the .ENTRY command's NORESET option, you will need to call M_RESET when you want the menu column reset (dropped up).

On Windows, M_RESET has no other effect than to rebuild menu columns if you have made a change to them but haven't done input since the change.

M_RPLENT – Replace menu entry names with their shortcuts



```
xcall M_RPLENT (string, [length][, not_found])
```

Arguments

string

The field to be modified. **(a)**

length

(optional) Returned with the trimmed length. **(n)**

not_found

(optional) Returned with the number of menu entries that weren't found. **(n)**

Discussion

The M_RPLENT subroutine scans a string for menu entry names to be replaced by their corresponding shortcut key text.

This subroutine searches for entry names enclosed in square brackets ([]) within the string and replaces the brackets and entry name with the shortcut text currently assigned to that entry name. If any entry name is not found, it is replaced by a single space. The text that follows the entry name may shift due to the differing lengths of the entry name and its shortcut text.

You can escape the leading bracket with a back slash (\). Any leading bracket that is preceded by a “\” will not be considered as delimiting a menu entry name, and the string “\[” will be replaced by “[”. If there is no ending bracket for a beginning bracket, all further replacement ceases. A set of brackets ([]) with no entry name is treated as an entry not found.

When *not_found* is passed, it is returned with a value of 0 if all menu entries are found and they all have shortcut text. Otherwise, *not_found* is returned with the number of menu entries that were either not on a placed column or had no associated shortcut text.

This subroutine is especially useful for creating messages and information lines that contain the names of keys to be pressed. Since different keys may be mapped with runtime terminal binding to the same function for different users or terminal types, you can use this subroutine to insert the correct key name at runtime. (See [“Customizing Key Mapping for Menu Shortcuts” on page 18-11](#) for information about runtime terminal binding.)

Examples

This example scans all currently placed columns for the entry name “O_HELP.” If found, and a shortcut key has been assigned to the entry, “[O_HELP]” in the string is replaced by the shortcut text and **nohelp** is returned with a value of zero. If the entry is not found or does not have a shortcut, “[O_HELP]” is replaced by a single space and **nohelp** is returned with a value of 1. For example, if the F1 key is assigned to the entry “O_HELP,” the resulting contents of **info** would be “Press F1 for Help.”

```
info = "Press [O_HELP] for Help"  
xcall m_rplent(info,, nohelp)
```

M_SCINFO – Get menu and shortcut entry information



```
xcall M_SCINFO (function, shortcut, entry, [number][, text])
```

Arguments

function

Identifies which menu entries will be processed. **(n)**

D_ENTS	All entries. (default)
D_ENAB_ENTS	Enabled entries.
D_DSAB_ENTS	Disabled entries.
D_SCS	Entries with shortcuts.
D_ENAB_SCS	Enabled entries with shortcuts.
D_DSAB_SCS	Disabled entries with shortcuts.
D_SINGLE_ENT	A specific entry.

shortcut

An array returned with the shortcut text for the menu entries specified by *function*. **(a)**

entry

An array returned with the entry names of the menu entries specified by *function*. **(a)**

number

(optional) The maximum number of elements to load in the *shortcut* and *entry* arrays. It is returned with the number of items found. If not passed, defaults to 1. **(n)**

text

(optional) An array returned with the text of the entries specified by *function*. **(a)**

Discussion

The M_SCINFO subroutine retrieves information about placed menu entries and their shortcuts and returns it in an array of entry names and an array of shortcut names.

If D_SINGLE_ENT is specified, the menu entry name for which to search must be passed as the first element in *entry*.

If *text* is passed, it is loaded with the corresponding menu text, minus the shortcut key names. The elements in *shortcut*, *entry*, and *text* will directly correspond with each other. Shortcut entries are blank if the corresponding menu entry does not have a shortcut defined.

Examples

The following example loads the shortcut text, entry names, and menu entry text for the first 98 enabled menu column entries. This information could be passed to a help routine to create a window that lists the available shortcuts.

```
count    ,i4
scuts    ,98a5
name     ,98a20
text     ,98a30

count=98
xcall m_scinfo(D_ENAB_ENTS, scuts, name, count, text)
```

%M_SIGNAL – Signal a menu entry

WT	WN	U	V

```
status = %M_SIGNAL (entry_name, [operation], {reserved})
```

or

```
xcall M_SIGNAL (entry_name, [operation], {reserved})
```

Return value

status

If you pass `D_SIGNAL` (or if you omit *operation*), `%M_SIGNAL` returns true (1) if *entry_name* is non-blank or false (0) if it is blank. If you specify `D_REMOVE`, `%M_SIGNAL` returns true if an entry is returned or false if there is no menu entry to cancel. (^VAL)

Arguments

entry_name

The menu entry name to signal or the returned name of the canceled menu entry. (a)

operation

(optional) One of the following: (n)

D_REMOVE Cancel the previously signaled menu entry.

D_SIGNAL Signal a menu entry for future processing. (default)

{reserved}

Reserved for internal use. (n)

Discussion

The `%M_SIGNAL` function provides a standardized way of signaling or canceling a signaled menu entry. See “[Simulating a menu entry or canceling a signaled menu entry](#)” on page 10-7 for recommendations on when to use `%M_SIGNAL`, and note the following:

- ▶ For `D_SIGNAL`, `%M_SIGNAL` sets **g_select** and **g_entnam**. If you specify `D_REMOVE`, `%M_SIGNAL` clears **g_select** and sets **g_entnam** to the name of the canceled entry.
- ▶ The `D_SIGNAL` operation does *not* call the routine that handles the signaled menu entry. It merely sets up the event to be “seen” by subsequent processing. Think of the `D_SIGNAL` operation as simply setting **g_select** and **g_entnam**. No other routine is called. Thus, code that follows the call to `%M_SIGNAL` will be executed before the code that responds to the event signaled. In the Synergy debugger, set the breakpoint in the code that responds to the event. If

you break before the %M_SIGNAL call and then step over, the action signaled by the event will not have occurred yet. (If you have Windows API programming experience, the D_SIGNAL operation is more like a PostMessage than a SendMessage.)

- ▶ The D_SIGNAL operation does not look at the current menu, so *entry_name* does not have to be on a placed menu column, and it doesn't have to be enabled. Thus, disabling the menu entry does not prevent %M_SIGNAL from generating the associated event. The application must avoid signaling or processing the event if it is not appropriate.
- ▶ As with all menu entry handling in Toolkit, the menu entry name is converted to uppercase to simplify subsequent conditional tests. However, the name passed to %M_SIGNAL is not validated as an identifier, so you can include characters that would not be legal for a menu entry, such as "@" and "#". The usefulness of these non-menu characters is left to the creativity of the developer.
- ▶ We don't recommend calling %M_SIGNAL in list load methods. Because of the way Toolkit synchronizes menu entry signaling, Toolkit may not execute such calls.
- ▶ If you signal a menu entry from and process the menu entry in a callback routine, use the D_REMOVE operation to cancel the menu entry after you've processed it. If you don't, the menu entry will be resigaled when processing returns from the callback routine.
- ▶ The menu entries E_CLEAR, E_COPY, E_CUT, E_MARK, and E_PASTE immediately perform their associated actions. %M_SIGNAL honors and dispatches these entries and sets **g_select** to 0, though *status* is returned as true.
- ▶ We recommend that you do not pass **g_entnam** as *entry_name*. **G_entnam** may be modified within this routine.
- ▶ As of version 8.3.1a, the Synergy DBL ACCEPT statement and the WD_ACCEPT subfunction to W_DISP no longer drop characters that are rapidly typed ahead by the user. If you create a custom ActiveX control, be sure not to call %M_SIGNAL in an event triggered by a "key down" (WM_KEYDOWN) message for a key that will be translated into a character. Instead, do this in response to a "key press" (WM_CHAR) message, to avoid duplicating the keystroke in type-ahead.

Examples

The following example, which is a change method, uses the "CUST_OK" menu entry to indicate whether a customer account could be found. The first call to %M_SIGNAL, which is in the *read_customer* subroutine, signals "CUST_OK" if it can read the account. The next call attempts to cancel the last signaled menu entry. If there's no menu entry to cancel, the method informs the user that the record cannot be found ("Invalid customer code"). If the call is able to cancel a menu entry, the method then checks the value returned in **g_entnam**. If this is "CUST_OK", the method proceeds to determine if there's a credit hold on the account or if the account is closed. If **g_entnam** is set to any other value, the method informs the user that the record cannot be found and then resignals the canceled entry.

```
; Customer Change Method
function cust_change ,^val ,reentrant
```

Menu Routines

%M_SIGNAL

```

    a_data_entered      ,a      ; A buffer containing the field data as
                                ; entered by the user.
    a_data_stored       ,a      ; A buffer for the final storage of the
                                ; data.
    a_pending_status    ,n      ; The result of Toolkit's field
                                ; validations.
.include "WND:inpinf.def"      ; Group argument of input info
.include 'INPUT_DATA' REPOSITORY , group='input'
                                ; The data_area argument passed to the
                                ; calling input routine.
    ; a_method_data     ,a      ; Optional method data argument.

.include "WND:tools.def"

.include 'CUSTOMER_MASTER' REPOSITORY ,record='cusmas'

proc
    clear cusmas
    cusmas.customer = input.customer
    xcall read_customer(cusmas)

    clear g_entnam
    ;Is there a menu signal waiting to be processed?
    if (%m_signal(g_entnam, D_REMOVE)) then
        begin
            ;Is the menu signal a "customer record read OK"?
            if (g_entnam.eq.'CUST_OK') then
                begin
                    using cusmas.status select
                    ('CH '),
                        begin ; CREDIT HOLD status
                            xcall u_message("Customer on credit hold")
                            a_pending_status = D_EMITTEDERR
                        end
                    ('C '),
                        begin ; CLOSED status
                            xcall u_message("Customer account has been closed")
                            a_pending_status = D_EMITTEDERR
                        end
                    endusing
                end
            else
                begin
                    xcall u_message("Invalid customer code")
                    a_pending_status = D_EMITTEDERR

                    ;Resignal the menu entry canceled above
                    xcall m_signal(g_entnam, D_SIGNAL)
                end
            end
        end
    end
endproc
```

```

        end
      end
    else
      begin
        xcall u_message("Invalid customer code")
        a_pending_status = D_EMITTEDERR
      end

      freturn a_pending_status
    endfunction

subroutine read_customer
  .include 'CUSTOMER_MASTER' REPOSITORY ,group='a_cusmas'
proc

  read(chan, a_cusmas, a_cusmas.customer, LOCK=Q_NO_LOCK) [ERR=nocust]

  ;Signal that customer record was read OK.
  xcall m_signal('CUST_OK', D_SIGNAL)

nocust,
  xreturn

endsubroutine

```

%M_TEXT – Modify and retrieve menu entry text

WT	WN	U	V

```
status = %M_TEXT (col_id, entry, [new_text], [old_text][, quick_select])
```

or

```
xcall M_TEXT (col_id, entry, [new_text], [old_text][, quick_select])
```

Return value

status

True (1) if successful, and false (0) if unsuccessful. (^VAL)

Arguments

col_id

The ID of the column in which the entry is located. (**n**)

entry

The name or row number of the entry to modify. (**a** or **n**)

new_text

(optional) The new menu entry text. (**a**)

old_text

(optional) Returned with the previous menu entry text. (**a**)

quick_select

(optional) The new quick-select character. (**a**)

Discussion

The %M_TEXT function enables you to retrieve and modify entry text for a menu or submenu. The entry can be a selectable entry, unselectable text, or blank. If the entry is selectable, you can also modify the quick-select character. Lines in a menu column, however, cannot be modified.

To modify a submenu, load *col_id* by calling M_LDCOL prior to calling %M_TEXT. Note the following. *Col_id* must be the ID of a loaded column or a fatal Toolkit error will occur. If the column becomes unloaded after calling %M_TEXT, the menu entry text will revert to its initial state when it is reloaded.

If *entry* is alphanumeric, it is presumed to be the entry name (**g_entnam**) associated with the entry to query and/or modify. If *entry* is numeric, it is presumed to be the row number of the entry, which enables you to access text, blank, and selectable entries by ordinal position. When determining ordinal position, all text, blank, line, and selectable entries must be counted.

If the specified entry doesn't exist or is a line, %M_TEXT returns false.

If *new_text* is passed, the menu entry text will be modified to contain *new_text*. Any shortcut text will remain unchanged. If *new_text* is shorter than the original text, the remainder will be filled with blanks. If *new_text* is wider than the widest entry in the column, *new_text* will be truncated.

If *old_text* is passed, it is returned with the text of the entry prior to being modified.

If *quick_select* is passed, the quick-select character will be modified to be the first occurrence (case insensitive) of the specified character. If *quick_select* is not passed and *new_text* is passed, the first non-blank character in *new_text* will be used to denote the quick-select character. If neither *quick_select* nor *new_text* is passed, the quick-select character is not modified.

Examples

The following example changes the menu text for the menu entry “entry1” of column **col_id** to be “New entry text.”

```
xcall m_text(col_id, "entry1", "New entry text")
```

The following example changes the menu text of the third menu entry of column **col_id** to be “Quick-select changed,” returns the previous menu text in **old_text**, and underlines the ‘s’ for the new quick-select character.

```
if (.not. %m_text(col_id, 3, "Quick-select changed", old_text, 's'))
    xcall u_message ("Unable to modify menu text")
```

M_USER – Retrieve a user text string for a menu entry

WT	WN	U	V

```
xcall M_USER (entry, text)
```

Arguments

entry

A menu entry name on a placed column. (a)

text

Returned with the text of the menu entry’s user-defined string. (a)

Discussion

The M_USER subroutine retrieves the user-defined string for the first menu entry that matches the entry name specified by the USER qualifier of the .ENTRY command. The menu columns are searched from left to right.

Entry is not case sensitive. If *entry* isn’t found on any placed column or if it doesn’t have a user-defined string, *text* remains unchanged.

You cannot retrieve user-defined text for submenu entries. The only way to access user strings for submenu entries is by testing **g_entusr** (defined in **tools.def**), which contains the user string associated with the last primary menu or submenu entry selected. If the entry has no user text associated with it, **g_entusr** is blank.

The maximum size of *text* is 80 characters.

See also

[.ENTRY on page 2-27](#) for information about specifying user strings.

Examples

The following example returns the user-defined text string for the menu entry “system” in **privs**. The user-defined text string could contain a security level that must be checked before the menu selection is processed.

```
xcall m_user("system", privs)
```

MB_BLANK – Add a blank row to a new column



```
xcall MB_BLANK(control)
```

Arguments

control

The menu control structure. (**a**)

Discussion

The MB_BLANK subroutine adds a blank row to a new menu column. It is used when building a menu column at runtime.

The blank row added by the MB_BLANK subroutine is not a blank entry; it's just a row that contains no text.

On Windows, blank rows can be highlighted, but the user will not be able to select them.

See also

- ▶ [MB_COLUMN on page 10-61](#) for more information on the menu control structure.
- ▶ [.BLANK on page 2-8](#) for more information on adding a blank row to a menu or a selection window column.

Examples

The following subroutine uses all the “MB_xxx” commands to build a menu at runtime.

```
main mbdemo
.include "WND:tools.def"
.include "WND:inpctl.def"

record
    colid      ,i4
    ctl        ,a12000

proc
    xcall u_start

    xcall mb_column(ctl, "c_general", "General", D_GLOBAL)
    xcall mb_entry(ctl, "O_HELP", "Help", F1_KEY)
    xcall mb_entry(ctl, "O_EXIT", "Exit", F4_KEY)
    xcall mb_entry(ctl, "U_ABOUT", "About")
    xcall mb_line(ctl)
    xcall mb_text(ctl, "Hot Entries", 2)
```

Menu Routines

MB_BLANK

```
xcall mb_blank(ctl)
xcall mb_entry(ctl, "U_VIEW", "View", F3_KEY,,, TRUE)
xcall mb_entry(ctl, "U_DIR", "Directory", F2_KEY,,, TRUE)
xcall mb_entry(ctl, "U_EDITREND", "Renditions")
xcall mb_entry(ctl, "U_EDITKEYS", "Key mapping")
xcall mb_entry(ctl, "U_APP", "Application Keypad",,"p")
xcall mb_list(ctl, "LIBOPEN", 2, "U_VIEW", "U_DIR")
xcall mb_end(ctl, colid)
xcall m_process
xcall u_finish
endmain
```

MB_COLUMN – Create a new column



```
xcall MB_COLUMN (control, column, text, [D_GLOBAL], [D_NOLOAD | D_NOPLC], [disabled],
& [justification] [, quick_select])
```

Arguments

control

The menu control structure. **(a)**

column

The name of the menu column to create (a maximum of 15 characters). **(a)**

text

The heading text for the menu column. **(a)**

D_GLOBAL

(optional) The global column flag. **(n)**

D_NOLOAD

(optional) Don't load the column (which implies D_NOPLC; see the Discussion below). **(n)**

D_NOPLC

(optional) Don't place the column. **(n)**

disabled

(optional) The true/false flag that indicates whether the column will be initially disabled. **(n)**

justification

(optional) One of the following justification flags: **(n)**

0 Left-justified beneath heading. (default)

1 Right-justified beneath heading.

2 Centered beneath heading.

quick_select

The quick-select character for the column. **(a)**

Discussion

The MB_COLUMN subroutine begins a menu column specification. This subroutine is used when building a new menu column at runtime.

The *control* argument must be large enough to include all of your column information. We recommend that *control* be several hundred bytes, but the size you'll need depends on the column information in your application. For example, a column with nine entries, one blank line, one horizontal line, and one text line would need 850 bytes. You can use this example as a guideline, but there is no simple algorithm to determine what you'll need for *control*. If the control area is too small for the column you're defining, Toolkit generates the following error:

overflow of column build array

As another example, the size of the control space used by the Script program is 24,540. This setting accommodates the largest possible menu column.



The maximum available memory for all loaded columns (including columns created with MB_COLUMN) is 65,565 bytes on Windows and UNIX and 8,192 bytes on OpenVMS. If your application exceeds this, Toolkit will generate an error. Note that this limit includes all information about the columns: information you've specified and internal information. If you get this error, you must reduce the amount of information in the columns. For example, you might be able to reduce user text or shortcuts associated with the menu entries.

If you pass a null string (“”) for *column*, Toolkit assigns a unique column name in the form *_W_nnn*, where *nnn* is the column ID.

If you pass D_GLOBAL, the column is classified as global rather than local.

By default the column will be loaded and placed. You can, however, can pass D_NOLOAD to prevent the column from being loaded, and D_NOPLC to prevent the column from being placed on the menu. If you are creating pop-up columns or submenu columns, or if you are creating a column just to save it to a window library, the D_NOLOAD option can save space in the memory control structure (see note above). D_NOLOAD implies D_NOPLC, but you can pass both by combining them with the bitwise operators .BOR. and I. Additionally, note the following restrictions for *unloaded* columns:

- ▶ You can't pass the column ID (*column_id* returned by MB_END) of an unloaded column to M_COLUMN or %M_TEXT. This will cause a fatal error.
- ▶ You can't pass the column ID (*column_id* returned by MB_END) of an unloaded column to M_ENABLE or M_DISABLE when using D_ENTRY, D_LIST, or D_COLUMN. This will cause a fatal error. You can, however, use an unloaded column with M_ENABLE and M_DISABLE when using D_SUB.

If you use D_NOLOAD or D_NOPLC, you can later load and/or place the column with M_LDCOL and M_COLUMN. See [M_LDCOL on page 10-37](#) and [M_COLUMN on page 10-10](#).

If *disabled* is present and non-zero, the column will initially be disabled when it is placed.

Use the *quick_select* argument to specify the quick-select character (case-insensitive) for the menu column. If *quick_select* is passed with more than one character, the first character becomes the quick-select character. If you don't pass *quick_select*, Toolkit won't establish a quick-select character for the column header.

To create a submenu column, define the column, and then define the parent menu column (the menu from which the submenu will cascade). In the submenu column's definition, use the `D_NOPLC` argument. In the parent menu column's definition, use the `D_SUBMENU` argument for the menu entry that will cause the submenu to be displayed, and use the submenu column's name (the *column* argument for this routine, `MB_COLUMN`) as the entry name (the *entry* argument for `MB_ENTRY`) for the menu entry. You must define the submenu before you define its parent menu. See the example below.

See also

[.COLUMN on page 2-20](#) for information on beginning a menu column definition.

Examples

For a complete example, see the `MB_BLANK` [Examples on page 10-59](#), which use all the “`MB_`xxx” subroutines.

The following example program has one subcolumn, “`MYSUB`”, that appears when a user selects “`My sub`” from the File menu.

```
.include 'WND:tools.def'
.include 'WND:inpctl.def'

record ids
    menu_id ,i4
    sub_id   ,i4

record
    control ,a1000

proc
    xcall u_start

    xcall mb_column(control,"MYSUB","My Sub",,D_NOPLC)
    xcall mb_entry(control,"SUB1","Sub1")
    xcall mb_entry(control,"SUB2","Sub2")
    xcall mb_entry(control,"SUB3","Sub3")
    xcall mb_end(control,sub_id)

    xcall mb_column(control,"FILE","File")
    xcall mb_entry(control,"O_EXIT","Exit",F4_KEY)
    xcall mb_entry(control,"MYSUB","My Sub",D_SUBMENU)
    xcall mb_end(control,menu_id)
```

Menu Routines

MB_COLUMN

```
repeat
  begin
    xcall m_process
    using (g_entnam) select
      ("O_EXIT"), exitloop
      (""), xcall u_message("Menu Entry: "+g_entnam)
    endusing
  end

  xcall u_finish
.end
```


MB_END – Define the end of a new column



```
xcall MB_END (control, column_id)
```

Arguments

control

The menu control structure. (**a**)

column_id

Returned with the ID of the menu column that is created. (**n**)

Discussion

The MB_END subroutine ends a menu column specification session. Use this subroutine when you are building a new menu column at runtime.

After you call MB_END, you can't change the contents of the column.

If you specified D_NOLOAD in the MB_COLUMN call, the value returned in *column_id* cannot be passed to some Toolkit routines. See the MB_COLUMN [Discussion on page 10-62](#) for details.

See also

- ▶ [MB_COLUMN on page 10-61](#) for more information on the menu control structure.
- ▶ [.END on page 2-26](#) for information on terminating the current window definition.

Examples

The following example creates a menu column with three entries. The ID of the new menu column is returned in **colid**.

```
xcall mb_column(menuctl, "colors", "Color Choices")
xcall mb_entry(menuctl, "color_1", "red")
xcall mb_entry(menuctl, "color_2", "blue")
xcall mb_entry(menuctl, "color_3", "yellow")
xcall mb_end(menuctl, colid)
```

Also see the MB_BLANK [Examples on page 10-59](#), which use all the “MB_xxx” subroutines.

MB_ENTRY – Add a menu entry to a new column



```
xcall MB_ENTRY (control, entry, text, [shortcut], [shortcut_text], [quick_select], [justification]
&               [, disabled])
```

Arguments

control

The menu control structure. **(a)**

entry

The name of the menu entry (a maximum of 10 characters). **(a)**

text

The display text for the entry. **(a)**

shortcut

(optional) The shortcut key code or **D_SUBMENU** to place a submenu. **(n)**

shortcut_text

(optional) The shortcut display text. (This argument is obsolete, though it is still available.) **(a)**

quick_select

(optional) The quick-select character. **(a)**

justification

(optional) The justification flag. **(n)**

0 Left-justified beneath heading. (default)

1 Right-justified beneath heading.

2 Centered beneath heading.

disabled

(optional) The true/false flag that indicates whether or not the entry will be initially disabled. **(n)**

Discussion

The MB_ENTRY subroutine adds a menu entry to a new column. This subroutine is used when building a new menu column at runtime. You can define up to 20 menu entries per menu column.

By defining D_SUBMENU for *shortcut*, you can cause the menu entry to display a submenu column. (Note that you must also pass the submenu column's name as the *entry* argument.) See the MB_COLUMN [Discussion on page 10-62](#) for more information.

If *quick_select* is not specified, the first non-blank character of text will be used as the quick-select character. If *disabled* is present and non-zero, the entry will initially be disabled on column placement.

To assign a shortcut to the menu entry, pass one of the identifiers defined in **inpctl.def** as the *shortcut* argument. For example, to assign the END key as the shortcut on Windows, pass F60_KEY as *shortcut*. To find out what to pass for a key or key combination, see the comments in **inpctl.def**, or open the key mapping utility and view the key map that will be used with the application.

The *shortcut_text* argument is now obsolete. Before version 2.5 of UI Toolkit, menu shortcut text was derived from the **keymap.ctl** file at script time, and thus also had to be supplied to MB_ENTRY so that it could be included in the column controls. With version 2.5, runtime terminal binding was added to Toolkit. This feature looks up the key code in the key map for the user's keyboard at runtime, and loads the shortcut text as the column is loaded.

See also

- ▶ [MB_COLUMN on page 10-61](#) for more information on the menu control structure.
- ▶ [.ENTRY on page 2-27](#) for information on defining a column entry.

Examples

See the MB_BLANK [Examples on page 10-59](#), which use all the “MB_xxx” subroutines.

MB_LINE – Add a line to a new column



```
xcall MB_LINE(control)
```

Arguments

control

The menu control structure. **(a)**

Discussion

The MB_LINE subroutine adds a horizontal line to a new column. This subroutine is used when building a new menu column at runtime.

The line added by the MB_LINE subroutine is not a menu entry; it's just a line in the menu column.

See also

- ▶ [MB_COLUMN on page 10-61](#) for more information on the menu control structure.
- ▶ [.LINE on page 2-79](#) for information about drawing a line within a window definition.

Examples

See the MB_BLANK [Examples on page 10-59](#), which use all the “MB_xxx” subroutines.

MB_LIST – Create a list entry in a new column



```
xcall MB_LIST(control, list, #entries, entry[, ...])
```

Arguments

control

The menu control structure. **(a)**

list

The name of the list to create (a maximum of 10 characters.) **(a)**

#entries

The number of entries in the list. **(n)**

entry

The name of an entry in the list. You can have up to 20 entry names, separated by commas. **(a)**

Discussion

The MB_LIST subroutine creates a list entry in a new column. This subroutine is used when building a new menu column at runtime.

You must already have defined each entry with the MB_ENTRY subroutine.

See also

- ▶ [MB_COLUMN on page 10-61](#) for more information on the menu control structure.
- ▶ [.LIST on page 2-81](#) for information on creating a set of menu entries.

Examples

See the MB_BLANK [Examples on page 10-59](#), which use all the “MB_xxx” subroutines.

MB_TEXT – Add a row of text to a new column



```
xcall MB_TEXT (control, text[, justification])
```

Arguments

control

The menu control structure. (**a**)

text

The text to add to the column. (**a**)

justification

(optional) One of the following justification flags: (**n**)

- 0** Left-justified beneath heading. (default)
- 1** Right-justified beneath heading.
- 2** Centered beneath heading.

Discussion

The MB_TEXT subroutine adds a row of text to a new column. This subroutine is used when building a new menu column at runtime.

The *text* added by MB_TEXT is not a menu entry; it's just a comment displayed in the menu column. On Windows, text in a menu column can be highlighted, but the user will not be able to select it.

If *justification* is present and has a value of **1**, *text* will be right-justified within the row. If *justification* has a value of **2**, *text* will be centered in the row.

See also

- ▶ [MB_COLUMN on page 10-61](#) for more information on the menu control structure.
- ▶ [.TEXT on page 2-101](#) for information on adding text to a window or menu column.

Examples

See the MB_BLANK [Examples on page 10-59](#), which use all the “MB_xxx” subroutines.

11

.NET Routines

UI Toolkit .NET routines are not available in Synergy .NET.

Adding and Using .NET Forms, .NET Controls, and WPF Elements 11-2

Discusses how you can use .NET forms, .NET controls, and WPF elements in Toolkit applications.

DOTNET_TKADDCTL – Add a form, control, or element to a previously-embedded form..... 11-6

DOTNET_TKINPUT – Process an embedded form 11-8

DOTNET_TKFORM – Retrieve form object from a Toolkit container 11-11

%DOTNET_TKWIN – Create a container window and embed a form, control, or element 11-12

Adding and Using .NET Forms, .NET Controls, and WPF Elements

The UI Toolkit .NET routines enable you to embed the following in a Toolkit application:

- ▶ .NET forms (System.Windows.Forms.Form)
- ▶ .NET controls (System.Windows.Forms.Control)
- ▶ Windows Presentation Foundation (WPF) elements (System.Windows.UIElement)

.NET Framework 4.0 or higher is required. These routines are not do not work with Synergy .NET. For information on using Toolkit programs with Synergy .NET, see [“Using UI Toolkit with the .NET Framework” on page 1-21](#).

Only .NET forms can be embedded directly into Toolkit windows, but you can embed .NET controls and WPF elements by either adding them to a .NET form and then embedding the form, or by instructing Toolkit to create a .NET form as you add a control or element.

With the Toolkit .NET routines, you can do the following:

- ▶ Use a .NET form as a Toolkit window. This includes using a .NET form as a child for a composite window. See [“Embedding a .NET form” on page 11-3](#).
- ▶ Use a .NET control or WPF element as a Toolkit window. Note that to add a .NET control or a WPF element to a Toolkit application, the control or element must be on a .NET form. You can instruct Toolkit to embed the control or element in a form for you. See [“Embedding .NET forms, .NET controls, and WPF elements” on page 11-4](#).
- ▶ Add a .NET form, .NET control, or WPF element to an already embedded .NET form. See [“Embedding .NET forms, .NET controls, and WPF elements” on page 11-4](#).
- ▶ Process input for a .NET form. See [step 4 in “Embedding a .NET form” on page 11-3](#).
- ▶ Find out if a Toolkit window contains a .NET form. See [D_WINDOW_TYPE on page 7-32](#).

The user sees the .NET form, .NET control, or WPF element as part of your Toolkit application. Input and menu entries work as they do with any other Toolkit window, and if a .NET form is part of a composite window, tabbing takes the user in and out of the form. Note, however, that unless the container window is part of a composite window, you won't be able to use TAB or SHIFT+TAB to access buttons on the Toolkit container window (the Toolkit window that contains the form, control, or element). See the [DOTNET_TKINPUT Discussion on page 11-9](#) for more information on how keyboard input is handled.

When you embed a .NET form, .NET control, or WPF element, you associate it with a Toolkit container window. Because the container is a Toolkit window, you can perform all standard window-related Toolkit functions (place, remove, delete, resize, and so forth) by referencing the window ID for the container. Note, however, that resizing the Toolkit container window doesn't automatically resize the form. If you resize the container, areas of the form can be cut off or portions of the Toolkit container window may be visible on the bottom or right of the form.

(You can adjust this by querying and setting the width and height properties of the form.) The only time Toolkit automatically sizes a form is when it automatically creates a form (when the *flags* argument is passed as DN_MAKEFORM or DN_WPF in a call to %DOTNET_TKWIN).

Embedding a .NET form

This is the basic procedure for embedding a .NET form in a Toolkit application and using the form as part of the application. (You can also add a .NET form to a previously embedded form. See [“Embedding .NET forms, .NET controls, and WPF elements” on page 11-4.](#))

1. Load the assembly that contains the type to be instantiated, and create an instance of a .NET form contained in the assembly.

The easiest way to do this is by using **gennet40**, **dblproto**, and the NEW keyword. For example, if you have a program named MyTkProg and you want to embed a form with a text editor named TextEditor that’s in a prebuilt assembly named **MyTextEd.dll**, you could do the following:

Run **gennet40** to generate a wrapper class for TextEditor:

```
gennet40 -output Editor.dbl "c:\MyForms\MyTextEd.dll"
```

Run **dblproto** to generate prototypes for the class. The output will be **Editor.dbp**.

```
dblproto Editor.dbl
```

Create an instance of TextEditor:

```
Record
    txt      ,@TextEditor
Proc
    txt = new TextEditor()
```

Compile the **.dbl** file produced by **gennet40** (**Editor.dbl**) and your Toolkit application (**MyProg.dbl**). Make sure the MyTkProg compile has access to the prototypes (**Editor.dbp**) generated by **dblproto**.

```
dbl -qrelaxed:interop Editor
dbl MyTkProg
```

Link your Toolkit application with the compiled Editor file:

```
dblink MyTkProg Editor
```

For more information, see

- ▶ [“Instantiating and using a class”](#) in the “Understanding Objects and Classes” chapter of the *Synergy DBL Language Reference Manual*.
- ▶ [“The Synergy Prototype Utility”](#) and [“The Gennet40 Utility”](#) in the “General Utilities” chapter of *Synergy Tools*.

Note that you can also do this directly with the Synergy .NET API (see the “[Synergy .NET Assembly API](#)” chapter of the *Synergy DBL Language Reference Manual*), but we recommend using **gennet40**, which simplifies the process.

2. Use `%DOTNET_TKWIN` to embed the .NET form in a Toolkit container window. If the .NET form will be part of a composite window, you’ll also need to use the `DC_ADD` subfunction for `%C_CONTAINER` to add the container window to the composite window.
3. Access properties, call methods, and register event handlers for the .NET form object as necessary. Note that registering an event handler involves adding a delegate to an event. See the “[Synergy .NET Assembly API](#)” chapter of the *Synergy DBL Language Reference Manual*, and note that **gennet40** creates a public variable for each event defined in a class. These variables contain event objects, and you can call an `AddHandler` method for these to add a delegate.
4. Do one of the following:
 - ▶ If the .NET form is *not* part of a composite window, call `DOTNET_TKINPUT` to accept input for the .NET form.
 - ▶ If the .NET form *is* part of a composite window, call `C_PROCESS` to process the composite window. (`C_METHNET`, the default composite window processing method for .NET form windows, calls `DOTNET_TKINPUT`.)
5. Respond to any new entries after the application exits from `DOTNET_TKINPUT` or `C_PROCESS`, and call methods or query properties for the .NET form to get the input results.
6. Delete the Toolkit container window, if it isn’t automatically deleted as a result of exiting the current environment (using `E_EXIT`). This releases Toolkit’s reference to the .NET form.

Embedding .NET forms, .NET controls, and WPF elements

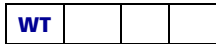
To add a .NET control or a WPF element to a Toolkit application, the control or element must be on a .NET form. But what if you have a .NET control or a WPF element that you’d like to use on its own (e.g., a grid control), and you don’t have a form? Or what if you want to add a .NET form or control to a .NET form that’s already embedded in a Toolkit window? Except for the first three steps listed above in “[Embedding a .NET form](#),” the procedure is the same.

1. Create an instance of the .NET form, .NET control, or WPF element. See [step 1](#) in “[Embedding a .NET form](#)” on [page 11-3](#), which discusses creating an instance of a .NET form. You can use the same procedure to create an instance of a .NET control or a WPF element.
2. Do one of the following:
 - ▶ If you want to add the .NET form, .NET control, or WPF element to a .NET window that’s already been embedded in a Toolkit container window, use `DOTNET_TKADDCTL`, passing the container window ID as the *wndid* argument.
 - ▶ If you want to add a .NET control or a WPF element, but you don’t want to add it to a .NET window that’s already been embedded (for example, if you want to add a grid control to a composite window, and you have no .NET form), use `%DOTNET_TKWIN`, passing the *flags* argument as `DN_MAKEFORM` (for a .NET control) or `DN_WPF` (for a WPF element). This

instructs %DOTNET_TKWIN to create a Toolkit window and a .NET form to contain the control or element. Pass the .NET control or WPF element (instead of a form) to %DOTNET_TKWIN, and the control or element will be placed on the automatically-generated form.

3. Follow [step 3](#) through [step 6](#) above in “[Embedding a .NET form.](#)”

DOTNET_TKADDCTL – Add a form, control, or element to a previously-embedded form



```
xcall DOTNET_TKADDCTL (window_id, object[, flags])
```

Arguments

window_id

The ID of a window created by %DOTNET_TKWIN. (Otherwise Toolkit throws an exception.) (n)

object

The .NET form, .NET control, or WPF element to add. This must be derived from System.Windows.Forms.Form, System.Windows.Forms.Control, or System.Windows.UIElement. If *object* is null or does not wrap a .NET form, .NET control or WPF element, Toolkit throws an exception. (@DotNetObject)

flags

(optional) Pass as DN_WPF if *object* is a WPF element; otherwise omit this argument. (n)

Discussion

On Windows, DOTNET_TKADDCTL adds a .NET form, a .NET control, or a WPF element to the Controls collection for a previously embedded .NET form (contained in *window_id*), but it does not adjust its size or position. (On UNIX and OpenVMS, this routine merely returns.)

To add a WPF element, pass *flags* as DN_WPF. If you pass *flags* as DN_WPF, *object* must be a WPF element (i.e., it must be instantiated from a class derived from System.Windows.UIElement); otherwise a DotNetException will be thrown.

See also

[%DOTNET_TKWIN](#) on page 11-12 for information on embedding a .NET form in a Toolkit window.

Examples

This example adds two .NET controls (a label and a field) to a composite window and processes the composite window. Toolkit automatically creates a .NET form for the controls. See [step 1 of “Embedding a .NET form” on page 11-3](#).

```
.include "MyForm.inc"           ;Include the .inc file generated by gennet40.
main
.include "WND:tools.def"
record
    wndid ,i4
    ctrid ,i4
    ctl   ,@TextBox
    lbl   ,@Label
    form  ,@DotNetObject
proc
    xcall u_start
    ctrid = %c_container(DC_CREATE, "", 20, 78) ;Create composite window

    ;Create Toolkit container window and empty .NET form. (If you pass
    ; the <object> argument as null, Toolkit creates a .NET form,
    ; regardless of the <flags> setting.)
    wndid = %dotnet_tkwin("", 18, 78, ^null)

    xcall dotnet_tkform(wndid, form)           ;Make sure form was created.
    if (form == ^null)
        throw new System.NullReferenceException()

    lbl = new Label()           ;Instantiate a Label object.
    lbl.Text = "Label1: "       ;Set Text property for Label object.
    xcall dotnet_tkaddctl(wndid, lbl) ;Add Label to the automatically-
                                   ; created .NET form.

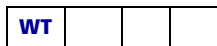
    ctl = new TextBox()         ;Instantiate a TextBox object.
    ctl.Text = "Lorem ipsum..." ;Set Text property of TextBox object.
    xcall dotnet_tkaddctl(wndid, ctl) ;Add TextBox to the automatically-
                                   ; created .NET form.

    ctl.Left = lbl.Left + lbl.Width + 10    ;Set TextBox position & width.
    ctl.Width = %u_winmetrics(D_CHARWIDTH, wndid) * 30

    ; Add and place .NET form.
    xcall c_container(DC_ADD, ctrid, DC_WINDOW, wndid, 1, 1)
    xcall u_window(D_PLACE, ctrid, 1, 1)

    ;Process composite window. Default method for .NET form window
    ; (C_METHNET) will call DOTNET_TKINPUT.
    xcall c_process(ctrid)
    xcall u_finish
endmain
```

DOTNET_TKINPUT – Process an embedded form



```
xcall DOTNET_TKINPUT (window_id, [help_id], [wait_time], [menu_flag][, focus])
```

Arguments

window_id

The ID of a window created by %DOTNET_TKWIN. (Otherwise Toolkit throws an exception.) (n)

help_id

(optional) A help identifier to be passed if the menu entry “O_HELP” is signaled or selected. (a)

wait_time

(optional) The time-out limit for I/O processing to be completed before returning to the calling routine. (n)

D_FOREVER or **-1**

Never time out.

D_GLOBAL or **-2**

Use the global value (see [g_wait_time on page 18-6](#)). (default)

D_IMMEDIATE or **0**

Time out immediately.

n

Wait up to *n* seconds (where *n* is a positive value) for the user to press a key.

menu_flag

(optional) If passed and true, a flag that causes DOTNET_TKINPUT to return utility menu entries (U_ and O_HELP) instead of processing them. (n)

focus

(optional) Indicates which component control (which .NET control or WPF form within the .NET form) to move focus to. (n)

D_FIRST

Focus the first control.

D_LAST

Focus the last control.

If any other value is passed as *focus*, or if *focus* is not passed, the form determines where to move focus. And if no suitable component control can be found, the form itself receives focus.

Discussion

On Windows, the DOTNET_TKINPUT routine processes a .NET form embedded in a Toolkit container window. DOTNET_TKINPUT moves focus to the specified form (contained in *window_id*) and enables your program to accept input for the form. At the same time, it allows menu entries to be selected and returned to Toolkit. (On UNIX and OpenVMS, this routine merely returns.)

When focus is on a .NET form, most keystrokes are sent first to the form as input. The form's `PreProcessMessage` method returns true if the form will consume the keyboard input, which prevents the Synergy runtime from processing the input. The `PreProcessMessage` method returns false if the form won't consume the keyboard input, which allows the Synergy runtime to process the input.

The exceptions are TAB and SHIFT+TAB. Toolkit sends TAB directly to the Toolkit container, rather than the form (enabling the Synergy runtime to process it) if the .NET form is part of a composite window and TAB is pressed when the last control in the form has focus. The same is true of SHIFT+TAB if pressed when the first control in the form has focus. TAB and SHIFT+TAB are otherwise sent to the form and used by the form to navigate among its controls.



If you have a .NET control that needs to consume a keystroke that is used by the form (e.g., SHIFT+TAB, ESCAPE, or an accelerator key), you will need to override the control's `ProcessCmdKey` or `IsInputKey` method or the form's `PreProcessMessage` method. See Synergex® KnowledgeBase article [100001973](#) for information.

If a keystroke can be mapped to a menu entry and isn't consumed by the .NET form, the menu entry is signaled. DOTNET_TKINPUT recognizes the following:

- ▶ “C_NEXT” and “C_PREV” if the .NET form is part of a composite window. If the form isn't part of a composite window, these menu entries are returned immediately to the calling routine. See [chapter 6, “Composite Window Routines,”](#) for more information.
- ▶ “O_HELP” if the *menu_flag* argument is not passed or is passed as false. Otherwise, this menu entry is returned to the calling routine.
- ▶ “U_*” menu entries if the *menu_flag* argument is not passed or is passed as false. Otherwise, these menu entries are returned to the calling routine.

DOTNET_TKINPUT returns only when a menu entry is signaled (**g_select** is true and **g_entnam** contains the name of the menu entry), an exception has been thrown (`@DotNetException`), or the operation times out. If the operation times out, **g_select** is set to true, and **g_entnam** is set to the entry name specified in **g_time_entry**, which is defined in **tkctl.def**. See [g_time_entry on page 18-6](#).

If, while the form is active, focus is diverted to another application, when focus returns to your Toolkit application, it is established on the form, and the form reestablishes focus among its component controls.

.NET forms in composite windows

If the Toolkit container window is part of a composite window, DOTNET_TKINPUT must be called from the processing method for the child window (which is C_METHNET by default), and TAB and SHIFT+TAB are processed by the Synergy runtime if pressed when the first or last control in a .NET form has focus (as described above). TAB moves forward through the following tabbing order, and SHIFT+TAB moves backward through the following.

1. Previous windows (or buttons if wrapping to the end) in the composite window's tabbing order
2. The .NET form
3. Buttons on the .NET form's Toolkit container window (in order)
4. Subsequent windows or buttons in the composite window's tabbing order

See [“Controlling tabbing, focus, and input context” on page 6-4](#) for more information.

Examples

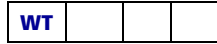
The following example embeds a .NET form in a Toolkit window and then processes it with DOTNET_TKINPUT. The container window created below is not part of a composite window, so DOTNET_TKINPUT is called directly. (If it was part of a composite window, the processing method for the child window, which is C_METHNET by default, would call DOTNET_TKINPUT.) Note that the first line uses .INCLUDE to include the .inc file generated by gennet40. See [step 1 of “Embedding a .NET form” on page 11-3](#).

```
.include "MyForm.inc"           ;Include the .inc file generated by gennet40.
main
.include "WND:tools.def"
record
    form      ,@MyForm
    wndid     ,i4
    ctrid     ,i4

proc
    xcall u_start
    form = new MyForm()         ;Instantiate a MyForm object.
    wndid = %dotnet_tkwin("", 18, 78, form) ;Create container window and
                                           ; embed form (MyForm object)

    xcall b_button(wndid, "OK")
    xcall u_window(D_PLACE, wndid, 1, 1)   ;Place container window.
    repeat
        begin
            xcall dotnet_tkinput(wndid,,, D_LAST);Process container window
        end
    .
    .
    .
    xcall u_finish
endmain
```


DOTNET_TKFORM – Retrieve form object from a Toolkit container



```
xcall DOTNET_TKFORM(window_id,form)
```

Arguments

window_id

The ID of the Toolkit window that contains the .NET form. (Otherwise Toolkit throws an exception.) (**n**)

form

The returned form object. (@DotNetObject)

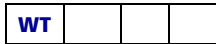
Discussion

The DOTNET_TKFORM routine returns the .NET form object (*form*) for the specified Toolkit container window (*window_id*), even if the container window was automatically generated by %DOTNET_TKWIN.

Examples

See the DOTNET_TKADDCTL [Examples on page 11-7](#).

%DOTNET_TKWIN – Create a container window and embed a form, control, or element



window_id = %DOTNET_TKWIN (*name*, *height*, *width*, *object*[, *flags*])

Return value

window_id

On Windows, the Toolkit container window ID. On UNIX and OpenVMS, returns 0. (^VAL)

Arguments

name

The name of the Toolkit container window to create. This can be up to 15 characters. If you specify a null string (“”), Toolkit assigns a unique window name in the form *_W_nnn*, where *nnn* is the window ID. (a)

height

The height of the window in character rows. If you pass either *height* or *width* as 0, Toolkit sets the window to the height and width of the application screen. (n)

width

The width of the window in character columns. If you pass either *height* or *width* as 0, Toolkit sets the window to the height and width of the application screen. (n)

object

The .NET form, .NET control, or WPF element to embed. This must be derived from System.Windows.Forms.Form, System.Windows.Forms.Control, or System.Windows.UIElement and wrapped in a Synergy class (i.e., as a DotNetObject) by **gennet40**. (@DotNetObject)

flags

(optional) Instructs %DOTNET_TKWIN to do one of the following: (n)

DN_MAKEFORM Create a .NET form for a .NET control. (If passed, *object* must be a .NET form or control or null.)

DN_WPF Create a .NET form for a WPF element. (If passed, *object* must be a WPF element or null.)

Note that although you can combine these bit flags with .BOR. or the vertical bar (|) operator (for a WPF element), this is not necessary because DN_WPF includes DN_MAKEFORM.

Discussion

On Windows, the %DOTNET_TKWIN function enables you to embed a .NET form, .NET control, or WPF element in a Toolkit application. To do this, it creates a Toolkit container window and, optionally, it creates a .NET form to contain the form, control, or element passed as *object*. (On UNIX and OpenVMS, this function returns zero, but otherwise does nothing.)

- ▶ To add a .NET form, pass the form as *object*, and omit the *flags* argument (unless you want the form to be treated as a .NET control and be added to an automatically-created .NET form).
- ▶ To embed a .NET control or a WPF element, pass the control or element as *object* and pass *flags* as DN_MAKEFORM (for a .NET control) or DN_WPF (for a WPF element). (If you want to add a .NET control or a WPF element to an already-embedded .NET form, use DOTNET_TKADDCTL.)

Before you pass a .NET form, .NET control, or WPF element as *object*, you must wrap it in a Synergy class by using **gennet40**. (See “[Embedding a .NET form](#)” on page 11-3 for information.) Otherwise Toolkit throws an exception.

If you pass *flags* as DN_MAKEFORM or DN_WPF, this function creates a Toolkit container window with an embedded .NET form. If *object* is null, the .NET form will be empty. If *object* is non-null, Toolkit embeds *object* as a control in the Toolkit-created .NET form (even if *object* is a .NET form). *Object* is added to the Controls collection for the Toolkit-created form. Also note that *object* is sized and positioned to fill the form. You can subsequently resize and move *object* by modifying the Toolkit-created form.

If you don’t pass *flags* (or if you pass it as a value other than DN_MAKEFORM or DN_WPF) and *object* is null, this function creates a Toolkit container window with an embedded .NET form. If *object* is non-null, this function creates a Toolkit container window, but it doesn’t create a .NET form. Instead, Toolkit embeds *object* (which must be a .NET form) in the container window. Note the following:

- ▶ The embedded .NET form is not sized automatically, which means that the form may be cropped or portions of the Toolkit container window may be visible on the bottom or right of the .NET form. You can adjust this by querying and setting the width and height properties of the .NET form.
- ▶ When a .NET form is embedded in a Toolkit container window, the frame around the .NET form is turned off, and the Toolkit container window becomes the parent window of the form’s outermost window. (And when the container window is deleted, the .NET form’s style is set to WS_POPUP, and its parent window is set to null.)

%DOTNET_TKWIN creates a container window by calling W_PROC. and passing *name*, *rows*, and *cols*. If W_PROC generates an error, you can trap it. (See [W_PROC](#) in the “Synergy Windowing API” chapter of the *Synergy DBL Language Reference Manual* for more information.) Note that the container window is logged with Toolkit at the current environment level. You can use U_GBLWND to promote it to global.

.NET Routines

%DOTNET_TKWIN

When a user clicks on an embedded .NET form, that event does not automatically trigger a call to any UWNDEVENTS_METHOD associated with the Toolkit container window. To trigger Windows event processing, the .NET form must fire an event of its own.

Examples

See the DOTNET_TKADDCTL [Examples on page 11-7](#).

12

Selection Routines

Introduction to Selection Processing 12-2

Explains what selection windows are, how they look, and how the user can make a selection from them.

S_SELBLD – Create a selection window.....	12-5
S_SELDFLT – Set or return default entry for a selection window	12-8
S_SELECT – Process a selection window	12-10
S_SELECTCB – Process a check box selection window.....	12-13
S_SELINF – Retrieve selection window information	12-15
S_SELLD – Load new selections into a selection window.....	12-17
S_UPDATECB – Update a check box selection window	12-18

Introduction to Selection Processing

Selection windows enable you to present the user with a fixed list of textual items from which to choose. These items are listed in one or more columns in the window. All items are padded out to the same length, although the text of each may be left-justified, right-justified, or centered within that length. A selection window can also have a border and a title.

The item currently being accessed in a selection window is always highlighted, regardless of whether the selection window is being processed.

A selection window can be attached to an input window field (using the `I_FLDSEL` subroutine) or processed directly. When a selection window is attached to an input field, the selected item becomes the input for that field. When a selection window is processed directly, either the text of the selected item or its ordinal position in the selection window can be returned to the calling routine.

You can generate selection windows from script files or at runtime (using the [S_SELBLD](#) routine).

Selection windows are like menu columns in the following ways:

- ▶ All the possible choices are visible.
- ▶ You navigate them with arrow keys and quick-select characters.
- ▶ To select an item, you press ENTER.

However, they are unlike menu columns in these ways:

- ▶ They are not attached to a menu bar.
- ▶ They may contain more than one column of selections.
- ▶ Items can be reloaded at runtime.
- ▶ Entries do not have shortcuts, entry names, or user text.

Selection windows compare and contrast with lists in the following ways:

- ▶ Like lists, you can load selection windows at runtime with a set of items drawn from any source (such as a file) so that the user has them available to select.
- ▶ While lists allow only one column of items, selection windows can contain more than one column of items.
- ▶ Unlike lists, selection windows only contain a fixed number of items, all of which must be visible.

Note that you can also create check box selection windows, which display a special character (usually a check mark or an “X”) to the left of any entry whose value is true. See [“Creating a check box selection window” on page 12-3](#).

Display renditions

Five rendition states are associated with selection windows:

- ▶ Border
- ▶ Title
- ▶ Background
- ▶ Selected item
- ▶ Entry quick-select character

When a selection window is processed, the above renditions are applied to the window. The item currently being accessed is displayed in the selected item rendition. All other items and any text, lines, or blank entries are displayed in the background rendition.

By default,

- ▶ the border and title renditions are unchanged from how they were defined when the window was created.
- ▶ the background rendition is unchanged, except that reverse is disabled.
- ▶ the selected item rendition is unchanged, except that reverse is forced.
- ▶ the entry quick-select character is set to underlined, color palette 10.

For information on editing these renditions, see [“Customizing the Look of Your Application” on page 18-24](#).

Making a selection

When a selection window is processed, the default item is highlighted. The user can then move the highlight to the desired item by pressing the arrow keys or a quick-select character. To select the item, the user presses ENTER. On Windows, the mouse can be used to highlight and select entries.

Creating a check box selection window

Check box selection windows enable you to display a special character (usually a check mark or an “X”) to the left of any entry that has a true value. These values correspond to an array that is passed to and from the check box routines. The routines [S_UPDATECB](#) and [S_SELECTCB](#) are used with check box selection windows.

Typically, you would design a check box selection window in the following manner:

1. Create your selection window, either in a script with the `.SELECT` command or in your code with the `S_SELBLD` subroutine. Leave the first character in each entry blank. This position will be used for the check mark.
2. In your program, load the selection window, and then call `S_UPDATECB` to mark any items that are already true. The check mark character is defined in `tkctl.def` by the global field `g_chk_char`. This field is initialized by `U_START` to an “X” on most environments. On UNIX and OpenVMS, you can change the check mark character by modifying the value of `g_chk_char`. On Windows, setting `g_chk_char` has no effect.
3. Call `S_SELECTCB` to enable the user to process the window and toggle the truth value of each item. This subroutine returns when the user selects a menu entry unrecognized by `S_SELECTCB`. An array is returned from this subroutine with the final values.

For more information, see [S_UPDATECB on page 12-18](#) and [S_SELECTCB on page 12-13](#).

S_SELBLD – Create a selection window



```
xcall S_SELBLD (window_id, window_name, fields, rows, array_data, [type], [quick_select],
&                [width], [default_entry][, nopad])
```

Arguments

window_id

Returned with the window ID. **(n)**

window_name

The name of the window to be created (a maximum of 15 characters). **(a)**

fields

The total number of items in the window, including text and lines. **(n)**

rows

The number of rows per column, including text and lines. **(n)**

array_data

The first element in the array containing the data to load. **(a)**

type

(optional) The array of flags indicating one of the following types of entries: **(n)**

D_SEL_ENTRY A field entry. (default)

D_SEL_TEXT Non-entry text.

D_SEL_LINE A horizontal line.

quick_select

(optional) The array of *quick_select* characters. **(a)**

width

(optional) The column width. The default width is `%len(array_data)`. **(n)**

default_entry

(optional) The default entry. **(n)**

nopad

(optional) The true/false flag that specifies whether to suppress the blank columns that occur to the left and right of each entry in the selection window. **(n)**

Discussion

The S_SELBLD subroutine creates a selection window and loads it with data.

You can specify a maximum of 32767 fields. The number of rows per column (*rows*) must be greater than zero. (Note, however, that because Synergy cannot place windows that do not fit on the screen, the practical maximum number of entries will be much lower. For instance, if the size of the Toolkit's virtual screen is 25 rows by 80 columns, a selection window cannot be placed if it has more than 2000 entries—and 2000 would only be possible if there is no padding between the columns and each entry is a single character.)

S_SELBLD loads the data from top to bottom and then left to right. The selection window will have a one-character margin on each side (unless you have passed *nopad*), except for horizontal line entries. Line entries in *array_data* will be overwritten by a horizontal line; use a blank element in *array_data*.

If you are processing with S_SELECT, on Windows systems, the window is automatically placed. On UNIX and OpenVMS, you must place the window with U_WINDOW(D_PLACE).

Use I_FLDSEL to associate the new selection window with a particular field.

Don't forget to log the window you create with the U_LOGWND subroutine. See [U_LOGWND on page 16-94](#).

If you pass a null string (“”) for *window_name*, Toolkit assigns a unique window name in the form *_W_nnn*, where *nnn* is the window ID.

If *quick_select* is not passed or is blank, the first non-blank character in each entry is used.

The elements in *array_data*, *type*, and *quick_select* correspond with each other.

On Windows, the *type* options D_SEL_TEXT and D_SEL_LINE, and the *quick_select* argument are ignored and serve only as placeholders. D_SEL_TEXT and D_SEL_LINE are not recognized because Windows environments do not categorize text and line items with selection items the user can select.

If *default_entry* is not specified, the first selectable entry in the selection window becomes the default entry. You can change or override the default entry with S_SELDFLT or when calling S_SELECT. *Default_entry* refers to the ordinal, selectable entry number (not counting text and line entries).

Note that horizontal lines do not connect with the selection window border.

If you have a selection window with more than one column, and that window does not have corresponding text or line entries in the same row of each column, left and right movement within the window won't be consistent. Therefore, we recommend that you either don't use text or line entries in a multi-column selection window, or make sure that each column in the selection window contains text or line entries in the same row.

Examples

The following example creates a window called choices with a maximum of 25 entries, 5 per column. It loads the data from **data**.

```
xcall s_selbld(selid, "choices", 25, 5, data)
```

The next example, a selection window named names, can be created with the data records listed below.

Team 1	Team 2	Team 3
Jim	Bill	Jan
Kim	Jill	Dan
Tim	Phil	Fran

```
data    ,15a6  ,"Team 1", " ", "Jim", "Kim", "Tim"
&      ,"Team 2", " ", "Bill", "Jill", "Phil"
&      ,"Team 3", " ", "Jan", "Dan", "Fran"
type    ,15d1  ,D_SEL_TEXT, D_SEL_LINE, D_SEL_ENTRY
&      ,D_SEL_ENTRY, D_SEL_ENTRY
&      ,D_SEL_TEXT, D_SEL_LINE, D_SEL_ENTRY
&      ,D_SEL_ENTRY, D_SEL_ENTRY
&      ,D_SEL_TEXT, D_SEL_LINE, D_SEL_ENTRY
&      ,D_SEL_ENTRY, D_SEL_ENTRY
```

```
xcall s_selbld(selid, "names", 15, 5, data, type)
```

S_SELDFLT – Set or return default entry for a selection window



```
xcall S_SELDFLT (window_id, [new_default] [, current_default])
```

Arguments

window_id

The ID of a loaded selection window. **(n)**

new_default

(optional) The item number of a new default entry. **(n)**

current_default

(optional) The returned item number of the current default entry (that is, the entry that was the default when S_SELDFLT was called). **(n)**

Discussion

S_SELDFLT sets the default entry for a selection window or a set of radio buttons or returns the last default set with S_SELDFLT. S_SELDFLT does not affect check box selection windows.

Selection items are numbered from top to bottom and from left to right. (Text and line entries are not numbered as items.)

Note the following:

- ▶ If *new_default* is not passed, is less than 1, or is greater than the number of items in the selection window, S_SELDFLT will not set a default.
- ▶ A default set with S_SELDFLT will override defaults specified with the DEFAULT option for .ITEM and the *default_entry* argument for S_SELBLD.
- ▶ If an input field is associated with selection window, and that field has a default (set with DEFAULT for .FIELD or D_FLD_DFLT for IB_FIELD), that default will override a default set with *new_default*.

The *current_default* argument is returned with the last default set with S_SELDFLT. If no default has been set with S_SELDFLT, *current_default* is returned with a value of 1. Note that S_SELDFLT does *not* return defaults set by the DEFAULT option for .ITEM, the *default_entry* argument for S_SELBLD, the DEFAULT option for .FIELD, or the D_FLD_DFLT qualifier for IB_FIELD.

Examples

The first S_SELDFLT call in the following example changes the default selection to **Bill**—assuming that **selid** is the selection window ID for the following selection window. (See S_SELBLD [Examples on page 12-7](#) for the code to create this selection window.)

Team 1	Team 2	Team 3
Jim	Bill	Jan
Kim	Jill	Dan
Tim	Phil	Fran

The second S_SELDFLT call retrieves the item number for the current default (which is 4) set with S_SELDFLT. Note that because *new_default* hasn't been passed in the second call, it doesn't change the default.

```
record
    selid          ,i4
    default_entry  ,i4

proc
.
.
.
xcall s_seldflt(selid, 4)
.
.
.
xcall s_seldflt(selid,, default_entry)
.
.
.
```

S_SELECT – Process a selection window



```
xcall S_SELECT (window_id, [initial_field], [field_data], [select_column], [selected_field],
&               [help_id][, wait_time])
```

Arguments

window_id

The ID of the selection window to process. **(n)**

initial_field

(optional) The number of the initial field. **(n)**

field_data

(optional) Returned with the data of the selected field. **(a)**

select_column

(optional) The selection column ID. **(n)**

selected_field

(optional) Returned with the ordinal number of the selected field. **(n)**

help_id

(optional) The help identifier. **(a)**

wait_time

(optional) The time-out limit for I/O processing to be completed before returning to the calling routine. **(n)**

D_FOREVER or **-1** Never time out.

D_GLOBAL or **-2** Use the global value (see [g_wait_time on page 18-6](#)). (default)

D_IMMEDIATE or **0** Time out immediately.

n Wait up to *n* seconds (where *n* is a positive value) for I/O processing to be complete.

Discussion

The S_SELECT subroutine processes selection windows.

On UNIX and OpenVMS, you must place the window with U_WINDOW(D_PLACE) before processing with S_SELECT. On Windows, the window is automatically placed; you don't need to call U_WINDOW(D_PLACE).

If *initial_field* is passed, the specified field will be initially highlighted before selection processing begins. If not passed, any previous call to S_SELDFLT or a default specified by S_SELBLD (or in a .SELECT window script using the .ITEM DEFAULT qualifier) will be used. In either case, if the resulting value is less than 1, 1 will be used. If *initial_field* is greater than the number of fields in the window, the last field will be highlighted.

If *field_data* is passed, it will be returned with the data of the highlighted field at the time that ENTER was pressed, or an unrecognized menu entry was selected.

On Windows, the *select_column* is ignored by default. This column usually contains reserved menu entries that are automatically supported by the Windows environment. Additionally, the excessive placing and removing of columns causes a Windows menu bar to flicker.

On UNIX and OpenVMS, if *select_column* is passed and non-zero, the specified column will be placed on the menu and removed on exit. The alternative to specifying this column is to place the column before calling S_SELECT.

The default behavior for each environment's handling of columns is controlled by the **g_plc_col_args** flag, which is defined in the distributed **tkctl.def** file. See [g_plc_col_args](#) on page 18-4 for information.

The following internal menu entry names are recognized by S_SELECT. Menu entries that start with E_ are recognized only on Windows and perform the associated Windows functionality, if applicable. See [“Appendix B: Reserved Menu Entries”](#) for information on reserved menu entries.

E_CLEAR	S_UP
E_COPY	S_DOWN
E_CUT	S_LEFT
E_MARK	S_RIGHT
E_PASTE	

If the name of the selected menu entry begins with “U_”, the EUTILS_METHOD subroutine is called and the input is redone. If the internal name of the selected menu entry is “O_HELP”, the EHELP_METHOD subroutine is called with *help_id* as its argument.

If some other menu entry is selected, control is returned to the calling routine. **G_select** will be true and **g_entnam** will hold the name of the entry.

You will need to clear **g_select** after you satisfy the event.

If the operation times out, **g_select** is set to true and **g_entnam** set to the entry name specified in **g_time_entry**, which is defined in **tkctl.def**. See [g_time_entry](#) on page 18-6.

On Windows, the mouse can be used to highlight and select entries.

Selection Routines

S_SELECT

The S_SELECT subroutine turns echo and the cursor off. Your calling routine should put an E_ENTER and E_EXIT around the call to this subroutine because S_SELECT does not do an E_ENTER, and it does not reset these parameters when done.



On Windows, for selection windows that are part of a composite window, TAB and SHIFT+TAB move focus to the next or previous child (unless TAB or SHIFT+TAB is a shortcut on a placed menu column). TAB instructs S_SELECT to signal "C_NEXT", and SHIFT+TAB instructs S_SELECT to signal "C_PREV".

Examples

This example loads a selection window and a menu column, then processes the selection window, starting on the third item. The S_SELECT subroutine returns the chosen selection's text in **selection** and its ordinal field number in **selnum**. If the user selects the menu entry "O_HELP," the EHELP_METHOD subroutine is called, passing the help ID argument of "sel_help."

```
xcall u_ldwnd(id, g_utlib, "select")
xcall m_ldcol(colid, g_utlib, "selcol", D_NOPLC)
xcall s_select(id, 3, selection, colid, selnum, "sel_help")
```


S_SELECTCB – Process a check box selection window



```
xcall S_SELECTCB (window_id, flag_array, [current_item], [select_col][, help_id])
```

Arguments

window_id

The ID of an existing selection window. **(n)**

flag_array

An array of true/false flags, which must have at least as many elements as the selection window has fields. **(n)**

current_item

(optional) The item number you want initially selected. It is returned with the final selected item number. **(n)**

select_col

(optional) The selection column ID. **(n)**

help_id

(optional) The help identifier. **(a)**

Discussion

S_SELECTCB enables the user to process a check box selection window.

When the user presses ENTER on an item, its truth value is toggled. If an entry is false, a space is displayed. If true, the check mark character, which is generally an “X”, appears in the first position (to the left) of the entry. On UNIX and OpenVMS, the value of **g_chk_char** determines the check mark character. On Windows, setting **g_chk_char** has no effect. This subroutine returns whenever an item is toggled or an unrecognized menu entry is signaled.

On Windows, the mouse can be used to highlight and toggle items.

Flag_array is an array of numeric true/false values. This array must already correspond to the visual truth values in the selection window (see [S_UPDATECB on page 12-18](#)) and must contain at least as many elements as there are selectable entries in the window.

On return, *flag_array* contains the updated true/false values resulting from the user’s input.

If you don’t pass *current_item*, S_SELECTCB defaults to initially highlight the first item in the window. When the user exits, *current_item* will return with the last item highlighted.

You can pass *select_col* with the column ID of an existing menu column that you want placed upon entry to, and removed upon exit from this subroutine.

Selection Routines

S_SELECTCB

If the name of the selected menu entry begins with “U_”, the EUTILS_METHOD subroutine is called and the input is redone. If the internal name of the selected menu entry is “O_HELP,” the EHELP_METHOD subroutine is called with *help_id* as its argument.

For information on designing check box selection windows, see [“Creating a check box selection window” on page 12-3](#).

See also

[S_SELECT on page 12-10](#) for a list of recognized menu entries.

S_SELINF – Retrieve selection window information

WT	WN	U	V

```
xcall S_SELINF (window_id, entry_number, entry_text, [fields][, numeric])
```

Arguments

window_id

The ID of the selection window. **(n)**

entry_number

The number of the selectable entry (supplied or returned). **(n)**

entry_text

The text of the selectable entry (supplied or returned). **(a)**

fields

(optional) Returned with the total number of selectable fields in the specified selection window. **(n)**

numeric

(optional) The true/false flag that specifies that *entry_text* is a numeric value. **(n)**

Discussion

The S_SELINF subroutine returns information about a specified selection window.

If *entry_number* is non-zero, S_SELINF retrieves the text for that entry and returns it in *entry_text*. If a matching entry is not found, *entry_text* returns a blank. Note that entry numbers don't include text or line entries, only selectable entries. If *entry_number* is zero, S_SELINF searches for *entry_text* in the specified selection window. If found, the associated entry number is returned in *entry_number*. If not found, zero is returned. Note that the match is case sensitive and leading blanks are not ignored. However, if there is a leading blank created by S_SELBLD or the script compiler, it is ignored.

If the selection window entries are numeric, pass true for the *numeric* argument.

If you want to get information about or modify a selection window whose ID you don't know, you can use the I_FLDINF subroutine to retrieve the ID of an attached selection window for a specified field.

Selection Routines

S_SELINF

Examples

With the following selection window, the first example returns “Bill” in **text**. The second example returns “4” in **enum** (provided that **enum** was zero upon entry) and “9” in **numflds**. (See S_SELBLD [Examples on page 12-7](#) for the code to create this selection window.)

Team 1	Team 2	Team 3
Jim	Bill	Jan
Kim	Jill	Dan
Tim	Phil	Fran

```
xcall s_selinf(wndid, 4, text)
xcall s_selinf(wndid, enum, "Bill", numflds)
```

S_SELLD – Load new selections into a selection window

WT	WN	U	V

```
xcall S_SELLD (window_id, array_data, [fields], [first_field][, quick_select])
```

Arguments

window_id

The ID of the selection window. (**n**)

array_data

The first element of the array containing the data to load, which does not include any replacement text and horizontal lines. (**a**)

fields

(optional) The number of fields to be loaded. Maximum in 100. If not specified, one field is loaded. (**n**)

first_field

(optional) The number of the field to load first. (**n**)

quick_select

(optional) The array of quick-select characters. (**a**)

Discussion

The S_SELLD subroutine replaces the selectable entries in an existing selection window (created using S_SELBLD).

If *first_field* is specified, the first element of the array is loaded into the specified field. The second element of the array is loaded into *first_field*+1, the third element into *first_field*+2, and so on.

If *first_field* is not specified, the first element of the array is loaded into the first field.

If *quick_select* is not passed or is blank, the first non-blank character in each entry is used.

If the selection window contains lines or non-selectable text, they are not overwritten by S_SELLD; only selectable entries are replaced. Use the Synergy DBL W_DISP subroutine to replace text in a selection window. (See [W_DISP](#) in the “Synergy Windowing API” chapter of the *Synergy DBL Language Reference Manual*.)

Examples

This example loads the selection window whose ID is stored in **selid** with the first 20 entries in **newdata**.

```
xcall s_selld(selid, newdata, 20)
```

S_UPDATECB – Update a check box selection window



```
xcall S_UPDATECB (window_id, num_flags, flag_array)
```

Arguments

window_id

The window ID of the loaded selection window. (n)

num_flags

The number of flags. (n)

flag_array

The array of true/false flags (which must have at least as many elements as the selection window has fields). (n)

Discussion

The S_UPDATECB subroutine updates the check mark characters in a loaded check box selection window.

Before you call S_UPDATECB, make sure the selection window meets the following criteria:

- ▶ The selection window was created with the .SELECT command, the SELECT option of the .FIELD command, or the S_SELBLD subroutine.
- ▶ Each entry in the selection window has a space as the first character. This space will be used for the check mark character.
- ▶ The selection window is already loaded.

Num_flags indicates both the number of elements in *flag_array* and the number of entries in the selection window.

Flag_array is an array of numeric true/false values. This subroutine displays a blank in the first position (the space) of each entry whose corresponding array element is false. For each true element, this subroutine displays the check mark character (which is generally an “X”) in the first position of the corresponding window entry. On UNIX and OpenVMS, the value of **g_chk_char** determines the check mark character. On Windows, setting **g_chk_char** has no effect.

For information on designing check box selection windows, see [“Creating a check box selection window” on page 12-3](#).

13

Tab Set Routines

Understanding Tab Set Processing 13-2

Discusses what a tab set is, why you would want to use one, and how tab set processing works. Also gives a brief overview of UI Toolkit tab set processing functions.

%TS_PROCESS – Process a tabbed dialog	13-4
%TS_TABSET – Create and manipulate a tabbed dialog	13-7

Understanding Tab Set Processing

Complex applications developed with Synergy/DE® often contain input requirements that will not fit on a single-screen input window. Historically, developers solved this problem either by using a single large window with a screen-sized movable display area or by using several windows, which were navigated with a PAGE UP/DOWN equivalent. A third solution is tabbed dialogs, where windows are rendered in the metaphor of tabbed file-folders.

The tab set processing functions in Toolkit enable you to associate windows or lists with the “tabs” of a tabbed dialog. Specific input window or list processing is handled using a method that you register with Toolkit.

The basic tab set functionality is as follows:

- ▶ A tab is associated with a window (or list) and a method, and is contained within a tab set. The associated window is placed immediately below (or beside) the set of tabs, and appears to be joined to its associated tab.
- ▶ When the tab is selected (with a menu entry or mouse click), the associated window (or list) is brought to the front, and the associated method routine is invoked.
- ▶ The entire set of tabs and windows is contained within a bounding window, which may also contain buttons.

A tab set is created with a call to `%TS_TABSET` using the `DTS_CREATE` subfunction. (Windows are added to the tab set with the `DTS_WINDOW` subfunction; lists are added with `DTS_LIST`.) The window or list is visually transformed into a tab page. It is placed adjacent to the tabs, a tab is created for it, and the border is rendered in a manner compatible with a tabbed page. On Windows, this is a blended 3-D effect.

On UNIX and OpenVMS systems, the top border of the window will be overlapped by a single-line window containing textually rendered tabs:

```
/Tab one \/Tab two \/Tab three \
```

Slashes are used to delimit the tabs, by default, and the underscore attribute is used to distinguish the inactive tabs from the active one. See the `DTS_CREATE`, `DTS_SELECT_ATTR`, and `DTS_NONSELECT_ATTR` subfunctions of `%TS_TABSET` on [page 13-7](#) for information on how to modify these separators and attributes.

By default, tabs are placed at the top of the tab set. You can change this by setting `DTS_BOTTOM`, `DTS_VERTICAL`, or `DTS_RIGHT` when creating a tab set. (Note, however, that for Windows, we recommend leaving tabs at the top of the tab set; `DTS_BOTTOM` and `DTS_VERTICAL` result in incorrectly drawn tabs.)

The text of the tab is the text of the window’s title, if any. Subsequently, the placement of the window is left at the discretion of the tab set.

If the size of the window or list is larger than the number of rows and columns specified for the tab set on `DTS_CREATE`, by default the window or list will be clipped. If a resize event is specified for the window or list, scroll bars will be displayed as necessary.

The order in which windows are added to the tab set defines the ordering of the tabs. Only one row of tabs will be displayed. If this is not sufficient to display all the tabs in the tab set, on Windows, scroll bars are used by default to indicate that more tabs exist. This style can be changed to use multi-line tabs instead of scroll bars by using the `DTS_MULTILINE` option when creating a tab set. On UNIX and OpenVMS systems, the active tab will always be visible.

A tab can be brought to the front programmatically by using the `DTS_ACTIVE` subfunction of `%TS_TABSET`.

To process a tab set, call `%TS_PROCESS`. When this routine is called, the method for the active tab is immediately invoked. When a tab switch occurs within `%TS_PROCESS`, the associated window is brought to the front, and its associated method is called. The only time that `%TS_PROCESS` returns is when a menu entry is selected that is not recognized by the currently active tab's method, nor by `%TS_PROCESS`.

To remove a tab from a tab set, use `DTS_REMOVE`. The user won't see the tab on the tab set, but the window will still reside in memory. You must use `U_WINDOW` with `D_DELETE` (or rely on an environment) to remove a window from memory.

To delete a tab set, do one of the following:

- ▶ Let an environment delete the tab set for you. By default the tab set container window is local to the current environment, so if you create and load the windows and tab set in an environment, Toolkit will delete the tab set and remove all the tab set windows from memory when you exit the environment.
- ▶ Use `U_WINDOW` with the `D_DELETE` operation. You can do this because a tab set container is a Toolkit window. (To use `D_DELETE`, you'll need to pass the window ID that was returned when the tab set was created.) However, individual tab windows (input windows, lists, and ActiveX containers) will not be deleted when you delete the tab set container window. You'll need to use `U_WINDOW` with `D_DELETE` to delete each of the individual tab windows (use `L_DELETE` for lists) after you delete the tab set container window.

%TS_PROCESS – Process a tabbed dialog



```
status = %TS_PROCESS (tabid[, method_arg, ...])
```

or

```
xcall TS_PROCESS (tabid[, method_arg, ...])
```

Return value

status

Always returns true. (^VAL)

Arguments

tabid

The ID of the tab set returned by the DTS_CREATE subfunction. (n)

method_arg

(optional) One of up to 20 arguments that are passed to the method for each tab as it becomes active.

Discussion

The %TS_PROCESS function enables your applications to process tabbed dialogs. When this function is called, the method for the active tab is immediately invoked. If no tab has been designated as the active tab (with DTS_ACTIVE), then the first defined tab is presumed. When the user clicks on a tab, the menu entry “TS_TAB*n*” is generated, where *n* is the index of the desired tab (there are exceptions for composite windows; see [“Controlling tabbing, focus, and input context” on page 6-4](#)). The tab method may “see” this menu entry and perform any cleanup or preparation for tab switching that may be needed, but it must allow this menu entry to be returned from the method in order for the automatic tab switching to occur. Clearing **g_select** or modifying **g_entnam** at this point will disable the tab switch.

On UNIX and OpenVMS, “TS_TAB*n*” menu entries can be placed on the menu to achieve immediate tabbing. Also, the entries “TS_TABNEXT” and “TS_TABPREV” will create the same effect as signaling the corresponding next or previous “TS_TAB*n*” menu entry. On Windows, CTRL+TAB and CTRL+SHIFT+TAB will be automatically mapped to these latter two entries, respectively. For cross-platform compatibility, place any “TS_XXX” menu entries on columns whose IDs are passed to Toolkit input routines. (By default, on UNIX and OpenVMS, these will be placed; on Windows, they won’t be placed.)

When a tab switch occurs within %TS_PROCESS, the associated window is brought to the front, and its associated method subroutine is called, passing the associated ID (window ID or list ID) and up to 20 optional arguments that were passed to %TS_PROCESS.

When the method for a tab returns, if a menu entry has been signaled (**g_select** is true), then if it is a menu entry which %TS_PROCESS recognizes (TS_xxx), it will be processed by %TS_PROCESS and the method for the active tab will be called. If the menu entry is not recognized by %TS_PROCESS, it is returned to the routine that called %TS_PROCESS. If no menu entry has been signaled (**g_select** is false) when the method returns, then the method is immediately called again.

If the method for a tab doesn't allow a TS_xxx menu entry to be returned to %TS_PROCESS, the method should reset the active tab by calling %TS_TABSET with the DTS_ACTIVE subfunction. Otherwise, the tab which is forward will not match the window displayed within. To prevent the tab from switching in the first place, remove or disable the other tabs.

Calls to %TS_TABSET are allowed while within %TS_PROCESS, even if they reference the same tab set.



We do not recommend using required fields in a tab set because when the user clicks OK, the tab set field validation behavior does not follow general Windows conventions. Instead, we recommend filling the windows with valid data, so that required field validation does not occur.

The only time that %TS_PROCESS returns is when a menu entry is selected that is not processed by the currently active tab's method subroutine, nor recognized by %TS_PROCESS.

Examples

The following subroutine loads a tab set and processes it:

```
subroutine process_tabs

.include "WND:tools.def"
record
    tabid          ,int
    inpid1         ,int
    inpid2         ,int
    lstid          ,int
    inp_record     ,a10
    list_record    ,a40
proc

    ;Create a tab container large enough for windows of 14 x 70.
    tabid = %ts_tabset(DTS_CREATE, "tab_container", 14, 70)

    ;Create the first tab as an input window.
    xcall i_ldinp(inpid1,, "i_tab1")
    xcall ts_tabset(DTS_WINDOW, tabid, inpid1, "inp_method")

    ;Create the second tab as a list.
    xcall i_ldinp(inpid2,, "i_tab2")
    xcall l_create(lstid, inpid2, list_record, g_utlib, "l_tab2")
```

Tab Set Routines

%TS_PROCESS

```
xcall ts_tabset(DTS_LIST, tabid, lstid, "list_method")

;Add a button to the tab set.
xcall ts_tabset(DTS_BUTTON, tabid, "O_EXIT", DSB_TEXT, "Done")

;Main input loop
repeat
  begin
    xcall ts_process(tabid, inp_record, list_record)
    if (g_select)
      begin
        using g_entnam select
          ("O_EXIT "), exitloop
        ;Add other cases as needed.
      endusing
    end
  end
xreturn
end

; Tab method for the input window
subroutine inp_method
  a_inpid      ,n    ;The input window ID
  a_inp_record ,a    ;The input record
  a_list_record ,a    ;The list record, though we don't use it
here
proc
  ;Just perform input on the default set of the window,
  ; updating a_inp_record.
  xcall i_input(a_inpid,, a_inp_record)
  xreturn
end

; Tab method for the list
subroutine list_method
  a_lstid      ,n    ;The list's ID
  a_inp_record ,a    ;The input record, though we don't use
  ; it here
  a_list_record ,a    ;The list record
record
  req          ,int ;A variable for the list request code
proc
  ;Just select an entry, and place its record in a_list_record.
  xcall l_select(a_lstid, req=D_LNOP, a_list_record)
  xreturn
end
```

%TS_TABSET – Create and manipulate a tabbed dialog



return = %TS_TABSET(*subfunction*, *arg*[, ...])

or

xcall TS_TABSET(*subfunction*, *arg*[, ...])

Return value

return

A value specific to the subfunction. (^VAL)

Arguments

subfunction

One of the following: (n)

DTS_CREATE	Create a tab set.
DTS_WINDOW	Add a window to a tab set.
DTS_LIST	Add a list to a tab set.
DTS_BUTTON	Add a button to a tab set. (Windows only)
DTS_BUTTONSET	Modify the button set for a tab set. (Windows only)
DTS_ACTIVE	Bring a tab to the front.
DTS_ENABLE	Enable a tab in a tab set.
DTS_DISABLE	Disable a tab in a tab set.
DTS_REMOVE	Remove a tab from a tab set.
DTS_SELECT_ATTR	Modify the attributes for the selected tab. (UNIX, VMS only)
DTS_NONSELECT_ATTR	Modify the attributes for non-selected tabs. (UNIX, VMS only)
DTS_DEFAULT	Modify the default tab settings.
DTS_GETDEFAULT	Obtain the default tab settings.
DTS_NUMTABS	Obtain the number of tabs in a tab set.
DTS_TABINDEX	Obtain the tab index for a window or list.
DTS_TABINFO	Obtain information for a tab.

Tab Set Routines

%TS_TABSET

DTS_TABSET

Obtain the ID of the tab set that contains a specified window or list. (UNIX, VMS only)

DTS_CTRCONTEXT

Set context of window on tab.

arg

One or more arguments, depending on *subfunction*.

Discussion

The %TS_TABSET function enables you to create and manipulate a tabbed dialog. See the individual subfunctions for details.

Examples

See the %TS_PROCESS [Examples on page 13-5](#) and the DTS_CREATE [Examples on page 13-11](#).

DTS_CREATE

```

window_id = %TS_TABSET (DTS_CREATE, name, rows, cols, [pre_tab_string],
&                      [post_tab_string][, style])

or

xcall TS_TABSET (DTS_CREATE, name, rows, cols, [pre_tab_string], [post_tab_string]
&                [, style])

```

Return value

window_id

The window ID of the created tab set container window. (^VAL)

Arguments

name

A name for the created tab set (a maximum of 15 characters). (a)

rows

The maximum number of rows that a contained window may contain. (n)

cols

The maximum number of columns that a contained window may contain. (n)

pre_tab_string

(optional) The string to put before each tab in a tab set container window. (UNIX, OpenVMS only) (a)

post_tab_string

(optional) The string to put after each tab in a tab set container window. (UNIX, OpenVMS only) (a)

style

(optional) One or more of the following options, combined using .BOR. or the vertical bar (|) operator. (Windows only) (n)

DTS_NONE	None of the following styles apply. (default)
DTS_MULTILINE	Multiple lines of tabs.
DTS_VERTICAL	Left or right side.
DTS_BOTTOM	Bottom (or right if DTS_VERTICAL is also specified).
DTS_RIGHT	Right (or bottom if DTS_VERTICAL is <i>not</i> specified).

Discussion

The `DTS_CREATE` subfunction of `%TS_TABSET` creates a tab set. A tab set can have up to 24 tabs. Since the window created is a UI Toolkit window, `U_WINDOW` can be used to place, remove, move, or delete the tab set as a whole. Deleting the tab set does not delete its member windows, but does disassociate them from the tab set. (See [“Understanding Tab Set Processing” on page 13-2](#) for more information.) Note also that the containing window is by default local to the current environment (for instance, it will be deleted by a subsequent `E_EXIT`), but it may also be promoted to global status with `U_GBLWND`.

Rows and *cols* must be greater than or equal to the height and width, respectively, of the window specified with `DTS_WINDOW`.

UNIX, VMS

Pre_tab_string and *post_tab_string* are used to define the two delimiter strings. The maximum length of each string is 10 characters. If *pre_tab_string* or *post_tab_string* is not passed, the default strings “*P*” and “*V*” will be used, respectively. If *post_tab_string* is specified as an empty string (“”), the string specified in *pre_tab_string* will be repeated after the last tab. If *pre_tab_string* is an empty string, the container window’s border acts as a border for the first tab.

WIN

When a tab set is created, the global default style is used. The global default style is determined by either the style specified in a prior call to `%TS_TABSET(DTS_DEFAULT)`, the style specified by the `TABSET_STYLE` initialization setting in **synergy.ini**, or the Toolkit’s default style. Toolkit’s default style is to display tabs on a single line (scrolling as necessary) on the top of the window.

You can override the default style currently in effect by specifying *style* for the `DTS_CREATE` subfunction. The style options for a tabbed dialog cannot be changed after a tab set has been created. *Style* options should be combined using `.BOR.` or the vertical bar (`|`) operator.

To restore the default style in effect back to the Toolkit default style, specify `DTS_NONE`.

If `DTS_MULTILINE` is specified, the tabs display on multiple rows if they cannot fit on one row. This will use more container space for the tabs, so you may need to adjust the number of rows and columns specified in *rows* and *cols* to allow the contained windows to display fully. If tabs are displayed on the top or left, contained windows will be shifted down or to the right, respectively. This means that rows or columns at the bottom or right of the contained window may be truncated.

If `DTS_VERTICAL` is specified, tabs are displayed vertically on the left or right side of the contained windows with the text rotated 90 degrees to follow the vertical border. Setting `DTS_VERTICAL` forces `DTS_MULTILINE` to be set.

If `DTS_RIGHT` or `DTS_BOTTOM` is specified along with `DTS_VERTICAL`, the tabs are displayed on the right side. If `DTS_RIGHT` or `DTS_BOTTOM` is specified and `DTS_VERTICAL` is not, the tabs are displayed on the bottom.

See also

[DTS_DEFAULT](#) on page 13-28 for setting the default style.

Examples

On UNIX and OpenVMS, the following example sets the tab delimiter to “|”, by setting *pre_tab_string* to “|” and *post_tab_string* to “”, as shown below.

```
tabid = %ts_tabset(DTS_CREATE, "Tabset1", 15, 50, "|", "")
```

```
!Tab 1|Tab 2|Tab 3!
```

On Windows, the following example specifies multi-line tabs on the bottom of the tab set.

```
tabid = %ts_tabset(DTS_CREATE, "Tabset2", 10, 30,,,
& DTS_MULTILINE .bor. DTS_BOTTOM)
```

Tab Set Routines

%TS_TABSET

DTS_WINDOW

index = %TS_TABSET (DTS_WINDOW, *tab_id*, *window_id*, *method*, [*elb*][, *tooltip*])

or

xcall TS_TABSET (DTS_WINDOW, *tab_id*, *window_id*, *method*, [*elb*][, *tooltip*])

Return value

index

The tab index of the new tab if successful, or 0 if not. (^VAL)

Arguments

tab_id

The ID of the tab set returned by the DTS_CREATE subfunction. (n)

window_id

The ID of a window to add to the tab set. (n)

method

The name of the subroutine to call when this tab is selected. (a)

elb

(optional) The **.elb** filename in which to find *method*. (a)

tooltip

(optional) The text of a ToolTip to be displayed if the mouse rests over the tab. (a)

Discussion

The DTS_WINDOW subfunction of %TS_TABSET adds a window to a tab set.

The window referenced by *window_id* is visually transformed into a tab page: It is placed adjacent to the tabs, a tab is created for it, the border is turned off, and the window is made to appear to be a tabbed page.

When a tab is created, the window is sized to fit the tab. If the window is larger than the tab, the display area is resized, and scroll bars appear. If the window is smaller than the tab, the blank portion of the tab is filled with the background window color. When a tab is resized, the window display area is also resized.



When a window is added to a tab set, UI Toolkit changes the color of the window to the color palette entry specified by the **g_tabfacecolor** global variable, if this variable is set to a value in the range 1 through 16. On Windows, **g_tabfacecolor** is set to 9 by default if Windows themes are used. On UNIX and OpenVMS or on Windows if themes are not enabled, the default is 0, a setting that instructs Toolkit to leave a window's color as is.

When a window is removed from a tab set, Toolkit does not restore the original color for the window. If the color was changed when the window was added to a tab set, that change will stand until the program or the user changes the renditions for the window.

Additionally, if unframed fields are displayed or cleared (initialized) after the input window for the fields is added to a tab set, Toolkit will apply the Input Field Display rendition or the Cleared Input Field rendition to the fields—renditions that might not match the **g_tabfacecolor** palette entry (if that has been applied to the field). In this case, do one of the following:

- ▶ Before performing the display (I_DISPLAY or I_DSPFLD) or clear (I_INIT), use U_REND to set the Input Field Display rendition or the Cleared Input Field rendition to **g_tabfacecolor**.
- ▶ After performing the display or clear, overwrite the renditions using W_AREA. For example:

```
xcall w_area(wndid, WA_COPY, WAC_WTOP, WA_COLOR,
&
           g_tabfacecolor)
```

For more information, see [g_tabfacecolor on page 18-5](#).

On UNIX and OpenVMS, the tabs are rendered as a single-line window. By default, slashes are used to delimit the tabs, and the underscore attribute is used to distinguish the inactive tabs from the active tab. (See under “[Understanding Tab Set Processing](#)” on page 13-2 for an example.)

Tab Set Routines

%TS_TABSET

DTS_LIST

index = %TS_TABSET (DTS_LIST, *tab_id*, *list_id*, *method*, [*elb*][, *tooltip*])

or

xcall TS_TABSET (DTS_LIST, *tab_id*, *list_id*, *method*, [*elb*][, *tooltip*])

Return value

index

The tab index of the new tab if successful, or 0 if not. (^VAL)

Arguments

tab_id

The ID of the tab set returned by the DTS_CREATE subfunction. (n)

list_id

The ID of a list to add to the tab set. (n)

method

The name of the subroutine to call when this tab is selected. (a)

elb

(optional) The .elb filename in which to find *method*. (a)

tooltip

(optional) The text of a ToolTip to be displayed if the mouse rests over the tab. (a)

Discussion

The DTS_LIST subfunction of %TS_TABSET adds a list to a tab set.

This subfunction is similar to the DTS_WINDOW subfunction, except that it's for lists. By default, when a tab is created, the list is sized to fit the tab regardless of the number of list items you've specified to display. If the tab is resized, the list is also resized. To override this behavior, specify the NOAUTOSIZE list class option.

DTS_BUTTON

```
xcall TS_TABSET (DTS_BUTTON, tab_id, name, [type], [type_data], [method], [elb],
&                                     [select_char], [position][, tooltip])
```

Return value

The return value of this subfunction is undefined; we recommend you use XCALL TS_TABSET in this case.

Arguments

tab_id

The ID of the tab set returned by the DTS_CREATE subfunction. **(n)**

name

The name of the button to add (a maximum of 30 characters). **(a)**

type

(optional) The button face type: **(n)**

DSB_TEXT Text. (default)

DSB_BITMAP Graphic.

type_data

(optional) Text if *type* is DSB_TEXT. The filename for a graphic if *type* is DSB_BITMAP. **(a)**

method

(optional) The name of the subroutine to call when this button is selected. **(a)**

elb

(optional) The name of the library from which to draw *method*. **(a)**

select_char

(optional) The quick-select character. **(a)**

position

(optional) The position the button will be inserted in the button set. **(a or n)**

tooltip

(optional) The text of a ToolTip to be displayed when the mouse pointer rests over the button. **(a)**

Discussion

This subfunction is only supported on Windows. (On UNIX and OpenVMS, it is ignored.)

The DTS_BUTTON subfunction of %TS_TABSET adds a button to a tab set. If there are no existing buttons, a default button set is created.

By default, button sets are placed at the bottom of the tab set, are left-justified, and can have as many buttons on a row as will fit before the row wraps. You can change these defaults with B_BUTTONSET.

The tab set will expand downward or to the right to accommodate the buttons, which may cause your window to extend beyond the bottom or right side of the application window. Also, buttons are a standard size; text that exceeds that size will not be visible.

For information on the arguments for this subroutine, see the DSB_ADD [Discussion on page 5-6](#). All the comments in that section apply to tab sets.

DTS_BUTTONSET

```
xcall TS_TABSET (DTS_BUTTONSET, tab_id, [placement], [per_row_col], [justification]
&                                     [, button_name])
```

Return value

The return value of this subfunction is undefined; we recommend you use XCALL TS_TABSET in this case.

Arguments

tab_id

The ID of the tab set returned by the DTS_CREATE subfunction. **(n)**

placement

(optional) One of the following: **(n)**

DSB_BOTTOM The set of buttons will be placed horizontally along the bottom of the window with a maximum of *per_row_col* buttons per row. (default)

DSB_RIGHT The set of buttons will be placed vertically along the right side of the window with a maximum of *per_row_col* buttons per column.

per_row_col

(optional) The maximum number of buttons to place on a row if DSB_BOTTOM is specified, or on a column if DSB_RIGHT is specified. The default behavior is to display as many as will fit and wrap to the next row or column. **(n)**

justification

(optional) One of the following: **(n)**

DSB_BEGIN The buttons will be left-justified if DSB_BOTTOM is specified, or positioned towards the top if DSB_RIGHT is specified. (default)

DSB_CENTER The buttons will be centered along the designated side.

DSB_END The buttons will be right-justified if DSB_BOTTOM is specified, or positioned towards the bottom if DSB_RIGHT is specified.

button_name

(optional) The name of the default button. **(a)**

Discussion

This subfunction is only supported on Windows. (On UNIX and OpenVMS, it is ignored.)

The DTS_BUTTONSET subfunction of %TS_TABSET modifies the button set for a tab set.

If a tab set container and its contained window both have buttons, the following rules are used to determine which button is the default:

- ▶ If only one, either the contained window or the tab set container, defines a default button, the specified button is the default button.
- ▶ If neither the contained window nor the tab set container defines a default button, the first button on the contained window is the default button.
- ▶ If both the contained object and the tab set container define a default button, the default button of the contained window is the default button.

If the E_STATE option D_RETURNBTN is set, the default button will be selected if you press RETURN, no matter what field you're on in the window. If *button_name* is not specified and D_RETURNBTN is set, the first button in the window is the default.

See also

- ▶ [B_BUTTONSET on page 5-14](#) for more information on defining the placement of a set of buttons.
- ▶ [.BUTTON_SET on page 2-17](#) for defining the placement of a window's button set in a script file.
- ▶ [L_BUTTON on page 9-55](#) for creating buttons within a list.
- ▶ [L_BUTTONSET on page 9-62](#) for information about specifying button placement for a list.
- ▶ [E_STATE on page 7-41](#) for information about the ENTER key selecting the default button.

DTS_ACTIVE

index = %TS_TABSET (DTS_ACTIVE, *tab_id*[, *new_index*])

or

xcall TS_TABSET (DTS_ACTIVE, *tab_id*[, *new_index*])

Return value

index

If *new_index* is passed, the index of the previously active tab. Otherwise, the index of the currently active tab. (^VAL)

Arguments

tab_id

The ID of the tab set returned by the DTS_CREATE subfunction. (n)

new_index

(optional) The index of the tab to bring to the front, which will be processed on the next call to %TS_PROCESS. (See [%TS_PROCESS on page 13-4](#) for more information.) (n)

Discussion

The DTS_ACTIVE subfunction of %TS_TABSET brings a tab to the front.

Tab Set Routines

%TS_TABSET

DTS_ENABLE

```
xcall TS_TABSET(DTS_ENABLE, tab_id, index)
```

Arguments

tab_id

The ID of the tab set returned by the DTS_CREATE subfunction. **(n)**

index

The index of the tab to enable. **(n)**

Discussion

The DTS_ENABLE subfunction of %TS_TABSET enables a tab in a tab set.

Index is the tab index returned from the DTS_WINDOW or DTS_LIST subfunctions.

DTS_DISABLE

```
xcall TS_TABSET(DTS_DISABLE, tab_id, index)
```

Arguments

tab_id

The ID of the tab set returned by the DTS_CREATE subfunction. (n)

index

The index of the tab to disable. (n)

Discussion

The DTS_DISABLE subfunction of %TS_TABSET disables a tab in a tab set. Be aware that the disabled tab is not rendered visually. For behavior that is more Windows-like, we recommend removing the tab.

Index is the tab index returned from the DTS_WINDOW or DTS_LIST subfunctions.

Tab Set Routines

%TS_TABSET

DTS_REMOVE

curr_index = %TS_TABSET (DTS_REMOVE, *tab_id*, *index*)

or

xcall TS_TABSET (DTS_REMOVE, *tab_id*, *index*)

Return value

curr_index

The index of the active tab. (^VAL)

Arguments

tab_id

The ID of the tab set returned by the DTS_CREATE subfunction. (n)

index

The index of the tab to remove. (n)

Discussion

The DTS_REMOVE subfunction of %TS_TABSET removes a tab from a tab set. (It does not remove the tab window from memory. See [“Understanding Tab Set Processing” on page 13-2](#) for more information.)

Index is the tab index returned from the DTS_WINDOW or DTS_LIST subfunctions.



Although UI Toolkit may change the color palette entry for a window when the window is added to a tab set, Toolkit does not restore the original palette entry setting for the window when it is removed from the tab set. See [DTS_WINDOW on page 13-12](#) for information.

DTS_SELECT_ATTR

status = %TS_TABSET (DTS_SELECT_ATTR, *tab_id*, [*option+attribute*][, *palette#*])

or

xcall TS_TABSET (DTS_SELECT_ATTR, *tab_id*, [*option+attribute*][, *palette#*])

Return value

status

True if successful; false if not. (^VAL)

Arguments

tab_id

The ID of the tab set returned by the DTS_CREATE subfunction. (n)

option

One of the following: (n)

DA_SET Set attributes.

DA_CLR Clear attributes.

attribute

One of the following flags defined in **tools.def**: (n)

A_NONE No attributes.

A_U Underlined.

A_R Reverse.

A_RU Reverse, underlined.

A_B | A_I Blinking (UNIX and OpenVMS) or italic (Windows). “I” can be substituted whenever “B” appears in the following flags.

A_BU Blinking, underlined.

A_BR Blinking, reverse.

A_BRU Blinking, reverse, underlined.

A_H Highlighted.

A_HU Highlighted, underlined.

A_HR Highlighted, reverse.

A_HRU Highlighted, reverse, underlined.

A_BH Blinking, highlighted.

Tab Set Routines

%TS_TABSET

A_BHU	Blinking, highlighted, underlined.
A_BHR	Blinking, highlighted, reverse.
A_BHRU	Blinking, highlighted, reverse, underlined.
A_ALL	All attributes (same as A_BHRU).

palette#

(optional) A palette number, in the range 1 to 16, to set for the selected tab. (**n**)

Discussion

This subfunction is only supported on UNIX and OpenVMS. (On Windows, it is ignored.)

The DTS_SELECT_ATTR subfunction of %TS_TABSET sets the rendition for the selected tab in a tab set. These renditions apply to the currently selected tab as well as any tab selected in the future.

Using DTS_SELECT_ATTR, you can set and clear various attributes for the selected tab to modify the effect of which tab is selected. See the example below.

If *option+attribute* is not passed, the default attribute will be used. The default is DA_CLR+A_U, unless it is modified through %TS_TABSET(DTS_DEFAULT). It is important to note that whatever attributes are specified using DTS_SELECT_ATTR overwrite the default attributes for a selected tab. For example, if DA_SET+A_R is specified, and the current default is DA_SET+A_U, the attribute of the selected tab will be reverse *only*.

If *palette#* is not passed, the default color will be used. The initial default color palette number is 1. If *palette#* is specified, but not between 1 and 16, a fatal error will be generated.

Changes to the selected tab's renditions will take effect either at the next call to the %TS_PROCESS function using *tab_id*, or within a call to %TS_PROCESS as a result of returning from a tab method.

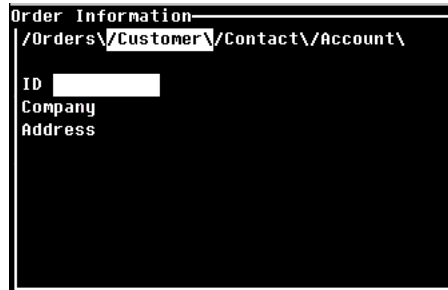
See also

- ▶ [DTS_DEFAULT on page 13-28](#) for information about setting the default rendition for tab sets.
- ▶ [DTS_NONSELECT_ATTR on page 13-26](#) for information about setting the rendition for non-selected tabs in a tab set.
- ▶ The “[Synergy Windowing API](#)” chapter of the *Synergy DBL Language Reference Manual* for more information on the effects of setting, changing, loading, and clearing attributes.
- ▶ “[Colors and the color palette](#)” in the “[Synergy Windowing API](#)” chapter of the *Synergy DBL Language Reference Manual* for more information on using color palettes.

Examples

The following example sets the rendition for the selected tab, in this case, Tab 2, in tab set **tabid** to be reverse video, and clears reverse non-selected tabs.

```
if (%TS_TABSET(DTS_SELECT_ATTR, tabid, DA_SET+A_R))
  status = %TS_TABSET(DTS_NONSELECT_ATTR, tabid, DA_CLR+A_U)
```



DTS_NONSELECT_ATTR

status = %TS_TABSET (DTS_NONSELECT_ATTR, *tab_id*, [*option+attribute*][, *palette#*])

or

xcall TS_TABSET (DTS_NONSELECT_ATTR, *tab_id*, [*option+attribute*][, *palette#*])

Return value

status

True if successful; false if not. (^VAL)

Arguments

tab_id

The ID of the tab set returned by the DTS_CREATE subfunction. (n)

option

One of the following: (n)

DA_SET Set attributes.

DA_CLR Clear attributes.

attribute

One of the following flags defined in **tools.def**: (n)

A_NONE No attributes.

A_U Underlined.

A_R Reverse.

A_RU Reverse, underlined.

A_B | A_I Blinking (on UNIX and OpenVMS) or italic (on Windows). “I” can be substituted whenever “B” appears in the following flags.

A_BU Blinking, underlined.

A_BR Blinking, reverse.

A_BRU Blinking, reverse, underlined.

A_H Highlighted.

A_HU Highlighted, underlined.

A_HR Highlighted, reverse.

A_HRU Highlighted, reverse, underlined.

A_BH Blinking, highlighted.

A_BHU	Blinking, highlighted, underlined.
A_BHR	Blinking, highlighted, reverse.
A_BHRU	Blinking, highlighted, reverse, underlined.
A_ALL	All attributes (same as A_BHRU).

palette#

(optional) A palette number, in the range 1 to 16, to set for the selected tab. (**n**)

Discussion

This subfunction is only supported on UNIX and OpenVMS. (On Windows, it is ignored.)

The DTS_NONSELECT_ATTR subfunction of %TS_TABSET sets the rendition for non-selected tabs in a tab set. These renditions apply to the currently non-selected tabs as well as any unselected tabs in the future.

Using DTS_NONSELECT_ATTR, you can set and clear various attributes for the non-selected tabs to modify the effect of which tabs aren't selected. See the %TS_TABSET [Examples on page 13-25](#).

If *option+attribute* is not passed, the default attribute will be used. The default is DA_SET+A_U, unless it is modified through %TS_TABSET(DTS_DEFAULT). It is important to note that whatever attributes are specified using DTS_NONSELECT_ATTR overwrite the default attributes for unselected tabs. For example, if DA_SET+A_R is specified, and the current default is DA_SET+A_U, the attribute of the non-selected tab will be reverse *only*.

If *palette#* is not passed, the default color will be used. The initial default color palette number is 1. If *palette#* is specified, but not between 1 and 16, a fatal error will be generated.

Changes to the non-selected tab's renditions will take effect either at the next call to the %TS_PROCESS function using *tab_id*, or within a call to %TS_PROCESS as a result of returning from a tab method.

See also

- ▶ [DTS_DEFAULT on page 13-28](#) for information about setting the default rendition for tab sets.
- ▶ [DTS_SELECT_ATTR on page 13-23](#) for information about setting the rendition for the selected tab in a tab set.
- ▶ The “[Synergy Windowing API](#)” chapter of the *Synergy DBL Language Reference Manual* for more information on the effects of setting, changing, loading, and clearing attributes.
- ▶ “[Colors and the color palette](#)” in the “[Synergy Windowing API](#)” chapter of the *Synergy DBL Language Reference Manual* for more information on using color palettes.

Examples

See the DTS_SELECT_ATTR [Examples on page 13-25](#).

DTS_DEFAULT

status = %TS_TABSET (DTS_DEFAULT, DTS_SELECT_ATTR, [*option+attribute*][, *palette#*])

status = %TS_TABSET (DTS_DEFAULT, DTS_NONSELECT_ATTR, [*option+attribute*]
[, *palette#*])

status = %TS_TABSET (DTS_DEFAULT, DTS_TABTEXT, [*pre_tab_string*][, *post_tab_string*])

status = %TS_TABSET (DTS_DEFAULT, DTS_STYLE [, *style*])

Return value

status

True if successful; false if not. (^VAL)

Arguments

DTS_SELECT_ATTR

Set rendition for the selected tab in a tab set. (UNIX, OpenVMS only) (n)

DTS_NONSELECT_ATTR

Set rendition for non-selected tabs in a tab set. (UNIX, OpenVMS only) (n)

option

One of the following:

DA_SET	Set attributes.
DA_CHNG	Change existing attributes.
DA_LOAD	Set listed attributes and clear all others.
DA_CLR	Clear attributes.

attribute

One of the following flags defined in **tools.def**:

A_NONE	No attributes.
A_U	Underlined.
A_R	Reverse.
A_RU	Reverse, underlined.
A_B A_I	Blinking.
A_BU	Blinking, underlined.
A_BR	Blinking, reverse.
A_BRU	Blinking, reverse, underlined.

A_H	Highlighted.
A_HU	Highlighted, underlined.
A_HR	Highlighted, reverse.
A_HRU	Highlighted, reverse, underlined.
A_BH	Blinking, highlighted.
A_BHU	Blinking, highlighted, underlined.
A_BHR	Blinking, highlighted, reverse.
A_BHRU	Blinking, highlighted, reverse, underlined.
A_ALL	All attributes (same as A_BHRU).

palette#

(optional) Palette number, in the range 1 to 16, to set for the selected tab. **(n)**

DTS_TABTEXT

Specify text to put before and after each tab in a tab set container window. (UNIX, OpenVMS only) **(n)**

pre_tab_string

(optional) String to put before each tab in a tab set container window. **(a)**

post_tab_string

(optional) String to put after each tab in a tab set container window. **(a)**

DTS_STYLE

Specify a tabbed dialog. **(n)**

style

One or more of the following options, combined using .BOR. or the vertical bar (|) operator. (Windows only) **(n)**

DTS_NONE	None of the following styles apply. (default)
DTS_MULTILINE	Multiple lines of tabs.
DTS_VERTICAL	Left or right side.
DTS_BOTTOM	Bottom (or right if DTS_VERTICAL is also specified).
DTS_RIGHT	Right (or bottom if DTS_VERTICAL is <i>not</i> specified).

Discussion

The DTS_DEFAULT subfunction of %TS_TABSET sets default settings for tab sets. Defaults will be used for the next tab set created using the %TS_TABSET(DTS_CREATE) function.

DTS_DEFAULT overrides Toolkit's default styles and any styles set by the TABSET_STYLE initialization setting.

If any of the optional arguments are not passed for a given *function*, no default settings will be set for that function.

On UNIX and OpenVMS, *pre_tab_string* and *post_tab_string* can be used to define the two delimiter strings. The maximum length of each string is 10 characters. If *post_tab_string* is specified as an empty string (""), the string specified in *pre_tab_string* will be repeated after the last tab. If *pre_tab_string* is an empty string, the container window's border acts as a border for the first tab.

See also

- ▶ [DTS_SELECT_ATTR on page 13-23](#) for information about setting the rendition for the selected tab in a tab set.
- ▶ [DTS_NONSELECT_ATTR on page 13-26](#) for information about setting the rendition for non-selected tabs in a tab set.
- ▶ [DTS_CREATE on page 13-9](#) for information about creating a tabbed dialog with specific delimiter strings and tabbed dialog style.
- ▶ [TABSET_STYLE](#) in the “Environment Variables” chapter of *Environment Variables & System Options* for setting the global default style in **synergy.ini**.
- ▶ “[Colors and the color palette](#)” in the “Synergy Windowing API” chapter of the *Synergy DBL Language Reference Manual* for more information on using color palettes.
- ▶ The “[Synergy Windowing API](#)” chapter of the *Synergy DBL Language Reference Manual* for more information on effects of setting, changing, loading, and clearing attributes.

Examples

The following example sets the default rendition for non-selected tabs to use reverse video, underline, and color palette number 10.

```
status = %TS_TABSET(DTS_DEFAULT, DTS_NONSELECT, DA_SET+A_RU, 10)
```

The following example sets the default tab delimiter to be “|”.

```
status = %TS_TABSET(DTS_DEFAULT, DTS_TABTEXT, "|", "")
```

DTS_GETDEFAULT

status = %TS_TABSET (DTS_GETDEFAULT, *function*)

or

xcall TS_TABSET (DTS_GETDEFAULT, *function*)

Discussion

The DTS_GETDEFAULT subfunction of %TS_TABSET can be called to get default settings for tab sets. It uses the same syntax as [DTS_DEFAULT on page 13-28](#).

If any of the optional arguments are not passed for a given *function*, no default settings will be retrieved for that function.

Tab Set Routines

%TS_TABSET

DTS_NUMTABS

tab_count = %TS_TABSET (DTS_NUMTABS, *tab_id*)

or

xcall TS_TABSET (DTS_NUMTABS, *tab_id*)

Return value

tab_count

The number of tabs currently in the tab set. (^**VAL**)

Arguments

tab_id

The ID of the tab set returned by the DTS_CREATE subfunction. (**n**)

Discussion

The DTS_NUMTABS subfunction of %TS_TABSET obtains the number of tabs associated with a tab set.

DTS_TABINDEX

index = %TS_TABSET (DTS_TABINDEX, *tab_id*, *type*, *type_id*)

or

xcall TS_TABSET (DTS_TABINDEX, *tab_id*, *type*, *type_id*)

Return value

index

The index of the tab, or 0 if the window or list is not contained within the specified tab set. (^VAL)

Arguments

tab_id

The ID of the tab set returned by the DTS_CREATE subfunction. (n)

type

One of the following: (n)

DTS_WINDOW A window.

DTS_LIST A list.

type_id

The ID of the window or list. (n)

Discussion

The DTS_TABINDEX subfunction of %TS_TABSET obtains the tab index that contains the specified window or list.

If *tab_id* is not a valid tab set ID, a fatal error occurs.

Because the index of a tab can change when other tabs are added and removed, this subfunction should be called immediately before the returned information is required.

Tab Set Routines

%TS_TABSET

DTS_TABINFO

status = %TS_TABSET (DTS_TABINFO, *tab_id*, *tab_index*, *type*, *type_id*[, *disabled*])

or

xcall TS_TABSET (DTS_TABINFO, *tab_id*, *tab_index*, *type*, *type_id*[, *disabled*])

Return value

status

True if successful; false if not. (^VAL)

Arguments

tab_id

The ID of the tab set returned by the DTS_CREATE subfunction. (n)

tab_index

The index of the tab. (n)

type

One of the following: (n)

DTS_WINDOW A window.

DTS_LIST A list.

type_id

The ID of the window or list. (n)

disabled

(optional) Returned true if the tab is disabled, false if enabled. (n)

Discussion

The DTS_TABINFO subfunction of %TS_TABSET obtains the type and ID of a contained object on a specified tab.

If *tab_id* is not a valid tab set ID, a fatal error occurs.

If *tab_index* is not in the range from 1 to the number of tabs in the specified tab set, the subfunction returns false with *type* and *type_id* set to 0.

If *tab_id* and *tab_index* are valid values, the subfunction returns true with values set for *type* and *type_id*.

Because the index of a tab can change when other tabs are added and removed, this subfunction should be called immediately before the returned information is required.

You can XCALL this subfunction if the return value is not required.

DTS_TABSET

```
tab_id = %TS_TABSET (DTS_TABSET, type, id[, container])
```

or

```
xcall TS_TABSET (DTS_TABSET, type, id[, container])
```

Return value

tab_id

The ID of the tab set that contains the window or list specified by *id*, or 0 if the window or list is not contained in a tab set. (^VAL)

Arguments

type

One of the following: (n)

DTS_WINDOW A window.

DTS_LIST A list.

id

The ID of the window or list. If *id* is not the ID for a window or list that has been logged with Toolkit, U_ABORT generates a Toolkit error. (n)

container

(optional) The returned ID of the composite window that contains *id* and is contained by the returned tab set (*tab_id*). If *id* is not part of a composite window on a tab set, *container* is returned as zero. (n)

Discussion

The DTS_TABSET subfunction of %TS_TABSET returns the ID of a tab set that contains the specified window or list (*id*), or it returns zero if the window or list is not contained in a tab set. This ID is returned if the window or list is directly placed on a tab set or if the window or list is part of a composite window that's placed on a tab set. If there are nested tab sets, DTS_TABSET returns the ID of the innermost tab set that contains the window or list. If you want the ID of the outermost tab set, call this function iteratively using the returned ID.

If the window or list is part of a composite window on the *tab_id* tab set, you can also get the ID of the composite window by passing the *container* argument. If there are nested composite windows, the *container* argument returns the ID of the outermost composite window, which is the composite window that is directly placed on the tab set. (To get the ID of the innermost container window use the DC_CONTAINER subfunction for %C_CONTAINER. See [DC_CONTAINER on page 6-29](#).)

DTS_CTRCONTEXT

`xcall TS_TABSET (DTS_CTRCONTEXT, tab_id, reason)`

Arguments

tab_id

The ID of the tab set returned by the DTS_CREATE subfunction. (**n**)

reason

The reason the tabbed dialog was activated, which can be blank (indicating that it was already activated), a program-supplied reason passed because it was activated programmatically, or one of the following: (**a**)

C_FIRST	Focus moved in attempt to reach first child in a composite window.
C_LAST	Focus moved in attempt to reach last child in a composite window.
C_NEXT	Focus moved from previous child in a composite window (e.g., by pressing TAB).
C_PREV	Focus moved from next child in a composite window (e.g., by pressing SHIFT+TAB).
C_SET	Child explicitly activated. For example, this could be the result of a mouse click, or it could be that the child was activated programmatically without a specified reason.

Discussion

DTS_CTRCONTEXT is designed for use in child processing methods for tab sets, where it enables users to tab through the buttons and input fields on the windows and lists in a tab set. It does this by calling routines for the various window types, which in turn use the reason code passed to the child method by C_PROCESS to determine what should have focus. (You could directly call the following, but DTS_CTRCONTEXT makes this easier by figuring out for you what type of window is on a tab.)

- ▶ For an input window on a tab, DTS_CTRCONTEXT calls I_CTRCONTEXT, passing the ID of the input window and *reason* on to that routine. See [I_CTRCONTEXT on page 8-9](#) for information.
- ▶ For a list on a tab, DTS_CTRCONTEXT calls L_CTRCONTEXT, passing the ID of the list and *reason* on to that routine. See [L_CTRCONTEXT on page 9-80](#) for information.
- ▶ For a composite container window on a tab, DTS_CTRCONTEXT calls the DC_CTRCONTEXT subfunction of %C_CONTAINER, passing the ID of the container and *reason* on to that routine. See [DC_CTRCONTEXT on page 6-33](#) for information.

- ▶ For a tab set on a tab (which we don't recommend), `DTS_CTRCONTEXT` recursively calls itself, passing the ID of the child tab set and *reason*.
- ▶ For any other type of window, `DTS_CTRCONTEXT` calls `U_CTRCONTEXT`, passing the ID of the window and *reason* on to that routine. See [U_CTRCONTEXT on page 16-24](#) for information.

To use in a child method, pass the reason passed to the child method (by `C_PROCESS`) as the *reason* argument for `DTS_CTRCONTEXT`. See the **ctrmeths.dbl** file included in your UI Toolkit distribution for an example. This file includes code for the `C_METHTS` method, which uses `DTS_CTRCONTEXT`.

Note that if `%TS_TABSET` is invoked as a function with `DTS_CTRCONTEXT`, the return value for the function will be true.

See also

- ▶ [C_PROCESS on page 6-42](#) for information on how it uses child processing methods.
- ▶ [“Controlling tabbing, focus, and input context” on page 6-4](#) for information on tabbing indices.

Examples

See the **ctrmeths.dbl** file included in your UI Toolkit distribution. It includes code for the `C_METHTS` method, which uses `DTS_CTRCONTEXT`.

14

Text Routines

Using Text Subroutines 14-2

Explains the capabilities of the text subroutines.

T_EDIT – Edit the text in a text window	14-3
T_GETLIN – Get lines of text from a text window	14-7
T_PUTTXT – Put text into a text window	14-8
T_SETUP – Specify a text window's characteristics	14-11
T_TXTRD – Read text from a window library	14-15
T_TXTWRT – Store text into a window library	14-17
T_VIEW – Control view through a text window	14-19
T_WNDRD – Load a text window from a window library.....	14-22
T_WNDWRT – Store a text window's contents in a window library.....	14-23

Using Text Subroutines

The text subroutines enable you to view (display) or edit text in a window. These subroutines optionally support margins and word wrap. Also, the window view can be scrolled if it is smaller than the entire window.

The view function supports both horizontal and vertical movement within the window, as well as character, word, and line movement.

The edit function gives you editing capabilities such as character, word, and line deletion, inserting a line, joining lines, and toggling insert and overstrike mode.

If desired, the text subroutines can store text entries in a compressed format. These can be stored to, or loaded from, either a text window or an array. Text windows also can be loaded from an array.

T_EDIT – Edit the text in a text window



```
xcall T_EDIT(window_id, [row], [column], [insert], [direction], [case], [col_id1], [col_id2],
&           [col_id3], [help_id][, decimal_value])
```

Arguments

window_id

The ID of the text window. **(n)**

row

(optional) The text row at which to begin editing; returned with a row value on exit. **(n)**

column

(optional) The text column at which to begin editing; returned with a column value on exit. **(n)**

insert

(optional) Indicates the default insert/overstrike mode. Returned with a mode value on exit. **(n)**

1 Insert. (default)

0 Overstrike.

direction

(optional) Indicates the editing direction. Returned with a direction value on exit. **(n)**

1 Forward. (default)

0 Reverse.

case

(optional) Indicates whether lowercase is allowed. Returned with a case value on exit: **(n)**

0 Lowercase allowed. (default)

1 Uppercase only allowed.

col_id1 through *col_id3*

(optional) The IDs of menu columns to place. **(n)**

help_id

(optional) The help identifier. **(a)**

decimal_value

(optional) The first character to input (specified as a decimal value). Ignored on Windows. **(n)**

Discussion

T_EDIT performs text editing in a window that T_SETUP has converted to a text window. You must call T_SETUP at least once for a window prior to using T_EDIT to edit it. (For information on an alternative text-editing routine that preserves hard returns, see [U_EDIT on page 16-34](#).)

The T_EDIT subroutine places any specified menu columns and sets the entry values for the row and column at which to begin editing, insert/overstrike mode, editing direction, and case. All of these values are stored and returned upon exit so the user can continue editing where they left off. Note the following:

- ▶ *Row* and *column* are the text line and position to edit, not the window line and position. For example, if *row* and *column* each have a value of 1, the edit position will be the first character of the first text line, regardless of any margins. The default *row* and *column* values are both 1.
- ▶ If you define a display area, it must be the same width as the window.

WIN

The following internal menu entry names are recognized by T_EDIT on Windows: E_CLEAR, E_COPY, E_CUT, E_MARK, E_MODE, E_PASTE.

Note the following:

- ▶ The *col_id* arguments are, by default, ignored when running on Windows. These columns usually contain reserved menu entries that are automatically supported by the Windows environment. Additionally, the excessive placing and removing of columns causes a Windows menu bar to “flicker.” Instead, we recommend that menu columns be placed before the call to T_EDIT and removed afterwards.
- ▶ If the height of the display area is less than the height of the window, a scroll bar is displayed. To prevent the scroll bar from occluding text entered in the text window, make the window size large enough to accommodate the scroll bar and the text, and then in the T_SETUP call, specify a right margin that is large enough for the scroll bar.

For example:

```
; Get the width of a character.
charwidth = %u_winmetrics(D_CHARWIDTH, WID_APP)
; Use character width to get width (in chars) of scrollbar.
extra_width = (%u_winmetrics(D_VSCROLL_X) + charwidth - 1) / charwidth
; Create window with extra width for scrollbar.
xcall w_proc(WP_CREATE, wndid, "text", 20,
&          textbox_cols + extra_width, WP_PLACE, wndid, 1, 1)
xcall t_setup(wndid, D_INIT, 0, extra_width, 0, 0, ,
&          1, 1, textbox_rows, textbox_cols + extra_width)
```

- ▶ We recommend that a fixed font be used for text in a text window. Using a proportional font may cause the text to wrap unpredictably.

- ▶ T_EDIT's handling of line ends in incoming data is controlled by the **g_txt_softreturn** field, which is defined in **tools.def**. See [g_txt_softreturn on page 18-7](#) for information.
- ▶ For text windows that are part of a composite window, TAB and SHIFT+TAB move focus to the next or previous child (unless TAB or SHIFT+TAB is a shortcut on a placed menu column). TAB instructs T_EDIT to signal "C_NEXT", and SHIFT+TAB instructs T_EDIT to signal "C_PREV".

UNIX, VMS

If any *col_ids* are passed, the specified columns will be placed on the menu and removed on exit. (The alternative to specifying menu columns in a T_EDIT call is to place them before calling T_EDIT.)

The following internal menu entry names are recognized by T_EDIT on UNIX and OpenVMS:

E_TOP	E_LMOV	E_DIR	E_XUP
E_BOTTOM	E_PDEL	E_MODE	E_XDOWN
E_UP	E_LDEL	E_WMOV	
E_DOWN	E_WDEL	E_EMOV	
E_LEFT	E_EDEL	E_BMOV	
E_RIGHT	E_CDEL	E_JOIN	
E_PMOV	E_LCLR	E_ILIN	

If passed, *decimal_value* is converted to its alpha (ASCII) equivalent and placed into the window as if it were the first character entered by the user. T_EDIT then clears the variable passed as *decimal_value*, which makes it easier to use T_EDIT in a loop.

In any environment, if the internal name of the selected menu entry begins with "U_," the EUTILS_METHOD subroutine is called and the input is redone. If the internal name of the selected menu entry is "O_HELP," the EHELP_METHOD subroutine is called, with *help_id* as its argument. If some other menu entry is selected, control is returned to the calling routine. **G_select** will be true and **g_entnam** will hold the name of the entry. You will need to clear **g_select** after you satisfy the event.

The default behavior for each environment's handling of columns is controlled by the **g_plc_col_args** flag, which is defined in the distributed **tkctl.def** file. See [g_plc_col_args on page 18-4](#) for information.

See also

- ▶ [EHELP_METHOD on page 7-73](#) for more information about help processing.
- ▶ [EUTILS_METHOD on page 7-82](#) for information about utility processing.
- ▶ ["Appendix B: Reserved Menu Entries"](#) for the functions performed by the UI Toolkit reserved menu entries.

Examples

This code fragment shows the three steps involved in editing a text window. The window in **wndid** is loaded with U_LDWND and converted to a text window with T_SETUP. Next, a menu column listing edit commands is loaded but not placed on the menu bar. The edit column is placed by T_EDIT. Editing begins at the first character of the first text line, in insert mode, in the forward direction by default. "H_KEYPAD" is passed to the EHELP_METHOD subroutine if the "O_HELP" entry is selected from the menu.

```
xcall u_ldwnd(wndid, g_utlib, "textwnd")
xcall t_setup(wndid)
xcall m_ldcol(ecolid, g_utlib, "edit_ctl", D_NOPLC)
do
  begin
    xcall t_edit(wndid,,,,,, ecolid,,, "H_KEYPAD")
    if(g_select)
      case(g_entnam) of
        begincase
"ENTRY1": call ENTRYONE ; one of these subroutines will
"ENTRY2": call ENTRYTWO ; set the done flag
        endcase
      end
  until(done)
```

T_GETLIN – Get lines of text from a text window



```
xcall T_GETLIN (window_id, obtained_lines, array_size, first_line, num_obtained)
```

Arguments

window_id

The ID of the text window to access. **(n)**

obtained_lines

Returned with the line(s) obtained. **(a)**

array_size

The number of lines to get, or the size of the array. **(n)**

first_line

The text line at which to start retrieval. Pass **1** for the first line, **2** for the second line, etc. **(n)**

num_obtained

Returned with the number of lines that were obtained. **(n)**

Discussion

The T_GETLIN subroutine retrieves one or more lines of loaded text from a text window. If the desired text is divided into 40 lines in the text window, T_GETLIN will retrieve up to 40 lines of text.

This subroutine assumes that the length of *obtained_lines* is large enough to hold the returned lines. *Obtained_lines* is either an alpha field or the first element of an alpha array.

If *array_size* is greater than 1, *obtained_lines* is considered to be the first element of an array that has at least as many elements as *array_size*. (If *array_size* is 1, the array is a single alpha field.)

First_line is the number of the text line within the window at which to start retrieval. It depends on the margins whether the number of this line corresponds with the row number. If *first_line* specifies a starting line that is less than one or greater than the number of text lines in the window, *num_obtained* will be returned as zero. A value of zero can also indicate that there are no more lines of text or that the first line is invalid. However, if the window is entirely empty and *first_line* is 1, *num_obtained* is returned as 1, and the first element of *obtained_lines* is returned blank.

Examples

This example starts at line one and tries to get 10 lines. It puts the lines in **newtext** and the actual number of lines in **numlines**.

```
xcall t_getlin(wndid, newtext, 10, 1, numlines)
```

T_PUTTXT – Put text into a text window



```
xcall T_PUTTXT (window_id, [text], [array_size][, option, ...])
```

Arguments

window_id

The ID of the text window to access. **(n)**

text

(optional) The text to load. **(a)**

array_size

(optional) The number of lines to load. **(n)**

option

(optional) One or more of the following options: **(n)**

D_NEWLN	Start new line before first line output.
D_PARA	Start new paragraph before first line output.
D_KEEPLEADING	Keep interline spacing for a window.
D_NOKEEPLEADING	Remove interline spacing for a window.
D_STRIP	Set trailing blank strip mode.
D_NOSTRIP	Turn off strip mode.
D_WRAP	Set wrap mode.
D_NOWRAP	Turn off wrap mode.
D_WRITES	Set WRITES mode.
D_NOWRITES	Turn off WRITES mode.
D_PERM	Make specified modes permanent.

Discussion

When T_PUTTXT loads text, it structures the text according to the structure of the text window. It automatically breaks lines when necessary and wraps the text if wrap is set. Thus, loading can be independent of the form of the text being loaded. For example, if you load 2,000 characters of straight text (with no line breaks), T_PUTTXT will break the text into lines and you might end up

with 40 lines of text (depending on the margins of your window). To change the text's appearance, you simply change the configuration of the window (such as margins and renditions) rather than re-creating the text itself.

You cannot use T_PUTTXT to load text into a window that has not been set up with T_SETUP. In other words, the window specified by *window_id* must be a text window, as opposed to an input window or a window created with the Synergy DBL W_PROC subroutine.

Text is either an alpha variable or the first element of an alpha array. If *array_size* is present and non-zero, *text* is considered to be the first element of an array whose size is equal to or greater than *array_size*. If *array_size* is omitted or has a value of zero, *text* is considered to be a single alpha field. There is no limit on the size of *text*. Note that *text* is optional; you can use T_PUTTXT to set permanent options for future text "puts" without loading any text at this time.

As with the T_SETUP subroutine, every *option* is processed before the text is output. Separate multiple *options* with commas. You can specify up to 10 *options*. If a particular mode is not specified, the current default (as set by T_SETUP or a previous T_PUTTXT) will be used.

If D_PERM is not present, the modes will only be in effect for the duration of this subroutine. If D_PERM is present, any modes specified in this subroutine will become the new defaults. The T_SETUP subroutine can also modify the defaults.

If D_NEWLNL or D_PARA are not present, the text will be output beginning at the current window position.

When entering T_EDIT or T_VIEW on Windows, D_KEEPLADING prevents Toolkit from removing the interline spacing for a window. With T_EDIT, this may cause more rows of text to become visible than the display area of the window would dictate. In addition, D_NOKEEPLADING restores default behavior: Toolkit removes interline spacing for a window, which makes the window vertically shorter. Note that D_KEEPLADING and D_NOKEEPLADING are ignored on Windows if D_PERM is not passed. These options are always ignored on UNIX and OpenVMS.

When strip mode is in effect, the trailing blanks are stripped from each element of text before the items are loaded into a text window. If wrap mode is also in effect, a single trailing blank is implicitly added to each stripped line.

In wrap mode, each line is considered to be one or more items that are separated by one or more blanks. Text is output until an item crosses the right margin. At this point, the item that would have crossed the margin is placed on the next line, and loading continues.

In WRITES mode, each line of output is displayed as if the Synergy DBL WRITES statement had been used; at the end of each line, the current position within the window automatically moves to the left margin of the next line in the text window.

If specified modifiers conflict, the last one to be processed will be in effect.

Examples

The following example loads 10 lines from **newtext** to the window, one line of text per window line.

```
xcall t_puttxt(wndid, newtext, 10, D_WRITES)
```

The next example loads the 15 elements of **newtext** into the window, stripping trailing blanks and wrapping to the next line on overflow. (This means the text will wrap, as described above.)

```
xcall t_puttxt(wndid, newtext, 15, D_STRIP, D_WRAP)
```

T_SETUP – Specify a text window’s characteristics



```
xcall T_SETUP (window_id, [D_INIT], [left_margin], [right_margin], [top_margin],
&               [bottom_margin], [paragraph], [origin_row], [origin_col], [rows],
&               [columns][, option, ...])
```

Arguments

window_id

The ID of the window that will be used. **(n)**

D_INIT

(optional) The flag to initialization the window. **(n)**

left_margin

(optional) The size (in columns) of the left margin (i.e., the offset from the left edge of the window). **(n)**

right_margin

(optional) The size (in columns) of the right margin (i.e., the offset from the right edge of the window). **(n)**

top_margin

(optional) The size (in rows) of the top margin (i.e., the offset from the top edge of the window). **(n)**

bottom_margin

(optional) The size (in rows) of the bottom margin (i.e., the offset from the bottom edge of the window). **(n)**

paragraph

(optional) Paragraph options of the form *xy* (a two-digit value), where *x* is the number of blank lines between paragraphs, and *y* is the number of spaces to indent a paragraph. If not specified, the number of blank lines and indent spaces default to zero. **(n)**

origin_row

(optional) The row at which the display area will begin. **(n)**

origin_col

(optional) The column at which the display area will begin. **(n)**

rows

(optional) The number of rows in the text window (i.e., the height of the display area). **(n)**

columns

(optional) The number of columns in the text window (i.e., the width of the display area). **(n)**

option

(optional) One or more of the following default load state flags: **(n)**

D_ATTR	Attributes. (See the table below.)
D_COLOR	Color palette number.
D_KEEPLEADING	Keep interline spacing for a window.
D_NOKEEPLEADING	Remove interline spacing for a window. (default)
D_STRIP	Set trailing blank strip mode. (default)
D_NOSTRIP	Turn off strip mode.
D_WRAP	Set wrap mode. (default)
D_NOWRAP	Turn off wrap mode.
D_WRITES	Set WRITES mode.
D_NOWRITES	Turn off WRITES mode. (default)

Discussion

The T_SETUP subroutine configures a low-level Synergy window (created with W_PROC) as a text window and sets the window's characteristics. You also can use T_SETUP to reconfigure an existing general or input window as a text window. For example, you can use T_SETUP to change an input window into a text window (but you will lose all input fields).

All arguments except *window_id* are optional and are useful only if you're going to be loading text into the window.

If D_INIT is specified, the window will be cleared and flagged as empty. If it is not specified, the window will not be cleared and is assumed to contain data. The number of text lines will be calculated by stripping all blank lines from the window.

Each margin argument (*left_margin*, *right_margin*, etc.) specifies how many columns or rows will be allotted to a margin, as measured from the window's edge. This determines how much text can be entered in the window. For example, if you have a 30 by 30 window, but you set the margins in T_SETUP to 1,1,1,10, there will be 28 rows and 19 columns for text (30 minus 2 for the left and right margins, and 30 minus 11 for the top and bottom margins). Any omitted arguments default to zero. If the window is not initialized and you specify a margin, the text processing system assumes that there is no data within that margin. (Text in a margin will be unavailable for editing.)

The display area arguments (*origin_row*, *origin_col*, *rows*, and *columns*) determine how much of the window you can see (not how much text you can enter in the window, which is determined by the size of the window minus any margins.) If none of the display arguments are present, the display area for the window is used.

You can specify up to eight *options*. If specified modifiers conflict, the last one to be processed will take effect.

T_SETUP clears all attributes if D_ATTR is not set. (ATTR_LOAD is assumed. See “[The Windowing Routines](#)” in the “Synergy Windowing API” chapter of the *Synergy DBL Language Reference Manual*.) If set, D_ATTR must be followed by one of the attribute flags listed in the following table. Each flag consists of a series of codes: B=blinking (italic on Windows), H=highlighted (bold), R=reverse, and U=underlined. Note that you can substitute I for B (for example, A_IR instead of A_BR). These are defined in **tools.def**.

Attribute Flag Definitions	
Attribute flag	Definition
A_NONE	No attributes.
A_U	Underlined.
A_R	Reverse.
A_RU	Reverse, underlined.
A_B	Blinking (on UNIX and OpenVMS) or italic (on Windows).
A_BU	Blinking, underlined.
A_BR	Blinking, reverse.
A_BRU	Blinking, reverse, underlined.
A_H	Highlighted.
A_HU	Highlighted, underlined.
A_HR	Highlighted, reverse.
A_HRU	Highlighted, reverse, underlined.
A_BH	Blinking, highlighted.
A_BHU	Blinking, highlighted, underlined.
A_BHR	Blinking, highlighted, reverse.
A_BHRU	Blinking, highlighted, reverse, underlined.
A_ALL	All attributes (same as A_BHRU).

D_COLOR must be followed by the color palette number (1-16). T_SETUP sets the color to 1 if D_COLOR is not set.

When entering T_EDIT or T_VIEW on Windows, D_KEEPLADING prevents Toolkit from removing the interline spacing for a window. With T_EDIT, this may cause more rows of text to become visible than the display area of the window would dictate. D_NOKEEPLADING restores default behavior: Toolkit removes interline spacing for a window, which makes the window vertically shorter. On UNIX and OpenVMS, D_KEEPLADING and D_NOKEEPLADING are ignored.

The D_STRIP, D_NOSTRIP, D_WRAP, D_NOWRAP, D_WRITES, and D_NOWRITES options define the defaults for T_PUTTXT. See [“T_PUTTXT” on page 14-8](#).

See also

- ▶ [.INPUT on page 2-76](#) or [.WINDOW on page 2-116](#) for information about creating windows.
- ▶ [W_PROC](#) in the “Synergy Windowing API” chapter of the *Synergy DBL Language Reference Manual* for information on creating a low-level Synergy window.

Examples

This example changes the low-level Synergy window “txtwnd” into a text window and initializes it. It will have the default text options.

```
xcall u_ldwnd(txtid, g_utlib, "txtwnd")
xcall t_setup(txtid, init)
```

The following example sets up the window specified by **txtid**. It will have a two-column left margin, there will be no paragraph indentation or spacing, the text will start two columns in from the upper left corner, and the display area will include 10 rows of 16 characters of text.

```
xcall t_setup(txtid,, 2, 0, 0, 0,, 1, 1, 10, 18)
```

T_TXTRD – Read text from a window library



```
xcall T_TXTRD (channel, entry_name, control, text, array_size, lines_read)
```

Arguments

channel

The channel on which the window library is opened. **(n)**

entry_name

The identifier (name) under which the text is stored in the library. **(a)**

control

One of the following processing controls on entry: **(n)**

D_FIRST Start with the first line.

D_NEXT Continue with line after last read.

text

The first element of the text array into which to read lines. **(a)**

array_size

The dimension of *text*. **(n)**

lines_read

Returned with the number of lines read. If zero, there are no more lines. **(n)**

Discussion

The T_TXTRD subroutine reads lines of text from a window library text entry created using T_TXTWRT.

The *channel* must be opened in update mode. This allows T_TXTRD to modify the control record in order to maintain read context within the text entry.

Control is returned with a value of D_NOFIND (defined in **tools.def**) if the specified entry does not exist. D_NOFIND is a negative number, so you can test for an error by checking whether *control* is less than zero.

Examples

This example reads the first 10 lines from the window “menuhelp” in the window library opened on **g_utlib** and stores them into the variable **helptxt**. If “menuhelp” is not found, **ctrl** is returned as **D_NOFIND**.

```
ctrl = D_FIRST  
xcall t_txtrd(g_utlib, "menuhelp", ctrl, helptxt, 10, rdsiz)
```

T_TXTWRT – Store text into a window library



```
xcall T_TXTWRT (channel, entry_name, control[, text, array_size])
```

Arguments

channel

The channel on which the window library is opened. **(n)**

entry_name

The identifier (name) of the text entry in the library (a maximum of 15 characters). Must be unique among window names stored in the library. **(a)**

control

One of the following processing controls on entry: **(n)**

D_CREATE Create new entry, no replacement.

D_REPLACE Create entry, replace any existing.

D_APPEND Append to existing entry.

text

(optional) The first element of the text array to be written. **(a)**

array_size

(optional) The number of entries in the *text* array. **(n)**

Discussion

T_TXTWRT creates and/or writes to text entries in a window library. These entries are stored in compressed form: trailing blanks are stripped and multiple blanks are compressed.

On exit, *control* is set to D_APPEND if the operation is successful or D_EXISTS if you request that an entry that already exists be created (with D_CREATE). *Control* is set to D_NOFIND (defined in the **tools.def** file) if you request an append operation to an entry that does not exist. D_EXISTS and D_NOFIND are negative numbers, and you can test for an error condition by checking whether *control* is less than 0.

If D_CREATE or D_REPLACE is successful and no *text* was passed, the entry within the library has been created and is empty. (In other words, if you create or replace an entry without specifying any text, you create an empty entry.)

If you write text to a library using T_TXTWRT, that text can only be read with T_TXTRD. You can add text to an entry at any time.

Examples

The following example takes 10 elements from **helptxt**, compresses them, and creates a text entry called “menuhlp” in the window library opened on **g_utlib**.

```
ctrl=D_CREATE  
xcall t_txtwrt(g_utlib, "menuhlp", ctrl, helptxt, 10)
```

T_VIEW – Control view through a text window



```
xcall T_VIEW(window_id, [col_id], [help_id][, unmapped_key])
```

Arguments

window_id

The ID of the text window to view. **(n)**

col_id

(optional) The ID of the view-control menu column. **(n)**

help_id

(optional) The help identifier. **(a)**

unmapped_key

(optional) Returned with the decimal value of any pressed key not mapped to a function code. **(n)**

Discussion

The T_VIEW subroutine enables the user to move the view of a text window using a view-control menu column. T_VIEW performs the following actions:

- ▶ Places the view-control column on the menu (if *col_id* is present and if running).
- ▶ Does view processing (in all environments).
- ▶ Turns off the cursor and echo in the current environment level (in all environments).

WIN

Unlike T_EDIT, T_VIEW does not display a scroll bar by default. However, you can enable the scroll bars by calling the W_BRDR windowing subroutine. For more information, see the “[Synergy Windowing API](#)” chapter of the *Synergy DBL Language Reference Manual*. The text window can also be navigated using arrow keys by setting the **g_plc_col_args** flag to true. (See [g_plc_col_args](#) on page 18-4 for information.)

The *col_ids* are, by default, ignored when running on Windows. These columns usually contain reserved menu entries that are automatically supported by the Windows environment. Additionally, the excessive placing and removing of columns causes a Windows menu bar to “flicker.”

If there’s more than one line in the text window, there will be unusable space at the bottom of the window. The amount of unusable space depends on the number of lines of text and is caused by leading that’s added as a result of the mapping from the native Windows edit control (used under the hood for text windows) to the rows of the Toolkit window used to display the text. This added

leading causes the text window to be larger than it needs to be to display the rows, resulting in the unusable space. If the text window is part of a tab set, however, you may be able to work around this by making the text window larger than the tab set. This allows the extra rows to be used by T_VIEW because the tab set container stretches the window's display area to encompass that size.

UNIX, VMS

On UNIX and OpenVMS, T_VIEW assumes that the column identified by *col_id* will contain one or more of the menu entries for scrolling listed below:

T_LINUP	T_RSIDE	T_TOP
T_LINDWN	T_LSIDE	T_BOTTOM
T_SCRUP	T_PAGUP	T_SCRLFT
T_SCRDWN	T_PAGDWN	T_SCRRT

The default behavior for each environment's handling of columns is controlled by the **g_plc_col_args** flag, which is defined in **tkctl.def**. See [g_plc_col_args on page 18-4](#) for details.

In all environments, if the internal name of a selected menu entry begins with "U_," the EUTILS_METHOD subroutine is called and the input is redone. If the internal name of the selected menu entry is "O_HELP," the EHELP_METHOD subroutine is called, with *help_id* as its argument.

WIN

T_VIEW recognizes the following menu entries on Windows. E_CLEAR, E_COPY, E_CUT, E_MARK, E_PASTE. These perform the associated Windows functionality, if applicable. See ["Appendix B: Reserved Menu Entries"](#) for information on reserved menu entries.

If a selection is made that is neither a T_VIEW entry, a utility entry, a help entry, nor one of the E_ menu entries listed above for Windows, control returns to the calling subroutine with **g_select** set to true and **g_entnam** containing the name of the selected entry in uppercase. You must clear **g_select** after you satisfy the event.

If *unmapped_key* is passed, when the user presses a key that is not mapped to a function code, the decimal value of the corresponding ASCII character will be returned in this argument. If passed and the user presses ENTER, *unmapped_key* returns zero. Passing *unmapped_key* may be useful if you want to test for a certain ASCII character and provide alternative processing if the test is true. For example, you could allow the user to press ENTER to exit the view function, even though you can't define the ENTER key as a regular shortcut. If you omit *unmapped_key*, any unmapped key the user presses will ring the terminal bell, but T_VIEW will continue viewing.

Creating the view-control column

Use the menu column script file commands `.COLUMN` and `.ENTRY` to create the view-control column and its entries. The following command creates a menu column named “text_view” whose text in the menu bar is “Scrolling.”

```
.column text_view, Scrolling
```

To enable scrolling on UNIX or OpenVMS, the view-control column must contain some subset of the menu entries listed above. Your script file commands will look something like this:

```
.entry t_linup, "Up 1 line", key(up)
.entry t_lindwn, "Down 1 line", key(down)
```

See also

- ▶ [EHELP_METHOD on page 7-73](#) for information about help processing.
- ▶ [EUTILS_METHOD on page 7-82](#) for information about utility processing.
- ▶ [“Appendix B: Reserved Menu Entries”](#) for the functions performed by Toolkit reserved menu entries.

Examples

This code fragment shows the three steps required to view a text window. The window is loaded with `U_LDWND` and converted to a text window with `T_SETUP`. Next, a menu column that lists view commands is loaded but not placed on the menu bar. The view column is placed by `T_VIEW` in this example. `T_VIEW` only returns when a non-view command is selected.

```
xcall u_ldwnd(wndid, g_utlib, "intro_text")
xcall t_setup(wndid)
xcall m_ldcol(vcolid, g_utlib, "view_ctl", D_NOPLC)
do
    begin
        xcall t_view(wndid, vcolid)
        if(g_select)
            case(g_entnam) of
                begincase
"ENTRY1": call ENTRYONE      one of these subroutines will
"ENTRY2": call ENTRYTWO     set the done flag
"ENTRY3": call ENTRYTHREE
                endcase
            end
    until(done)
```

T_WNDRD – Load a text window from a window library



```
xcall T_WNDRD (channel, entry_name, window_id, status[, lines_loaded])
```

Arguments

channel

The channel of the window library. **(n)**

entry_name

The identifier (name) under which the text is stored in the library. **(a)**

window_id

The ID of the text window to process. **(n)**

status

The returned status flag. **(n)**

lines_loaded

(optional) Returned with the number of lines loaded. **(n)**

Discussion

The T_WNDRD subroutine reads the data for a text window from a window library and puts it into the window. This subroutine obtains data using one or more calls to T_TXTRD and stores that data into the text window by calling T_PUTTXT.

Status is returned with a value of zero if the operation is successful, or D_NOFIND if the text entry cannot be found in the library. D_NOFIND is a negative number, therefore you can test for an error situation by checking whether *status* is less than zero.

Examples

The following example finds the text entry “menuhelp” in the window library opened on **libchn** and puts the text out to the window identified by **txtid**.

```
xcall t_wndrd(libchn, "menuhelp", txtid, status, numlines)
```

T_WNDWRT – Store a text window’s contents in a window library



```
xcall T_WNDWRT (channel, entry_name, window_id, control[, lines_written])
```

Arguments

channel

The channel of the window library. (n)

entry_name

The identifier (name) of the text entry in the library (a maximum of 15 characters). (a)

window_id

The ID of the text window to process. (n)

control

One of the following processing controls on entry: (n)

D_CREATE Create a new entry; generate an error if it already exists.

D_REPLACE Create a new entry and replace any existing entry.

lines_written

(optional) The number of lines written. (n)

Discussion

The T_WNDWRT subroutine stores the data within a text window as a text entry in a window library. This subroutine obtains data using one or more calls to T_GETLIN and writes that data out to the library by calling T_TXTWRT.

Upon exit, *control* is returned with a value of zero if the operation is completed successfully, or D_EXISTS if *control* was set to D_CREATE on entry and the text entry already exists. D_EXISTS is a negative number, therefore you can test for an error condition by checking whether *control* is less than zero.

Examples

This example takes the text from **txtid**, makes it into a text entry, and stores it as “menuhlp” in the window library opened on **libchn**.

```
ctrl = D_CREATE
xcall t_wndwrt(libchn, "menuhlp", txtid, ctrl, numlines)
```


15

Toolbar Routines

Understanding Toolbar Processing 15-2

Discusses what toolbar processing is, how it works, and why you would want to use a toolbar.

%TB_BUTTON – Load and manipulate toolbar buttons.....	15-3
%TB_INFO – Return toolbar information	15-15
%TB_TKCREATE – Create and log a toolbar with the environment	15-32
TB_TKDEL – Delete a toolbar that’s logged with the environment.....	15-34
TB_TKGLOBAL – Promote a toolbar to global	15-35
TB_TKLOG – Log a toolbar with the environment.....	15-36
%TB_TOOLBAR – Create and manipulate a toolbar	15-37

Understanding Toolbar Processing

The toolbar functions enable you to create and manipulate an application window toolbar. The toolbar is a variable-size rectangle containing buttons that the user can select with a mouse.

Toolbars are used as extensions to—or replacements for—the menu bar, providing an alternative method for invoking functions within an application. Toolbars are supported only on Windows.

Horizontal toolbars will occlude the entire width of the top or bottom of the application window. Similarly, vertical toolbars will blank out the entire vertical area of the application window on the specified side. Toolbars are sized to the full extent of the application window with respect to the border where they are defined.

A toolbar button can display either text or a graphic. See [“Toolkit button graphics” on page 2-15](#) for more information. You can apply 3-D borders to toolbar buttons by setting the `SYN_3D_TOOLBAR` environment variable to 1. For information, see [SYN_3D_TOOLBAR](#) in the “Environment Variables” chapter of *Environment Variables & System Options*.

Selecting a toolbar button can trigger the simulation of one or more characters pressed on the keyboard, or it can invoke an entire subroutine. A common approach is to have a function key invoke a particular action such as Find, Select, Page Up, and so forth. It’s also common to use `%M_SIGNAL` as the button method to signal the name of the button as a menu entry. Toolbar button actions are specified at runtime when the button is loaded onto the toolbar.

A special type of button, the numeric bar, contains the numbers 0 through 9. To load the numeric bar, pass `“#NUMBER”`, a special reserved name, to `%TB_BUTTON`.

Toolbars and toolbar buttons can be managed by Toolkit’s environment control. You can do this by creating the toolbar with `%TB_TKCREATE` or by using `%TB_TKLOG` to log the toolbar with Toolkit if you created it with `%TB_TOOLBAR`.

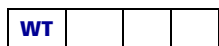
When you log a toolbar with an environment, the following are controlled by the environment:

- ▶ Deletion of the toolbar when its environment is exited
- ▶ Placement of the toolbar
- ▶ The state of toolbar buttons, including enablement and forced-down state. Note that toolbar buttons created in an environment will be deleted when the environment is exited, but no button deleted in an environment will be restored when that environment is exited.

When using toolbars, you must use Toolkit environment processing consistently or not at all. Once a toolbar has been logged with Toolkit, you must leave the management of the toolbar to Toolkit, and you must use `TB_TKDEL` to explicitly delete the toolbar.

You may use `TB_PUSHSTATE` and `TB_POPSTATE` when you create a toolbar with `%TB_TOOLBAR`, but we recommend that you don’t use them for toolbars that are under environment control; this may confuse Toolkit.

%TB_BUTTON – Load and manipulate toolbar buttons



```
return = %TB_BUTTON (id, subfunction, arg[, ...])
```

or

```
xcall TB_BUTTON (id, subfunction, arg[, ...])
```

Return value

return

On Windows, a value specific to the subfunction. On UNIX and OpenVMS, always returns false. (^VAL)

Arguments

id

The toolbar identifier, the value of which is returned from the TB_CREATE subfunction of %TB_TOOLBAR. (n)

subfunction

The toolbar function to perform: (n)

TBB_LOAD	Add a button to the toolbar.
TBB_GROUP	Define a button group on the toolbar.
TBB_SELECT	Force a button to appear in either an up or down state; or simulate a mouse click.
TBB_DELETE	Delete (and remove) a toolbar button.
TBB_ENABLE	Enable a disabled toolbar button.
TBB_DISABLE	Disable a toolbar button.
TBB_STATE	Return the up or down state of a toolbar button.

arg

One or more arguments, depending on *subfunction*.

Discussion

The %TB_BUTTON function enables you to load and manipulate toolbar buttons. See the individual subfunctions for details.

You must include the file **toolbar.def** to use this function.

Examples

The following example uses %TB_BUTTON to add buttons to a toolbar. If the user chooses to change the image for a button, a call to %TB_BUTTON deletes the button and adds a new one with the selected image.

```

;Create a toolbar at the top of the application, with unspecified
; width and a height of 70 pixels
tb_toolbar(TB_CREATE, tbid, TBC_TOP, 0, 70)

;Add a 60x60 pixel button named "O_EXIT" at 5,5 using the image
exit.png
;On left click, call m_signal with the button name
;On right click, call mod_button with the button name
tb_button(tbid, TBB_LOAD, "O_EXIT", 5, 5, 60, 60, TBLOAD_TYPE_BMAP,
&      "exit.png", "Exit the application",,, "m_signal",,,,,
&      "mod_button")

;Now do the same for "O_HELP", but placed at 75,5 and using the
; image help.png
tb_button(tbid, TBB_LOAD, "O_HELP", 75, 5, 60, 60, TBLOAD_TYPE_BMAP,
&      "help.png", "Get Help",,, "m_signal",,,,, "mod_button")

;Place the toolbar
tb_toolbar(TB_PLACE, tbid)

repeat
begin
    m_process("")          ;Process the menu
                           ; (you could call any input routine here)
    if (g_select)
        using g_entnam select
            ("O_EXIT "),    exitloop    ;Handle O_EXIT by exiting
                                     ;We don't need to handle O_HELP,
                                     ; because M_PROCESS does
        endusing
    end
.
.
.
function mod_button, ^val, reentrant
;
; Routine to allow the user to select a new image for a button
;
in req a_name ,a      ;Name of the button

external common
    tbid      ,int    ;Toolbar ID

```



```

record                                ;Place to save off button information
    x                                ,int
    y                                ,int
    cx                               ,int
    cy                               ,int
    image                            ,a256
tooltip                             ,a256
proc
    x = %tb_info(tbid, TBI_LEFT, a_name)
    y = %tb_info(tbid, TBI_TOP, a_name)
    cx = %tb_info(tbid, TBI_WIDTH, a_name)
    cy = %tb_info(tbid, TBI_HEIGHT, a_name)
    tb_info(tbid, TBI_TYPE, a_name, image)
    tb_info(tbid, TBI_TOOLTIP, a_name, tooltip)

    if (%u_getfilename(image, "Choose a new image",,,, image))
        begin
            ;User selected a new image, so first delete the old button
            tb_button(tbid, TBB_DELETE, a_name)

            ;Then add it back using the new image
            tb_button(tbid, TBB_LOAD, a_name, x, y, cx, cy, TBLOAD_TYPE_BMAP,
                & image, %atrim(tooltip),,, "m_signal",,,,, "mod_button")
        end

        m_signal("O_UPDT")            ;Force an update
        freturn TRUE
    end
end

```

Toolbar Routines

%TB_BUTTON

TBB_LOAD

```
status = %TB_BUTTON (id, TBB_LOAD, name, [left], [top], width, height, type_spec, type_arg,  
& [tooltip], [left_vk_code], [left_string], [left_method], [mid_vk_code], [mid_string],  
& [mid_method], [right_vk_code], [right_string] [, right_method])  
  
or  
  
xcall TB_BUTTON (id, TBB_LOAD, name, [left], [top], width, height, type_spec, type_arg,  
& [tooltip], [left_vk_code], [left_string], [left_method], [mid_vk_code], [mid_string],  
& [mid_method], [right_vk_code], [right_string] [, right_method])
```

Return value

status

The status of the operation. (^VAL)

TB_SUCCESS 1

TB_FAILURE 0

Arguments

id

The toolbar identifier, the value of which is returned from the TB_CREATE subfunction of %TB_TOOLBAR. (n)

name

The name of the button (maximum 30 characters). “#NUMBER” is a reserved name that will load the numeric bar. (a)

left

(optional) The pixel location of the left edge of the button relative to the top left corner of the toolbar. (n)

top

(optional) The pixel location of the top edge of the button relative to the top left corner of the toolbar. (n)

For horizontal toolbars, if *left* or *top* are not specified, the button will be placed automatically, following a left-to-right, top-to-bottom pattern.

For vertical toolbars, if *left* or *top* are not specified, the button will be placed automatically, following a top-to-bottom, left-to-right pattern.

width

The width of the button in pixels. (n)

height

The height of the button in pixels. **(n)**

type_spec

The button face type: **(n)**

TBLOAD_TYPE_TEXT Text.

TBLOAD_TYPE_BMAP Graphic.

type_arg

The button type data. If *type_spec* is **TBLOAD_TYPE_TEXT**, this is the text string to load on the face of the button. If *type_spec* is **TBLOAD_TYPE_BMAP**, this is the filename for the graphic to be placed on the face of the button. (See [“Toolkit button graphics” on page 2-15](#) for information.) **(a)**

tooltip

(optional) The text of a ToolTip to be displayed if the mouse rests over the button. **(a)**

left_vk_code

mid_vk_code

right_vk_code

(optional) “Virtual keys” generated when the left, middle, and right mouse buttons are used to select this toolbar button. Example virtual key definitions are: **VK_LBUTTON**, **VK_TAB**, and so forth, listed in the file **toolbar.def**. **(n)**

left_string

mid_string

right_string

(optional) Strings generated when the left, middle, and right mouse buttons are used to select this toolbar button. The strings generated will precede any virtual key generation that may be defined for this button. **(a)**

left_method

mid_method

right_method

(optional) The names of the subroutines invoked when the left, middle, and right mouse buttons are used to select this toolbar button. **(a)**

Discussion

The **TBB_LOAD** subfunction of **%TB_BUTTON** adds a button to the toolbar. If you don’t specify a position for the button and no other button have been added, the button will be placed at the leftmost position on the toolbar. If you don’t specify a position for the button but other buttons have been added, the button will be placed to the right of the last specified button—even if the last specified button has been deleted (which will create a gap before the new button on the toolbar.)

Toolbar Routines

%TB_BUTTON

When a mouse button is clicked while on a toolbar button, the specified method routine is called. This method routine is passed one argument, the name of the button. If the method returns a value of false, the action sequence terminates and no further actions are taken on that keypress. If the method returns a value of true, any string is posted, followed by any VK_xxx virtual key.

See also

[“Appendix D: Methods”](#).

TBB_GROUP

status = %TB_BUTTON (*id*, TBB_GROUP, *name1*[, *name2*, ...])

or

xcall TB_BUTTON (*id*, TBB_GROUP, *name1*[, *name2*, ...])

Return value

status

The status of the operation. (^VAL)

TB_SUCCESS 1

TB_FAILURE 0

Arguments

id

The toolbar identifier, the value of which is returned from the TB_CREATE subfunction of %TB_TOOLBAR. (n)

name1

The name of the button in the group that will initially receive the down state. (a)

name2, ...

(optional) The name of one or more buttons that define the rest of the group. (a)

Discussion

The TBB_GROUP subfunction of %TB_BUTTON is used to define a button group. A button group is a set of buttons on a toolbar. When a button is selected, it exhibits a down state. If that button is part of a group, all other buttons in the group exhibit an up state. At all times, one and only one button in a group will exhibit a down state. Regardless of state, all buttons are always selectable if enabled.

When specified, all button name arguments given in TBB_GROUP must be successfully loaded by TBB_LOAD prior to the TBB_GROUP call. An error will result if a TBB_GROUP button name argument is not a loaded button. A button can only be a member of one group. You cannot delete a group directly, but you can use TBB_DELETE to delete buttons from the group. When the last button in the group is deleted, the group is deleted. See the TBB_DELETE [Discussion on page 15-11](#) for more information on deleting buttons in a group.

TBB_SELECT

status = %TB_BUTTON (*id*, TBB_SELECT, *name*, *option*)

or

xcall TB_BUTTON (*id*, TBB_SELECT, *name*, *option*)

Return value

status

The status of the operation. (^VAL)

TB_SUCCESS 1

TB_FAILURE 0

Arguments

id

The toolbar identifier, the value of which is returned from the TB_CREATE subfunction of %TB_TOOLBAR. (**n**)

name

The name of the button. (**a**)

option

One of the following toolbar button-state options: (**n**)

TBBS_HOLD Forces the button to appear in the down state until TBBS_RELEASE is called for that button. The down state indicates that a selection is currently being processed.

TBBS_RELEASE Releases a TBBS_HOLD button from its forced down state. The up state indicates that the selection being processed is now completed.

TBBS_LEFT Simulates a left mouse button press on a toolbar button. All events defined for the button with TBB_LOAD are executed.

TBBS_MIDDLE Simulates a middle mouse button press. All events defined for the button with TBB_LOAD are executed.

TBBS_RIGHT Simulates a right mouse button press. All events defined for the button with TBB_LOAD are executed.

Discussion

The TBB_SELECT subfunction of %TB_BUTTON forces a button to appear in either an up or down state. It can also simulate a user mouse-press on a button. If a method, string, or virtual key is defined for the toolbar button, the actions are invoked in the sequence described under TBB_LOAD.

TBB_DELETE

return = %TB_BUTTON (*id*, TBB_DELETE, *name*)

or

xcall TB_BUTTON (*id*, TBB_DELETE, *name*)

Return value

return

The status of the requested operation. (^VAL)

TB_SUCCESS 1

TB_FAILURE 0

Arguments

id

The toolbar identifier, the value of which is returned from the TB_CREATE subfunction of %TB_TOOLBAR. (**n**)

name

The name of the button. (**a**)

Discussion

TBB_DELETE removes a toolbar button from the toolbar. You can delete a button that's part of a group. If the button is the selected member of a group, the first remaining member (in the order the members were specified when the group was created) becomes the selected member, and its left click event is signaled. If the button is the only remaining button in the group, the group is deleted.

Toolbar Routines

%TB_BUTTON

TBB_ENABLE

return = %TB_BUTTON(*id*, TBB_ENABLE, *name*)

or

xcall TB_BUTTON(*id*, TBB_ENABLE, *name*)

Return value

return

The status of the requested operation. (^VAL)

TB_SUCCESS 1

TB_FAILURE 0

Arguments

id

The toolbar identifier, the value of which is returned from the TB_CREATE subfunction of %TB_TOOLBAR. (**n**)

name

The name of the button. (**a**)

Discussion

TBB_ENABLE enables a disabled toolbar button.

TBB_DISABLE

return = %TB_BUTTON (*id*, TBB_DISABLE, *name*)

or

xcall TB_BUTTON (*id*, TBB_DISABLE, *name*)

Return value

return

The status of the requested operation. (^VAL)

TB_SUCCESS 1

TB_FAILURE 0

Arguments

id

The toolbar identifier, the value of which is returned from the TB_CREATE subfunction of %TB_TOOLBAR. (**n**)

name

The name of the button. (**a**)

Discussion

TBB_DISABLE disables a toolbar button. The toolbar button remains on the screen, but nothing happens if it is selected. When a button is disabled, it is grayed out.

Toolbar Routines

%TB_BUTTON

TBB_STATE

return = %TB_BUTTON (*id*, TBB_STATE, *name*)

or

xcall TB_BUTTON (*id*, TBB_STATE, *name*)

Return value

return

The state of the specified button. (^VAL)

Down (selected) 1

Up (not selected) 0

Arguments

id

The toolbar identifier, the value of which is returned from the TB_CREATE subfunction of %TB_TOOLBAR. (**n**)

name

The name of the button. (**a**)

Discussion

TBB_STATE returns the up or down state of a toolbar button. The state is returned as the value of the %TB_BUTTON function.

%TB_INFO – Return toolbar information

WT			

return = %TB_INFO (*tbid*, *subfunction*[, *arg*, ...])

or

xcall TB_INFO (*tbid*, *subfunction*[, *arg*, ...])

Return value

return

On Windows, a value specific to the subfunction. On UNIX and OpenVMS, always returns false. (^VAL)

Arguments

tbid

The ID of a toolbar whose information you want returned. (n)

subfunction

One of the following to return the specified information: (n)

TBI_BUTTONS	Number of buttons on the toolbar and optionally the button names.
TBI_CLICK	Click actions for a toolbar button.
TBI_ENABLED	Enabled state of a toolbar or a toolbar button.
TBI_GROUP	Number of buttons in a button group and optionally the button names.
TBI_HEIGHT	The height (in pixels) of a toolbar or a toolbar button.
TBI_HWND	The window handle for a toolbar or a toolbar button.
TBI_LEFT	The horizontal position of a toolbar button.
TBI_ORIENTATION	The side of the application window that the toolbar occupies.
TBI_PLACED	The placed state of a toolbar.
TBI_TOOLTIP	The ToolTip text for a toolbar button.
TBI_TOP	The vertical position of a toolbar button.
TBI_TYPE	The type of a toolbar button (text or bitmap) and return the caption or graphic path.
TBI_WIDTH	The width (in pixels) of a toolbar or a toolbar button.

arg

(optional) One or more arguments as required for *subfunction*.

Discussion

The %TB_INFO function returns information about toolbars and toolbar buttons depending on the value of *subfunction*.

The value passed in *tbid* must be the ID of a toolbar created in Toolkit, or Toolkit will generate a runtime error. (The Toolbar does not need to be placed.) And the value passed as *subfunction* must be one of the TBI_ values listed above, or Toolkit will generate a runtime error.

TBI_BUTTONS

return = %TB_INFO (*tbid*, TBI_BUTTONS, [*max*, *buttons*])

or

xcall %TB_INFO (*tbid*, TBI_BUTTONS, [*max*, *buttons*])

Return value

return

The number of buttons on the toolbar specified by *tbid*. (^VAL)

Arguments

tbid

The ID of a toolbar created in Toolkit. (n)

max

(optional) The maximum number of button names to return in the *buttons* array. (n)

buttons

(optional) The first element of an alphanumeric array of at least *max* elements. This is required if a value greater than zero is passed as *max*. (a)

Discussion

The TBI_BUTTONS subfunction returns the number of buttons on the specified toolbar (*tbid*). It returns zero if there are no buttons.

TBI_BUTTONS also returns the names of buttons on the specified toolbar if you pass *max* and *buttons*, and if *max* is greater than zero. It returns these names in all uppercase characters as elements in the *buttons* array. (The number of returned button names is the lesser of *return* and *max*.) Note that if you pass a value that's greater than zero as *max*, you must also pass *buttons*. If you don't pass *max*, the *buttons* argument is ignored (so button names are not returned).

Examples

The following example returns the number of buttons on the **tb_id** toolbar.

```
btn_num = %tb_info(tb_id, TBI_BUTTONS)
```

The next example returns the number of buttons on the **tb_id** toolbar, and it returns the names of the buttons. The first button name is stored in the first element of the array **btn_array**. Subsequent names are returned in subsequent array elements.

```
btn_num = %tb_info(tb_id, TBI_BUTTONS, btn_max, btn_array)
```

TBI_CLICK

```
return = %TB_INFO (tbid, TBI_CLICK, button_name, mouse_button, [vk_code], [string]  
&          [, method])
```

or

```
xcall %TB_INFO (tbid, TBI_CLICK, button_name, mouse_button, [vk_code], [string]  
&          [, method])
```

Return value

return

One or more of the following bit values (indicating which actions will be taken on a mouse click to the specified toolbar button), or 0 if no action is taken. (**^VAL**)

TBIC_VK A keystroke is sent to the application.

TBIC_STRING A string is sent to the application.

TBIC_METHOD A method is invoked.

Arguments

tbid

The ID of a toolbar created in Toolkit. (**n**)

button_name

The name (case-insensitive) of a button on the toolbar specified by *tbid*. (**a**)

mouse_button

One of the following: (**n**)

TBIC_LEFT Left click.

TBIC_MIDDLE Middle click.

TBIC_RIGHT Right click.

vk_code

(optional) The virtual key code for the specified *mouse_button*, or zero if there is no virtual key code. (**n**)

string

(optional) The string generated by the specified *mouse_button*, or all spaces if no string is generated. (**a**)

method

(optional) The method for the specified *mouse_button*, or all spaces if there is no method. Method names are returned in all uppercase characters. (**a**)

Discussion

The `TBI_CLICK` subfunction enables you to determine if any action will be taken on a mouse click to a toolbar button (*button_name*), and returns information about actions taken (i.e., any virtual key generated, string generated, or method called).



If a mouse button does not have a method, Toolkit will not generate a virtual key code or a string, even if these are assigned to the button. See the `TBB_LOAD` [Discussion on page 15-7](#).

If *button_name* isn't a valid button name for *tbid*, Toolkit will generate a runtime error.

Examples

The following example determines if there are any actions for the button specified by **btn_name**. Each of the following IF statements (which use the bitwise operator `&`) returns true if the specified bit is set in **btn_actions** or zero if it isn't.

```
btn_actions = %tb_info(tb_id, TBI_CLICK, btn_name, mouse)
if (btn_actions & TBIC_VK)           ;True if button has a virtual key
.
.
.
if (btn_actions & TBIC_STRING)       ;True if button has a string
.
.
.
if (btn_actions & TBIC_METHOD)       ;True if button has a method
.
.
.
```

The next example returns the virtual key code, string, and method for the specified button.

```
btn_actions = %tb_info(tb_id, TBI_CLICK, btn_name, mouse, btn_vk,
&                btn_string, btn_method)
```

TBI_ENABLED

```
return = %TB_INFO (tbid, TBI_ENABLED[, button_name])
```

or

```
xcall %TB_INFO (tbid, TBI_ENABLED[, button_name])
```

Return value

return

True if the button or toolbar is enabled; false if the button or toolbar is disabled. (^VAL)

Arguments

tbid

The ID of a toolbar created in Toolkit. (n)

button_name

(optional) The name (case-insensitive) of a button on the toolbar specified by *tbid*. (a)

Discussion

If *button_name* is passed, TBI_ENABLED returns the enabled state of the button. (If *button_name* is passed as a non-blank value that isn't a button name for *tbid*, Toolkit will generate a runtime error.)

If *button_name* is not passed or is blank, TBI_ENABLED returns the enabled state of the toolbar specified by *tbid*.

Examples

The following example returns the enabled state of the **tb_id** toolbar.

```
tb_status = %tb_info(tb_id, TBI_ENABLED)
```

The next example returns the enabled state of the toolbar button **btn_name**.

```
btn_status = %tb_info(tb_id, TBI_ENABLED, btn_name)
```


TBI_GROUP

return = %TB_INFO (*tbid*, TBI_GROUP, *button_name*, [*max*, *members*])

or

xcall %TB_INFO (*tbid*, TBI_GROUP, *button_name*, [*max*, *members*])

Return value

return

The number of buttons in the group that *button_name* belongs to, or 0 if *button_name* is not part of a button group. (^VAL)

Arguments

tbid

The ID of a toolbar created in Toolkit. (n)

button_name

The name (case-insensitive) of a button on the toolbar specified by *tbid*. (a)

max

(optional) The maximum number of button names to return in the *members* array. (n)

members

(optional) The first element of an alphanumeric array of at least *max* elements. This is required if *max* is passed and is greater than zero. (a)

Discussion

The TBI_GROUP subfunction returns the number of buttons in the group that *button_name* belongs to. It returns zero if *button_name* is not a member of a button group. If *button_name* is not a valid button name for *tbid*, Toolkit will generate a runtime error.

TBI_GROUP also returns the names of buttons in the group if you pass *max* and *members*, and if *max* is greater than zero. It returns these names as elements in the *members* array. (The number of returned button names is the lesser of *return* and *max*.) Note that if you pass a value that's greater than zero as *max*, you must also pass *members*. If you don't pass *max*, the *members* argument is ignored (so button names are not returned).

Examples

The following example returns the number of buttons in the group that the button **btn_name** belongs to. If **btn_name** doesn't belong to a button group, the following returns zero.

```
btn_number = %tb_info(tb_id, TBI_GROUP, btn_name)
```

The next example returns the number of buttons in the button group that **btn_name** belongs to, and it returns the names of the buttons in that group. The first button name is stored in the first element of the array **btn_array**. Subsequent names are returned in subsequent array elements.

```
btn_number = %tb_info(tb_id, TBI_GROUP, btn_name, btn_max, btn_array)
```

TBI_HEIGHT

```
return = %TB_INFO (tbid, TBI_HEIGHT[, button_name])
```

or

```
xcall %TB_INFO (tbid, TBI_HEIGHT[, button_name])
```

Return value

return

The height in pixels of a toolbar or a toolbar button. (**^VAL**)

Arguments

tbid

The ID of a toolbar created in Toolkit. (**n**)

button_name

(optional) The name (case-insensitive) of a button on the toolbar specified by *tbid*. (**a**)

Discussion

If *button_name* is passed and isn't blank, TBI_HEIGHT returns the height of the specified button in pixels. (If *button_name* is passed as a non-blank value that isn't a valid button name for *tbid*, Toolkit will generate a runtime error.)

If *button_name* is not passed or is blank, TBI_HEIGHT returns the height of the toolbar (*tbid*) in pixels. Note that this is the actual height of the toolbar's window (whether placed or not), which may differ from the height passed when the toolbar was created (i.e., the height passed to the TB_CREATE subfunction of %TB_TOOLBAR or %TB_TKCREATE).

Examples

The following example returns the height of the **tb_id** toolbar.

```
tb_height = %tb_info(tb_id, TBI_HEIGHT)
```

The next example returns the height of the toolbar button specified by **btn_name**, unless **btn_name** is blank (in which case it returns the toolbar height).

```
tb_height = %tb_info(tb_id, TBI_HEIGHT, btn_name)
```

TBI_HWND

```
return = %TB_INFO (tbid, TBI_HWND[, button_name])
```

or

```
xcall %TB_INFO (tbid, TBI_HWND[, button_name])
```

Return value

return

The handle to a toolbar or a toolbar button. (^VAL)

Arguments

tbid

The ID of a toolbar created in Toolkit. (n)

button_name

(optional) The name (case-insensitive) of a button on the toolbar specified by *tbid*. (a)

Discussion

The TBI_HWND subroutine returns the window handle for a toolbar or a toolbar button. This is useful if you want to subclass the window or query its characteristics.

If *button_name* is passed and isn't blank, TBI_HWND returns the window handle for the specified toolbar button. (If *button_name* is passed as a non-blank value that isn't a valid button name for *tbid*, Toolkit will generate a runtime error.)

If *button_name* is not passed or is blank, TBI_HWND returns the window handle for the toolbar (*tbid*).

Examples

The following example returns the handle of the **tb_id** toolbar.

```
tb_handle = %tb_info(tb_id, TBI_HWND)
```

The next example returns the handle of a button (**btn_name**) on the **tb_id** toolbar.

```
tb_handle = %tb_info(tb_id, TBI_HWND, btn_name)
```

TBI_LEFT

return = %TB_INFO (*tbid*, TBI_LEFT, *button_name*)

or

xcall %TB_INFO (*tbid*, TBI_LEFT, *button_name*)

Return value

return

The horizontal position in pixels of the left side of *button_name*. (^VAL)

Arguments

tbid

The ID of a toolbar created in Toolkit. (n)

button_name

The name (case-insensitive) of a button on the toolbar specified by *tbid*. (a)

Discussion

The TBI_LEFT subfunction returns the horizontal position in pixels of the left side of the button specified by *button_name*. This position is relative to the toolbar's client area.

If *button_name* isn't a valid button name for *tbid*, Toolkit will generate a runtime error.

Examples

The following example returns the horizontal position of the left side of button **btn_name**.

```
left_side = %tb_info(tb_id, TBI_LEFT, btn_name)
```

TBI_ORIENTATION

```
return = %TB_INFO (tbid, TBI_ORIENTATION)
```

or

```
xcall %TB_INFO (tbid, TBI_ORIENTATION)
```

Return value

return

The side of the application window that the toolbar occupies: (^VAL)

TCB_TOP Top.

TCB_BOTTOM Bottom.

TCB_LEFT Left side.

TCB_RIGHT Right side.

Arguments

tbid

The ID of a toolbar created in Toolkit. (n)

Discussion

The TBI_ORIENTATION subfunction returns the side of the application window that the specified toolbar (*tbid*) occupies (or will occupy if the toolbar is not placed).

Examples

The following example returns the position of the **tb_id** toolbar in the application window.

```
position = %tb_info(tb_id, TBI_ORIENTATION)
```

TBI_PLACED

```
return = %TB_INFO(tbid, TBI_PLACED)
```

or

```
xcall %TB_INFO(tbid, TBI_PLACED)
```

Return value

return

True if the toolbar is placed; otherwise false. (^VAL)

Arguments

tbid

The ID of a toolbar created in Toolkit. (n)

Discussion

The TBI_PLACED subfunction returns the placed state of the toolbar specified by *tbid*.

Examples

The following example determines if the **tb_id** toolbar is placed.

```
position = %tb_info(tb_id, TBI_PLACED)
```

TBI_TOOLTIP

return = %TB_INFO (*tbid*, TBI_TOOLTIP, *button_name*[, *text*])

or

xcall %TB_INFO (*tbid*, TBI_TOOLTIP, *button_name*[, *text*])

Return value

return

The number of characters in the ToolTip for the specified button, or 0 if there is no ToolTip for the button. (^VAL)

Arguments

tbid

The ID of a toolbar created in Toolkit. (n)

button_name

The name (case-insensitive) of a button on the toolbar specified by *tbid*. (a)

text

(optional) The text of the ToolTip, or all spaces if there is no ToolTip. (a)

Discussion

The TBI_TOOLTIP subfunction enables you to determine if a toolbar button has a ToolTip and returns the length of the ToolTip if there is one. It also returns the text for the ToolTip if you pass *text*. Note that *return* is the actual length of the ToolTip, which may be longer than *text*.

If *button_name* isn't a valid button name for *tbid*, Toolkit will generate a runtime error.

Examples

The following example returns the text for the ToolTip for **btn_name**.

```
%tb_info(tb_id, TBI_TOOLTIP, btn_name, tooltip)
```


TBI_TOP

return = %TB_INFO (*tbid*, TBI_TOP, *button_name*)

or

xcall %TB_INFO (*tbid*, TBI_TOP, *button_name*)

Return value

return

The vertical position in pixels of the top side of *button_name*. (^VAL)

Arguments

tbid

The ID of a toolbar created in Toolkit. (n)

button_name

The name (case-insensitive) of a button on the toolbar specified by *tbid*. (a)

Discussion

The TBI_TOP subfunction returns the vertical position in pixels of the top side of the button specified by *button_name*. This position is relative to the client area.

If *button_name* isn't a valid button name for *tbid*, Toolkit will generate a runtime error.

Examples

The following example returns the vertical position of the top side of button **btn_name**.

```
top_side = %tb_info(tb_id, TBI_TOP, btn_name)
```

TBI_TYPE

return = %TB_INFO (*tbid*, TBI_TYPE, *button_name*[, *text*])

or

xcall %TB_INFO (*tbid*, TBI_TYPE, *button_name*[, *text*])

Return value

return

The type of a button: (^VAL)

TBLOAD_TYPE_TEXT A text button.

TBLOAD_TYPE_BMAP An image button.

Arguments

tbid

The ID of a toolbar created in Toolkit. (n)

button_name

The name (case-insensitive) of a button on the toolbar specified by *tbid*. (a)

text

(optional) The caption for a text button or the path for the image for an image button. (a)

Discussion

The TBI_TYPE subfunction returns the type of a button and, if *text* is passed, one of the following:

- ▶ If *return* is TBLOAD_TYPE_TEXT, it returns the caption of the specified button in *text*.
- ▶ If *return* is TBLOAD_TYPE_BMAP, it returns the path of the image file (the path passed to the TBB_LOAD subfunction of %TB_BUTTON) as *text*.

If *button_name* isn't a valid button name for *tbid*, Toolkit will generate a runtime error.

Examples

The following example uses the type of button **btn_name** to determine which routine to call, and it passes the caption or the image path to the routine.

```
using (btn_type = %tb_info(tb_id, TBI_TYPE, btn_name, btn_text))
& select
    (TBLOAD_TYPE_TEXT), xcall change_text_btn(btn_name, btn_text)
    (TBLOAD_TYPE_BMAP), xcall change_bmap_btn(btn_name, btn_text)
endusing
```

TBI_WIDTH

```
return = %TB_INFO (tbid, TBI_WIDTH[, button_name])
```

or

```
xcall %TB_INFO (tbid, TBI_WIDTH[, button_name])
```

Return value

return

The width in pixels of the toolbar or toolbar button. (**^VAL**)

Arguments

tbid

The ID of a toolbar created in Toolkit. (**n**)

button_name

(optional) The name (case-insensitive) of a button on the toolbar specified by *tbid*. (**a**)

Discussion

If *button_name* is passed, TBI_WIDTH returns the width of the specified button in pixels. (If *button_name* is passed as a non-blank value that isn't a button name for *tbid*, Toolkit will generate a runtime error.)

If *button_name* is not passed or is blank, TBI_WIDTH returns the width of the toolbar (*tbid*) in pixels. Note that this is the actual width of the toolbar's window (whether placed or not), which may differ from the width passed when the toolbar was created (i.e., the width passed to the TB_CREATE subfunction of %TB_TOOLBAR or to %TB_TKCREATE).

Examples

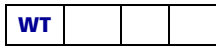
The following example returns the width of the **tb_id** toolbar.

```
tb_width = %tb_info(tb_id, TBI_WIDTH)
```

The next example returns the width of the toolbar button specified by **btn_name**, unless **btn_name** is blank (in which case it returns the toolbar width).

```
tb_width = %tb_info(tb_id, TBI_WIDTH, btn_name)
```

%TB_TKCREATE – Create and log a toolbar with the environment



```
tbid = %TB_TKCREATE ([orientation], [width], [height], [D_NOPLC][, D_GLOBAL])
```

or

```
xcall TB_TKCREATE ([orientation], [width], [height], [D_NOPLC][, D_GLOBAL])
```

Return value

tbid

On Windows, the ID of the toolbar created by this function. On UNIX, always returns false.
On OpenVMS, always returns true. (^**VAL**)

Arguments

orientation

(optional) The toolbar's position in the application container. (**n**)

TBC_TOP Top of container. (default)

TBC_BOTTOM Bottom of container.

TBC_LEFT Left side of container.

TBC_RIGHT Right side of container.

width

(optional) The width of the toolbar in pixels. Pass this when placing the toolbar on the left or right side of the container (which places the toolbar vertically). (**n**)

height

(optional) The height of the toolbar in pixels. Pass this when placing the toolbar on the top or bottom of the container (which places the toolbar horizontally). (**n**)

D_NOPLC

(optional) The no-placement flag; the toolbar will not be placed until you place it with the TB_PLACE subfunction of %TB_TOOLBAR. If not passed, it is placed immediately. (**n**)

D_GLOBAL

(optional) The global toolbar flag; the toolbar is logged at the global environment level. If not passed, the toolbar is logged at the current environment level. (**n**)

Discussion

`%TB_TKCREATE` creates a toolbar and logs it with the current environment.



Once a toolbar has been assigned to an environment, you must leave the management of the toolbar to Toolkit. Use `TB_TKDEL` if you want to explicitly delete the toolbar.

TB_TKDEL – Delete a toolbar that’s logged with the environment



```
xcall TB_TKDEL (tbid)
```

Arguments

tbid

The ID of a toolbar that has been logged with the environment. (n)

Discussion

TB_TKDEL deletes a toolbar that’s been logged with the current environment or the global environment. On UNIX and OpenVMS, this routine is ignored.

If *tbid* is *not* the ID of a toolbar that’s been logged with the current environment or the global environment, a fatal error will occur.

Note that if you use TB_DELETE to delete a toolbar that’s been logged with an environment, a fatal error will occur on the next call to E_ENTER or E_EXIT.

TB_TKGGLOBAL – Promote a toolbar to global



```
xcall TB_TKGGLOBAL (tbid)
```

Arguments

tbid

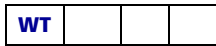
The ID of a toolbar that has been logged at the current Toolkit environment level. (**n**)

Discussion

TB_TKGGLOBAL promotes a toolbar and the current state of all its buttons to the global environment level. On UNIX and OpenVMS, this routine is ignored.

If *tbid* is an invalid toolbar ID or has not been logged with the current environment, a fatal error will occur. (If *tbid* is already global, no error occurs.)

TB_TKLOG – Log a toolbar with the environment



```
xcall TB_TKLOG (tbid[, D_GLOBAL])
```

Arguments

tbid

The ID of a toolbar created by the TB_CREATE subfunction of %TB_TOOLBAR. **(n)**

D_GLOBAL

(optional) The global toolbar flag; the toolbar is logged at the global environment level. If not passed, the toolbar is logged at the current environment level. **(n)**

Discussion

TB_TKLOG logs a toolbar with the environment. On UNIX and OpenVMS, this routine is ignored.

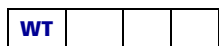
Use this function only if you created the toolbar with the TB_CREATE subfunction of %TB_TOOLBAR. If you pass D_GLOBAL, the toolbar is logged with the global environment. If not passed or is passed as false, the toolbar is logged with the current environment.

If *tbid* is an invalid toolbar ID or has already been logged with an environment, a fatal error will occur.



Once a toolbar has been assigned to an environment, you must leave the management of the toolbar to Toolkit. Use TB_TKDEL if you want to explicitly delete the toolbar.

%TB_TOOLBAR – Create and manipulate a toolbar



```
status = %TB_TOOLBAR (function, id, [orientation], [width][, height])
```

or

```
xcall TB_TOOLBAR (function, id, [orientation], [width][, height])
```

Return value

status

On Windows, the status of the operation: (^VAL)

TB_SUCCESS 1

TB_FAILURE 0

On UNIX and OpenVMS, always returns false.

Arguments

function

The toolbar function to perform: (n)

TB_CREATE Creates the toolbar. It does not display the toolbar on the application window. (To display the toolbar, use TB_PLACE.)

TB_DELETE Deletes the toolbar.

TB_DISABLE Disables a toolbar.

TB_ENABLE Enables a disabled toolbar.

TB_PLACE Places (shows) the toolbar.

TB_POPSTATE Restores a previously saved state of the toolbar.

TB_PUSHSTATE Saves the present state of the toolbar.

TB_REMOVE Removes (hides) the toolbar.

id

The toolbar identifier; the value of *id* is returned on TB_CREATE and passed on all other toolbar functions. (n)

orientation

(optional) Used with TB_CREATE. *Orientation* defines the application window side where the toolbar will be placed. It must be one of the following: **(n)**

TBC_TOP	Top of container (default)
TBC_BOTTOM	Bottom of container
TBC_LEFT	Left side of container
TBC_RIGHT	Right side of container

width

(optional) Used with TB_CREATE. *Width* defines the width of the toolbar in pixels. If *orientation* is TBC_TOP or TBC_BOTTOM, this argument is optional and ignored since the toolbar will be created the full width of the window. **(n)**

height

(optional) Used with TB_CREATE. *Height* defines the height of the toolbar in pixels. If *orientation* is TBC_LEFT or TBC_RIGHT, this argument is optional, ignored, and defaults to the full height of the window. **(n)**

Discussion

The %TB_TOOLBAR function enables you to create and manipulate an application-window toolbar. You must include the file **WND:toolbar.def** to use this function.

TB_CREATE creates a toolbar, but does not display it on the application window, and it does not assign it to an environment. To display the toolbar, use TB_PLACE. To create a toolbar and assign it to an environment, use %TB_TKCREATE. (You can also use TB_TKLOG to assign a toolbar to an environment if you create it with TB_TOOLBAR.) For more information, see [%TB_TKCREATE on page 15-32](#) and [TB_TKLOG on page 15-36](#).

TB_DELETE deletes a toolbar that's not assigned to an environment. If a toolbar is assigned to an environment, use TB_TKDEL; if you use TB_DELETE, a fatal error will occur on the next call to E_ENTER or E_EXIT.

You can place up to four toolbars per application window: one at the top, left, right, and bottom of the window. (You can create up to 256 toolbars per application.) The toolbars will be displayed in the order in which they were placed (with TB_PLACE). When placed, the toolbars will blank out the entire width (or height) of that area of the screen. Note that if you place a toolbar on a side where another toolbar is already placed, the original toolbar is removed, though if it's logged with Toolkit, a subsequent call to E_EXIT could remove the new toolbar and replace it with the original.

If TB_DISABLE is passed, the toolbar remains on the screen, but nothing happens if a button is selected. Disabling a toolbar disables all of its buttons.

TB_PUSHSTATE saves the current state of a Toolbar by pushing it onto a stack. This enables you to save a state, make changes to it, and then restore the state. To restore a state saved by TB_PUSHSTATE, use TB_POPSTATE. The stack follows normal last-on, first-off behavior, and there's no limit to the size of the stack other than available memory. If you pass TB_POPSTATE and there's no saved state for the toolbar, *status* will be TB_FAILURE, and nothing will happen to the toolbar.

TB_PUSHSTATE and TB_POPSTATE may be used for toolbars created with %TB_TOOLBAR, but we recommend that you don't use them for toolbars that are under environment control; this may confuse Toolkit.

Examples

See the %TB_BUTTON [Examples on page 15-4](#).

16

Utility Routines

Introduction to the Utility Routines 16-4

U_ABORT – Process an abort.....	16-5
U_ABOUT – Display an “About” help window	16-8
U_BAR – Place or remove the menu bar or information line	16-10
U_BEEP – Ring the terminal bell.....	16-11
U_CHARSB – Copy or load scroll bar characters	16-12
U_CHECKVERSION – Check for minimum version of any product.....	16-14
U_CHNINF – Return information about a channel.....	16-15
U_CHR – Accept one character from the terminal	16-16
U_CLOSE – Close or purge a channel.....	16-18
%U_CMDLLOPT – Search for a command line option	16-19
U_CNVDAT – Convert lunar and Julian dates	16-21
U_CREATE_SB – Create a scroll bar window	16-22
U_CTRCONTEXT – Establish context within a child window.....	16-24
U_DCDDAT – Decode an alpha date string into its numeric components.....	16-26
U_DCDTIM – Decode an alpha time string into its numeric components	16-29
U_DEBUG – Invoke the UI Toolkit window debugger.....	16-31
U_DIALOG – Generate and process a dialog box	16-32
U_EDIT – Edit text and control line breaks in a window	16-34
U_EDITKEYS – Edit a key map.....	16-40
U_EDITREND – Create and modify rendition schemes.....	16-42
%U_ENUMWNDS – Return a list of windows in a library.....	16-45
U_FINISH – Terminate the UI Toolkit environment.....	16-46
U_FLASH – Flash a message window.....	16-47
U_FLD – Accept one field from the terminal	16-48
U_FMTDAT – Format a date value into an alpha date string.....	16-51
U_FMTTIM – Format a time value into an alpha time string.....	16-53

U_FNDFIL – Search for a file	16-55
U_GBLCHN – Give channel global status	16-57
U_GBLWND – Give window global status	16-58
U_GETCHN – Obtain a channel	16-59
%U_GETFILENAME – Prompt user for filename	16-61
U_GETTXT – Get text from the text library	16-67
%U_GETWNDERR – Get a window system error	16-69
U_HELPREND – Use help window renditions	16-70
%U_HTMLHELP – Invoke HTML Help	16-72
%U_ICON – Set an icon or get an icon count	16-87
U_INITREND – Reset Toolkit renditions	16-89
U_LDREND – Load a rendition scheme	16-90
U_LDWND – Load a window from a library	16-92
U_LOGWND – Log a window as global or local	16-94
U_MESSAGE – Display a message	16-95
U_MINVERSION – Check for minimum Synergy/DE product versions	16-98
U_MODREND – Modify the current rendition scheme	16-99
U_MONTHS – Assign alpha characters to represent months	16-101
%U_MSGBOX – Pop up a Windows-style message box	16-102
U_OPEN – Open a channel/file	16-106
U_PARSEVERSION – Parse a version number	16-108
U_POPUP – Pop up a window	16-109
%U_PRINTQUERY – Retrieve printer characteristics	16-111
%U_PRINTSETUP – Display a Print Setup dialog box	16-113
U_REDRAW – Clear screen and redraw it	16-115
U_REND – Process display renditions	16-116
U_RESIZE – Resize the screen	16-122
U_RESIZESB – Resize a scroll bar window	16-123
U_RLSCHN – Release a channel	16-124
U_RSTREND – Restore renditions	16-125
U_SAVE – Save a window to a window library	16-127
%U_SAVELOG – Save an initialization setting to synergy.ini or synuser.ini	16-128
%U_SAVESETTINGS – Save application window settings to synergy.ini or synuser.ini	16-130

U_SAVREND – Save a rendition scheme to a file	16-132
%U_SELPALETTE – Select a color palette entry	16-134
U_START – Initialize the UI Toolkit environment	16-136
U_STOREND – Store a rendition scheme	16-139
U_TEXT – Load and modify text library entries	16-141
U_UPDATE – Update screen	16-143
U_UPDATESB – Update a scroll bar window	16-145
U_VERSION – Return Toolkit version number	16-147
U_WAIT – Wait for a user response	16-148
%U_WINCELLS – Convert pixels to fractional rows or columns	16-150
%U_WINCOLOR – Perform Windows-specific color operations	16-152
U_WINDOW – Place/move/remove/delete windows	16-159
%U_WINHELP – Invoke WinHelp Help	16-162
%U_WININFO – Return Windows-specific information	16-166
%U_WINMETRICS – Return information about a window	16-170
%U_WNDEVENTS – Associate methods with mouse events	16-203
%U_WNDFONT – Set/retrieve font information on Windows	16-213
%U_WNDSTYLE – Change the style of a list or window	16-232
%U_WNDTYPE – Return a window's type	16-233

The Utility Methods 16-235

USTART_METHOD – Perform application start-up processing	16-236
%UWNDEVENTS_METHOD – Perform window event processing	16-237

Introduction to the Utility Routines

In addition to the routines that handle menus, input windows, text windows, selection windows, and lists, UI Toolkit consists of many other routines which simplify and standardize your application. These are classified as “utility” routines.

Two of the most important Toolkit utility routines are U_START and U_FINISH. These routines initialize and shut down the UI Toolkit environment. U_START must be called before any other Toolkit subroutine.

A subset of the utility routines enables you to load and manipulate windows. These routines are U_GBLWND, U_LDWND, U_LOGWND, U_SAVE, U_WINDOW, U_UPDATE, and U_REDRAW. The routines U_CHR and U_FLD accept a character or field of input from the terminal.

File I/O and channel maintenance are managed using U_CHNINF, U_CLOSE, U_FNDFIL, U_GBLCHN, U_GETCHN, U_OPEN, and U_RLSCHN.

The routines U_DIALOG, U_FLASH, U_MESSAGE, U_POPUP, U_ABORT, U_BAR, and the function %U_MSGBOX enable you to standardize error and status messages.

Key mapping can be performed using U_EDITKEYS.

Rendition utilities are provided through U_EDITREND, U_HELPREND, U_LDREND, U_MODREND, U_REND, U_RSTREND, U_SAVREND, and U_STOREND.

Scroll bars for UNIX and OpenVMS environments may be defined and manipulated through U_CHARSB, U_CREATE_SB, U_UPDATE_SB, and U_RESIZE_SB.

A subset of the utility routines support Windows-specific features: %U_GETFILENAME displays a standard Windows “Open” or “Save As” dialog box. You can set window icons with %U_ICON. %U_PRINTQUERY retrieves printer information, and %U_PRINTSETUP displays a standard Windows “Print Setup” dialog box. Initialization settings can be written to **synergy.ini** or **synuser.ini** using U_SAVELOG and U_SAVESETTINGS. Providing Help in a Windows environment can be done with %U_WINHELP. %U_WNDEVENTS allows you to associate methods with various mouse events, and %U_WNDFONT displays the standard Windows “Font” dialog for changing the font at runtime.

Additional utility routines provide for conversion of date and time values, access to the text message file, access to the UI Toolkit version number, and access to the UI Toolkit window debugger.

U_ABORT – Process an abort



```
xcall U_ABORT (message1[, message2, ..., message5])
```

Arguments

message1 to
message5

Up to five error messages. *Message1* is required. *Message2* through *message5* are optional. (a)

Discussion

When a Toolkit error or a window error is generated, the error is automatically sent to U_ABORT, unless the error is returned in an argument for a Toolkit routine, such as the *error* argument for L_CREATE. (If this happens, the error number is returned in the argument, and you can use %U_GETWNDERR to get the error text.) However, you can also call U_ABORT directly when you detect a fatal error condition in your application.

The U_ABORT subroutine behaves differently depending on the value of **g_throwabort**, a global defined in **tkctl.def**. The default behavior (active when **g_throwabort** is set to zero) may be adequate for interactive Toolkit applications, where the resulting fatal error message can be displayed to the screen. But for error trapping, use the second type of U_ABORT processing (active when **g_throwabort** is set to a non-zero value). Error trapping gives you greater control over errors generated by U_ABORT and enables server-side applications and applications running as services to pass error information on to client programs. You can set the **g_throwabort** global from your program or you can change its initial setting by setting the DTK_THROW_ABORT environment variable.

If **g_throwabort** is set to zero (the default), U_ABORT performs a STOP and displays a fatal error message. The STOP will have an exit status of D_EXIT_FAILURE, and U_ABORT will not return to the calling routine. (Most operating systems have commands that enable you to test the exit status. See your operating system documentation for information.) The error message will have the following format:

**** ERROR ** ERROR ** ERROR ****

followed by the contents of the *message1* to *message5* argument(s). If you pass more than one *message* argument, each message will start a new physical line of text. You can pass up to five messages.

Additionally, note the following:

- ▶ On Windows, this routine creates a Windows-style message box with a stop sign icon and an OK button. Each message can wrap, so the total length of all messages is limited only by your desktop's size.
- ▶ On UNIX and OpenVMS, each message can be up to 70 characters long. Additional characters are truncated.
- ▶ U_ABORT cannot time out on Windows.
- ▶ On UNIX and OpenVMS, if a time-out occurs **g_select** is set to true and **g_entnam** is set to the entry name specified in **tkctl.def** for **g_time_entry**. By default, **g_time_entry** is defined as "O_TIMEOUT".

If **g_throwabort** is set to a non-zero value, U_ABORT calls U_FINISH to shut down the Toolkit environment and then calls EXITE to throw a trappable error with the mnemonic \$ERR_TOOLKIT and the error text "Toolkit error." Message arguments passed to U_ABORT are passed on as informational messages for the thrown error and can be retrieved using %ERR_TRACEBACK once the error has been trapped. Because U_ABORT shuts down the Toolkit environment, you will need to call U_START to perform any further Toolkit operations.

If **g_throwabort** is set to throw an error, your application should be written to catch and handle the thrown error. Otherwise the application may continue to execute in situations where it would have terminated had U_ABORT performed a STOP. For example, if input is redirected but the application's error catching mechanism does not terminate the application after a certain number of EOFs, U_ABORT may cause the application to run indefinitely. Additionally, you may need to modify any automated procedures (batch files and so forth) that invoke U_ABORT so that they work with the new behavior.

If an interactive Toolkit application fails to trap an error thrown by U_ABORT

- ▶ on Windows, the Synergy™ runtime will display a fatal error message box (which is different than the message displayed with U_ABORT's default behavior).
- ▶ on UNIX and OpenVMS, the Synergy runtime will generate a fatal error traceback (which does not require user input and which returns immediately to the command prompt). Additionally, U_FINISH will clear the screen unless the error occurs within U_START.

See also

- ▶ **ONERROR** in the "Synergy DBL Statements" chapter of the *Synergy DBL Language Reference Manual*.
- ▶ **%ERR_TRACEBACK** in the "System-Supplied Subroutines and Functions" chapter of the *Synergy DBL Language Reference Manual*.
- ▶ **DTK_THROW_ABORT** in the "Environment Variables" chapter of *Environment Variables & System Options*.

Examples

The following call generates the specified messages (in addition to the ****ERROR**** message) if **g_throwabort** is set to 0. If **g_throwabort** is set to a non-zero value, it throws a trappable error and passes the messages on as informational error messages for the thrown error.

```
xcall u_abort("Customer file is corrupt", "Please call your distributor")
```

The next example is written for error trapping (active when **g_throwabort** is non-zero). It uses **ONERROR** to enable error trapping and **%ERR_TRACEBACK** to retrieve error information.

```
proc
  xcall u_start
  .
  .
  .
  onerror ($ERR_CATCH) catch      ;Need this to activate error trapping.
  xcall u_open(chan, "I", " cusmas ")
  reads(chan, cusmas, bad_cusmas) ;We need to have a control record.
  if (cusmas.control .ne. 0)      ;If it is not a control record...
    begin
bad_cusmas, ;Either we have no record, or it is not a control record
      xcall u_abort("This is the first line of the message.",
&          "This is the second line of the message.")
    end
  .
  .
  .
catch,
  offerror      ;Prevents an error within catch from looping back to
                ; the catch label.
  xcall u_start ;Need this because U_ABORT shuts down Toolkit, but we
                ; need Toolkit to call U_MSGBOX.
  set cnt, len = 1
  while cnt      ;Loop through messages passed by U_ABORT.
    begin
      cnt = %err_traceback(errtext(len, ^size(errtext)))
      if (cnt)
        begin
          len = %trimz(errtext) + 1
          errtext(len:2) = "
          len += 2
        end
      end
    end
  xcall u_msgbox(errtext(1:len), D_MICONSTOP, "Trapped error")
  .
  .
  .
```

U_ABOUT – Display an “About” help window



```
xcall U_ABOUT (application, version, compiled)
```

Arguments

application

The name of the application. **(a)**

version

The version of the application. **(a)**

compiled

The date the application was compiled. **(a)**

Discussion

The U_ABOUT subroutine displays an “About” help window. An “About” window contains general information about the current application:

- ▶ Application name
- ▶ Version
- ▶ Date compiled
- ▶ Current version of Synergy DBL
- ▶ Current version of UI Toolkit
- ▶ Application icon (in Windows environments only)

You pass the text for the first three items (in *application*, *version*, and *compiled*) when you call U_ABOUT. U_ABOUT supplies the information for the next two items. If you have specified an application icon with %U_ICON, it will be displayed in the window.

U_ABOUT creates a window named “u_about” and displays it on the screen. It then waits for the user to press RETURN (or click the OK button in a Windows environment). After the user responds, the window is deleted, and this subroutine returns. Typically, you will call this subroutine within a case statement in your EUTILS_METHOD to support an “About” hot entry for your application.

The About window uses the current help window renditions. It displays 16 lines above the information line, and is centered horizontally on the screen. Each of the items in the bulleted list above is displayed on a separate line. The text is centered horizontally. Each line of information can contain a maximum of 76 characters. The About window title displays as “About *application*.”

Static text in the window is maintained in the text message file, under facility DTK. Mnemonics are ABOUT, VERSION, COMPILED, and DTKVERS.

If a window named “u_about” already exists when this subroutine is called, a fatal error occurs.

In a Windows environment, this subroutine creates a Windows-style message box with an OK button and any application icon you have specified (using %U_ICON). The text within the message box will be left-justified rather than centered. Note that U_ABOUT cannot time out on Windows.

If a time-out occurs on UNIX or OpenVMS, **g_select** is set to true and **g_entnam** is set to the entry name specified in **tkctl.def** for **g_time_entry**. See [g_time_entry](#) on page 18-6.

Examples

The following displays an “About” window like the ones in [figure 16-1](#) and [figure 16-2](#).

```
xcall u_about("Repository", "8.3.1", %datecompiled)
```

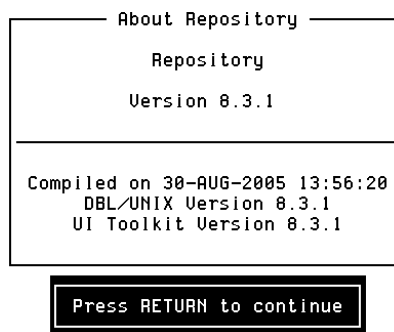


Figure 16-1. An “About” window in a UNIX or OpenVMS environment.

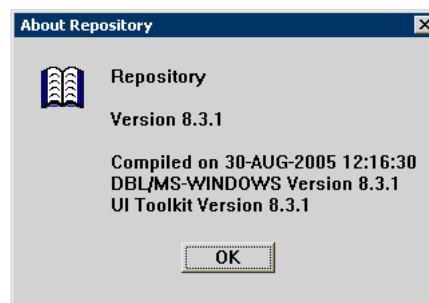


Figure 16-2. A Windows-style “About” window.

U_BAR – Place or remove the menu bar or information line



```
xcall U_BAR (/D_PLACE | D_REMOVE,] bar[, ...])
```

Arguments

D_PLACE

(optional) Place the menu bar or information line. (default) **(n)**

D_REMOVE

(optional) Remove the menu bar or information line. **(n)**

bar

One or both of the following items to place or remove: **(n)**

D_MENU Menu bar.

D_INFO Information line.

Discussion

The U_BAR subroutine places or removes the menu bar and/or information line from the screen. It can have a maximum of three arguments.

On Windows, D_REMOVE has no effect on a menu bar. A menu bar always displays if any menu columns have been placed.

On UNIX and OpenVMS, if the menu bar is removed from the screen and the process-menu key is pressed, the menu bar will temporarily be placed on the screen until an entry is selected or until the process-menu key is pressed again. You can remove the menu bar permanently by redefining the process-menu key to be nothing.

When D_REMOVE is used, the extra row in the display is only available for the placement and display of an unbounded window.

Examples

The following example removes both the menu bar and the information line from the screen.

```
xcall u_bar(D_REMOVE, D_MENU, D_INFO)
```

U_BEEP – Ring the terminal bell



```
xcall U_BEEP
```

Discussion

The U_BEEP subroutine rings the terminal bell. It can replace the following DISPLAY statement, which also rings the terminal bell:

```
display(g_terminal, 7)
```

Using U_BEEP rather than the above DISPLAY statement enables you to set a debugger breakpoint at the U_BEEP call in your code, and then examine the traceback to track down the source of an unwanted terminal bell.

Note that U_BEEP has no effect if **g_beep** is set to false. See [g_beep on page 18-2](#).

U_CHARSB – Copy or load scroll bar characters



```
xcall U_CHARSB (D_LOAD | D_COPY, characters)
```

Arguments

D_LOAD

Set scroll bar characters. (**n**)

D_COPY

Return current scroll bar characters. (**n**)

characters

The series of scroll bar characters. (**a12**)

Discussion

The U_CHARSB subroutine copies or loads the characters you want to use for denoting the various states of a scroll bar window in UNIX or OpenVMS environments. (This routine does work on Windows, but the results are non-standard.)

If D_LOAD is specified, the contents of *characters* will supersede any previously-defined scroll bar characters. D_COPY returns to the current settings for the scroll bar characters in *characters*.

Characters contains 12 individual characters to display in a scroll bar. The table below shows the order of specification and the “initial” values of the characters upon program setup. The horizontal and vertical fill characters are set to the horizontal and vertical line-drawing characters. This makes the indicator appear to “slide” along a line, which many times coincides with the window’s border. Renditions for scroll bar sections may be modified with the routines U_REND, U_MODREND, or U_EDITREND.

Scroll Bar Characteristics	
Description	Initial setting
Scroll to “more” upwards	^
Scroll to “more” downwards	v
Scroll to “more” to the left	<
Scroll to “more” to the right	>
Screen position already at the top edge; there is “no more” upwards	^

Scroll Bar Characteristics (Continued)	
Description	Initial setting
Screen position already at the bottom edge; there is “no more” downwards	v
Screen position already at the left edge; there is “no more” to the left	<
Screen position already at the right edge; there is “no more” to the right	>
Vertical pointer	—
Horizontal pointer	
Horizontal area fill	—
Vertical area fill	

Examples

The following example sets the “more” upwards and “more” downwards scroll bar characters to ‘+’ and sets the “no more” characters and “fill” characters to blank. The other characters are set to the default values.

```
chars  ,a12

chars = "++<>  - |  "
xcall u_charsb(D_LOAD, chars)
```

U_CHECKVERSION – Check for minimum version of any product



```
xcall U_CHECKVERSION(product, version, minvers)
```

Arguments

product

The product name (displayed in the error message if the comparison results in a fatal error). **(a)**

version

The version of the product in use. **(a)**

minvers

The minimum version required. **(a)**

Discussion

This subroutine checks *version* to see if it is at least as high as *minvers*. If not, a fatal error occurs.

The *version* and *minvers* strings are expected to be in the form *v*[*r*][*m*], where *v* is the application version number, *r* is the major revision number, and *m* is the minor revision number. Note that you can use an uppercase letter for a two-byte version number. For example, you can use A for 10 as in A.1.1 (for 10.1.1). But if you do use an uppercase letter, it must be the first character of *v*, *r*, or *m*, and a second character for *v*, *r*, or *m* is ignored. For example, 1A.2.3 is interpreted as 1.2.3, and A1.2.3 is interpreted as 10.2.3.

Examples

The following example compares the current Toolkit version in **version**, with the minimum allowed version, defined by MIN_VRSN. If **version** is less than MIN_VRSN, a fatal error occurs.

```
xcall u_version(version)
xcall u_checkversion("UI Toolkit", version, MIN_VRSN)
```

U_CHNINF – Return information about a channel



```
xcall U_CHNINF (channel, [filename], [mode][, class])
```

Arguments

channel

The channel about which to obtain information. **(n)**

filename

(optional) Returned with the name of the file open on the channel. **(a)**

mode

(optional) Returned with the I/O mode in which the file is open. **(a)**

class

(optional) Returned with the environment level that owns the channel if the channel is local or one of the following class values: **(n)**

- 0** Channel is not in use (according to Toolkit).
- 1** Channel is global.
- 2** Channel is a Toolkit system channel.

Discussion

The U_CHNINF subroutine returns information about a channel.

If *channel* is not in use, *filename* and *mode* will be returned blank.

Examples

In the following, U_CHNINF returns the name of the file open on the channel stored in **chnl**, the I/O mode for which the file is open, and the class of the channel (local, global, and so forth).

```
xcall u_chninf(chnl, filnam, iomode, class)
```

This example closes all global channels.

```
for chnl from 1 thru 32
  begin
    xcall u_chninf(chnl, , , class)
    if (class.eq.-1)
      xcall u_close(chnl)
  end
```

U_CHR – Accept one character from the terminal



```
xcall U_CHR (character, [window_id], [row, col], [help_id][, wait_time])
```

Arguments

character

Returned with the character that was input. **(a)**

window_id

(optional) The ID of the window in which to perform input. This is required if echo is on. **(n)**

row, col

(optional) The window row and column positions this subroutine will be reading from. **(n)**

help_id

(optional) The help identifier. **(a)**

wait_time

(optional) The time-out limit for I/O processing to be completed before returning to the calling routine. **(n)**

D_FOREVER or **-1**

Never time out.

D_GLOBAL or **-2**

Use the global value (see [g_wait_time on page 18-6](#)). (default)

D_IMMEDIATE or **0**

Time out immediately.

n

Wait up to *n* seconds (where *n* is a positive value) for I/O processing to be complete.

Discussion

The U_CHR subroutine accepts one character of input from the terminal. If you are performing input to a list item, use the L_CHR subroutine instead of U_CHR. While waiting for character input, this subroutine will do any menu or shortcut processing.

On Windows, U_CHR recognizes the following reserved menu entries on Windows. E_PASTE pastes a single character if echo is on. The others do nothing when U_CHR is being processed.

E_CLEAR

E_COPY

E_CUT

E_MARK

E_PASTE

If echo is on (in other words, if the D_ECHO state for E_STATE is on), you must pass *window_id*. If echo is off, *window_id* is optional. In that case, U_CHR accepts a character (in *character*) without displaying it.

If you pass *row*, you must also pass *col*. If you omit both, U_CHR uses the current display location in the window. Note that if echo is on, the display location (whether it's the default or a location set by *row* and *col*) must be visible. If not, Toolkit will generate a fatal error.

If **g_select** is set on return, a menu entry has been selected, and **g_entnam** contains the name of the selected entry. If the internal name of the selected menu entry begins with "U_," the EUTILS_METHOD subroutine is called and the input is redone. If the internal name of the selected menu entry is "O_HELP," the EHELP_METHOD subroutine is called with *help_id* as its argument.

You will need to clear **g_select** after you satisfy the event.

On return, the field **gi_sts** (defined in **inpctl.def**) will be set with the status. If a printable character was entered, **gi_sts** will be NORMAL_INPUT (zero) and the character received will be in *character*. If a control character (ASCII value 1-31) was input, **gi_sts** will be non-zero and the control character will be in *character*. If any other character was input, **gi_sts** will be non-zero, and *character* will be a null (binary zero).

If the operation times out, **g_select** is set to true and **g_entnam** set to the entry name specified in **g_time_entry**, which is defined in **tkctl.def**. See [g_time_entry on page 18-6](#).

See also

[L_CHR on page 9-66](#) for information on performing input to a list item.

Examples

The following example returns the character that was input to the variable **chr**. This input comes from the current position in the window whose ID is stored in **wndid[i]**.

```
xcall u_chr(chr, wndid[i])
```

The next example also returns the character that was input to **chr**. Input comes from the current input window at position (15,30). The subroutine passes the identifier "glhelp" to the EHELP_METHOD subroutine.

```
xcall u_chr(chr,, 15, 30, "glhelp")
```

U_CLOSE – Close or purge a channel



```
xcall U_CLOSE (close_channels[, D_PURGE, purge_channels])
```

```
xcall U_CLOSE (D_PURGE, purge_channels)
```

Arguments

close_channels

One or more channels to be closed. If there's more than one channel, separate them with commas. (**n**)

D_PURGE

(optional) Purge the following channels. (**n**)

purge_channels

(optional) One or more channels to be purged. If there's more than one channel, separate them with commas. (**n**)

Discussion

The U_CLOSE subroutine closes or purges one or more channels.

If D_PURGE is not passed, all specified channels are closed.

If D_PURGE is specified, channels that follow it are purged, and if any channels precede it, they are closed.

Examples

In this example, the channel stored in **chn** is closed:

```
xcall u_close(chn)
```

In the following example, the channels stored in **chn_1** and **chn_2** are closed, and the channel stored in **purchn** is purged:

```
xcall u_close(chn_1, chn_2, D_PURGE, purchn)
```

In the next example, the channels stored in **purchn_1** and **purchn_2** are purged:

```
xcall u_close(D_PURGE, purchn_1, purchn_2)
```

%U_CMDLINOPT – Search for a command line option

WT	WN	U	V

status=%U_CMDLINOPT (*option*, [*max*], [*args*][, *argcount*])

Return value

status

True if the option is found; false if not. (^VAL)

Arguments

option

Command line option to search for. (**a**)

max

(optional) Maximum number of arguments to return for the option. (**n**)

args

(optional) Array for returned arguments. (**a**)

argcount

(optional) Number of returned arguments. (**n**)

Discussion

%U_CMDLINOPT searches the command line for the command line option specified by *option*. A command line option is defined as a dash (-) or a slash (/) (if you've set DBLOPT 34) followed immediately by a string that identifies the option. The string can be up to 80 characters. If the option has arguments, they follow the option with separating spaces.

%U_CMDLINOPT parses the entire command line. If the option occurs multiple times, the argument sets are concatenated. Unless the last character in *option* is a space, %U_CMDLINOPT selects each option that starts with the characters specified by *option*, even if an option has more characters than *option* specifies. For example, if you specify "n" for *option*, %U_CMDLINOPT will select **-number**. However, if you specify "n " (note the space), %U_CMDLINOPT selects only **-n** or **/n**, not **-number** or **/number**.

If *max* is passed, it specifies the maximum number of arguments that may be returned for this option. If *max* is greater than 0, *args* is presumed to be the base of an array with *max* or greater elements that contains the returned arguments. If *argcount* is passed, it is returned with the actual number of arguments found for the option, even if that number is greater than *max*. (Note, however, that only *max* arguments will be returned in *args*.)

For example, to retrieve the first two options (opt1 and opt2) from the command line

```
dbr myprogram -a "opt1" "opt2" "opt3"
```

your program could have the following:

```
status = %U_CMDLINOPT("a", 2, options_array, count)
```

Switches that precede the program name on the command line are ignored. For example, %U_CMDLINOPT will always return false for the following command:

```
dbr -n program_name
```

It may return true, however, for the following:

```
dbr program_name -n
```

Examples

The following example looks for an option with **-r** (or **/r** if DBLOPT 34 is set):

```
if (%u_cmdlinopt("r"))
```

Note that **-rec** would qualify because it starts with **-r**. To search for a command line option that contains only **-r** or **/r**, specify the following:

```
if (%u_cmdlinopt("r "))
```

Note the space after the r in the above example; this prevents anything other than **-r** (**-rec** for example) from qualifying.

In the next example, %U_CMDLINOPT looks for **-names** or **/names**. Note that it will retrieve only 10 arguments and that the address of the returned arguments will be assigned to nmnames.

```
if (%u_cmdlinopt("names ", 10, names, nmnames))
```


U_CNVDAT – Convert lunar and Julian dates

WT	WN	U	V

```
xcall U_CNVDAT (mode, year, month, day, julian)
```

Arguments

mode

Determines the type of conversion: **(n)**

1 Convert lunar to Julian.

2 Convert Julian to lunar.

year

The year to convert. **(n)**

month

The month to convert or is returned with the converted month. **(n)**

day

The day to convert or is returned with the converted day. **(n)**

julian

The Julian day or is returned with the converted Julian day. **(n)**

Discussion

The U_CNVDAT subroutine converts lunar (month-oriented) dates to Julian and vice versa.

U_CNVDAT returns either a converted month and day in *month* and *day* or a converted Julian day in *julian*, depending on whether the mode is 1 or 2. The year stays the same regardless of *mode*.

If any of the input arguments are invalid, number 9s will be returned.

Examples

The following example places the Julian day for January 25, 1994 into **julie**.

```
xcall u_cnvdatt(1, 94, 1, 25, julie)
```

The next example places the month and day of the 265th day of 1993 into **mn** and **dy**.

```
xcall u_cnvdatt(2, 93, mn, dy, 265)
```

U_CREATESB – Create a scroll bar window



```
xcall U_CREATESB (scroll_id, name, size, [type][, endpoints])
```

Arguments

scroll_id

Returned with the ID of the scroll bar window created. **(n)**

name

The name of the scroll bar window (a maximum of 15 characters). **(a)**

size

The length or height of the scroll bar. **(n)**

type

(optional) The type of scroll bar window to create. **(n)**

D_VERTICAL Vertical scroll bar. (default)

D_HORIZONTAL Horizontal scroll bar.

endpoints

(optional) A flag that specifies whether or not endpoint indicators should be displayed. **(n)**

D_ENDPOINTS Display endpoint indicators. (default)

D_NOENDPOINTS Do not display endpoint indicators.

Discussion

The U_CREATESB subroutine creates a scroll bar window in UNIX and OpenVMS environments. A scroll bar window is one character wide if vertical, one character high if horizontal. (This routine does work on Windows, but the results are non-standard.)

Endpoints are single-character indicators at either end of the scroll bar which indicate whether or not further progress in that direction is possible. These characters, along with the character for the background fill for the scroll bar, can be changed with the U_CHARSB subroutine.

A scroll bar window indicates progress and regress with the movement of an indicator (through the window's display area). For example, if you use a scroll bar with a list, the entire scroll bar represents the whole list; the location of the indicator within the scroll bar area represents the location of the current item relative to the list.

As you move through the list, you would update the scroll bar to represent the relative movement through the list using U_UPDATESB. Upon creation, the indicator is not placed within the scroll bar. U_UPDATESB must be called to place the indicator.

By default, U_CREATESB creates scroll bar windows without borders, but you can change this by calling W_BRDR. For more information, see [W_BRDR](#) in the “Synergy Windowing API” chapter of the *Synergy DBL Language Reference Manual*.

U_CREATESB automatically logs the scroll bar window to the current environment.

If you pass a null string (“”) for *name*, Toolkit assigns a unique window name in the form `_W_nnn`, where *nnn* is the window ID.

A scroll bar window is deleted when the environment in which it was created is exited using the E_EXIT subroutine. To place, remove, and explicitly delete a scroll bar, use U_WINDOW.

See also

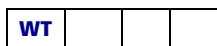
- ▶ [U_CHARSB on page 16-12](#) for information on changing the character used to denote various states of a scroll bar.
- ▶ [U_RESIZESB on page 16-123](#) for information on resizing a scroll bar window.
- ▶ [U_UPDATESB on page 16-145](#) for information on updating a scroll bar window.
- ▶ [U_WINDOW on page 16-159](#) information about placing, moving, removing and deleting windows.

Examples

The following example creates a vertical scroll bar window “vscroll,” which is 10 characters tall, and has no endpoint indicators.

```
xcall u_createsb(scrlid, "vscroll", 10,, D_NOENDPOINTS)
```

U_CTRCONTEXT – Establish context within a child window



```
xcall U_CTRCONTEXT (window_id, reason)
```

Arguments

window_id

The ID of the child window. (**n**)

reason

The reason the window was activated (which determines what U_CTRCONTEXT will move context to). This argument can be passed as blank (which indicates that the window was already activated), a program-supplied reason passed because it was activated programmatically, or one of the following: (**a**)

C_FIRST	Focus moved in attempt to reach the first child in the composite window. This instructs U_CTRCONTEXT to move focus to the window itself (rather than a button on the window).
C_LAST	Focus moved in attempt to reach the last child in the composite window. This instructs U_CTRCONTEXT to move context to the last active button for the window. If the window is part of a tab set with an active button, this moves context to the last active button for the tab set. If there is no active button for the window or tab set, this moves focus to the window itself.
C_NEXT	Focus moved from the previous child in the composite window—e.g., by pressing TAB. This instructs U_CTRCONTEXT to move context to the window itself (rather than a button on the window).
C_PREV	Focus moved from the next child in the composite window—e.g., by pressing SHIFT+TAB. This instructs U_CTRCONTEXT to move context to the last active button for the window. If the window is part of a tab set with an active button, this moves context to the last active button for the tab set. If there is no active button for the window or tab set, this moves focus to the window itself.
C_SET	Window or button explicitly activated. For example, this could be the result of a mouse click, or it could be that the window or button was activated programmatically without a specified reason. With this reason, U_CTRCONTEXT does not change focus.

If *reason* is blank or any value other than the C_ values above, U_CTRCONTEXT does not change focus

Discussion

When used in a child processing method, U_CTRCONTEXT enables users to tab through buttons in windows that are part of a composite window. For example, if a composite window has a text window and a list, and the user presses SHIFT+TAB to move from the list back to the text window, U_CTRCONTEXT moves focus to the last button in the text window.

U_CTRCONTEXT uses *reason* to determine where to establish focus, and since C_PROCESS passes a reason code to child processing methods, this reason code can then be passed as *reason* to U_CTRCONTEXT. This enables focus to initially reflect the reason the window was activated.

There are several similar Toolkit routines that are designed to be called from child processing methods and that use a reason code to control focus in a child, but U_CTRCONTEXT is for child windows that are not input windows, container windows for lists, tab set windows, or composite container windows. For example, it is used in the default methods C_METHAX (for ActiveX windows), C_METHSEL (for selection windows), and C_METHTXT (for text windows). (For the other types of windows listed above, see [I_CTRCONTEXT](#) on page 8-9, [L_CTRCONTEXT](#) on page 9-80, [DTS_CTRCONTEXT](#) on page 13-36, and [DC_CTRCONTEXT](#) on page 6-33.)

For examples of how to use U_CTRCONTEXT in a child processing method, see the **ctrmeths.dbl** file included in the UI Toolkit distribution. This file has the code for the default methods C_METHAX, C_METHSEL, and C_METHTXT, all of which use U_CTRCONTEXT.

On UNIX and OpenVMS and in Synergy .NET, this routine is ignored.

See also

- ▶ [C_METHAX](#) on page 6-47.
- ▶ [C_METHSEL](#) on page 6-61.
- ▶ [C_METHTXT](#) on page 6-65.
- ▶ “Controlling tabbing, focus, and input context” on page 6-4 for information on tabbing indices.

U_DCDDAT – Decode an alpha date string into its numeric components

WT	WN	U	V

```
xcall U_DCDDAT (status, [string], type, [D_TODAY], year, month, day)
```

Arguments

status

Returned with the decoding status (0 if successful, non-zero if unsuccessful). **(n)**

string

(optional) The string to convert to numeric form. **(a)**

type

The type of date into which *string* will be converted: **(n)**

- 1** *PP/YYYY* (or *PP/YY* if *year* accepts only two digits)
- 2** *PP/YYYY* (or *PP/YY* if *year* accepts only two digits)
- 3** *DD/MM/YYYY* (or *DD/MM/YY* if *year* accepts only two digits)
- 4** *DD/MM/YYYY* (or *DD/MM/YY* if *year* accepts only two digits)

D_TODAY

(optional) Specifies that the date should default to today if *string* is passed and blank. **(n)**

year

Returned with the year. **(n)**

month

Returned with the month or period. **(n)**

day

Returned with the day. **(n)**

Discussion

The U_DCDDAT subroutine decodes an alpha date string into separate numeric values for period and year (*PP/YYYY*), or for month, day, and year (*DD/MM/YYYY*).

String can consist of numbers only (up to eight digits), or it can consist of numbers separated by a delimiter. If *string* is not passed, the current date is returned and *type* is ignored.

If *string* is delimited,

- ▶ one non-alphanumeric delimiter must separate date components (e.g., '1-23-2006').
- ▶ the month can be a three-letter abbreviation. The default is the first three letters of the English name for a month, but this can be overridden with U_MONTHS.
- ▶ the current day, month, or year is used for any missing date component. For example, if *string* is 10//2006 and **g_date_order** is 0, the current day will be used for the missing day.

If *string* consists only of numbers (no delimiter), *string* is parsed and interpreted according to the number of digits passed in *string* and the value passed for *type*, as listed in the table below.

Digits in string	type	Interpretation of string
1 or 2	1 or 2	<i>String</i> is a period in the current year.
	3 or 4	<i>String</i> is a day in the current month.
3 or 4	1 or 2	If g_date_order is 0 or 1, the two lowest-order digits of <i>string</i> are a year in the default century, and the one or two remaining digits are the period. If g_date_order is 2, the two lower-order digits are the period, and the one or two remaining digits are the year.
	3 or 4	<i>String</i> is the month and day in the current year. The two lowest-order digits are the second component of the date, as defined by g_date_order , and the one or two remaining digits are the first component.
5 or 6	1 or 2	If g_date_order is 0 or 1, the two lowest-order digits are the year, and one or two remaining digits are the period. If g_date_order is 2, the two lowest-order digits are the period, and the one or two remaining digits are the year.
	3 or 4	If g_date_order is 0 or 1, the two lowest-order digits are a year in the default century, the next two digits are the second component of the date (as defined by g_date_order). The one or two remaining digits are the first component of the date. If g_date_order is 2, the two lowest-order digits are the day, the next two are the month, and the one or two remaining digits are the year in the default century.
7 or 8	1 or 2	<i>Invalid (results in error).</i>
	3 or 4	If g_date_order is 0 or 1, the four lowest-order digits are the year, the next two digits are the second component of the date, and the one or two remaining digits are the first date component. If g_date_order is 2, the two lowest-order digits are the day, the next two digits are the month, and the three or four remaining digits are the year.

The default century is the century for the current system date unless a default century method has been registered (with E_METHOD), in which case the method must return the default century.

Examples

The following example decodes the date stored in **date** to the form *DD/MM/YYYY* and returns it in **yr**, **mn**, and **dy**. (For example, if **date** contains 1/30/1999, **yr** will be returned with 1999 (or 99 if it accepts only two digits), **mn** will be returned with 01, and **dy** will be returned with 30.) **Sts** will contain the decoding status.

```
xcall u_dcddat(sts, date, 4, , yr, mn, dy)
```

The next example decodes today's date to the form *DD/MM/YYYY* (or *DD/MM/YY* if **yr** accepts only two digits) and returns it in **yr**, **mn**, and **dy**.

```
xcall u_dcddat(sts,, 3, D_TODAY, yr, mn, dy)
```


U_DCDTIM – Decode an alpha time string into its numeric components

W	T	W	N	U	V

```
xcall U_DCDTIM (status, [string], [D_NOW], hour, minute[, second])
```

Arguments

status

Returned with the decoding status (zero if valid, non-zero if not). **(n)**

string

(optional) The time string to convert to numeric form. **(a)**

D_NOW

(optional) Specifies that the time should default to the current system time if *string* is passed and blank. **(n)**

hour

Returned with the converted hour. **(n)**

minute

Returned with the converted minute. **(n)**

second

(optional) Returned with the converted second. **(n)**

Discussion

The U_DCDTIM subroutine decodes an alpha time string into separate numeric values for hour, minute, and second.

If you don't pass *string*, the current system time will be returned and D_NOW, if passed, will be ignored.

String can be either numeric or delimited. A delimited time string has any number of non-digit characters separating each of the time's components.

If *string* is a numeric string containing one or two digits, UI Toolkit considers it to be the hour. If it contains three or four digits, it is considered to be the hour and minute. (The rightmost two digits will be the minute, and the leftmost digits will be the hour.) If five or more digits are present, they are considered to be the hour, minute, and second. The rightmost two digits will be the second, the next two digits are the minute, and the leftmost digits are the hour.

If *string* is delimited, and if any component of the time is missing, the missing part will be assumed to be zero. If one of the non-numeric components is the post-meridian indicator (pm), twelve hours will be added to the time specified.

The post-meridian indicator is the text defined in the UI Toolkit message file as facility name DTK, mnemonic PM. It is not case sensitive. As distributed, this is set to “PM.” Any abbreviation of this text will be recognized. Thus “1:45p” will be recognized as 13:45 military time, so long as the text file entry for PM starts with a ‘p’ (uppercase or lowercase).

Examples

The following example decodes the time string in **time** to the form *HHMMSS* and returns the *HH* value in **hr**, and the *MM* value in **mn**. The *SS* value is ignored. (For example, if **time** contains “03:45 pm,” **hr** will be returned with 15, and **mn** will be returned with 45.) **Sts** will contain the decoding status.

```
xcall u_dcdtim(sts, time,, hr, mn)
```

The next example decodes the current system time to the form *HHMMSS* and returns the *HH* value in **hr**, the *MM* value in **mn**, and the *SS* value in **sc**.

```
xcall u_dcdtim(sts,, D_NOW, hr, mn, sc)
```

U_DEBUG – Invoke the UI Toolkit window debugger



```
xcall U_DEBUG
```

Discussion

The U_DEBUG subroutine invokes the UI Toolkit window debugger. See [“Debugger” on page 17-1](#) for information on using the Toolkit window debugger.

On Windows, U_DEBUG brings up the Toolkit window debugger in a separate window. The initial size and location of this window is user-definable using the DBG_X, DBG_Y, DBG_WIDTH, and DBG_HEIGHT environment variables. The font is user-definable using the FONT_DEBUG environment variable. See more information see the following in the “Environment Variables” chapter of *Environment Variables & System Options*: [DBG_X](#), [DBG_Y](#), [DBG_WIDTH](#), [FONT_DEBUG](#).

U_DIALOG – Generate and process a dialog box



```
xcall U_DIALOG (response, [row, col], [border_flag], message[, ...])
```

Arguments

response

Returned with the response entered by the user in the dialog box's input field. **(a)**

row, col

(optional) The row and column where the window will be placed on the screen. **(n)**

border_flag

(optional) Specifies whether the window border around the dialog box is on or off: **(n)**

D_ON Border is on. (default)

D_OFF Border is off.

message

A line of message text. You can have up to 12 lines of text, but each line must be specified as a separate *message* argument. **(a)**

Discussion

The U_DIALOG subroutine generates and processes a simple dialog box.

The dialog box contains one input field for a user response and up to 12 lines of text (specified in *message*). The height of the box is determined by the number of message lines. The width of the box is the width of the longest message line, plus the response field. The response field immediately follows the text of the last message line. The size of the response field is determined by the data declaration of *response*. The window is centered if row and column are not specified or if they have values of 0.

To obtain a default response, set the response argument to the default value before calling U_DIALOG.

You can set the window rendition using the U_REND subroutine and the D_RND_DLG display rendition.

On Windows, U_DIALOG recognizes the following menu entries, which are described in [“Appendix B: Reserved Menu Entries”](#): E_CLEAR, E_COPY, E_CUT, E_MARK, E_PASTE.

If a time-out occurs, **g_select** is set to true and **g_entnam** is set to the entry name specified in **tkctl.def** for **g_time_entry**. By default, **g_time_entry** is defined as “O_TIMEOUT”.

Examples

This example generates a dialog box that contains a line of text and an input field for the user's response, which will be stored in **resp**. The dialog box is placed at row 1, column 1 of the screen body and has a border. "N" is the default response.

```
resp="N"
xcall u_dialog(resp, 1, 1, D_ON,
&              "Are you sure you want to delete that file? [Y/N] ")
```

U_EDIT – Edit text and control line breaks in a window



```
xcall U_EDIT(window_id, text_array, [line], [character], [flags], [col_id1], [col_id2],
&          [col_id3], [help_id][, decimal_value])
```

Arguments

window_id

The ID of the window that will be the container for text editing. **(n)**

text_array

A System.Collections.ArrayList array for passing initial text to be inserted when U_EDIT starts, and for returning the final state of text in the window when program control returns from U_EDIT. (System.Collections.ArrayList)

line

(optional) A writable variable whose value determines the line that the cursor will be initially positioned on when U_EDIT starts. When control returns from U_EDIT, *line* returns the line that the cursor was on when last used for input. *Line* defaults to 1. **(n)**

character

(optional) A writable variable whose value determines the cursor's character position in *line* when program control moves to U_EDIT. When control returns from U_EDIT, *character* returns the character position last occupied by the cursor. *Character* defaults to 1. **(n)**

flags

(optional) A writable variable that sets one or more of the following flags and that returns the final state of these flags (except D_EDIT_SELECTALL, which is never returned). By default these are set to off. They are defined in **tools.def** and can be combined using .BOR. or the vertical bar (|) operator: **(n)**

D_EDIT_SELECTALL Initially select all text (Windows only).

D_EDIT_OVERSTRIKE Use overstrike mode.

D_EDIT_BACKWARD Use backward navigation.

D_EDIT_SOFT Insert soft return for ENTER.

col_id1

col_id2

col_id3

(optional) The IDs of menu columns to place or remove. **(n)**

help_id

(optional) A help identifier. **(a)**

decimal_value

(optional) A writable variable whose value specifies an initial character to be input. This value must be passed as a decimal value for an ASCII character (e.g., pass 65 for the letter A). **(n)**

Discussion

The U_EDIT routine enables users to enter and edit text in a window. U_EDIT is similar to T_EDIT in that you control where editing starts, whether insert mode or overstrike mode is used, and the direction for certain menu entries. You can also optionally specify menu columns to load while the routine is processing, a help ID, and an initial character for input. And when control returns from U_EDIT, it returns the line and character position last used for input, so your program can enable users to continue editing where they left off.

However, unlike T_EDIT, the U_EDIT routine does the following;

- ▶ Preserves hard returns, and enables you to specify the kind of return (hard or soft) that is created when the user presses ENTER.
- ▶ Enables you to supply initial text to be displayed when U_EDIT starts (this text can include hard returns), and returns the resulting edited text in a way that preserves the location of hard returns. (You supply initial text in *text_array*, and U_EDIT returns text in *text_array*.)
- ▶ Uses System.Collections.ArrayList arrays.
- ▶ Doesn't require T_SETUP to be called for the window. If T_SETUP has been called, its settings are ignored.

U_EDIT uses the entire visible area of the window (*window_id*) as the editing area. The visible area of a window is usually defined by its display area, but note that on Windows systems this can be affected by altering the container size with the WP_SIZE subfunction of W_PROC. To create an editing area that occupies only part of the window, turn off the window's border and make the window a child of a larger window.

On Windows, if the window is resized while U_EDIT is processing (e.g., if the window has a size event and is therefore resizable by the user, or if some other asynchronous event resizes the window), the editing area is resized accordingly.

The maximum length for text in the editing area is 65,535 characters. On Windows, that includes two characters for every hard return and three characters for every user-inserted soft return. On UNIX and OpenVMS, a user-inserted soft return requires one character, while a hard return doesn't require a character.

If the text supplied in *text_array* exceeds this maximum when control moves to U_EDIT, the text will be truncated to 65,535 characters before it's displayed. Text lost in truncation will not be returned when control returns from U_EDIT.

If a user attempts to insert characters (by typing or pasting) that would cause the maximum length to be exceeded, the insertion is ignored.

If it is not possible to wrap text on a space, the text will be wrapped (with a soft return) at the character that exceeds the available width.

Hard returns are logical characters and can be deleted using any available method for deleting a character. Soft returns are not logical characters. Inserting or deleting text before a soft return (but after the preceding hard return) may remove the soft return. Furthermore, U_EDIT does not preserve soft returns in the text it returns to the calling routine.

The window specified by *window_id* must be the ID of a window registered and placed with Toolkit. Otherwise Toolkit will generate a U_ABORT error.

Note the following for *text_array*:

- ▶ When program control moves to U_EDIT, each element of the ArrayList is converted to a string and appended as a line of text in the editing area, with a hard return between each element. (If you don't want hard returns in the text, pass all text in one element.)
- ▶ When control returns from U_EDIT, all ArrayList elements are removed and replaced by strings that represent the updated state of each line.

Values passed as *line* and *character* set the current cursor location when program control moves to U_EDIT, and these arguments are returned with the updated position when control returns from U_EDIT. If *line* or *character* is not passed or contains a value that's less than 1, the default (1) is used.

Note the following for *line*:

- ▶ *Line* corresponds to an element in *text_array*. When U_EDIT returns, *line* contains the line number (not considering soft returns) where the cursor was last located. Note that because System.Collections.ArrayList arrays are 0-based, *line* is the *text_array* element number for that text plus 1. It's not necessary to take this into consideration when using *line* in U_EDIT calls, but you will need to account for this if you use the ArrayList outside of U_EDIT.
- ▶ If you pass a number that's greater than the number of elements in *text_array*, *line* is assumed to be specifying the last line. (So passing a value greater than 65535 ensures that you're specifying the last line.)

Note the following for *character*:

- ▶ *Character* is the character position within *line*, beginning at 1.
- ▶ If you pass a value that's greater than the length of the specified line, the cursor is positioned after the last character in the line. (So passing a value that's greater than or equal to 65535 ensures that the cursor will be positioned at the end of the line.)
- ▶ When U_EDIT returns, *character* contains the position at which the cursor was last located within a line of text.

Note the following for *flags*:

- ▶ D_EDIT_BACKWARD determines the direction used for the following menu entries: E_LMOV, E_WMOV, E_PDEL, E_LDEL, and E_WDEL. By default, the direction is forward, but if D_EDIT_BACKWARD is set to on, the direction is backward. The E_DIR menu entry enables users to toggle the direction.
- ▶ D_EDIT_SOFT controls the type of line wrap inserted when the user presses ENTER. By default, ENTER inserts a hard return. If D_EDIT_SOFT is set to on, ENTER inserts a soft return. The E_HARD menu entry enables users to toggle this setting.
- ▶ The E_MODE menu entry enables users to toggle insert/overstrike mode. When text is selected on Windows, typing a character replaces the selection regardless of insert/overstrike mode.

If you pass any of the *col_id* arguments while the global variable **g_plc_col_args** is set to true, the specified columns are placed when program control moves to U_EDIT and are removed when control returns from U_EDIT.

On Windows, *col_id* arguments are ignored by default because these arguments are used only when **g_plc_col_args** is set to true, these arguments are ignored by default on Windows. These columns are often unnecessary because they usually contain reserved menu entries that are automatically supported by Windows. Additionally, the excessive placing and removing of columns caused by the *col_id* arguments causes Windows menu bars to “flicker.” If you want to place menu entries, we instead recommend placing them before a call to U_EDIT and removing them after control returns from U_EDIT.

If *decimal_value* is passed as a non-zero value, it is converted to its alpha (ASCII) equivalent and placed in the window as if it were the first character entered by the user (after text supplied by *text_array*). U_EDIT then clears the variable passed as *decimal_value*, which makes it easier to use U_EDIT in a loop.

U_EDIT calls EUTILS_METHOD for menu entries whose names begin with U_, and it calls EHELP_METHOD for the O_HELP menu entry, passing *help_id* as the argument. (If *help_id* is not passed to U_EDIT, no help ID argument is passed to EHELP_METHOD.) If an entry other than these or the entries listed below is called, control is returned to the calling routine, **g_select** is set to true, and **g_entnam** stores the name of the entry.

U_EDIT recognizes the following internal menu entry names on all platforms:

C_NEXT	E_DIR	E_LEFT	E_RIGHT
C_PREV	E_HARD	E_MODE	E_TOP
E_BOTTOM	E_ILIN	E_PDEL	E_WDEL
E_CDEL	E_JOIN	E_PMOV	E_WMOV

On Windows (but not with Synergy .NET), U_EDIT also recognizes the following:

E_CLEAR	E_MARK
E_COPY	E_PASTE
E_CUT	

On UNIX and OpenVMS, U_EDIT also recognizes the following:

E_BMOV	E_LCLR	E_XDOWN
E_DOWN	E_LDEL	E_XUP
E_EDEL	E_LMOV	
E_EMOV	E_UP	

Note the following:

- ▶ The E_ILIN menu entry inserts a hard return regardless of the way D_EDIT_SOFT is set.
- ▶ For the E_LMOV, E_EMOV, and E_LDEL menu entries, a line is considered the text delimited by either hard or soft returns.
- ▶ For the E_PMOV and E_PDEL menu entries, a paragraph is considered a logical line of text (i.e., a string of text ending in a hard return).
- ▶ The E_JOIN menu entry removes the next hard return (the return at or following the current cursor location).

For text windows that are part of a composite window, TAB and SHIFT+TAB move focus to the next or previous child, unless TAB or SHIFT+TAB is already associated with a menu entry on a placed menu column. TAB instructs U_EDIT to signal C_NEXT, and SHIFT+TAB instructs U_EDIT to signal C_PREV.

See also

- ▶ [T_EDIT](#) on page 14-3.
- ▶ [System.Collections.ArrayList](#) in the “System-Supplied Classes” chapter of the *Synergy DBL Language Reference Manual*.
- ▶ [EHELP_METHOD](#) on page 7-73 for more information about help processing.
- ▶ [EUTILS_METHOD](#) on page 7-82 for information about utility processing.
- ▶ “Appendix B: Reserved Menu Entries” for the functions performed by the UI Toolkit reserved menu entries.

Examples

The following example results in a window that accepts text that can include hard and soft returns:

```
import System.Collections      ;U_EDIT uses System.Collections.ArrayList
                                ; for the text_array argument.

main
.include "DBLDIR:windows.def"
.include "WND:tools.def"

record
    text      ,@Arraylist
    wndid     ,int
    line      ,int
    char       ,int
    flags     ,int
    dchr       ,int
proc
    u_start
    .
    .
    .
    u_window(D_PLACE, wndid, 1, 1)
    text = new ArrayList()
    repeat
        begin
            xcall u_edit(wndid, text, line, char, flags,,, "help", dchr)
            if (g_select)
                using g_entnam select
                .
                .
                .
                ( ),      exitloop
            endusing
        end
    u_finish
end
```

U_EDITKEYS – Edit a key map



```
xcall U_EDITKEYS ([help_id])
```

Arguments

help_id

(optional) The help identifier. **(a)**

Discussion

U_EDITKEYS enables the user to create or modify the key map definitions for any terminal type. This subroutine is used by Proto to support the “Key mapping” function.

On Windows, shortcut keys are mapped to a native Windows accelerator table to enable their use. The key mapping function available through Proto or U_EDITKEYS has an effect only on the shortcut text displayed in the menus. The actual keystrokes required for each function code are fixed, and are reflected in the MSWINDOWS key map.

The current key map is initially loaded when you call U_EDITKEYS. Changes made to the current key map take effect after U_EDITKEYS returns.

Refer to [“Customizing Key Mapping for Menu Shortcuts” on page 18-11](#) for information on editing key maps.

This subroutine recognizes the following menu entry names if they are included in a placed menu column.

O_ACCEPT	Accept the shortcut sequence directly.
O_CANCEL	Cancel changes.
O_CHGTERM	Change which terminal key map is being modified.
O_COPYTERM	Copy a terminal key map definition.
O_DELTERM	Delete a terminal key map definition.
O_EXIT	Exit and save all changes.
O_HELP	Call the EHELP_METHOD subroutine and pass <i>help_id</i> .
S_UP	Move to the previous selection window item.
S_DOWN	Move to the next selection window item.
S_LEFT	Move to the selection item in the previous column.
S_RIGHT	Move to the selection item in the next column.

Any other menu entries cause U_EDITKEYS to return with **g_select** set to true and **g_entnam** set to the selected menu entry.

On Windows, U_EDITKEYS cannot time out.

On UNIX and OpenVMS, if a time-out occurs, **g_select** is set to true and **g_entnam** is set to the entry name specified in **tkctl.def** for **g_time_entry**. See [g_time_entry](#) on page 18-6.

Examples

The following subroutine provides the entire “Key mapping” utility included in Proto.

```
subroutine keymap
  .include "WND:tools.def"
  record
    colid    ,i4
  proc
    xcall e_enter                ;Enter new environment
    xcall m_column(D_REMOVE, D_LOCAL) ;Remove all local columns
    xcall m_ldcol(colid, g_utlib, "c_editkeys");Load key editing
  column
    xcall m_defcol(9)            ;Make it default column
    xcall u_editkeys             ;Call edit routine
    xcall e_exit                 ;Exit environment
    xreturn
  endsubroutine
```

U_EDITREND – Create and modify rendition schemes



```
xcall U_EDITREND ([filename], [number], [description], [windows], [areas],
&                [field_numbers][, help_id])
```

Arguments

filename

(optional) The name of the rendition file in which to save the rendition schemes. **(a)**

number

(optional) The number of user renditions to change. The default is 0. **(n)**

description

(optional) The array of user rendition descriptions to load in the “Select Screen Area” selection window. If *description* is not passed and user renditions are to be changed, the default descriptions (“User rendition X”) will be supplied. **(a)**

windows

(optional) The array of window IDs for the user renditions to load in the “Sample Windows” window on UNIX or OpenVMS. Each window will be placed on the screen at its default position. (Ignored on Windows.) **(n)**

areas

(optional) The array of the screen areas within *windows* that will be affected on UNIX or OpenVMS. These areas can be added together for combined effects. If *areas* is 0, then none are affected. The default is all three areas. (Ignored on Windows.) **(n)**

D_RND_AREA Processing area.

D_RND_BRDR Border area.

D_RND_TITLE Title area.

field_numbers

(optional) The array of field numbers within *windows* that will be affected on UNIX or OpenVMS, or **WFN_ALL** to affect all fields. The default is none. (Ignored on Windows.) **(n)**

help_id

(optional) The help identifier. **(a)**

Discussion

The U_EDITREND subroutine enables the user to create or modify rendition schemes and save them to a rendition file.

U_EDITREND affects the default and user renditions referenced in the U_REND subroutine section. (These renditions, as listed in the **tools.def** file, are D_RND_MNU_ENA through D_RND_USER_5).



On Windows, U_EDITREND cannot set the Input Field Processing rendition.

On all platforms, U_EDITREND cannot set Disabled Input Field, Read-only Input Field Display, or Read-Only Field Processing. (These can be set only by U_REND and only on UNIX and OpenVMS.)

This subroutine is used by Proto to support the task of editing renditions. See [“Using the Renditions window \(Windows\)” on page 18-36](#) for more information.

On Windows, U_EDITREND invokes the Renditions Control where users can graphically modify and maintain window system renditions. See [“Customizing the Look of Your Application” on page 18-24](#) for more information on editing renditions in a Windows environment.

If *filename* doesn't exist, it is created. If *filename* is not specified, UI Toolkit searches for the rendition file as follows:

- ▶ If the environment variable DTKRNDFIL is defined, the rendition filename is the value of DTKRNDFIL.
- ▶ If DTKRND is defined instead of DTKRNDFIL, the rendition file is opened as **DTKRND:dtkrnd.ism**.
- ▶ If neither environment variable is defined, **dtkrnd.ism** is opened in the current directory.

If *windows* is not specified and user renditions are used, default sample windows are created. All created windows are deleted upon exit.

U_EDITREND recognizes the following menu entry names if they are included in a placed menu column:

O_CANCEL	Cancel changes to area with which you are working.
O_EXIT	Exit and save all changes.
O_HELP	Call the EHELP_METHOD subroutine and pass <i>help_id</i> .
O_DEFAULT	Set renditions to their default values.
O_LDSCHEME	Load rendition schemes from the rendition file.
O_PALETTE	Modify a color palette.
O_RESET	Reset to the working rendition scheme that was saved on entry.

O_RMScheme	Remove a rendition scheme from the rendition file.
O_STScheme	Store rendition schemes to the rendition file.
S_UP	Move to the previous selection window item.
S_DOWN	Move to the next selection window item.
S_LEFT	Move to the selection item in the previous column.
S_RIGHT	Move to the selection item in the next column.

Any other menu entries cause this subroutine to return with **g_select** set to true and **g_entnam** set to the selected menu entry.

O_EXIT causes U_EDITREND to return to the calling program, but it does not clear **g_select** and **g_entnam**. This enables a layered calling routine to continue a seamless exit.

This subroutine always stores UI Toolkit's current renditions to the rendition file as the working rendition settings upon both entry and exit of the subroutine.

See also

[“Customizing the Look of Your Application” on page 18-24.](#)

Examples

The following subroutine provides the entire “Renditions” utility included in Proto.

```
subroutine editrnd
.include "WND:tools.def"
record
    colid ,i4
proc
    xcall e_enter                ;Enter new environment
    xcall m_column(D_REMOVE, D_LOCAL) ;Remove all local columns
    xcall m_ldcol(colid, g_utlib, "c_editrend") ;Load edit column
    xcall m_defcol(9)            ;Make it the default column
    xcall u_editrend            ;Edit renditions
    xcall e_exit                ;Exit environment
    xreturn
endsubroutine
```


%U_ENUMWNDS – Return a list of windows in a library

WT	WN	U	V

```
nmwnds=%U_ENUMWNDS (channel, [max][, array])
```

Return value

nmwnds

The number of windows in the window library specified by *channel*. (^VAL)

Arguments

channel

A window library channel. (**n**)

max

(optional) Maximum number of window names to return. (**n**)

array

(optional) The first element in an array for returned names. (**a**)

Discussion

The %U_ENUMWNDS function returns the number of windows in a window library and optionally returns the window names.

Channel must be an open file channel for a window library. If it isn't open, Toolkit will generate a runtime error ("Channel has not been opened"), and if the file that *channel* is open on is not a window library, unpredictable behavior will result.

If *max* is less than the number returned in *nmwnds*, not all window names will be returned in *array*. If *max* is not passed or is less than 1, *array* is ignored.

If *max* is passed and is greater than zero, *array* must be the first element of an array (pseudo or real) of at least *max* elements.

If any read failure occurs (other than EOF), Toolkit generates a trappable runtime error.

Examples

The following example returns (in **num_windows**) the number of windows in the specified channel (**chn**), and it returns the names of the windows in **names**.

```
num_windows = %u_enumwnds(chn, max_names, names)
```

U_FINISH – Terminate the UI Toolkit environment



```
xcall U_FINISH ([D_NOCLR]/, cancel])
```

Arguments

D_NOCLR

(optional) Don't clear the screen. By default, the screen is cleared when Toolkit processing is terminated. **(n)**

cancel

(optional) Returned true if the close method subroutine cancels the shutdown. **(n)**

Discussion

U_FINISH shuts down the UI Toolkit environment. It purges all channels U_START or U_OPEN opened in output mode except the terminal (if it was opened by U_START). U_FINISH is called by your application. It is also called internally by UI Toolkit.

If you have used E_METHOD to register an ECLOSE_METHOD with the environment, U_FINISH calls that ECLOSE_METHOD to perform any additional processing before shutting down the application.

If you pass *cancel*, it is passed on to your ECLOSE_METHOD. If you want your ECLOSE_METHOD to cancel termination, have it set *a_cancel* to true, and it will get passed back to your application through *cancel*.

If you do not pass *cancel*, your ECLOSE_METHOD will not have the opportunity to cancel the shutdown.

U_FINISH's default is to clear the screen when Toolkit processing is terminated. If you pass D_NOCLR, the screen will not be cleared.

On Windows, if the application has previously called U_START and interrupt is enabled, the user can also close the application through the System-menu box or with the interrupt key. In both cases, UI Toolkit will internally call U_FINISH.

See also

[E_METHOD on page 7-35](#) and [ECLOSE_METHOD on page 7-59](#) for more information about environment-level events (such as shutting down the application).

Examples

The following example terminates Toolkit processing without clearing the screen.

```
xcall u_finish(D_NOCLR)
```

U_FLASH – Flash a message window



```
xcall U_FLASH ([channel], window_name, [window_id][, flag])
```

Arguments

channel

(optional) The channel of the window library containing the window to flash. **(n)**

window_name

The name of the window to load from the window library. **(a)**

window_id

(optional) Returned with the window ID. **(n)**

flag

(optional) A flag returned non-zero if the specified window is not found. **(n)**

Discussion

The U_FLASH subroutine flashes a message window on the screen.

The window will be loaded (if it's not already) and placed. The current message window renditions will be applied to the window. The screen will be updated in order to make the window visible. The window will then be removed upon the next screen update.

The window will be deleted when the current environment is exited, or it may be explicitly deleted with the U_WINDOW subroutine.

This subroutine is ideal for handling “Please wait” type messages because any input or other visible change to the screen will remove the message. However, if *window_id* is passed, the calling routine can place or remove the window at will after this subroutine is called.

If *channel* is not passed, **g_utlib** will be used.

Examples

This example will try to load the window named “wait” from the window library opened on **g_utlib**. If the window is found, **nofind** will be returned as zero, the *window_id* will be returned in **wndid**, and the window will be displayed on the screen until the next screen update occurs.

```
xcall u_flash(g_utlib, "wait", wndid, nofind)
```

U_FLD – Accept one field from the terminal

WT	WN	U	V

```
xcall U_FLD (data_area, [window_id], [row], column | field#, [paint], [length],
&          [begin_position], [mode], [help_id][, wait_time])
```

Arguments

data_area

(required if *row* is passed) Returned with the set of characters that was input. **(a)**

window_id

(optional) The ID of the window (created by U_LDWND) from which the input will come. If not passed, the current input window is used. **(n)**

row

(optional) The window row from which this subroutine will be reading. **(n)**

column or
field#

The column (if *row* is also specified) from which to read or the field number (if *row* is not specified) at which to do the input. **(n)**

paint

(optional) The paint character. **(a)**

length

(optional) The initial data length and is returned with the final data length on exit. **(n)**

begin_position

(optional) The initial field position and is returned with the final input position on exit. **(n)**

mode

(optional) An insert/overstrike mode variable. (Ignored on Windows.) **(n)**

0 Overstrike. (default)

1 Insert.

help_id

(optional) The help identifier. **(a)**

wait_time

(optional) The time-out limit for I/O processing to be completed before returning to the calling routine. (**n**)

D_FOREVER or **-1** Never time out.

D_GLOBAL or **-2** Use the global value (see [g_wait_time](#) on page 18-6). (default)

D_IMMEDIATE or **0** Time out immediately.

n Wait up to *n* seconds (where *n* is a positive value) for I/O processing to be complete.

Discussion

The U_FLD subroutine accepts a set of characters from the terminal.

U_FLD is designed to perform field input in a non-input window. You cannot use it with an input window (one defined by the .INPUT script command or by the IB_INPUT subroutine). See [I_INPFLL](#) on page 8-51 and [I_INPUT](#) on page 8-55 for information about performing input in an input window.

This subroutine allows menu or shortcut processing.

If you want input performed at a specific row and column within the window, pass a non-zero value to *row* and the column number as the next argument (which UI Toolkit will assume to be *column*). Together these two arguments define the position within the window at which to do the input. This is known as a “position-mode call.”

When processing a position-mode call, the *data_area* argument is required and specifies the maximum length of the input. If no field set is defined, UI Toolkit will create and use one with a single field.

If you want input performed on a field number, omit *row* and pass the window field number as the following required argument. (When *row* is not passed, UI Toolkit assumes this next argument to be the *field#*). This is known as a “field-mode call.”

When processing a field-mode call, the *data_area* field is optional. If *data_area* is not present, it is the responsibility of the calling routine to ensure that the specified field’s contents match the passed parameters and to retrieve the input data from the window field upon completion of the call. The maximum width of the input is determined by the length of the specified window field, regardless of the size of any *data_area* that is specified.

If *length* is present, it is the number of data characters in the field upon entry and exit. The initial data length in the field will be *length*’s entering value. On exit, *length* will be loaded with the final data length. If *length* is not specified, the entering data length is assumed to be 0.

If the entering data length is non-zero, the number of characters from *data_area* are placed in the leftmost part of the field before the initial input.

If *begin_position* is present, it is the position of input within the field upon entry and exit. The initial input will begin at this character position (a value of 1 means the input position is at the field's first character, 2 means it's at the second character, and so forth). On exit, *begin_position* will be loaded with the final input position.

If *begin_position* is not present or has a value of zero, the initial position will be the initial data length plus one.

If *mode* is present and non-zero, the input will be performed in insert mode. Any characters at and to the right of the initial input position (as determined by the entering data length and the initial position) will be “pushed around” as data is entered and deleted during input.

On UNIX and OpenVMS, if *mode* is not present or has a value of zero, input will be performed in overstrike mode. Characters to the right of the input position will be replaced by entered characters, while characters to the left will be replaced by the paint character on deletes.

On Windows, U_FLD recognizes the following menu entries, which are described in “[Appendix B: Reserved Menu Entries](#)”: E_CLEAR, E_COPY, E_CUT, E_MARK, E_PASTE.

If **g_select** is set on return, a menu entry has been selected, and **g_entnam** contains the name of the selected entry. If the selected menu entry begins with “U_” the EUTILS_METHOD subroutine is called and the input is redone. If the selected menu entry is “O_HELP,” the EHELP_METHOD subroutine is called, with the *help_id* as its argument.

You will need to clear **g_select** after you satisfy the event.

The field **gi_sts** will be loaded with the type of the input. If the input is NORMAL_INPUT (zero), the input was terminated normally by a ENTER. If **gi_sts** is non-zero, the input was terminated by a special entry.

If the operation times out, **g_select** is set to true and **g_entnam** set to the entry name specified in **g_time_entry**, which is defined in **tkctl.def**. See [g_time_entry on page 18-6](#).

Examples

The first example returns the set of characters that was input to the variable **custnam**. This input comes from the 4th field of the window whose ID is stored in **wndid[i]**.

```
xcall u_fld(custnam, wndid[i], , 4)
```

The second example also returns the characters that were input to **custnam**. Input comes from the current input window at position (15,30). The routine passes the identifier “cushelp” to the EHELP_METHOD subroutine.

```
xcall u_fld(custnam, , 15, 30, , , , "cushelp")
```

U_FMTDAT – Format a date value into an alpha date string



```
xcall U_FMTDAT (type, date, [year], [month | julian_day], [day], [D_LEFT], [order]
&             [, separator])
```

Arguments

type

The code for the type of date format: **(n)**

Code	Default display	Storage format
1	PP/YY	YYPP
2	MM/DD/YY	YYJJJ
3	MM/DD/YY	YYMMDD
4	PP/YYYY	YYYYPP
5	MM/DD/YYYY	YYYYJJJ
6	MM/DD/YYYY	YYYYMMDD

date

Returned with the formatted date. **(a)**

year

(optional) The year of the date or the entire date in storage format. **(n)**

month or

julian_day

(optional) The month of the date if *type* 1, 3, 4, or 6 is specified. The Julian day if *type* 2 or 5 is specified. **(n)**

day

(optional) The day of the date if *type* is 1, 3, 4, or 6 and if month is passed. *Types* 2 and 5 ignore *day*. **(n)**

D_LEFT

(optional) The left-justification flag. **(n)**

order

(optional) The code for the date order, applicable for *types* 2, 3, 5, and 6: **(n)**

0 Month, day, year (default)

1 Day, month, year

2 Year, month, day

separator

(optional) The separator between parts of the date (e.g., a slash between day, month, and year). **(a)**

Discussion

The U_FMTDAT subroutine formats separate date values into an alpha date string.

If *type* is **2** or **5** (YYJJJ or YYYYJJJ) and you omit *year* but pass *julian_day*, U_FMTDAT will use the day passed and default to use the current year. If *year* is passed but *julian_day* is omitted, U_FMTDAT assumes *year* contains the entire date in storage format. If neither argument is passed, U_FMTDAT defaults to use the current system date.

If *type* is **1**, **3**, **4**, or **6** and if *year* is not passed, *month* and *day* will be ignored and U_FMTDAT will default to use the current system date. (If you omit all three arguments, U_FMTDAT will also default to use the current date.) If *year* is passed but *month* is not, *day* is ignored and *year* is assumed to be the entire date in storage format.

The returned format for *types* **2**, **3**, **5**, and **6** is further controlled by the value of *order*, which determines the order of their date components. The default for *order* is the value in **g_date_order** (defined in **tkctl.def**).

The separator for all format types is loaded from *separator*. The default for *separator* is the value in **g_date_sep**, which is defined in **tkctl.def** and initialized by U_START to a forward slash (/).

If *type* is **5** or **6**, and the length of *date* is exactly 8, a two-digit year will be displayed.

See also

See the .FIELD qualifiers [DATE](#), [NODATE](#) on page 2-42 for more information about date processing.

Examples

In the following example, the date (returned in **date**) is left-justified and formatted as *DD-MM-YY*.

```
xcall u_fmtdat(2, date, yr, mn, dy, D_LEFT, 1, "-")
```

In the next example, the date is returned in **date**. **stor** contains the entire date in storage format in the form *YYYYJJJ*.

```
xcall u_fmtdat(5, date, stor)
```


U_FMTTIM – Format a time value into an alpha time string

WT	WN	U	V

```
xcall U_FMTTIM (type, time, [hour], [minute], [second], [D_LEFT][, separator])
```

Arguments

type

The code for the type of time format. **(n)**

Code	Display	Storage format
1	HH:MM	HHMM
2	HH:MM am/pm	HHMM
3	HH:MM:SS	HHMMSS
4	HH:MM:SS am/pm	HHMMSS

time

Returned with the formatted time. **(a)**

hour

(optional) The hour of the time or the entire time in storage format. If not passed, the current system time is assumed. **(n)**

minute

(optional) The minute of the time. **(n)**

second

(optional) The second of the time. **(n)**

D_LEFT

(optional) The left-justification flag. **(n)**

separator

(optional) The separator between parts of the time (for example, a colon between hour, minute, and second). **(a)**

Discussion

The U_FMTTIM subroutine formats separate time values into an alpha time string.

If *hour* is passed but *minute* is not passed, *second* is ignored and *hour* is assumed to be the entire time in storage format.

The separator for all the format types will be loaded from *separator*. The default for *separator* is the value in **g_time_sep**, which is defined in **tkctl.def** and initialized by U_START to a colon (:).

The meridian indicators are the text defined in the UI Toolkit message file as facility DTK, mnemonics “AM” and “PM,” respectively. The actual text is “AM” and “PM,” but can be modified.

Examples

In the following example, the time (returned in **time**) is left-justified and formatted as HH:MM AM/PM.

```
xcall u_fmttim(2, time, hr, mn,, D_LEFT)
```

In the second example, the time is returned in **time**, formatted as HH:MM:SS AM/PM. **Stor** contains the entire time in storage format in the form HHMMSS.

```
xcall u_fmttim(4, time, stor)
```

U_FNDFIL – Search for a file



```
xcall U_FNDFIL([filename], [mode], [channel], [base_channel][, I/O_mode])
```

Arguments

filename

(optional) The name of the file to find. **(a)**

mode

(optional) The mode in which the file is opened. **(a)**

channel

(optional) Returned with the channel on which the file is opened. **(n)**

base_channel

(optional) The base channel examined in the search for a file. **(n)**

I/O_mode

(optional) Returned with the mode of the found file. **(a)**

Discussion

The U_FNDFIL subroutine searches for a file and determines if it is already open. It enables you to search for a file yourself and is most useful when you are converting existing applications.

U_OPEN can perform this file searching automatically.

If *filename* is omitted or blank, U_FNDFIL searches for the file specified in the last U_FNDFIL call. If there's no previous call, U_FNDFIL returns a value of 0 for *channel* and returns *I/O_mode* as blank.

If *mode* is omitted or blank, the first channel on which the file is opened, regardless of *I/O_mode*, is returned.

Channel is returned with a value of 0 if the file or the mode is not found. If channel is omitted, U_FNDFIL will not perform a search.

If you specify *base_channel*, all channels less than or equal to *base_channel* will not be examined for the *filename*. If you do not specify *mode*, U_FNDFIL will find the first channel after *base_channel*. You can thus create a loop, using a *base_channel* specification to get past the previous channel that was examined until the file is found.

Examples

The first example looks for a file “cusmas” that was opened in mode “I:I” and returns the channel on which the file is opened in variable **cuschn**.

```
xcall u_fndfil("cusmas", "i:i", cuschn)
```

The second example also looks for “cusmas” and returns the channel on which the file is opened in **cuschn**, but it also returns the I/O mode of the file in variable **iomode**.

```
xcall u_fndfil("cusmas",, cuschn,, iomode)
```

The third example looks for the file named “cusmas” opened in mode “I:I,” ignoring channels 1 through 3 and beginning with channel 4. It returns the channel on which the file is opened in **cuschn**.

```
xcall u_fndfil("cusmas", "i:i", cuschn, 3)
```

U_GBLCHN – Give channel global status



```
xcall U_GBLCHN(channel)
```

Arguments

channel

The channel to be promoted. (**n**)

Discussion

The U_GBLCHN subroutine promotes a local channel to global status.

A global channel stays open even after the environment that opened it is exited.

If *channel* is local, but not local to the current environment, a fatal error occurs. If *channel* is already global, this subroutine is ignored.

Examples

In the following example, the variable **archn** is returned with the allocated channel, and the U_GBLCHN subroutine promotes the channel stored in **archn** to global status.

```
xcall u_open(archn, "i:i", "aropen")
xcall u_gblchn(archn)
```

U_GBLWND – Give window global status



```
xcall U_GBLWND (window_id)
```

Arguments

window_id

The ID of the window to be promoted. (**n**)

Discussion

The U_GBLWND subroutine promotes a local window to global status.

A global window is not deleted even when the environment that created it is exited.

If *window_id* is not local to the current environment, a fatal error occurs. If *window_id* is already global, this subroutine is ignored.

When promoting an input window to global status, any selection windows associated with the SELECT qualifier of the .FIELD command will also be promoted.

However, if an associated selection window was linked to an input field using the SELWND qualifier of the .FIELD command or the I_FLDSEL subroutine, UI Toolkit cannot guarantee that the associated selection window was created in the current environment. In this situation, you must promote the selection window yourself.

Examples

In the following example, the variable **wndid** is returned with the allocated channel, and the U_GBLWND subroutine promotes the window whose ID is stored in **wndid** to global status.

```
xcall u_ldwnd(wndid, g_utlib, "cusmas")  
xcall u_gblwnd(wndid)
```

U_GETCHN – Obtain a channel



```
xcall U_GETCHN (allocated_channel, [D_GLOBAL][, requested_channel])
```

Arguments

allocated_channel

Returned with the allocated channel. (n)

D_GLOBAL

(optional) Allocate channel as global. (n)

requested_channel

(optional) The channel number requested. (n)

Discussion

The U_GETCHN subroutine obtains a channel for your own use.

If D_GLOBAL is present, the channel is allocated as global rather than local (meaning that it has to be explicitly purged rather than being purged implicitly upon exit from the current environment). U_FINISH will purge the global channels.

If you do not specify *requested_channel*, the channel returned in *allocated_channel* is the first channel available. *Allocated_channel* is returned as 0 if *requested_channel* is not available for usage. If *requested_channel* is specified and available, it is used.

You can use this subroutine in conjunction with specifying U_OPEN's D_NOALLOC flag. Its primary use is in converting existing applications. If you allocate channels ahead of time and make them global, your existing logic will work.

We recommend that you use U_GETCHN only when converting your applications to use UI Toolkit. Channel allocation is best handled by UI Toolkit's U_OPEN subroutine and this subroutine can help you coincide your application's current channel usage with U_OPEN's future channel allocation. During conversion, you can request a certain channel number (with *requested_channel*) and then have U_OPEN use *that* channel rather than allocating a new one.

After conversion, we recommend that you don't use U_GETCHN to explicitly allocate channels. Allowing U_OPEN to maintain channel usage will help you keep your application modular.

See also

[U_RLSCHN](#) on page 16-124 for information about releasing channels allocated with U_GETCHN.

Examples

The following example allocates channel 15 and logs it as a global channel.

```
xcall u_getchn(chn, D_GLOBAL, 15)
```

The second example gets the first available channel as a local channel.

```
xcall u_getchn(chn)
```


%U_GETFILENAME – Prompt user for filename



```
status = %U_GETFILENAME (filename, title, [#filters], [filter_desc], [filters], [default_name],
&      [extension], [input_column], [select_column], [list_column], [help_id][, option, ...])
```

Return value

status

On Windows, the filter index (1-based) that is returned by the Microsoft Open or Save As dialog box, or false if the Cancel button is pressed on the dialog box or if the dialog box returns an error.

On UNIX and OpenVMS, true if a filename is selected, or false if no filename is selected or an error occurred. (^VAL)

Arguments

filename

Returned with the entered or selected filename. (a)

title

The title for the input window or dialog box. (a)

#filters

(optional) The number of file filters for browsing. (n)

filter_desc

(required if *#filters* passed) The array of filter descriptions. (a)

filters

(required if *#filters* passed) The array of filters. (a)

default_name

(optional) The default filename. (a)

extension

(optional) The default extension. (a)

input_column

(optional) The ID of the input column. (UNIX, OpenVMS only) (n)

select_column

(optional) The ID of the selection column. (UNIX, OpenVMS only) (n)

list_column

(optional) The ID of the list column. (UNIX, OpenVMS only) **(n)**

help_id

(optional) The help identifier. (UNIX, OpenVMS only) **(a)**

option

(optional) One or more of the following flags for Windows platforms. For traditional Synergy, you can pass more than one option by combining them with .BOR. or the vertical bar (|) operator. For Synergy .NET, only D_OFN_SAVEAS is available. (Windows only) **(n)**

D_OFN_CENTER Center dialog box over application window.

D_OFN_NONETWORK No Network button.

D_OFN_SAVEAS Save as, instead of Open.

D_OFN_NOTRANS Don't translate logicals.

Discussion

The %U_GETFILENAME function pops up a dialog box or window that enables the user to select a file either by entering a filename in a field or by browsing and then selecting a filename from a list. Note that to actually open the selected file, your program must call U_OPEN.

If the user selects a filename, **g_select** is set to false. Otherwise, **g_select** is set as follows:

- ▶ If the user pressed the “Cancel” button, **g_select** will be false. (Windows only)
- ▶ If the user selected the “O_EXIT” or “O_ABANDON” menu entry, **g_select** will be false.
- ▶ If the user selected some other menu entry that is not recognized by %U_GETFILENAME, **g_select** will be true.

On Windows, %U_GETFILENAME calls the standard file “Open” or “Save as” dialog box, and users can change directories as they browse.

On UNIX and OpenVMS, %U_GETFILENAME goes through two or three stages:

1. It opens a window with the “File” input field. The user can either enter a filename into this field or “browse” files. To browse, the user presses ENTER on the empty field or selects the menu entry “O_BROWSE.” (You must include “O_BROWSE” on one of your menus to make it available.) The browse is limited to the initial directory.

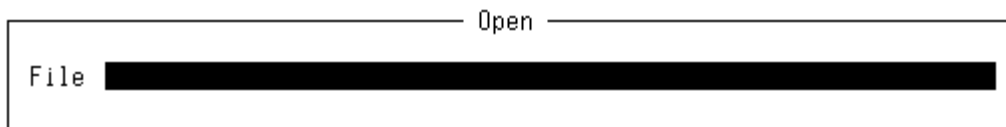


Figure 16-3. Stage 1.

2. (optional) If you passed the filter arguments (*#filters*, *filter_desc*, and *filters*) and if *#filters* is greater than one, when the user requests a browse, %U_GETFILENAME will display a filter description box. The user can select one of the filters, or select either the “O_EXIT” or “O_ABANDON” menu entry to return to the “File” input field. (These menu entries must be included on one of your menus.)

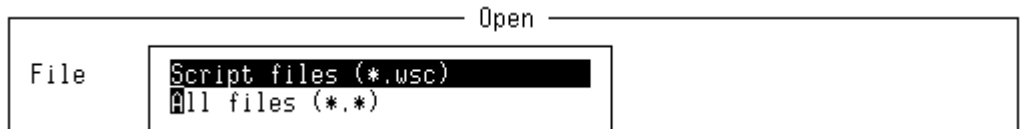


Figure 16-4. Stage 2.

3. %U_GETFILENAME removes the filter description box (if displayed) and displays a file browse box containing the filenames associated with the selected filter. The title of the browse box is the filter description. The user can select a file, or select “O_EXIT” or “O_ABANDON” to go back to the previous input point (which would be either the filter description box or the “File” input field).

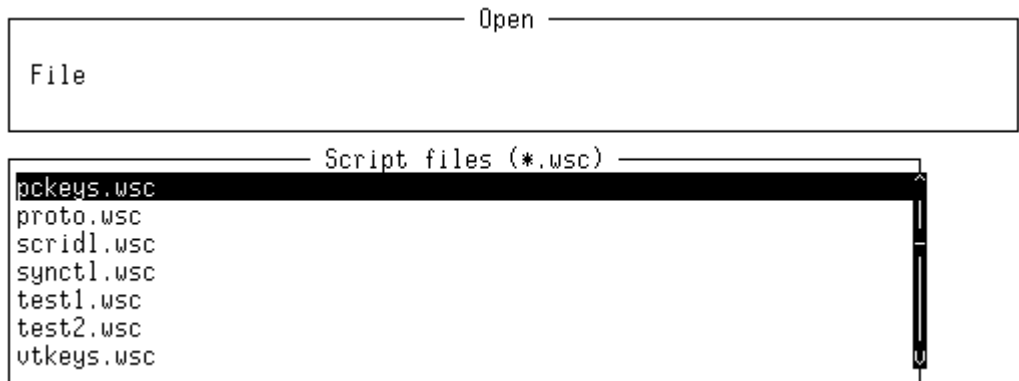


Figure 16-5. Stage 3.

On UNIX and OpenVMS, %U_GETFILENAME uses *I_INPUT* to process the “File” input field, *S_SELECT* to process filter selections, and *L_SELECT* for the file browser box. So you can use reserved menu entries for these routines when %U_GETFILENAME is using them. For example, you can use the reserved menu entries for *S_SELECT* while your program processes filter selections. To use reserved menu entries for these routines, pass *input_column*, *select_column*, or *list_column* with the name of the menu column that contains the entries.

%U_GETFILENAME does not time out on Windows. If a time-out occurs on UNIX and OpenVMS, *g_select* is set to true and *g_entnam* is set to the entry name specified by *g_time_entry* (see *g_time_entry* on page 18-6).

filename

Filename is returned with the name of the selected file—whether entered directly into the “File (Name)” field or selected from the file browse box. If the user entered the filename into the “File (Name)” field, the returned *filename* also will contain any path specification explicitly typed. Native path specification or standard Synergy DBL logical translation (in the form of *LOGICAL:filename*) is supported.

If the variable passed as *filename* is not big enough to hold the full path specification, %U_GETFILENAME returns false, and *filename* will be blank. We recommend a variable size of 256.

On Windows, if the user enters a filename but no path, the current path of the dialog box will be prefixed to the returned *filename*. When the user first enters the dialog box, the initial path will be the path contained in *default_name*. If no path is specified in *default_name* or if *default_name* is not passed, the path will be the current working directory. (Remember, however, that on Windows the user may change the directory of the dialog box.)

On UNIX and OpenVMS, if the user doesn’t include a path when entering the filename into the “File” field, any path contained in *default_name* will be prefixed to the returned *filename*.

title

On Windows, *title* becomes the title for the “Open” or “Save as” dialog box.

On UNIX and OpenVMS, *title* becomes the title for the input window containing the “File” input field.

#filters, filter_desc, and filters

If *#filters* is passed, the *filter_desc* and *filters* arguments are required. If you do not pass the *#filters* argument, *filter_desc* and *filters* are ignored if they are passed.

Filter_desc must be an alphanumeric array of at least *#filters* elements. Each element must contain the description for one of the *filters*.

Filters must also be an alphanumeric array of at least *#filters* elements. The elements in the *filters* array contain the filters themselves. Be sure to use the appropriate form for your application’s system when specifying the elements in filters. For example, if an application generates script files, one of the filter elements might be *.wsc.

If *#filters* is not passed, the file browse box will contain all files and the filter description box will contain only the “All files (*.*)” filter.

On UNIX and OpenVMS, the file browse box is built as a list with the filter description as the title.

default_name

If *default_name* is passed, *filename* will be returned with the contents of *default_name* if the user doesn't select a filename.

If *default_name* is not passed and the user doesn't select a filename, the contents of *filename* will be blank.

Default_name can include a complete path name (directory and filename). If no path is specified, %U_GETFILENAME searches for the default filename in the current working directory. You can use native path specification or standard Synergy DBL logical translation (in the form of *LOGICAL:filename*) as the path specification in *default_name*. See “filename” above for ramifications of including a path name.

extension

If *extension* is passed, when the user enters a filename without an extension into the “File (Name)” field, %U_GETFILENAME will append a period (“.”) followed by *extension* to the returned *filename*. If the user selects a file from the file browse box, *extension* is ignored.

input_column, select_column, and list_column (UNIX and OpenVMS)

If *input_column* is passed, that column is placed before and removed after performing input to the “File (Name)” field.

If *select_column* is passed, that column is placed before and removed after processing the filter description box.

If *list_column* is passed, that column is placed before and removed after processing the file browse box.

help_id (UNIX and OpenVMS)

If you pass *help_id*, %U_GETFILENAME will pass it to the current input routine. The current input routine will be

- ▶ I_INPUT if the “File (Name)” field is active.
- ▶ S_SELECT if the filter description box is active.
- ▶ L_SELECT if the file browse box is active.

option (Windows)

D_OFN_CENTER centers the dialog box over the application window, which matches behavior on UNIX and OpenVMS. Default positioning on Windows is determined by Windows Common Dialogs and is generally in the upper left-hand corner of the screen.

D_OFN_NONETWORK suppresses the inclusion of the “Network” button in the dialog box. This button is included by default. If included, it enables the user to browse other file systems connected to the current system.

Utility Routines

%U_GETFILENAME

D_OFN_SAVEAS indicates that the standard “Save as” dialog box should be used instead of the “Open” dialog box. %U_GETFILENAME uses the “Open” dialog box by default.

D_OFN_NOTRANS indicates that any logicals entered by the user should not be translated in the filename that is returned.

U_GETTXT – Get text from the text library



```
xcall U_GETTXT (facility, text_number, text[, text_length])
```

Arguments

facility

The name of the facility to choose from in the text message library. **(a3)**

text_number

The text message number. **(n)**

text

Returned with the message text. **(a)**

text_length

(optional) Returned with the length of the text. **(n)**

Discussion

The U_GETTXT subroutine gets a line of text from the text message library.

The text message library is opened by U_START. It contains all Toolkit error and text messages, as well as any customized messages you add using the Synergy UI Toolkit Control Panel.

The facility name for Toolkit text messages is DTK. If you've created your own facility in the text message library, use your own unique *facility* name.

The maximum size of Toolkit messages is 200 characters.

See also

See [“The Synergy UI Toolkit Control Panel”](#) in the “General Utilities” chapter of *Synergy Tools* for information on determining the text message file in use and for information on using the Synergy UI Toolkit Control Panel for customizing text messages.

Examples

The following example uses the UI Toolkit error “A window with that name already exists” as its own error message.

```
subroutine create_window
    a_name    , a    ; Window name to create

    .include "WND:tools.def"
    .include "WND:dtkmsg.def"
```

Utility Routines

U_GETTXT

```
record
    sts      ,i4
    len      ,i4
    text     ,a200
    .
    .
    .
proc
    call chk_if_exists      ; If it exists, sts set to 1
    if (sts)
        begin
            xcall u_gettxt("DTK", DTK_DUPNAME, text, len) ;DTK_DUPNAME
            xcall u_message(text(1:len))                  ; is defined in dtkmsg.def
            xreturn
        end
    .
    .
    .
endsubroutine
```


%U_GETWNDERR – Get a window system error



```
window_error = %U_GETWNDERR ([text])
```

or

```
xcall U_GETWNDERR ([text])
```

Return value

window_error

The window system error. (^VAL)

Arguments

text

(optional) Returned with the error message text. (a)

Discussion

The %U_GETWNDERR function returns the error number of the last window error returned as a result of Toolkit calling a Synergy windowing API routine (which it often does when you call a Toolkit routine). For information on window errors, see “[Window error messages](#)” in the “Error Messages” chapter of *Synergy Tools*.

If a Toolkit routine returns a Toolkit error, %U_GETWNDERR returns the accompanying window error number (not the Toolkit error number) in *window_error*. (The routine that returned the error returns the number for the Toolkit error.) For example, if you pass the error argument for L_CLASS and Toolkit is unable to create the list class because one already exists with the name you specified, Toolkit returns the Toolkit error number (519) in the *error* argument for L_CLASS, and it returns the window error number (14) in *window_error*.

If you pass *text*, %U_GETWNDERR also returns the text of the window or Toolkit error.

If no Toolkit error has been returned and no window error was returned by the last windowing subroutine call within Toolkit, *window_error* is returned with 0 and *text* is returned blank. Note, however, that this function should only be called immediately after a Toolkit routine has signaled that a window error has been trapped. Otherwise, errors that are part of Toolkit’s normal processing may be erroneously returned, or subsequent Toolkit calls to Synergy window API routine may overwrite the error information.

See also

See “[Window error messages](#)” in the “Error Messages” chapter of *Synergy Tools*, and see [L_CLASS](#) on page 9-68, [L_CREATE](#) on page 9-75, and [I_LDINP](#) on page 8-62 for Toolkit error messages.

U_HELPREND – Use help window renditions



```
xcall U_HELPREND (window_id)
```

Arguments

window_id

The ID of the window. (**n**)

Discussion

The U_HELPREND subroutine sets the renditions of the window specified by *window_id* to the help window renditions. The default help window renditions can be set in U_REND using the following rendition codes.

D_RND_HLP_BCK Help window background.

D_RND_HLP_BDR Help window border.

D_RND_HLP_TTL Help window title.

You also can use U_EDITREND or Proto to set these renditions.

You can use U_HELPREND within your EHELP_METHOD subroutine to change the window's renditions before it is placed on the screen.

See also

- ▶ [U_EDITREND on page 16-42.](#)
- ▶ [“Proto” on page 3-1.](#)

Examples

```
subroutine my_help_method
    a_help_id      ,a      ;optional help identifier

    .include "WND:tools.def"

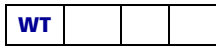
record
    colid          ,i4      ;Text editing column id
    wndid          ,i4      ;Window id
    srch           ,i4

proc
    xcall e_enter
    if (%passed(a_help_id)) then
```

```
        xcall u_ldwnd(wndid, g_utlib, a_help_id,, srch)
    else
        xcall u_ldwnd(wndid, g_utlib, "h_general",, srch)

    xcall t_setup(wndid)
    xcall u_helprend(wndid)
    xcall m_ldcol(colid, g_utlib, "c_text",, srch)
    xcall t_view(wndid)
    xcall e_exit
    clear g_select
    xreturn
endsubroutine
```

%U_HTMLHELP – Invoke HTML Help



```
result = %U_HTMLHELP (subfunction[, arg, ...])
```

or

```
xcall U_HTMLHELP (subfunction[, arg, ...])
```

Return value

result

On Windows, a value specific to the subfunction. (For most subfunctions, this is the handle for the outermost Help window.) On UNIX and OpenVMS, always returns 0. (^VAL)

Arguments

subfunction

The operation to perform (defined in **tools.def**):

D_HH_ALINK_LOOKUP	Look up topics for specified Alink names.
D_HH_CLOSE_ALL	Close all Help windows associated with application.
D_HH_DISPLAY_INDEX	Display the Index tab.
D_HH_DISPLAY_SEARCH	Display the Search tab.
D_HH_DISPLAY_TEXT_POPUP	Display text in a pop-up Help window.
D_HH_DISPLAY_TOC	Display the Contents tab.
D_HH_DISPLAY_TOPIC	Open a Help topic.
D_HH_GET_WIN_HANDLE	Return the handle for the window created for a specified window type.
D_HH_HELP_CONTEXT	Display the Help topic for a given context ID.
D_HH_KEYWORD_LOOKUP	Look up topics for specified keywords.

arg

(optional) One or more arguments if required for the subfunction. (**a** or **n**)

Discussion

%U_HTMLHELP wraps much of Microsoft's HTML Help API to enable you to access HTML Help (.chm files) from a Toolkit application. Each *subfunction* corresponds to an HTML Help API command. See the Microsoft documentation for the HTML Help API for more information.

D_HH_ALINK_LOOKUP

result = %U_HTMLHELP (D_HH_ALINK_LOOKUP, *help_file*, *alinks*, [*window*], *fail_action*)

or

xcall U_HTMLHELP (D_HH_ALINK_LOOKUP, *help_file*, *alinks*, [*window*], *fail_action*)

Return value

result

On Windows, the handle (D_ADDR) of the outermost Help window. (^VAL)

Arguments

help_file

The path and filename for the HTML Help file. The default extension is **.chm**. (a)

alinks

One or more associative keywords (Alink names) that will be used to look up topics. You can search for multiple Alink names by passing them in a semicolon-delimited string. (a)

window

(optional) The name of the window type to use to display a looked-up topic or a topic specified for D_HH_KEYWORD_FAIL_PATH. (a)

fail_action

One of the following to specify how a failure to find any of the *alinks* values should be handled: (n)

D_HH_KEYWORD_FAIL_INDEX

Display the Index tab and populate its field with the keywords.

D_HH_KEYWORD_FAIL_MSG, *text*[, *title*]

Pop up a message box, where *text* is the text for the pop-up message and *title* is an optional title for the message box. (a)

D_HH_KEYWORD_FAIL_PATH, *path*

Display a topic, where *path* is the path and filename for a topic in the HTML Help file. (a)

Discussion

The D_HH_ALINK_LOOKUP subfunction looks up topics for *alinks* for the specified Help file (*help_file*), opening the Help file if necessary. Alink names (sometimes called bookmarks) mark locations within topics, so when a looked-up topic is opened, the topic is scrolled down to the location marked by the *alinks* value.

For *help_file*, you can use an environment variable to specify the path, part of the path, or the full path and filename. If the specified file does not exist, a catchable runtime error (\$ERR_FNF) is generated. You cannot use an xfServer path (e.g., myhelpfile.chm@myserver) to access an HTML Help file on a remote machine. (Because of a Microsoft security feature, you may not be able to access an HTML help file on a remote machine in any case. See Synergex® KnowledgeBase article [100001779](#) for information.)

If you don't pass *window*, if *window* is not in the file specified by *help_file*, or if you pass an empty string for *window*, the default window type for the Help file is used to display the topic.

For *path*, pass the path and filename used in the HTML Help file structure. For example, to specify the topic **my_topic.html** in the /my_folder/my_subfolder directory, pass **/my_folder/my_subfolder/my_topic.html** (including the first slash) for *path*. Note that if the topic specified by *path* cannot be found, **Hhctrl.ocx**, the ActiveX control for HTML Help, determines the result. This is generally a message such as "The page cannot be displayed."

Examples

The following example looks up topics associated with the Alink name "orders". If there are no topics for this, it displays a pop-up message box with the title "Error" and the text "No topic found."

```
.define D_HELPFILE, "MYAPPDIR:bin\myhelp.chm"
.
.
.
help_handle = %u_htmlhelp(D_HH_ALINK_LOOKUP, D_HELPFILE, "orders", mywin,
&                      D_HH_KEYWORD_FAIL_MSG, "No topic found",
&                      "Error")
```

D_HH_CLOSE_ALL

```
xcall U_HTMLHELP(D_HH_CLOSE_ALL)
```

Discussion

The D_HH_CLOSE_ALL subfunction closes all Help windows associated with the application. If this is called as a function, it always returns zero.

If D_HH_CLOSE_ALL has not been called when the application shuts down, it is automatically invoked, unless the program is in a chain.

D_HH_DISPLAY_INDEX

```
result = %U_HTMLHELP(D_HH_DISPLAY_INDEX, help_file[, keyword])
```

or

```
xcall U_HTMLHELP(D_HH_DISPLAY_INDEX, help_file[, keyword])
```

Return value

result

On Windows, the handle (D_ADDR) of the outermost Help window. (^VAL)

Arguments

help_file

The path and filename for the HTML Help file. For information, see the D_HH_ALINK_LOOKUP [Discussion on page 16-73](#). (a)

keyword

(optional) A keyword (index entry) to highlight on the Index tab. (a)

Discussion

The D_HH_DISPLAY_INDEX subfunction displays the Index tab for the specified Help file (*help_file*), opening the Help file if necessary. If you pass a valid *keyword*, the keyword is highlighted on the Index tab.

Examples

The following example displays the Index tab with the keyword “vendors” highlighted on the tab:

```
.define D_HELPFILE, "MYAPPPDIR:bin\myhelp.chm"
.
.
.
help_handle = %u_htmlhelp(D_HH_DISPLAY_INDEX, D_HELPFILE, "vendors")
```


D_HH_DISPLAY_SEARCH

```
result = %U_HTMLHELP(D_HH_DISPLAY_SEARCH, help_file)
```

or

```
xcall U_HTMLHELP(D_HH_DISPLAY_SEARCH, help_file)
```

Return value

result

On Windows, the handle (D_ADDR) of the outermost Help window. (^VAL)

Arguments

help_file

The path and filename for the HTML Help file. For information, see the D_HH_ALINK_LOOKUP [Discussion on page 16-73](#). (a)

Discussion

The D_HH_DISPLAY_SEARCH subfunction displays the Search tab for the specified Help file (*help_file*), opening the Help file if necessary.

Examples

The following example displays the Search tab:

```
.define D_HELPFILE, "MYAPPDIR:bin\myhelp.chm"
.
.
.
help_handle = %u_htmlhelp(D_HH_DISPLAY_SEARCH, D_HELPFILE)
```

D_HH_DISPLAY_TEXT_POPUP

```
result = %U_HTMLHELP (D_HH_DISPLAY_TEXT_POPUP, text_string | text_id, x, y,
& [foreground],[background],[left], [top], [right], [bottom][, font])
```

or

```
xcall U_HTMLHELP (D_HH_DISPLAY_TEXT_POPUP, text_string | text_id, x, y,
& [foreground], [background], [left], [top], [right], [bottom][, font])
```

Return value

result

On Windows, the handle (D_ADDR) of the outermost Help window. (^VAL)

Arguments

text_string

The text to be displayed. (a)

text_id

A series of three arguments, *id*, *help_file*, *path*, where

id is the map ID of a text string within *help_file*. (n)

help_file is the path and filename for an HTML Help file. For information, see the D_HH_ALINK_LOOKUP [Discussion on page 16-73](#). (a)

path is the path within the *help_file* to the text file that contains the text string. For information, see the D_HH_ALINK_LOOKUP [Discussion on page 16-73](#). (a)

x

The screen-relative x-axis pixel coordinate for the top-center of the pop-up window. (n)

y

The screen-relative y-axis pixel coordinate for the top-center of the pop-up window. (n)

foreground

(optional) An RGB value for the text color. See [GetCellColor on page 9-23](#) for information on the format of this value. (n)

background

(optional) An RGB value for the pop-up window's background color. See [GetCellColor on page 9-23](#) for information on the format of this value. (n)

left

(optional) The width in pixels of the margin between the text and the pop-up window's left edge. (n)

top

(optional) The width in pixels of the margin between the text and the pop-up window's top edge. **(n)**

right

(optional) The width in pixels of the margin between the text and the pop-up window's right edge. **(n)**

bottom

(optional) The width in pixels of the margin between the text and the pop-up window's bottom edge. **(n)**

font

(optional) A string with the following format:

"[*facename*], [*point_size*], [*charset*][, BOLD ITALIC UNDERLINE]"

facename

(optional) The name of the type face. **(a)**

point_size

(optional) The font size in points. **(n)**

charset

(optional) The character set. **(n)**

Discussion

The `D_HH_DISPLAY_TEXT_POPUP` subfunction displays text in a pop-up Help window. The text is taken from one of the following, depending on the data type of the second argument (the argument that follows `D_HH_DISPLAY_TEXT_POPUP`):

- ▶ If you pass *text_string* (in other words, if the second argument is an alpha value), the text is taken from the specified text string.
- ▶ If you pass *text_id* (in other words, if the second argument is a numeric value), the text is taken from a text file included in the HTML Help file.

If you omit *foreground*, *background*, *left*, *top*, *right*, or *bottom*, or pass -1 for any of these, the system default is used.

If you don't pass *font* or one or more of its components, the omitted font characteristics are determined by system defaults. For example, if you omit *point_size*, the text in the pop-up window will have the default point size for the user's system.

For *font*, you can pass one, two, or all of the following: BOLD, ITALIC, and UNDERLINE. Use spaces to separate them. See example below.

Examples

The following example displays a pop-up Help window with the text “Text to display” in 14 point bold and underlined Courier:

```
help_handle = %u_htmlhelp(D_HH_DISPLAY_TEXT_POPUP, "Text to display",  
& 100, 100,,,,,, "Courier, 14,, BOLD UNDERLINE")
```

The next example displays a pop-up Help window with text from the text file **/info.txt**, which is contained in D_HELPFILE. System defaults are used for all optional settings.

```
.define D_HELPFILE, "MYAPPDIR:bin\myhelp.chm"  
.  
.  
.  
help_handle = %u_htmlhelp(D_HH_DISPLAY_TEXT_POPUP, 61288, D_HELPFILE,  
& "/info.txt", 100, 100)
```

D_HH_DISPLAY_TOC

```
result = %U_HTMLHELP (D_HH_DISPLAY_TOC, help_file[, path])
```

or

```
xcall U_HTMLHELP (D_HH_DISPLAY_TOC, help_file[, path])
```

Return value

result

On Windows, the handle (D_ADDR) of the outermost Help window. (^VAL)

Arguments

help_file

The path and filename for the HTML Help file. For information, see the D_HH_ALINK_LOOKUP [Discussion on page 16-73](#). (a)

path

(optional) The path within the HTML Help file to the topic whose Contents entry you want highlighted. For information, see the D_HH_ALINK_LOOKUP [Discussion on page 16-73](#). (a)

Discussion

The D_HH_DISPLAY_TOC subfunction displays the Contents tab for the specified Help file (*help_file*), opening the Help file if necessary. Passing a valid *path* may cause D_HH_DISPLAY_TOC to highlight the Contents entry for the topic specified by *path* and to display the topic. For example, this will occur if D_HH_DISPLAY_TOC opens the Help file, but it may not occur if the Help file is already open in a window that displays the navigation pane (the pane that displays the Contents, Index, and Search tabs).

Examples

The following example displays the Contents tab but does not change highlighting on the tab or the displayed topic because *path* is omitted.

```
help_handle = %u_htmlhelp(D_HH_DISPLAY_TOC, "MYAPPDIR:bin\myhelp.chm")
```

D_HH_DISPLAY_TOPIC

result = %U_HTMLHELP(*D_HH_DISPLAY_TOPIC*, *help_file*, *path*[, *window*])

or

xcall U_HTMLHELP(*D_HH_DISPLAY_TOPIC*, *help_file*, *path*[, *window*])

Return value

result

On Windows, the handle (D_ADDR) of the outermost Help window. (^VAL)

Arguments

help_file

The path and filename for the HTML Help file you want to open. For information, see the D_HH_ALINK_LOOKUP [Discussion on page 16-73](#). (a)

path

The path within the HTML Help file to the topic you want displayed. For information, see the D_HH_ALINK_LOOKUP [Discussion on page 16-73](#). (a)

window

(optional) The name of the window type to use to display the topic. (a)

Discussion

The D_HH_DISPLAY_TOPIC subfunction opens a specified Help topic (*path*) in the default window or in a specified window (*window*).

If you don't pass *window*, if *window* is not in the file specified by *help_file*, or if you pass an empty string for *window*, the default window type for the Help file is used to display the topic.

Examples

The following example displays the **/entry/orders.htm** topic with the mywin window type.

```
.define D_HELPFILE, "MYAPPDIR:bin\myhelp.chm"
.
.
.
help_handle = %u_htmlhelp(D_HH_DISPLAY_TOPIC, D_HELPFILE,
&                      "/entry/orders.htm", mywin)
```

D_HH_GET_WIN_HANDLE

```
result = %U_HTMLHELP(D_HH_GET_WIN_HANDLE, help_file, window)
```

or

```
xcall U_HTMLHELP(D_HH_GET_WIN_HANDLE, help_file, window)
```

Return value

result

On Windows, the handle (D_ADDR) of the window for the specified window type, or 0 if there is no such window. (^VAL)

Arguments

help_file

The path and filename for the HTML Help file. For information, see the D_HH_ALINK_LOOKUP [Discussion on page 16-73](#). (a)

window

The name of the window type whose handle you want to retrieve. (a)

Discussion

The D_HH_GET_WIN_HANDLE subfunction returns (in *result*) the handle for the window of the type specified in *window*. If no window has yet been created for the specified window type (i.e., if no topic has been displayed using the specified type), *result* is returned with a value of 0.

Examples

The following example returns the handle for the Help window of type mywin, or it returns a value of 0 if no Help window of that type has been created.

```
.define D_HELPFILE, "MYAPPDIR:bin\myhelp.chm"  
.  
.  
.  
help_handle = %u_htmlhelp(D_HH_GET_WIN_HANDLE, D_HELPFILE, mywin)
```

D_HH_HELP_CONTEXT

result = %U_HTMLHELP (D_HH_HELP_CONTEXT, *help_file*, *context_id*[, *window*])

or

xcall U_HTMLHELP (D_HH_HELP_CONTEXT, *help_file*, *context_id*[, *window*])

Return value

result

On Windows, the handle (D_ADDR) of the outermost Help window. (^VAL)

Arguments

help_file

The path and filename for the HTML Help file. For information, see the D_HH_ALINK_LOOKUP [Discussion on page 16-73](#). (a)

context_id

A context ID mapped to a topic in the HTML Help file. (n)

window

(optional) The name of the window type to use to display the topic. (a)

Discussion

The D_HH_HELP_CONTEXT subfunction displays the topic specified by *context_id*, which is a numeric ID that maps to a topic in *help_file*, opening the Help file if necessary.

If you don't pass *window*, if *window* is not in the file specified by *help_file*, or if you pass an empty string for *window*, the default window type for the Help file is used to display the topic.

Examples

The following example displays the Help topic with the context ID of 102 in a mywin type window:

```
.define D_HELPFILE, "MYAPPDIR:bin\myhelp.chm"
.
.
.
help_handle = %u_htmlhelp(D_HH_HELP_CONTEXT, D_HELPFILE, 102, "mywin")
```


D_HH_KEYWORD_LOOKUP

```
result = %U_HTMLHELP (D_HH_KEYWORD_LOOKUP, help_file, keywords, [window],  
& fail_action)
```

or

```
xcall U_HTMLHELP (D_HH_KEYWORD_LOOKUP, help_file, keywords, [window],  
& fail_action)
```

Return value

result

On Windows, the handle (D_ADDR) of the outermost Help window. (^VAL)

Arguments

help_file

The path and filename for the HTML Help file. For information, see the D_HH_ALINK_LOOKUP [Discussion on page 16-73](#). (a)

keywords

The keyword(s) whose topic(s) will be looked up. You can search for multiple keywords by passing them in a semicolon-delimited string. (a)

window

(optional) The name of the window type to use to display a looked-up topic or a topic specified for D_HH_KEYWORD_FAIL_PATH. (a)

fail_action

One of the following to specify how a failure to find any of the keywords should be handled: (n)

D_HH_KEYWORD_FAIL_INDEX

Display the Index tab and populate its field with the keywords.

D_HH_KEYWORD_FAIL_MSG, *text*[, *title*]

Pop up a message box, where *text* is the text for the pop-up message and *title* is an optional title for the message box. (a)

D_HH_KEYWORD_FAIL_PATH, *path*

Display a topic, where *path* specifies the path and filename for a topic in the HTML Help file. For information, see the D_HH_ALINK_LOOKUP [Discussion on page 16-73](#). (a)

Discussion

The `D_HH_KEYWORD_LOOKUP` subfunction looks up topics for *keywords* in the specified Help file (*help_file*), opening the Help file if necessary.

If you don't pass *window*, if *window* is not in the file specified by *help_file*, or if you pass an empty string for *window*, the default window type for the Help file is used to display the topic.

Examples

The following example looks up topics associated with the keyword "accounts." If there are topics for this, it displays a pop-up message box with the title "Error" and the text "No topic found."

```
.define D_HELPFILE, "MYAPPDIR:bin\myhelp.chm"
.
.
.
help_handle = %u_htmlhelp(D_HH_KEYWORD_LOOKUP, D_HELPFILE, "accounts",
&                               mywin, D_HH_KEYWORD_FAIL_MSG, "No topic found",
&                               "Error")
```

The next example does the same, but instead of displaying a pop-up message box if no topics are found, it displays the Index tab, populating the field on the tab with the keyword "accounts."

```
.define D_HELPFILE, "MYAPPDIR:bin\myhelp.chm"
.
.
.
help_handle = %u_htmlhelp(D_HH_KEYWORD_LOOKUP, D_HELPFILE, "accounts",
&                               mywin, D_HH_KEYWORD_FAIL_INDEX)
```

%U_ICON – Set an icon or get an icon count

WT	WN		

```
status = %U_ICON (D_ICONSET, filename, index[, window_id])
```

```
status = %U_ICON (D_ICONCOUNT, filename, count)
```

or

```
xcall U_ICON (D_ICONSET, filename, index[, window_id])
```

```
xcall U_ICON (D_ICONCOUNT, filename, count)
```

Return value

status

On Windows, true (1) if the icon file exists and contains icons, or false (0) if the icon file does not exist or contains no icons. On UNIX and OpenVMS, always returns false. (^VAL)

Arguments

D_ICONSET

Set an icon. (n)

filename

The name of the file that contains the icon to set or the icons to count. (a)

index

The icon index for D_ICONSET (1-based). (n)

window_id

(optional) The ID of the window. (n)

D_ICONCOUNT

Get the count of icons in the file. (n)

count

The icon count returned from D_ICONCOUNT. (n)

Discussion

%U_ICON sets an icon for a window or gets a count of icons in a file.

Filename can be an **.ico** file, a DLL, an executable program (**.exe**), or a bitmap image (**.bmp**). You can use environment variables in *filename*.

In Synergy .NET, this routine works only for the container window, and it uses only the first icon in the icon file. It ignores the *index* and *count* arguments (though you still must pass them) and the *window_id* argument.

Setting an icon for a window

To set an icon for a window, pass D_ICONSET and the *index* (1-based) for the icon (i.e., pass the icon's position in the file). An index value of 0 will set the icon to the default Windows system icon. For an **.ico** file, you must pass 1 as *index*.

Window_id determines which window the icon will be applied to:

- ▶ To apply the icon to the application window, omit *window_id* or pass zero. In either case, the icon is displayed in the About box if U_ABOUT is subsequently called, and the application uses the icon for its system menu and its taskbar button.
- ▶ To apply the icon to a window other than the application window, pass the window ID as *window_id*. In this case, the icon is used for the system menu, which is displayed if a close, move, size, minimize or maximize method is associated with the window. It can also be used if a DLL routine adds a system menu, changes the window frame style, or minimizes the window.

Getting a count of icons in a file

To get a count of icons in a file, pass D_ICONCOUNT and a variable as *count* to accept the returned count.

See also

[%L_ICON](#) on page 9-96 for setting a list icon.

Examples

The following example sets the application icon to the third icon in a file named **iconfil.dll**. If the icon cannot be found, an error is displayed.

```
if (.not. (%u_icon(D_ICONSET, "iconfil.dll", 3)))  
    xcall u_message("Could not set icon")
```

U_INITREND – Reset Toolkit renditions

	WN	U	V

```
xcall U_INITREND
```

Discussion

The U_INITREND subroutine resets UI Toolkit renditions to initial settings established by U_START. For information on these initial settings, see the “[U_START initialization](#)” columns of the “[Renditions](#)” table on [page 18-26](#).

U_LDREND – Load a rendition scheme



```
xcall U_LDREND (channel, [scheme][, status])
```

Arguments

channel

The channel on which the rendition file is already opened. **(n)**

scheme

(optional) The rendition scheme name. **(a)**

status

(optional) Returned with zero (0) if the load operation is successful, or with a Synergy DBL error number if it is unsuccessful. **(n)**

Discussion

The U_LDREND subroutine loads a rendition scheme from an open rendition file. Once the rendition scheme is loaded, the rendition definitions are used for subsequent screen displays.

If *scheme* is not passed or is blank, the working rendition settings are loaded.

See also

[“Customizing the Look of Your Application” on page 18-24.](#)

Examples

The following utility subroutine loads and saves personalized rendition schemes from a rendition file.

```
subroutine pers_rend
    a_scheme      ,a
    a_ldflg       ,n      ;Load flag: 1=load, 0=store

    .include "WND:tools.def"

    global data section mygb
    record
        rendchn      ,i4
    endglobal

    record
        error        ,i4
    proc
```

```
    if (a_ldflg) then
        xcall u_ldrend(rendchn, a_scheme, error)
    else
        xcall u_storend(rendchn, a_scheme, error, TRUE)
    if (error)
        xcall proc_error
    xreturn
endsubroutine
```

U_LDWND – Load a window from a library



```
xcall U_LDWND (window_id, [channel], window_name, [D_NOPLC], [search], [error]
&           [, saved_name], {reserved})
```

Arguments

window_id

Returned with the ID of the window. **(n)**

channel

(optional) The channel of the window library. **(n)**

window_name

The name of the window to load. **(a)**

D_NOPLC

(optional) The no-placement flag. If passed, the window is not placed on the screen. **(n)**

search

(optional) The search variable. **(n)**

error

(optional) The error return variable. **(n)**

saved_name

(optional) The name of the window when it was originally saved. **(a)**

{reserved}

Reserved for internal use. **(n)**

Discussion

The U_LDWND subroutine loads a window from a window library.

Use U_LDWND to load text windows, selection windows, or general windows from a window library. Use I_LDINP to load input windows, and M_LDCOL to load menu columns. Lists must be created using L_CREATE, although you can load the list class into memory with U_LDWND.

If *channel* is not passed, **g_utlib** will be used.

If you pass a null string (“”) for *window_name*, Toolkit assigns a unique window name in the form *_W_nnn*, where *nnn* is the window ID. You must pass *saved_name* in this case.

If you want to create multiple instances of the same window, you can call U_LDWND as many times as you need, using *window_name* to give each succeeding window a new unique name. You must also pass *saved_name* if *window_name* is different than the name under which the window was saved.

To create another instance of the same window, pass "" for *window_name*. You must pass *saved_name* in this case. A window name is automatically generated for you.

If *search* is present, U_LDWND searches for an existing window with the specified name before attempting to load the window. If the window is found, a successful load is simulated, *window_id* is set to the ID of the window that was found, and the *search* variable is set to true. If the window is not found, the *search* variable is cleared and the load continues normally.

If *error* is present, it will be returned with a zero (0) if the requested window was found in the specified library or a non-zero value (the error code number) if the window was not found. A fatal error occurs if *error* is not present and the window cannot be found; we recommend that you pass it.

See also

- ▶ [I_LDINP on page 8-62](#) for loading input windows.
- ▶ [M_LDCOL on page 10-37](#) for loading menu columns.
- ▶ [L_CREATE on page 9-75](#) for creating lists.

Examples

The first example loads a window with the name in **wndname(i)** from the window library on the channel stored in **chnl**, and returns the ID to **wndid(i)**. This window will not be placed on the screen. **Sts** is the error variable and is returned with a value of 0 if the operation is successful.

```
xcall u_ldwnd(wndid(i), chnl, wndname(i), D_NOPLC,, sts)
```

The second example loads the window whose name is "password" from the library on the channel stored in **chnl**, and returns the ID to **passid**. The routine will look for an existing window named "password" before loading. Again, **sts** is the error variable. The window is placed on the screen.

```
xcall u_ldwnd(passid, chnl, "password",, found, sts)
```

U_LOGWND – Log a window as global or local



```
xcall U_LOGWND (window_id [, D_GLOBAL])
```

Arguments

window_id

The ID of the window to be logged. (**n**)

D_GLOBAL

(optional) Log the window as global. (**n**)

Discussion

The U_LOGWND subroutine logs a window ID as either global or local.

The window is logged as global if D_GLOBAL is present. The window is entered into the current (local) environment if D_GLOBAL is not present.

Always use this subroutine with a window created outside of the UI Toolkit utilities (for example, if you created it yourself with the Synergy DBL W_xxx subroutines, loaded it from your own library, and so forth). If you don't, UI Toolkit environment will not recognize its existence.

Examples

The following example logs the window whose ID is stored in the **passid** variable as local.

```
xcall u_logwnd(passid)
```

U_MESSAGE – Display a message



```
xcall U_MESSAGE (message ([D_BELL | D_NOBELL], [D_ALERT | D_ERROR])
```

Arguments

message

The message to display. (**a**)

D_BELL

(optional) Ring the terminal bell. (default) (**n**)

D_NOBELL

(optional) Don't ring the terminal bell. (**n**)

D_ALERT

(optional) Do alert processing. (default) (**n**)

D_ERROR

(optional) Do error processing. (**n**)

Discussion

The U_MESSAGE subroutine displays a message in a message window. If your message requires a Yes/No or OK/Cancel type of response, see [%U_MSGBOX on page 16-102](#).

UNIX, VMS

On UNIX and OpenVMS, U_MESSAGE performs the following steps:

1. Looks for an existing message window called `u_message`.

If it cannot find an existing `u_message` window, it looks in the window library opened on `g_utlib`. If it cannot find a `u_message` window in the library, U_MESSAGE creates a message window and places it 12 lines above the information line. (You can change the rendition of the message window with the `U_REND` subroutine.)

2. Displays *message* in the `u_message` window.

If *message* is longer than 76 characters it will be divided up into a maximum of 5 lines with up to 76 characters each. This will be done by searching for the space or tab nearest to, but not beyond, the 76th character after the previous break.

If the `u_message` window already exists, it should be 5 lines long. U_MESSAGE will shorten the display area of the existing `u_message` window, if necessary, to the appropriate height (based on the number of lines *message* takes).

If `U_MESSAGE` creates the window, the created window's display area will be two columns wider than the length of the message's longest line and have a height determined by the number of lines (not to exceed five lines).

3. Calls `U_WAIT`, which creates another message window telling the user what to do, and waits for a user response.



If you supply a window for `U_MESSAGE`, you may also want to supply a window for `U_WAIT`. If you don't, the `U_WAIT` window will be centered three lines above the information line, which may not match the placement you specified for the `u_message` window. For more information, see [U_WAIT on page 16-148](#). (Note that this is *not* true for `%U_MSGBOX`; if you supply a window for `%U_MSGBOX`, the selection window will be placed relative to the `%U_MSGBOX` window.)

If a time-out occurs, `g_select` is set to true and `g_entnam` is set to the entry name specified in `tkctl.def` for `g_time_entry`. By default, `g_time_entry` is defined as "O_TIMEOUT".

4. Removes the message window from the screen prior to exiting the subroutine.



The terminal bell will not ring if `g_beep` is set to false. See [g_beep on page 18-2](#).

WIN

In Synergy .NET, Toolkit uses a cell-based-style message box if `g_netnativemsgbox` (a field in `tools.def`) is set to false, which is the default. If `g_netnativemsgbox` is set to true, Toolkit uses a native Windows message box.

In traditional Synergy, `U_MESSAGE` creates a Windows-style message box with an icon and an OK button. The icon is an information icon (i) if you pass `D_ALERT` or an exclamation point (!) if you pass `D_ERROR`.

Insert `\n` to specify explicit line breaks in *message*. To include the characters "`\n`", prefix the characters with another backslash. For example, the following displays a two-line message with "First" in the first line and "Second" in the second:

```
xcall u_message("First\nSecond")
```

The next example displays only one line, "C:\maps\northHaven":

```
xcall u_message("C:\maps\\northHaven")
```

`U_MESSAGE` cannot time out on Windows.

For abort processing, see [U_ABORT on page 16-5](#). The `U_ABORT` subroutine creates its own message window.

See also

- ▶ [U_REND on page 16-116](#) for information on changing window renditions.
- ▶ [U_WAIT on page 16-148](#) for more information on the wait flags.

Examples

This example displays the message “Invalid Customer Account” in the message window and waits in the error state for a user response.

```
xcall u_message("Invalid Customer Account", D_ERROR)
```

U_MINVERSION – Check for minimum Synergy/DE product versions



```
xcall U_MINVERSION([minimum_toolkit][, minimum_runtime])
```

Arguments

minimum_toolkit

(optional) The minimum UI Toolkit version required. **(a)**

minimum_runtime

(optional) The minimum Synergy Runtime version required. **(a)**

Discussion

The U_MINVERSION subroutine checks the version of UI Toolkit against *minimum_toolkit* and the version of the Synergy Runtime against *minimum_runtime*. If the version of UI Toolkit or the Synergy Runtime is lower than a minimum passed to U_MINVERSION, Toolkit generates a fatal error.

Use the following syntax for *minimum_toolkit* and *minimum_runtime* (where *v* is the version number, *r* is the major revision number, and *m* is the minor revision number):

v[.r][.m]

You can use an uppercase letter for a two-byte version number. For example, you can use A for 10, as in A.1.1 for 10.1.1. But note that if you do use an uppercase letter, it must be the first character of *v*, *r*, or *m*. In this case, the second character is ignored—for example, 1A.2.3 is interpreted as 1.2.3, and A1.2.3 is interpreted as 10.2.3. Additionally, note the following:

- ▶ If *v* or *m* is omitted, a major or minor revision number of 0 is assumed.
- ▶ Characters other than the following are ignored: numbers, periods (.), and capital letters that are used for two-byte version numbers.

Examples

The following generates a fatal Toolkit error if the Synergy Runtime version is lower than 9.5.3:

```
xcall u_minversion(", "9.5.3")
```

The following examples check to make sure that the version of both UI Toolkit and the Synergy Runtime are at least 10.1.1:

```
xcall u_minversion("10.1.1", "10.1.1")
xcall u_minversion("A.1.1", "A.1.1")
```

U_MODREND – Modify the current rendition scheme



```
xcall U_MODREND ([number], [descriptions], [windows], [areas], [field_numbers][, help_id])
```

Arguments

number

(optional) The number of user renditions to use. The default is 0. **(n)**

descriptions

(optional) The array of user rendition descriptions. If an array is not specified and user renditions are used, a default description is supplied. **(a)**

windows

(optional) The array of window IDs for user renditions on UNIX and OpenVMS. Each window is placed by a move 0,0. If an array is not specified and user renditions are used, a default window is created. All created windows are deleted upon exit of this subroutine. (Ignored on Windows.) **(n)**

areas

(optional) The array of which areas within *windows* will be affected on UNIX and OpenVMS. These areas can be added together for combined effects. If *areas* is 0, then none are affected. The default is all three areas. (Ignored on Windows.) **(n)**

D_RND_AREA Processing area.

D_RND_BRDR Border area.

D_RND_TITLE Title area.

field_numbers

(optional) The array of field numbers that will be affected on UNIX or OpenVMS or **WFN_ALL** to affect all fields. The default is none. (Ignored on Windows.) **(n)**

help_id

(optional) The help identifier. **(a)**

Discussion

The U_MODREND subroutine enables the user to modify the current rendition scheme without saving the changes. (U_EDITREND saves and restores renditions.)



U_MODREND cannot set the Input Field Processing rendition on Windows (though it can on UNIX and OpenVMS).

It cannot set the following on any platform: Disabled Input Field, Read-only Input Field Display, and Read-Only Field Processing. (These can be set only by U_REND and can be set only on UNIX and OpenVMS.)

U_MODREND modifies the rendition scheme that is held in memory and does not call any of the renditions file manipulation routines. So you can save the rendition scheme in any manner you choose.

U_MODREND recognizes the following menu entry names if they are included in a placed menu column:

O_CANCEL	Cancel changes.
O_EXIT	Exit.
O_HELP	Call the EHELP_METHOD subroutine and pass <i>help_id</i> .
O_DEFAULT	Set renditions to their default values.
O_PALETTE	Modify a color palette.
S_UP	Move to the previous selection window item.
S_DOWN	Move to the next selection window item.
S_LEFT	Move to the selection item in the previous column.
S_RIGHT	Move to the selection item in the next column.

Any other menu entries cause this subroutine to return with **g_select** set to true and **g_entnam** set to the selected menu entry.

O_EXIT causes U_MODREND to return to the calling program, but it does not clear **g_select** and **g_entnam**. This enables a layered calling routine to continue a seamless exit.

If a time-out occurs, **g_select** is set to true and **g_entnam** is set to the entry name specified in **tkctl.def** for **g_time_entry**. By default, **g_time_entry** is defined as "O_TIMEOUT".

See also

[“Customizing the Look of Your Application” on page 18-24.](#)

U_MONTHS – Assign alpha characters to represent months

WT	WN	U	V

```
xcall U_MONTHS (months)
```

Arguments

months

The first element of a 12-element array containing abbreviations for months. (a)

Discussion

The U_MONTHS subroutine enables you to specify alpha character strings for months. Note that U_MONTHS reads up to five characters in each string (the first five characters), and strings must consist of capital letters. These strings default to the first three characters of each month in English. So if you do not use U_MONTHS, the string for January is **jan**, the string for February is **feb**, and so on.

U_MONTHS is particularly useful for assigning month abbreviations for foreign languages. For example, the Spanish word for “January” is “Enero.” Thus, U_MONTHS could enable users to enter **ene** (rather than **jan**) for the first month of the year. You could specify the entire string “**enero**” in this subroutine as well.

Examples

In this example, **mon** is the array that consists of month strings.

```
xcall u_months(mon)
```

%U_MSGBOX – Pop up a Windows-style message box



```
button = %U_MSGBOX (message, [styles], [title], [select_col][, help_id])
```

or

```
xcall U_MSGBOX (message, [styles], [title], [select_col][, help_id])
```

Return value

button

False if the user doesn't select a message box button, but instead tries to select a menu entry. If a message box button is selected, returns the corresponding option: (**^VAL**)

D_MIDOK	OK button selected.
D_MIDCANCEL	Cancel button selected.
D_MIDABORT	Abort button selected.
D_MIDRETRY	Retry button selected.
D_MIDIGNORE	Ignore button selected.
D_MIDYES	Yes button selected.
D_MIDNO	No button selected.

Arguments

message

The message text or string. (**a**)

styles

(optional) One or more style options to set values for the dialog box, the buttons and the icon on the dialog box, and modality. See the “[Styles options](#)” table on page 16-104. If *styles* is not passed, %U_MSGBOX defaults to the documented default values. (**n**)

title

(optional) The title of the message box. (**a**)

select_col

(optional) The selection column ID (UNIX and OpenVMS only). (**n**)

help_id

(optional) The help identifier, which is passed to EHELP_METHOD if the user selects “O_HELP” while in this function. (**a**)

Discussion

The %U_MSGBOX function pops up a Windows-style message box.

- ▶ On UNIX and OpenVMS, %U_MSGBOX follows the same steps to create a message window as U_MESSAGE does. (See the U_MESSAGE [Discussion on page 16-95](#).) %U_MSGBOX, however, looks for a window named u_msgbox, creates a selection window (rather than a “Press... to continue” message), and processes the selection window for valid options. If a menu entry is selected, the function returns false and **g_select** is cleared. Otherwise, %U_MSGBOX returns the value that corresponds to the selected button.
- ▶ On Windows in traditional Synergy, %U_MSGBOX creates a Windows-style message box with one to three buttons, depending on the arguments passed. The icon can be a stop sign, exclamation point (!), question mark (?), or the information icon (i). Button choices are Yes, No, OK, Cancel, Abort, Retry, and Ignore. Since this function invokes the standard Windows MessageBox function, which is application-modal, no menu processing (or any other type of processing) is available while a message box is displayed.
- ▶ On Windows in Synergy .NET, Toolkit uses a cell-based-style message box if **g_netnativemsgbox** (a field in **tools.def**) is set to false, which is the default. If **g_netnativemsgbox** is set to true, Toolkit uses a native Windows message box.

If a time-out occurs on UNIX or OpenVMS, **g_select** is set to true and **g_entnam** is set to the entry name specified in **tkctl.def** for **g_time_entry**. By default, **g_time_entry** is defined as “O_TIMEOUT”. %U_MSGBOX cannot time out on Windows.

Message is wrapped according to the same rules as apply to the U_MESSAGE subroutine.

The *styles* argument enables you to set optional values for the dialog box, the buttons and the icon on the dialog box, and modality. The dialog box, icon, and modality options have no effect on UNIX and OpenVMS. See the table below for valid options.

- ▶ Default positioning on Windows is determined by Windows Common Dialogs and is generally centered in the screen. Specifying D_MCENTER, however, centers the dialog box over the application window, which matches UNIX and OpenVMS behavior.
- ▶ On Windows, the close button for the message box is disabled if you pass D_MABTRETIGN or D_MYYESNO. If you pass a *styles* option that includes the Cancel button, the close button causes %U_MSGBOX to return D_MIDCANCEL. If you pass D_MOK, the close button causes %U_MSGBOX to return D_MIDOK.

The following options, defined in **tools.def**, can be combined using .BOR. or the vertical bar (|) operator.

Styles options		
Category	Option	Definition
Dialog	D_MCENTER	Center dialog box over application window (Not available in Synergy .NET)
Button	D_MOK	OK button only (default)
	D_MOKCANCEL	OK and Cancel buttons
	D_MABTRETIGN	Abort, Retry, Ignore buttons
	D_MYESNOCANCEL	Yes, No, Cancel buttons
	D_MYESNO	Yes and No buttons
	D_MRETRYCANCEL	Retry and Cancel buttons
Default button	D_MDEFBUTTON1	Set default to first button (default)
	D_MDEFBUTTON2	Set default to second button
	D_MDEFBUTTON3	Set default to third button
Icon style (no icon is the default)	D_MICONSTOP	Stop sign
	D_MICONQUESTION	Question mark (?)
	D_MICONEXCLAM	Exclamation point (!)
	D_MICONINFO	Information (i)
Modality	D_MAPPLMODAL	Application modal (default)
	D_MSYSYSTEMMODAL	System modal

If you omit *title* in a Windows environment with traditional Synergy, the message box inherits the application's window title. With Synergy .NET, the default title is the Windows default, "Error". In UNIX and OpenVMS environments, the default for *title* is blank.

On UNIX and OpenVMS, if *select_col* is passed and non-zero, the specified column is placed on the menu and removed on exit. As an alternative, you can place the column before calling %U_MSGBOX. The default behavior for each environment's handling of columns is controlled by the **g_plc_col_args** flag, which is defined in the distributed **tkctl.def** file. See [g_plc_col_args on page 18-4](#) for information.

On Windows, *select_col* is ignored. (Columns passed as *select_col* usually contain reserved menu entries that are automatically supported on Windows. Excessive placing and removing of columns causes Windows menu bars to flicker.)

See also

- ▶ [U_ABORT](#) on page 16-5 for information about abort processing that creates its own message window.
- ▶ [U_ABOUT](#) on page 16-8 for information about displaying an “About” message box.
- ▶ [U_MESSAGE](#) on page 16-95 for information about a more simple message box.
- ▶ [EHELP_METHOD](#) on page 7-73 for information about help processing.

Examples

The following example pops up a window titled “Inquire” that contains the text “Error opening file”. On UNIX and OpenVMS, a single-item selection window is displayed containing the item “OK”. On Windows, a stop sign icon is displayed, along with an OK button. See [figure 16-6](#).

```
rtn = %u_msgbox("Error opening file", D_MOK|D_MICONSTOP, "Inquire")
```

If the user selected “OK”, **rtn** would contain D_MIDOK. If the user selected a menu entry (available only on UNIX and OpenVMS), **rtn** would be false, and **g_select** would be true.

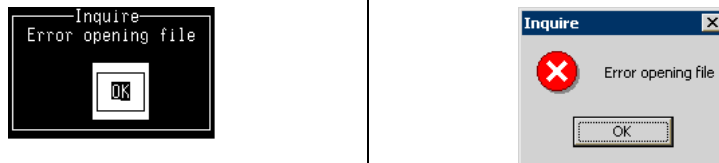


Figure 16-6. Sample %U_MSGBOX window for Unix or OpenVMS (left) and Windows (right).

U_OPEN – Open a channel/file



```
xcall U_OPEN (channel, [I/O_mode], filename, [options], [search], [error] [, D_NOALLOC])
```

Arguments

channel

Returned with the allocated channel. **(n)**

I/O_mode

(optional) The I/O mode in which to open the file. If not passed, defaults to **I:I**. **(a)**

filename

The name of the file to open. **(a)**

options

(optional) The open OPTIONS string. **(a)**

search

(optional) The search variable. **(n)**

error

(optional) Returned with the error value. **(n)**

D_NOALLOC

(optional) The no channel allocation flag; use the passed *channel*. **(n)**

Discussion

The U_OPEN subroutine allocates a global or local channel and opens a file.

The open is performed using an OPTIONS qualifier on the OPEN statement. The contents of the *I/O_mode* argument are appended to the string “IO=” and the result is placed into the OPTIONS string. *I/O_mode* must be one of the valid Synergy DBL open modes for the file (**I** for input, **U:I** for ISAM update, **O:S** for output sequential, and so forth).

If *options* is present and non-blank, a slash and the contents of that variable are appended to the OPTIONS string discussed above.

If *search* is not present, the file is opened on a new channel in the specified mode.

U_OPEN will search for *filename* in the specified *I/O_mode* if *search* is present and the file is not being opened for output. If *filename* is found, a successful open is simulated, *channel* is set to the channel on which the file is opened, and the *search* variable is set to true. If *filename* is not found,

the *search* variable is cleared and the open continues normally. Note that the *I/O_mode* and *filename* must match in content *exactly* and that *filename* is case sensitive and can have up to 256 characters. (*I/O_mode* is not case sensitive.)

If *error* is present, it is returned with a value of **0** if the open request is successful, or the number of the Synergy DBL error that occurs if the open fails. The *channel* argument is not loaded if an error occurs. If *error* is not present, any error that occurs will cause a fatal Toolkit error.

If **D_NOALLOC** is present, no channel will be allocated and the *channel* argument is assumed to contain the channel to use on entry. This is useful for converting existing applications that have “hard-wired” channels. **D_NOALLOC** is ignored if *search* is present.

Rather than explicitly allocating channels yourself, it is best to enable **U_OPEN** to manage channel allocation for you. Enabling **U_OPEN** to maintain channel usage will help you keep your application modular.

The available *channels* are from 1 to 255. (The maximum number of channels Toolkit can log is 255.)

See also

- ▶ [U_CLOSE on page 16-18](#) for information about how to close or purge channels.
- ▶ [U_FINISH on page 16-46](#) for information about how to close and purge channels at application shut down.
- ▶ [U_START on page 16-136](#) for information about how to open channels to the window library, text library, and the terminal and to determine the number of channels available to your application.
- ▶ [U_GETCHN on page 16-59](#) for information about how to convert applications.

Examples

In the first example, the variable **chnl** is returned with the allocated channel. The “cusmas” file is opened in ISAM update mode.

```
xcall u_open(chnl, "u:i", "cusmas")
```

The variable **chnl** is returned with the allocated channel in this example also. The file named “tagfil” is opened in output mode and the success of the operation is reported in **sts**.

```
xcall u_open(chnl, "o", "tagfil",,, sts)
```

The third example is the same as the first example, except that **U_OPEN** will see if the file is already opened in **U:I** mode before opening it. If it is already opened, the existing channel number is returned in **chnl**. If it isn’t already opened, **U_OPEN** tries to open it and returns **0** to **sts** if the open is successful or the number of the Synergy DBL error if the operation is not successful.

```
xcall u_open(chnl, "u:i", "cusmas",, srchflg, sts)
```

U_PARSEVERSION – Parse a version number



```
xcall U_PARSEVERSION(ver_number, version, revision, minor)
```

Arguments

ver_number

The incoming version number (*v.r.m*). (**a**)

version

Returned with the application version number. (**n**)

revision

Returned with the major revision number. (**n**)

minor

Returned with the minor revision number. (**n**)

Discussion

The U_PARSEVERSION subroutine parses a version number string (*ver_number*) into numeric values.

The *version_num* string is expected to be in the form *v.r.m*, where *v* is the application version number, *r* is the major revision number, and *m* is the minor revision number.

You can use an uppercase letter for a two-byte version number. For example, you can use A for 10, as in A.1.1 (for 10.1.1). But note that if you use an uppercase letter, it must be the first character of *v*, *r*, or *m*. In this case, the second character is ignored—for example, 1A.2.3 is interpreted as 1.2.3, and A1.2.3 is interpreted as 10.2.3.

See also

[U_CHECKVERSION](#) on page 16-14 for information about testing version numbers.

U_POPUP – Pop up a window



```
xcall U_POPUP ([channel], window_name, [search], [D_BELL | D_NOBELL],
&             [D_ALERT | D_ERROR], [error][, D_HELP])
```

Arguments

channel

(optional) The window library channel. **(n)**

window_name

The name of the window to load. **(a)**

search

(optional) The search variable. **(n)**

D_BELL

(optional) Ring the terminal bell. (default) **(n)**

D_NOBELL

(optional) Don't ring the terminal bell. **(n)**

D_ALERT

(optional) Do alert processing. (default) **(n)**

D_ERROR

(optional) Do error processing. **(n)**

error

(optional) Returned with the error value. **(n)**

D_HELP

(optional) Use help window renditions. **(n)**

Discussion

The U_POPUP subroutine pops up a window. On Windows, U_POPUP creates a Windows-style message box with an “OK” button.

U_POPUP calls U_LDWND to load the window from the specified library. If *channel* is not passed, **g_utlib** will be used.

If *search* is present, U_POPUP will search for an existing window with the specified name before trying to load it from the library. If it isn't already loaded and it isn't in the library, a fatal Toolkit error occurs.

The D_ALERT and D_ERROR arguments are passed to the U_WAIT subroutine to wait for a user response. If neither is specified, U_POPUP assumes an alert state. The terminal bell (and MessageBeep on Windows) will not ring if **g_beep** is set to false (see [g_beep on page 18-2](#)).

If D_HELP is present, the renditions of *window_name* will be set to the help window renditions.

If the window was loaded by U_POPUP, it will be deleted before the subroutine returns. A “Window not found” error occurs if the specified window cannot be found.

See also

[U_WAIT on page 16-148](#) for more information on the wait flags.

Examples

This example pops up the window named “custhlp,” located in the window library on the channel stored in **chnl**. U_POPUP assumes an error-style waiting state and displays the window with help window renditions.

```
xcall u_popup(chnl, "custhlp", , , D_ERROR, , D_HELP)
```

%U_PRINTQUERY – Retrieve printer characteristics



```
status = %U_PRINTQUERY (subfunction, info_record)
```

or

```
xcall U_PRINTQUERY (subfunction, info_record)
```

Return value

status

On Windows, true if successful, and false if unsuccessful (if the specified printer is not found). If D_ENUM_PRINTERS or D_ENUM_WITH_INFO is specified on Windows, a value of -1 (D_EOLIST) is returned if there are no more printers configured in the print manager. On UNIX and OpenVMS, always returns false. (^VAL)

Arguments

subfunction

One of the following: (n)

D_DEVICEINFO	Return info about a specific printer.
D_ENUM_PRINTERS	Return the next printer ID.
D_ENUM_WITH_INFO	Return the next printer ID and its information.
D_ENUM_RESET	Reset D_ENUM_PRINTERS to its initial state.

info_record

Returned with a structure containing printer characteristics. (a)

Discussion

On Windows, the %U_PRINTQUERY function retrieves information about the currently selected printer or a specified printer from your Windows Print Manager. It can also retrieve the names of all printers configured in your Windows Print Manager. The *info_record* format is

```
record info_record
    lpp          ,i4      ;Lines per page
    cpl          ,i4      ;Characters per line
    flags        ,i4      ;Miscellaneous flags
    unused       ,a20     ;Reserved space
    printerid    ,a80     ;For pre-7.5 compatibility
    driver       ,a32     ;Printer driver name
    extprinterid ,a256    ;Extended printer name and port(s)
```



Windows requires support for 256 character printer names. Because of this, you must use the **extprinterid** field, added in version 7.5, rather than **printerid** on Windows.

If `D_DEVICEINFO` is specified, *info_record* is returned with information about the printer designated by **printerid**. If **printerid** is blank, information on the currently selected printer is returned.

If `D_ENUM_PRINTERS` is specified, **printerid** is filled in with the name of the next printer from the Windows Print Manager list of installed printers. The first call retrieves the ID for the first installed printer, the second call retrieves the ID for the second installed printer, etc. When all printer IDs have been returned through this function, a call to `D_ENUM_PRINTERS` returns `D_EOLIST`, and **printerid** is cleared. (The next call to `D_ENUM_PRINTERS` will return the first printer ID again.)

`D_ENUM_WITH_INFO` works in the same way as `D_ENUM_PRINTERS`: the first call returns information on the first printer in the list, and so forth. But with `D_ENUM_WITH_INFO`, the entire *info_record* is returned.

If `D_ENUM_RESET` is specified, the `D_ENUM_PRINTERS` function is reset with the next `D_ENUM_PRINTERS` call returning the first printer ID.

The **flags** field contains bit flags for various printer characteristics. To test a flag, use **.band.** with the .defined value from **tools.def**. Currently, we provide the flag for proportional spacing only. To test for proportional spacing, use

```
proportional = (info_record.flags .band. D_PRT_PROPORTIONAL)
```

%U_PRINTSETUP – Display a Print Setup dialog box



status = %U_PRINTSETUP

Return value

status

On Windows, true if a printer was selected, or false if the user cancels or an error occurs.

On UNIX and OpenVMS, always returns false. (^VAL)

Discussion

The %U_PRINTSETUP function displays a standard Microsoft Print Setup dialog box, where the user can change the default printer or printer properties to which an LPQUE statement will print when the LPNUM qualifier is *not* specified.

Before %U_PRINTSETUP is called, the default printer is the printer selected as the default in “Devices and Printers” in Windows Control Panel. %U_PRINTSETUP does not change this default; the default is changed only for the life of the current Toolkit program or until %U_PRINTSETUP is called again. When a Toolkit program is run again, the default reverts to the printer specified in Devices and Printers. See “[Printing](#)” in the “Windows Development” chapter of the *Professional Series Portability Guide* for additional information.

Using the Print Setup dialog box

With the Print Setup dialog box ([figure 16-7](#)), the user can select a printer from the list setup in Print Manager, select form and orientation characteristics, and select 2-sided printing (if supported by the selected printer).

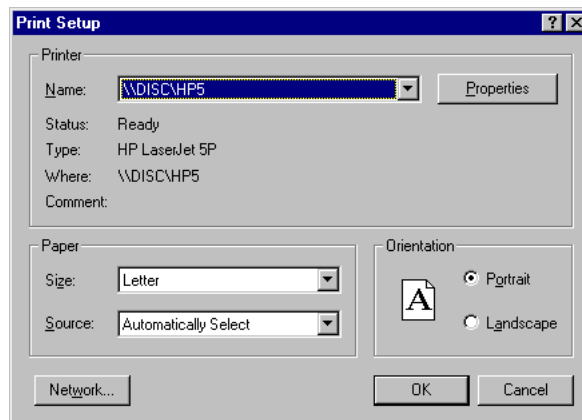


Figure 16-7. The Print Setup dialog box.

Selecting the Properties button displays the Document Properties dialog (figure 16-8), where the user can set up graphics resolutions, select color or monochrome printing, and set optimizations.

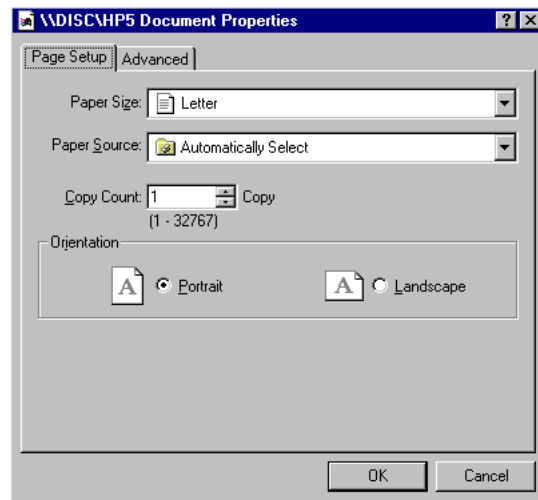


Figure 16-8. The Document Properties dialog box.

Selecting the Network button displays the Connect to Printer dialog box (figure 16-9). Here the user can specify a printer, view the available shared network printers, and view printer information such as the job description, status, and number of documents waiting.

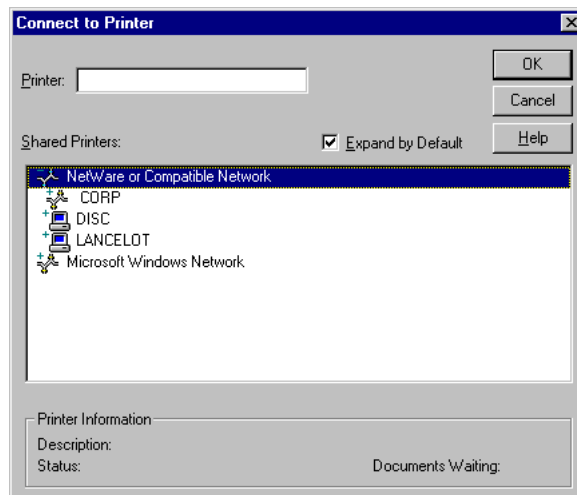


Figure 16-9. The Connect to Printer dialog box.

U_REDRAW – Clear screen and redraw it



```
xcall U_REDRAW
```

Discussion

The U_REDRAW subroutine clears the screen and redraws it from scratch on the next update.

Normally, you don't need to clear the screen and redraw it; as changes are made to the screen, the affected characters are repainted as needed.

The situations in which you would be most likely to use this subroutine are as follows:

- ▶ When you want to enable the user to redraw the screen with a key(s) other than CTRL+R.
- ▶ When your application is returning from a task done in the shell. (Since the windowing system does not keep track of shell activities, the windowing environment must be redrawn upon re-entry.)
- ▶ When a portion of your application uses display statements to the terminal, and you subsequently return to the UI Toolkit portion of your code.

U_RENDER – Process display renditions



```
xcall U_RENDER ([D_COPY | D_LOAD], [renditions, attr, color], [...],
& [D_PALETTE, palette#, backgrd, foregrd][, D_DSPLY_ALL, ctrl_record])
```

Arguments

D_COPY

(optional) Copy (retrieve) the renditions. **(n)**

D_LOAD

(optional) Set the renditions. (default) **(n)**

renditions

(optional) A display rendition. See the “[Rendition Code Definitions](#)” table below. **(n)**

attr

The attributes to set if loading (with D_LOAD) or the current attribute values if copying (with D_COPY). See the “[Attribute Flag Definitions](#)” table below. **(n)**

color

The palette number to use for the specified rendition if loading or the current color value if copying. **(n)**

D_PALETTE

(optional) Signifies that color palette codes will follow. **(n)**

palette#

Either the palette number to reference or **D_ALL_CODES** to indicate all palette codes. **(n)**

backgrd

Either the background color code (if *palette#* is a palette number) or the base of a 16-element array for all background color codes (if *palette#* is D_ALL_CODES). **(n)**

foregrd

Either the foreground color code (if *palette#* is a palette number) or the base of a 16-element array for all foreground color codes (if *palette#* is D_ALL_CODES). **(n)**

D_DSPLY_ALL

(optional) Signifies that all rendition controls will be accessed. **(n)**

ctrl_record

The name of a variable that holds the entire rendition control record. **(a425)**

Discussion

The U_REND subroutine retrieves (copies) or sets (loads) display renditions.

D_COPY and D_LOAD determine whether the codes that follow will define copy or load operations. D_COPY copies the current renditions into the specified variables (*attr* and *color*, *backgrd* and *foregrd*, or *ctrl_record*) so you can examine the current renditions and save them so you can reset them later. D_LOAD sets the renditions specified by the *attr* and *color*, *backgrd* and *foregrd*, or *ctrl_record* variables.

The table below lists the valid options for the *renditions* argument. Each rendition code must be followed by *attr* and *color* arguments. Attributes that aren't referenced in the *attr* argument are reset to off. (For information on the default state for a rendition, see [“Renditions” on page 18-25](#).)



On Windows, the environment overrides many of the renditions set with U_REND (as well as those set by Proto and U_EDITREND). See [“Renditions” on page 18-25](#) for information.

Rendition Code Definitions	
Rendition code	Definition
D_RND_MNU_ENA	Enabled Menu Entry
D_RND_MNU_DSA	Disabled Menu Entry
D_RND_MNU_SEL	Selected Menu Entry
D_RND_MNU_HQS	Menu Header Quick-Select
D_RND_MNU_EQS	Menu Entry Quick-Select
D_RND_INP_DSP	Input Field Display
D_RND_INP_PRC	Input Field Processing
D_RND_INP_CLR	Cleared Input Field
D_RND_INP_BCK	Input Window Background
D_RND_INP_TTL	Input Window Title
D_RND_INP_BDR	Input Window Border
D_RND_INP_DSA	Disabled Input Field
D_RND_INP_ROD	Read-Only Input Field Display
D_RND_INP_ROP	Read-Only Input Field Processing
D_RND_SEL_SEL	Select Wnd Selected Item

Rendition Code Definitions (Continued)	
Rendition code	Definition
D_RND_SEL_BCK	Selection Window Body (and unselected items)
D_RND_SEL_TTL	Selection Window Title
D_RND_SEL_BDR	Selection Window Border
D_RND_SEL_EQS	Sel Wnd Quick-Selects
D_RND_WIN_BCK	Generic Window Background
D_RND_WIN_TTL	General Window Title
D_RND_WIN_BDR	General Window Border
D_RND_MSG	Message Window Body
D_RND_MSG_BDR	Message Window Border
D_RND_HLP_BCK	Help Window Body
D_RND_HLP_TTL	Help Window Title
D_RND_HLP_BDR	Help Window Border
D_RND_HDR	Screen Header
D_RND_INF	Information Line
D_RND_FTR	Screen Footer
D_RND_DLG	Dialog Box Body
D_RND_DLG_BDR	Dialog Box Border
D_RND_LST_ENA	Enabled List
D_RND_LST_ENA_B	Enabled List Border
D_RND_LST_ENA_T	Enabled List Title
D_RND_LST_HDR	Enabled List Header
D_RND_LST_FTR	Enabled List Footer
D_RND_LST_SEL	Selected Item in List (used in place of D_RND_INP_DSP, D_RND_INP_CLR, or D_RND_INP_BCK if the input window is associated with a list)

Rendition Code Definitions (Continued)	
Rendition code	Definition
D_RND_LST_DSA	Disabled List (body, border, and title)
D_RND_LST_DSA_S	Disabled Selected Item
D_RND_SCR_BAR	Scroll Bar
D_RND_SCR_IND	Scroll Bar Indicators
D_RND_USER_1	User Rendition 1
D_RND_USER_2	User Rendition 2
D_RND_USER_3	User Rendition 3
D_RND_USER_4	User Rendition 4
D_RND_USER_5	User Rendition 5

The table below lists the flags you can pass as the *attr* argument. Each flag consists of a series of codes: B=blinking (italic on Windows), H=highlighted (bold), R=reverse, and U=underlined. Note that you can substitute I for B (for example, A_IR instead of A_BR).

Attribute Flag Definitions	
Attribute flag	Definition
A_NONE	No attributes.
A_U	Underlined.
A_R	Reverse.
A_RU	Reverse, underlined.
A_B	Blinking (on UNIX and OpenVMS) or italic (on Windows).
A_BU	Blinking, underlined.
A_BR	Blinking, reverse.
A_BRU	Blinking, reverse, underlined.
A_H	Highlighted.
A_HU	Highlighted, underlined.

Attribute Flag Definitions (Continued)	
Attribute flag	Definition
A_HR	Highlighted, reverse.
A_HRU	Highlighted, reverse, underlined.
A_BH	Blinking, highlighted.
A_BHU	Blinking, highlighted, underlined.
A_BHR	Blinking, highlighted, reverse.
A_BHRU	Blinking, highlighted, reverse, underlined.
A_ALL	All attributes (same as A_BHRU).

On Windows, U_REND can accept or return a color number from 0 through 511 for the *backgrd* and *foregrd* arguments, so variables for these arguments must be at least an **i2** or **d3**. We recommend using an **i4**.

If you want to save two different sets of display renditions, you can use the D_DSPLY_ALL argument to recall an entire set of renditions. D_DSPLY_ALL requires a single argument that is used to access the entire rendition control information. If D_COPY was specified, the *ctrl_record* variable must be large enough to hold the control record (**a425**).

See also

- ▶ [U_EDITREND on page 16-42](#) for information about creating and modifying rendition schemes.
- ▶ [“Customizing the Look of Your Application” on page 18-24.](#)

Examples

The first example sets the blinking attribute for the information line.

```
xcall u_rend(D_LOAD, D_RND_INF, A_B)
```

The second example sets the highlighted and underlined attributes for selected menu entries.

```
xcall u_rend(D_LOAD, D_RND_MNU_SEL, A_HU)
```

The following example saves the current rendition of the dialog box, changes the dialog box's rendition for the next U_DIALOG subroutine, and then restores the previous rendition.

```
xcall u_rend(D_COPY, D_RND_DLG, att_sav, clr_sav)
xcall u_rend(D_LOAD, D_RND_DLG, att_new, clr_new)
xcall u_dialog(. . .)
xcall u_rend(D_LOAD, D_RND_DLG, att_sav, clr_sav)
```

The final example copies all rendition controls into **rendctl**, where **rendctl** is at least 404 bytes in size. You might use a subroutine like this to save all the current renditions so you can reset them later.

```
xcall u_rend(D_COPY, D_DSPLY_ALL, rendctl)
```

U_RESIZE – Resize the screen



```
xcall U_RESIZE ([height][, width])
```

Arguments

height

(optional) The new screen height in rows. (**n**)

width

(optional) The new screen width in columns. (**n**)

Discussion

The U_RESIZE subroutine resizes the screen without reinitializing the windowing system.

If you specify only one of the arguments (either *height* or *width*), the other dimension will remain unchanged.

If the *height* is changed, only the screen body will be affected.

If the *width* is changed, all the screen sections will be resized. U_RESIZE will also rejustify any centered (D_CENTER) or right-justified (D_RIGHT) text within the new dimensions. If re-justifying causes any text to overlap, the centered text will prevail, truncating the right- or left-justified text.

Placement of any windows (other than the screen sections) will remain unaffected, except as limited by the window system.

On Windows, you can control the font used by the application window. See [“Using Fonts on Windows” on page 18-40](#).

Examples

The first example sets the number of screen rows to 25, and the number of screen columns to 132.

```
xcall u_resize(25,132)
```

The second example sets the number of screen columns to 80 and leaves the number of screen rows unaffected.

```
xcall u_resize(,80)
```

U_RESIZESB – Resize a scroll bar window



```
xcall U_RESIZESB (scroll_id, new_length)
```

Arguments

scroll_id

The ID of a scroll bar window. **(n)**

new_length

The new length in characters for the scroll bar window. **(n)**

Discussion

The U_RESIZESB subroutine changes the length of a scroll bar window. It changes the height of a vertical scroll bar, or it changes the width of a horizontal scroll bar.

Scroll_id must be a window ID returned by U_CREATESB.

See also

[U_CREATESB](#) on page 16-22 for information on creating scroll bar windows.

U_RLSCHN – Release a channel



```
xcall U_RLSCHN(channel)
```

Arguments

channel

The channel to release. (**n**)

Discussion

The U_RLSCHN subroutine is used to release a channel previously allocated with the U_GETCHN subroutine.

Do *not* use U_RLSCHN to deallocate a channel that was allocated with U_OPEN. Use U_CLOSE instead.

Examples

This example releases the channel contained in **chn**.

```
xcall u_rlschn(chn)
```


U_RSTREND – Restore renditions



```
xcall U_RSTREND ([filename], [scheme][, status])
```

Arguments

filename

(optional) The name of the rendition file. **(a)**

scheme

(optional) The rendition scheme name. If not specified or blank, the currently loaded renditions are used. **(a)**

status

(optional) Returned with zero (0) if the restore operation is successful, or with a Synergy DBL error number if it is unsuccessful. **(n)**

Discussion

The U_RSTREND subroutine restores a rendition scheme from a rendition file.

U_RSTREND opens *filename* in **I:I** mode and closes it upon exit.

If *filename* does not exist and *status* is not passed, errors during the open are ignored and no renditions are restored. If *filename* is not specified, UI Toolkit searches for the rendition file as follows:

- ▶ If the environment variable DTKRNDFIL is defined, the rendition filename is DTKRNDFIL's value.
- ▶ If DTKRND is defined instead of DTKRNDFIL, the rendition file is opened as **DTKRND:dtkrnd.ism**.
- ▶ If neither environment variable is set, **dtkrnd.ism** is opened in the current directory.

If Toolkit can't find the renditions file, U_RSTREND does nothing. Additionally, if Toolkit can't open a renditions file, no error occurs and no action is taken. You can use *status* to find out if the operation failed or was successful.

Once a file is opened, U_RSTREND loads the rendition scheme. If *scheme* is passed, it is used as the key (which is not case sensitive) to the file. If *scheme* is not passed, a blank key is used. A blank key marks the current rendition scheme. If the desired scheme cannot be found in the file, no error occurs and no renditions are modified. You can use *status* to find out if the operation failed or was successful.

Once the rendition scheme is loaded, the rendition definitions are used for subsequent screen displays.

See also

[“Customizing the Look of Your Application” on page 18-24.](#)

Examples

The following utility subroutine restores and stores personalized rendition schemes from a rendition file.

```
subroutine pers_rend
  a_scheme    ,a
  a_rstflg    ,n      ;Restore flag: 1=restore, 0=save

  .include "WND:tools.def"
  .define RENDFILE  , "RND:persrnd"

  record
    error      ,i4

  proc
    if (a_rstflg) then
      xcall u_rstrend(RENDFILE, a_scheme, error)
    else
      xcall u_savrend(RENDFILE, a_scheme, error, TRUE)
    if (error)
      xcall proc_error
    xreturn
  endsubroutine
```

U_SAVE – Save a window to a window library



```
xcall U_SAVE(channel, window_id, window_name[, error])
```

Arguments

channel

The I/O channel to use. This channel must already be open in update mode. **(n)**

window_id

The ID of the window to be saved. **(n)**

window_name

The window name. **(a)**

error

(optional) Returns false (0) if the save was successful. If the save failed, this argument returns the error number of the resulting Synergy DBL error. **(n)**

Discussion

The U_SAVE subroutine saves a window to a window library.

If you save a window that already exists in the window library open on *channel*, the new window replaces the existing one.

See also

The “[Error Messages](#)” chapter of *Synergy Tools* for information about Synergy DBL error numbers.

Examples

The following example saves the window whose ID is stored in **help_id** to a sequential file or window library on the channel stored in **help_chan**. If the name stored in **help_verb** is already present, the existing window will be overwritten.

```
xcall u_save(help_chan, help_id, help_verb)
```

%U_SAVELOG – Save an initialization setting to synergy.ini or synuser.ini

WT			

```
status = %U_SAVELOG (setting_name, [value], [section][, all_users])
```

or

```
xcall U_SAVELOG (setting_name, [value], [section][, all_users])
```

Return value

status

On Windows, true if the function is successful, or false if it is not. On UNIX, always returns false. On OpenVMS, always returns true. (^VAL)

Arguments

setting_name

The name of the initialization setting. (**a**)

value

(optional) The value to save for the logical. (**a** or **n**)

section

(optional) The section of the initialization file you want to save the setting in. (**a**)

all_users

(optional) One of the following: (**n**)

TRUE Save initialization setting to **synergy.ini**.

FALSE Save initialization setting to **synuser.ini**. (default)

Discussion

The U_SAVELOG subroutine writes an initialization setting (*setting_name* and *value*) to **synergy.ini** or **synuser.ini**. The setting does not take effect until the next time the runtime is started. (Chaining does not cause Toolkit to re-read **synergy.ini** or **synuser.ini**.) If the function fails, which causes *status* to be returned as false, you can use %SYSERR to retrieve the system-specific error code.

If *value* is numeric, it is converted to a string. If *value* is not passed, the current value of the initialization setting, *setting_name*, is used.

If “synergy” is specified for *section*, this symbol will apply to all Synergy programs and tools. If a program name is passed, it will only apply to that program. If *section* is not passed, the name of the currently executing **.dbr** file is used. *Section* is case insensitive. If *section* doesn’t exist, it will be created.

If the file (**synergy.ini** or **synuser.ini**) Toolkit is attempting to write to does not exist, it will be created. The **synergy.ini** file will be created in the directory specified by SFWINIPATH if it’s set. If this variable isn’t set, **synergy.ini** will be created in the directory specified by DBLDIR. The **synuser.ini** file will be created in the user’s application data directory. For more information, see [“Synergy initialization files”](#) in the “Environment Variables” chapter of *Environment Variables & System Options*.

Examples

The following example writes the initialization setting “APP_STATE=MAXIMIZED” to the [synergy] section of **synuser.ini**. (Note that *all_users* isn’t passed.)

```
xcall u_savelog("APP_STATE", "MAXIMIZED", "synergy")
```

%U_SAVESETTINGS – Save application window settings to synergy.ini or synuser.ini

WT			

```
status = %U_SAVESETTINGS ([flags])
```

or

```
xcall U_SAVESETTINGS ([flags])
```

Return value

status

On Windows, true if the function is successful, or false if it is not. On UNIX, always returns false. On OpenVMS, always returns true. (^VAL)

Arguments

flags

(optional) One or more of the following, combined using .BOR. or the vertical bar (|) operator, to indicate where to save window settings. If not passed, settings are saved to **synuser.ini**. (n)

D_APP_ALLUSERS	Save the settings to synergy.ini .
D_APP_STATE	Save the application window state.
D_APP_POS	Save the application position.
D_APP_SCALE	Save the scaling percentage.
D_APP_SIZE	Save the application height and width.

Discussion

The %U_SAVESETTINGS function saves the state, position, scaling percentage, and size of the application window to **synuser.ini** or, if you pass D_APP_ALLUSERS, **synergy.ini**. For the size, %U_SAVESETTINGS saves the size at U_START or as of the last call to U_RESIZE. For state, position, and scaling percentage, %U_SAVESETTINGS saves current settings. Settings are saved to the section in the initialization file that's labeled with the name of the currently executing .dbr file.

If the file (**synergy.ini** or **synuser.ini**) that Toolkit is attempting to write to does not exist, it will be created. The **synergy.ini** file will be created in the directory specified by SFWINIPATH if it's set. If it isn't set, **synergy.ini** will be created in the directory specified by DBLDIR. The **synuser.ini** file will be created in the user's application data directory. For more information, see [“Synergy initialization files”](#) in the “Environment Variables” chapter of *Environment Variables & System Options*.

If %U_SAVESETTINGS fails (which causes *status* to be returned as false), you can use %SYSERR to retrieve the system-specific error code.

See also

- ▶ [%U_SAVELOG](#) on page 16-128 for information on saving other initialization settings to **synergy.ini** or **synuser.ini**.
- ▶ [APP_X](#), [APP_Y](#), [APP_HEIGHT](#), [APP_WIDTH](#), and [APP_SCALE](#) in the “Environment Variables” chapter of *Environment Variables & System Options*.

Examples

The following example saves the application size (as of U_START or the last U_RESIZE) to **synergy.ini**.

```
xcall u_savesettings(D_APP_SIZE|D_APP_ALLUSERS)
```

The next example does the same, but by calling U_RESIZE with the current application window height and width, it causes %U_SAVESETTINGS to save the current size of the application window to **synergy.ini**.

```
stack record
    wnd_handle ,i4
    group rect ,a
        left ,i4
        top ,i4
        right ,i4
        bottom ,i4
    endgroup
    row_num ,i4
    col_num ,i4
```

```
;Get the window handle for the application window.
wnd_handle = %W_INFO(WIF_HWND, WID_APP)
```

```
;Get coordinates for client area of application window.
xcall u_winmetrics(D_CLIENT_RECT, wnd_handle, rect)
```

```
;Get number of last row.
row_num = %u_winmetrics(D_YTOROW, WID_APP, bottom-1)
```

```
;Get number of last column.
col_num = %u_winmetrics(D_XTOCOL, WID_APP, right-1)
```

```
;Call U_RESIZE with current height & width so app window doesn't change)
xcall u_resize (row_num, col_num)
```

```
;Save application window position and size to synergy.ini.
xcall u_savesettings(D_APP_SIZE|D_APP_ALLUSERS)
```

U_SAVREND – Save a rendition scheme to a file



```
xcall U_SAVREND ([filename], [scheme], [status][, flag])
```

Arguments

filename

(optional) The name of the rendition file. **(a)**

scheme

(optional) The rendition scheme name (a maximum of 30 characters). **(a)**

status

(optional) Returned with zero (0) if the save operation is successful, or a Synergy DBL error number if it is unsuccessful. **(n)**

flag

(optional) If true (non-zero), U_SAVREND prompts the user before replacing an existing scheme of the same name. If false, it is replaced without warning. **(n)**

Discussion

The U_SAVREND subroutine saves a rendition scheme to a rendition file.

U_SAVREND opens *filename* in **U:I** mode and closes it upon exit. If *filename* doesn't exist, it is created.

If *filename* is not specified, UI Toolkit searches for the rendition file as follows:

- ▶ If the environment variable DTKRNDFIL is defined, the rendition filename is DTKRNDFIL's value.
- ▶ If DTKRND is defined instead of DTKRNDFIL, the rendition file is opened as **DTKRND:dtkrnd.ism**.
- ▶ If neither environment variable is set, **dtkrnd.ism** is opened or created in the current directory.

If *scheme* is not specified or is blank, the currently loaded renditions are saved to the working rendition scheme (blank key).

See also

[“Customizing the Look of Your Application” on page 18-24.](#)

Examples

The following utility subroutine loads and saves personalized rendition schemes from a rendition file.

```

subroutine pers_rend
    a_scheme      ,a
    a_rstflg      ,n      ;Restore flag: 1=restore, 0=save
    .include "WND:tools.def"
    .define RENDFILE  ,"RND:persrnd"

    record
        error      ,i4

    proc
        if (a_rstflg) then
            xcall u_rstrend(RENDFILE, a_scheme, error)
        else
            xcall u_savrend(RENDFILE, a_scheme, error, TRUE)
        if (error)
            xcall proc_error
        xreturn
    endsubroutine

```

%U_SELPALETTE – Select a color palette entry



```
user_selection = %U_SELPALETTE (initial_selection)
```

Return value

user_selection

On Windows, the number of the palette entry chosen by the user, or false if canceled. On UNIX and OpenVMS, always returns false. (^VAL)

Arguments

initial_selection

The number (in the range 1-16) of the palette entry you want selected when the window opens. If the value passed is invalid, an error message is displayed and the function immediately returns 0. (n)

Discussion

A call to %U_SELPALETTE opens a selection window from your application that enables users to select a Synergy color combination. Typically this is used to customize user preferences. See [figure 16-10](#).

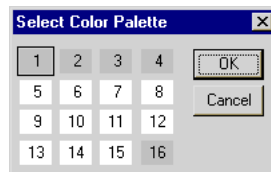


Figure 16-10. Palette selection window.

Each numbered square in the palette window corresponds to a color palette entry in the Synergy runtime and displays the foreground and background color for the entry. (The color of the number is the foreground color. The color of the box is the background color.) The current palette settings determine the initial colors, and the *initial_palette* argument specifies the initial selection. A thin-line box surrounds the selected entry. In [figure 16-10](#), for example, entry 1 is selected.

The user selects the entry and clicks OK or presses ENTER. %U_SELPALETTE then returns the number for the selected entry.

%U_SELPALETTE returns 0 if the Cancel button is clicked, if the close button in the caption bar is clicked, or if ESC is pressed while the selection window is active.

Examples

In the following example, the variable **color** refers to the color of a background window (whose ID is **backwnd**) placed within the application window. This section of code enables the user to select a new color and apply it.

```
.align
stack record
    color      ,i4
    newcolor   ,i4
    retvals    ,6d4
    backwnd    ,i4

proc
    xcall w_info(WI_ATTRS, backwnd, retvals) ;Get current window color
    color = retvals(2)                      ;Window color palette
    if ((newcolor = %u_selpalette(color) .and.
        & (newcolor .ne. color))            ;New color selected?
    begin
        color = newcolor                    ;Apply the change
        xcall w_area(backwnd, WA_COPY, WAC_WTOP,
            & WA_COLOR, color)              ;Change the window area
        xcall w_disp(backwnd, WD_COLOR, color) ; and the current
                                                ; display color.
    end
```

See also

- ▶ [U_REND](#) on page 16-116.
- ▶ The [WP_PALET](#) function for W_PROC in the “Synergy Windowing API” chapter of the *Synergy DBL Language Reference Manual*.
- ▶ The [WA_COLOR](#) function for W_AREA in the “Synergy Windowing API” chapter of the *Synergy DBL Language Reference Manual*.
- ▶ “[Colors and the color palette](#)” in the “Synergy Windowing API” chapter of the *Synergy DBL Language Reference Manual*.

U_START – Initialize the UI Toolkit environment



```
xcall U_START ([util_lib], [header_lines], [footer_lines], [pool_size], [screen_rows],
&             [screen_columns], [text_lib], [first_channel], [#env_levels], [last_channel]
&             [, start_method[, elb]])
```

Arguments

util_lib

(optional) The utility (window) library filename. **(a)**

header_lines

(optional) The number of lines (in the range 0 to 4) in the screen header section. If not passed or 0, the header section is not created or displayed on the screen. **(n)**

footer_lines

(optional) The number of lines (in the range 0 to 4) in the screen footer section. If not passed or 0, the footer section is not created or displayed on the screen. **(n)**

pool_size

(optional) The size of the windows pool, in bytes. This argument is deprecated for Synergy DBL version 5 and higher, and is therefore ignored. **(n)**

screen_rows

(optional) The number of rows on the screen. **(n)**

screen_columns

(optional) The number of columns on the screen. **(n)**

text_lib

(optional) The text library filename. **(a)**

first_channel

(optional) The number (from 1 to 253) of the first channel available. If not passed, defaults to 1. **(n)**

#env_levels

(optional) The number (from 2 to 32) of allowed environment levels. If not passed, defaults to 16 to conserve memory. **(n)**

last_channel

(optional) The number (from 3 to 255) of the last channel available. If not passed or greater than 255, defaults to 255. **(n)**

start_method

(optional) The name of a USTART_METHOD subroutine you have written. **(a)**

elb

(optional) The name of the library from which *start_method* will be drawn. If not passed, Toolkit searches for *start_method* in the program image or previously opened ELBs. **(a)**

Discussion

U_START initializes the UI Toolkit environment. It should be the first call in your application's Toolkit layer. It detects and records (but does not change) any pre-existing environment states. If you want to modify these states, use E_STATE.

If *util_lib* is present and non-blank, the specified library

- ▶ is opened in **I:I** mode with the “nolock” (**NL**) option set.
- ▶ is promoted to global status.
- ▶ has its allocated channel placed in the **g_utlib** field (defined in **tools.def**).

On Windows, if *header_lines* is specified as 1 (one) line, the Windows environment application “caption” is used to display the header instead of the typical Toolkit header section. If the header is more than one line in size, an area is still allocated within the application window. The text for a Windows application window caption can be explicitly referenced using the D_CAPTION argument flag in the E_SECT subroutine.

The default value for *screen_rows* is 24 (25 on Windows). The default value for *screen_columns* is 80. (You cannot call U_START to resize the screen.) On Windows, if *screen_rows* or *screen_columns* is not passed, UI Toolkit uses the APP_WIDTH and APP_HEIGHT initialization settings to determine the screen size. If those are not specified, the size is 25 rows by 80 columns.

The *text_lib* argument enables you to use translated text in your Toolkit applications by specifying a text message file. This text message file, or library, is an ISAM file that will be opened in read-only mode. If you don't specify *text_lib*, UI Toolkit uses **SYNTXT:syntxt.ism** by default. If not found, UI Toolkit uses **DBLDIR:syntxt.ism**.

If *first_channel* and *last_channel* are passed, U_START closes only channels in use in the range defined by these arguments. If they are not passed, U_START closes all channels in use from 1 to 255, which could close channels in use by other Synergy programs that are using the runtime. To prevent this, specify a range. For information on how this could affect xfServerPlus, see [“Removing User Interface Elements”](#) in the “Preparing Your Synergy Server Code” chapter of the *xfNetLink & xfServerPlus User's Guide*.

If you specify *first_channel* and *last_channel*, any channel previously in use within the range specified by these arguments will be purged. If you don't specify these arguments, UI Toolkit defaults to “owning” the entire range (1 to 255).

First_channel is the first channel number available to Toolkit. If not passed, it defaults to 1. This will also be the channel used for the terminal unless *W_INIT* has been called previously and specifies another channel. All channel numbers less than *first_channel* will be logged as “system” channels to Toolkit, and should not be referenced again through any Toolkit subroutine.

Last_channel is the last channel number available to Toolkit. Since Toolkit requires at least 3 channels, *last_channel* should never be less than 3. If it is not passed or more than 255, it defaults to 255. All channel numbers greater than *last_channel* will be logged as “system” channels to UI Toolkit, and should not be referenced again through any Toolkit subroutine.

If *start_method* is passed, just prior to returning to the calling routine, U_START calls this method to allow application-specific start-up processing. Note that if DTKDBG is set, the Toolkit debugger checks to make sure the method can be invoked (see [DTKDBG](#) in the “Environment Variables” chapter of *Environment Variables & System Options*).

See also

- ▶ The “[Renditions](#)” table on [page 18-26](#) for information on initial rendition settings established by U_START.
- ▶ [U_FINISH](#) on [page 16-46](#) for information about closing and purging channels at application shut down.
- ▶ [U_OPEN](#) on [page 16-106](#) for information on opening channels.
- ▶ [U_CLOSE](#) on [page 16-18](#) for information on closing or purging channels.
- ▶ [U_GETCHN](#) on [page 16-59](#) for information about how to allocate a channel (usually used only to convert applications).
- ▶ [E_STATE](#) on [page 7-41](#) for information about modifying the environment states.
- ▶ [E_SECT](#) on [page 7-38](#) for information about how to display text in screen sections.
- ▶ [USTART_METHOD](#) on [page 16-236](#) for information about writing your application start-up method.

Examples

In the following example, U_START opens the window library **WND:util** and initializes the Toolkit environment. The header section has 3 lines, and the footer section has 2 lines.

```
xcall u_start("WND:util", 3, 2)
```

The following example opens a window library called **WND:util** and initializes the Toolkit environment. The header and footer sections are not created.

```
xcall u_start("WND:util")
```

In this example, the opened window library is also called **WND:util**. The header and footer screen sections are not created, the screen has 25 rows, and text messages are located in **WND:txtlib**.

```
xcall u_start("WND:util",,,, 25,, "WND:txtlib")
```

U_STOREND – Store a rendition scheme



```
xcall U_STOREND (channel, [scheme], [status][, flag])
```

Arguments

channel

The channel on which the rendition file is opened. **(n)**

scheme

(optional) The name of the rendition scheme (a maximum of 30 characters). **(a)**

status

(optional) Returned with zero (0) if the store operation is successful, or a Synergy DBL error number if it is unsuccessful. **(n)**

flag

(optional) If true (non-zero), U_STOREND prompts the user before replacing an existing rendition scheme of the same name. If false, it is replaced without warning. **(n)**

Discussion

The U_STOREND subroutine stores a rendition scheme to an open rendition file.

Channel must be open in **U:I** mode.

If *scheme* is not passed or is blank, the currently loaded renditions are stored as the working rendition scheme (blank key).

See also

[“Customizing the Look of Your Application” on page 18-24.](#)

Examples

The following utility subroutine loads and stores personalized rendition schemes from a rendition file.

```
subroutine pers_rend
  a_scheme      ,a
  a_ldstflg     ,n      ;Load flag: 1=restore, 0=save

  .include "WND:tools.def"

  global data section mygb
  record
```

Utility Routines

U_STOREND

```
        rendchn      ,i4
endglobal

record
    error            ,i4

proc
    if (a_ldflg) then
        xcall u_ldrend(rendchn, a_scheme, error)
    else
        xcall u_storend(rendchn, a_scheme, error, TRUE)
        if (error)
            xcall proc_error
        xreturn
    endsubroutine
```


U_TEXT – Load and modify text library entries



```
xcall U_TEXT (channel, facility, message#, text, [length], [no_shortcuts][, parameter, ...])
```

Arguments

channel

The channel on which the text library is opened. (**n**)

facility

The desired facility name. (**a**)

message#

The number of the desired message. (**n**)

text

Returned with the message text. (**a**)

length

(optional) Returned with the length of the message text. (**n**)

no_shortcuts

(optional) The true/false flag that tells U_TEXT whether to prohibit shortcut text mapping. (**n**)

parameter

(optional) One or more (up to 10) S_BLD parameters. (**a** or **n**)

Discussion

The U_TEXT subroutine retrieves a text entry from a text message file, optionally performs an S_BLD on it, and optionally replaces menu entry names with their shortcut text. (Note that the S_BLD occurs before the entry name replacement.)

If no *parameters* are passed, Toolkit will not perform an S_BLD. This avoids errors which might occur if a “%” is in the text.

If *no_shortcuts* is passed and true, then no mapping of entry names to shortcut text will occur. Otherwise, M_RPLENT is called with the text after any S_BLD.

If *length* is passed, it is returned with the trimmed length of the resulting text. The text is returned in *text*.

Text message libraries may be created and maintained by using the Synergy UI Toolkit Control Panel. See [“The Synergy UI Toolkit Control Panel”](#) in the “General Utilities” chapter of *Synergy Tools* for more information.

See also

- ▶ [S_BLD](#) in the “System-Supplied Subroutines and Functions” chapter of the *Synergy DBL Language Reference Manual* for an explanation of various parameters.
- ▶ [M_RPLENT](#) on page 10-48 for a description of the replacements which can be performed or avoided.

Examples

The following example reads the text library which is open on the channel **mylib**, looking for a text message with facility “GEN” and a message number defined by D_EXITINFO. The text of the message is scanned for any entry names, and if they are found they are replaced by their corresponding shortcut text. The resulting text message is returned in **info**, and its trimmed length is returned in **infoflen**.

```
xcall u_text(mylib, "GEN", D_EXITINFO, info, infoflen)
```

In the second example, the text library is searched for message facility “ERR” and message code D_CANTWRITE. The text of this message is passed on to S_BLD, with **chnl** and **%ernum** as additional arguments to S_BLD. The text will not be scanned for menu entry replacement. The resulting text is returned in **msg**, and its trimmed length is returned in **msglen**.

```
xcall u_text(mylib, "ERR", D_CANTWRITE, msg, msglen,  
&                TRUE, chnl, %ernum)
```

U_UPDATE – Update screen



```
xcall U_UPDATE([auto_update])
```

Arguments

auto_update

(optional, Windows only) The update flag: **(n)**

TRUE Turns on updates. (default)

FALSE Turns off updates.

Discussion

A call to U_UPDATE causes an immediate screen update. Because input processing updates the screen, U_UPDATE is necessary only if you want to update the screen without input. For example, U_UPDATE can be used to display a “Please wait” message during processing.

On UNIX and OpenVMS, *auto_update* is ignored.

On Windows, a call to U_UPDATE(TRUE) is equivalent to a call to U_UPDATE() (without an argument). Both cause an immediate screen update and reinstate screen updating if it’s been turned off by U_UPDATE(FALSE). Screen updating is the default, so U_UPDATE(TRUE) isn’t necessary unless you want an immediate update or you’ve turned screen updating off with U_UPDATE(FALSE). A call to U_UPDATE(FALSE) suspends screen updating until the next call to U_UPDATE(), U_UPDATE(TRUE), W_UPDT, or a Toolkit or Synergy windows input routine.



Use U_UPDATE(FALSE) only if you expect rapid changes to the display and you know an update will soon follow (for example, when entering or exiting environments and placing or removing menu columns and other windows). If updates are turned off for too long, there may be repainting problems—for example, “bleed through,” which may happen if windows for other applications are updated while updates are turned off for your application.



To prevent menu flicker on Windows, you can use U_UPDATE(FALSE) prior to every call made to the E_EXIT subroutine.

Examples

The following example prevents menu flicker when the system is removing columns.

```
xcall e_enter
xcall m_1dcol(id1,, "col_1")
xcall m_1dcol(id2,, "col_2")
.
```

```
.  
. (processing loop)  
.   
.   
.   
xcall u_update(FALSE)  
xcall e_exit
```

To set the *auto_update* flag for all E_ENTER and E_EXIT subroutine calls, you can write your own version of the E_EXIT and E_ENTER subroutines as follows:

```
subroutine my_enter  
  a_shr ,n  
proc  
  xcall u_update(FALSE)  
  xcall e_enter(a_shr)  
  xreturn  
endsubroutine  
  
subroutine my_exit  
proc  
  xcall u_update(FALSE)  
  xcall e_exit  
  xreturn  
endsubroutine
```

U_UPDATESB – Update a scroll bar window



```
xcall U_UPDATESB (scroll_id, maximum, current)
```

Arguments

scroll_id

The ID of the scroll bar window. **(n)**

maximum

The new maximum value. **(n)**

current

The new current value. **(n)**

Discussion

The U_UPDATESB subroutine updates a scroll bar window in UNIX and OpenVMS environments. (This routine does work on Windows, but the results are non-standard.)

A scroll bar's progress/regress indicator shows the relationship of what is currently displayed versus the total possible. (For example, you may only be viewing 10 out of a possible 100 entries in a list.) This relationship is referred to as the “ratio of current to maximum.”

The ratio of current to maximum is used to determine the placement of the progress/regress indicator within the scroll bar window. There are a few special cases where this ratio is superseded:

- ▶ If *maximum* is 0, the indicator is not placed.
- ▶ If *current* is 1, the indicator is always placed at position 1.
- ▶ If *current* is equal to *maximum*, the indicator is always placed at the last position.
- ▶ If the ratio of *current* to *maximum* results in an indicator position outside the window (for example, less than 1 or greater than the window size), the indicator's position will be limited to those bounds.

The endpoint indicators, if used, are displayed according to the following rules:

- ▶ If *current* is 1 or less, the “no more at beginning” indicator is displayed, otherwise the “more at beginning” indicator is displayed.
- ▶ If *current* is equal to or greater than *maximum*, the “no more at end” indicator is displayed, otherwise the “more at end” indicator is displayed.

See also

- ▶ [U_CREATE_SB](#) on page 16-22 for information about how to create a scroll bar window.
- ▶ [U_CHARSB](#) on page 16-12 for information about how to copy or load scroll bar characters.

Examples

The following example would place the indicator at position 1. This could indicate that the user is viewing the first part of the list. If present, the endpoint indicators would be showing “no more at beginning” and “more at end.”

```
xcall u_updatesb(scrlid, 50, 1)
```

The second example updates the indicator which could reflect that the user is viewing approximately the middle of the list. If present, the endpoint indicators would be showing “more at beginning” and “more at end.”

```
xcall u_updatesb(scrlid, 50, 25)
```

U_VERSION – Return Toolkit version number



```
xcall U_VERSION([version][, version_full])
```

Arguments

version

(optional) Returned with the current Toolkit version with a one-digit version number. **(a)**

version_full

(optional) Returned with the current Toolkit version with a two-digit version number. **(a)**

Discussion

The U_VERSION subroutine returns the current UI Toolkit version. If your application is linked against **tklib.elb**, **tklib_qcheck.elb**, **TKLIB_SH.EXE**, or if it uses **Synergex.SynergyDE.tklib.dll**, *version* and *version_full* reflect the version of the ELB or DLL in use.

Version is returned in *v.r.mm* form, where *v* is the version number, *r* is the major revision number, and *mm* is the minor revision number. For the version number (*v*), versions higher than 9 are represented by Latin letters (A, B, etc.).

Version_full is in *vv.r.mm* form, where *vv* is the version number, *r* is the major revision number, and *mm* is the minor revision number. The version number (*vv*) is in decimal notation, so versions higher than 9 are represented by two decimal digits (10, 11, etc.). Version 9 is represented by a single decimal digit preceded by a space (i.e., “ 9”).

Examples

In the following example, *vers* is returned with the number of the UI Toolkit version currently in use. For example, if the current version is 9.1, *vers* will contain “9.1.1”:

```
xcall u_version(vers)
```

If the current version is 10.1, *vers* in the following will contain “A.1.1”, and *vers_full* will contain “10.1.1”:

```
xcall u_version(vers, vers_full)
```

U_WAIT – Wait for a user response



```
xcall U_WAIT (/D_BELL|D_NOBELL/, /D_ALERT|D_ERROR/)
```

Arguments

D_BELL

(optional) Ring the terminal bell. (default) **(n)**

D_NOBELL

(optional) Don't ring the terminal bell. **(n)**

D_ALERT

(optional) Do alert processing. (default) **(n)**

D_ERROR

(optional) Do error processing. **(n)**

Discussion

The U_WAIT subroutine displays a message in a window and waits for a user response. The waiting is either alert-style (D_ALERT) or error-style (D_ERROR). The primary difference between the two styles is how you want the message to be acknowledged.

An *alert state* is a normal part of processing a given function. For example, it could be a completion message after some lengthy processing has completed. Alert-style processing displays a message (“Press RETURN to continue” by default) and waits for the user to press RETURN (ENTER).

An *error state* is an exception condition. It does not occur unless an invalid status exists. In this case, you want the user to read the displayed message and then correct the error condition (entry of invalid data, for example). You don't want a simple RETURN (ENTER) to terminate the error state because users often enter data at the terminal before it has actually been requested (referred to as “type-ahead”). If the user receives an error condition with type-ahead pending, the first RETURN in the type-ahead data would inadvertently terminate the error condition. The data in any additional type-ahead would then be entered where the error originally occurred, either causing another error or entering valid but incorrect data.

For this reason, error-style processing purges all type-ahead data before displaying an error message. By default, this message is “Press ^G to continue” on UNIX and OpenVMS and “Press RETURN to continue” on Windows. It then waits for a response. On UNIX and OpenVMS, the terminal bell rings each time an incorrect continuation character is entered. (The terminal bell will not ring if **g_beep** is set to false. See [g_beep on page 18-2](#).)

UNIX, VMS

On UNIX and OpenVMS, U_WAIT looks for an existing wait window called `u_wait_msg`. If it can't find this window, it looks in the window library opened on `g_utlib`. If it can't find `u_wait_msg` in the library (or a window library is not defined), U_WAIT creates a one-line window of that name and places it three lines above the bottom of the screen body. The wait message will be displayed in this window.

If you create your own `u_wait_msg` window, you can control the window's vertical placement, but not the horizontal placement. Toolkit recognizes a vertical placement setting for this window but ignores a horizontal placement setting.

WIN

In Synergy .NET, Toolkit uses a cell-based-style message box if `g_netnativmsgbox` (in `tools.def`) is set to false, which is the default. If `g_netnativmsgbox` is set to true, Toolkit uses a native Windows message box.

In traditional Synergy, U_WAIT creates a Windows-style message box that contains the alert or error message, an icon, and an OK button. The icon is an information icon (**i**) if you pass `D_ALERT`, or an exclamation point (!) if you pass `D_ERROR`.

On Windows, U_WAIT cannot time out, and the `g_beep` setting has no effect (see [g_beep on page 18-2](#)).

UNIX, VMS

You can redefine the continuation key code and the continuation message for both the error and alert states by changing the contents of `g_wt_err_key`, `g_wt_err_msg`, `g_wt_alrt_key`, and `g_wt_alrt_msg`, respectively, in the `tkctl.def` file.

If a time-out occurs, `g_select` is set to true and `g_entnam` is set to the entry name specified in `tkctl.def` for `g_time_entry`. By default, `g_time_entry` is defined as "O_TIMEOUT".

Examples

The first example displays a message in an alert state but does not ring the terminal bell. The second example displays a message in an error state; it will ring the terminal bell.

```
xcall u_wait(D_NOBELL, D_ALERT)
xcall u_wait(D_ERROR)
```

%U_WINCELLS – Convert pixels to fractional rows or columns



```
cells = %U_WINCELLS (D_ROWS | D_COLS, window_id, pixels)
```

or

```
xcall U_WINCELLS (cells, D_ROWS | D_COLS, window_id, pixels)
```

Return value

cells

On Windows, the number of rows or columns. This may be fractional (implied decimal). **(d)**

On UNIX, always returns false. On OpenVMS, always returns true. **(n)**

Arguments

D_ROWS

Interpret *pixels* as a vertical series of pixels. *Cells* will be returned as the number of rows needed to span these pixels. **(n)**

D_COLS

Interpret *pixels* as a horizontal series of pixels. *Cells* will be returned as the number of columns needed to span these pixels. **(n)**

window_id

The window ID or one of the following: **(n)**

WID_APP The application window (excluding border and caption).

WID_CONTAINER The container window for the Synergy application (including the border and caption).

pixels

The number of pixels to convert. **(n)**

Discussion

%U_WINCELLS returns the number of columns or rows needed to span a given number of pixels.

Cells can be a fractional number. If, for example, the number of columns needed to span 80 pixels is 2.5, *cells* will be returned as 2.5 (implied decimal).

Examples

The following example returns the number of rows it takes to span the number of pixels (specified with the pixels variable) for the specified window:

```
rows = %u_wincells(D_ROWS, wndid, pixels)
```

The next example returns the number of columns it takes to span the number of pixels (specified with the pixels variable) for the specified window:

```
cols = %u_wincells(D_COLS, wndid, pixels)
```

%U_WINCOLOR – Perform Windows-specific color operations

WT			

value = %U_WINCOLOR (*subfunction*[, *arg*, ...])

or

xcall U_WINCOLOR (*subfunction*[, *arg*, ...])

Return value

value

On Windows, a value specific the to subfunction. On UNIX, always returns false.

On OpenVMS, always returns true. (^VAL)

Arguments

subfunction

The operation to perform: (**n**)

D_CHOOSSECOLOR Display the Color dialog box with an initial color selection.

D_GETCOLORS Return RGB values associated with Synergy colors.

D_HYPER_FG Return the RGB value for Toolkit hyperlink prompts.

D_PALETTE_RGB Return the foreground and background colors (as RGB values) for a Synergy palette entry.

D_SETCOLOR Use an RGB value to set a Synergy color.

arg

(optional) One or more arguments if required for *subfunction*.

Discussion

The %U_WINCOLOR function performs Windows-specific color operations. See the individual subfunctions for details.

If a value other than one of those listed above is passed for *subfunction*, a fatal Toolkit error will occur.

D_CHOOSSECOLOR

value = %U_WINCOLOR (D_CHOOSSECOLOR, *color*, [*help_filename*][, *help_context_id*])

Return value

value

On Windows, the user selected color, or the color passed as *color* if the user cancels. (^VAL)

Arguments

color

The color (RGB value) you want initially selected when the Color dialog box opens. (n)

help_filename

(optional) The name of the help file. (a)

help_context_id

(optional) The context identifier for help. Required if *help_filename* is passed. (n)

Discussion

The D_CHOOSSECOLOR subfunction displays the Color dialog box with an initially selected color (*color*). This function returns either the color selected by the user or, if the user cancels, the color specified by *color*. See the GetCellColor [Discussion on page 9-23](#) for information on RGB values.

If *help_filename* is passed, the dialog will have a Help button. *Help_filename* can be the name of either a WinHelp file (.hlp) or an HTML Help file (.chm):

- ▶ If the file extension is **.hlp**, *help_context_id* is passed as a context number to the HELP_CONTEXT command for the WinHelp API. See the Microsoft documentation for WinHelp for more information on this command.
- ▶ If the file extension is **.chm**, the HtmlHelp API is used, and *help_context_id* is passed on to the HTML Help API command HH_HELP_CONTEXT. See the Microsoft documentation for the HTML Help API for more information on this command.

Examples

The following example displays a dialog box with the initial value specified by **color (i4)** and returns the color chosen by the user (if any) in **new_color (i4)**.

```
new_color = %u_wincolor(D_CHOOSSECOLOR, color)
```

The next example includes a help button on the dialog box.

```
new_color = %u_wincolor(D_CHOOSSECOLOR, color, "helpfile.hlp",
&                                select_color_id)
```

D_GETCOLORS

value = %U_WINCOLOR (D_GETCOLORS, *color_array*, [*start*][, *end*])

or

xcall U_WINCOLOR (D_GETCOLORS, *color_array*, [*start*][, *end*])

Return value

value

On Windows, true. (^VAL)

Arguments

color_array

The first element in an array of returned RGB values for Synergy colors. If this array is not large enough to store all the returned colors, a fatal error may result (though results are unpredictable). (n)

start

(optional) The index of the first color to return (0 through 511). A value that is less than 0 or greater than 511 will cause a fatal runtime error. (n)

end

(optional) The index of the last color to return (0 through 511). A value that is less than 0 or greater than 511 will cause an error. If the value is less than *start*, no colors (or error) will be returned. (n)

Discussion

The D_GETCOLORS subfunction returns an array filled with the RGB values for a range of Synergy colors. D_GETCOLORS gets this information from the current color definitions. These can be the memory-resident defaults, colors defined in **synergy.ini** or **synuser.ini**, or colors set with D_SETCOLOR.

Start and *end* values correspond to numbers for Synergy colors. For example, if you pass 1 as *start*, the first element in the array will be the value for Synergy color 1. If you don't pass *start*, the array index will be one higher than the accessed color. Element 1 will contain Synergy color 0, element 2 will contain Synergy color 1, and so forth. This is because Synergy DBL arrays start with element 1, but Synergy colors start with color 0.

Note the following:

- ▶ If you don't pass *start* or *end*, `D_GETCOLOR` returns a 256 element array whose first element is Synergy color 0 and whose final element is Synergy color 255.
- ▶ If you don't pass *start*, but do pass *end*, `D_GETCOLOR` returns an array whose first element is Synergy color 0 and whose last element is specified by *end*. (The number of elements in the array will be *end* plus 1.)
- ▶ If you don't pass *end*, but do pass *start*, `D_GETCOLOR` returns an array whose first element is determined by *start* and whose final element is Synergy color 255.

See also

- ▶ The GetCellColor [Discussion on page 9-23](#) for information on RGB values.
- ▶ “[Colors and the color palette](#)” in the “Synergy Windowing API” chapter of the *Synergy DBL Language Reference Manual* for information on Synergy colors.

Examples

The following example returns in **color_array (256i4)** the RGB values for Synergy colors 0 through 255:

```
success = %u_wincolor(D_GETCOLORS, color_array)
```

The next example returns in **win_sys_colors (256i4)** the RGB values for Synergy colors 256 through 511:

```
success = %u_wincolor(D_GETCOLORS, win_sys_colors, 256, 511)
```

Utility Routines

%U_WINCOLOR

D_HYPER_FG

value = %U_WINCOLOR (D_HYPER_FG)

Return value

value

On Windows, the color of Toolkit hyperlink prompts as an RGB value. (^**VAL**)

Discussion

The D_HYPER_FG subfunction returns the color of Toolkit hyperlink prompts as an RGB value. See the GetCellColor [Discussion on page 9-23](#) for information on RGB values. (**n**)

Examples

The following example returns in **hyper_color (i4)** the RGB value associated with a Toolkit hyperlink prompt.

```
hyper_color = %u_wincolor(D_HYPER_FG)
```


D_PALETTE_RGB

value = %U_WINCOLOR (D_PALETTE_RGB, *palette*, *fg_color*, *bg_color*[, *attr*])

or

`xcall U_WINCOLOR (D_PALETTE_RGB, palette, fg_color, bg_color[, attr])`

Return value

value

On Windows, true if successful, or false if not. (^VAL)

Arguments

palette

A Synergy palette entry number (1-16). (n)

fg_color

The returned foreground color. This is returned as an RGB value. (n)

bg_color

The returned background color. This is returned as an RGB value. (n)

attr

Optional attribute to apply. You can supply only one *attr* argument, but its value can be composed by adding together the defined literals in **tools.def** or by using the supplied combination definitions (for example, A_HR for highlight and reverse). (n)

Discussion

The D_PALETTE_RGB subfunction returns the foreground and background colors associated with a Synergy palette entry. Each is returned as an RGB value. (See the GetCellColor [Discussion on page 9-23](#) for information on RGB values.) If *attr* is passed, the specified attribute (A_* in **tools.def**) are applied before evaluating the colors. (See the “[Attribute Flag Definitions](#)” table on [page 16-119](#).)

See also

“[Customizing the Look of Your Application](#)” on [page 18-24](#) for information on Synergy palettes.

Examples

This example returns the foreground and background colors associated with Synergy palette entry 6.

```
success = %u_wincolor(D_PALETTE_RGB, 6, fg_color, bg_color)
```

D_SETCOLOR

value = %U_WINCOLOR (D_SETCOLOR, *color*, *rgb*)

or

xcall U_WINCOLOR (D_SETCOLOR, *color*, *rgb*)

Return value

value

On Windows, true if successful, or false if not (for example, if *color* is invalid). (^VAL)

Arguments

color

A Synergy color (0-255). A value less than 0 or greater than 255 will cause a fatal runtime error. (n)

rgb

An RGB value. (n)

Discussion

The D_SETCOLOR subfunction sets a Synergy color (specified by *color*) to the RGB value represented by the value passed *rgb*.

D_SETCOLOR settings override color settings in **synergy.ini** and **synuser.ini**, as well as memory-resident defaults. But D_SETCOLOR settings last only until the program is exited. (Use the %U_SAVELOG function if you wish to save this new color to the [colors] section of **synergy.ini** or **synuser.ini**. See [%U_SAVELOG on page 16-128](#) for information.)

See also

- ▶ The GetCellColor [Discussion on page 9-23](#) for information on RGB values.
- ▶ “[Colors and the color palette](#)” in the “Synergy Windowing API” chapter of the *Synergy DBL Language Reference Manual* for information on Synergy colors.

Examples

The following example sets Synergy color 77 to the RGB value specified in **color** (i4).

```
success = %u_wincolor(D_SETCOLOR, 77, color)
```

U_WINDOW – Place/move/remove/delete windows



```
xcall U_WINDOW (D_PLACE, window_id, [row_placement], [col_placement])
```

```
xcall U_WINDOW (D_MOVE, window_id, [row_movement], [col_movement])
```

```
xcall U_WINDOW (D_REMOVE, remove_window_id [, ...])
```

```
xcall U_WINDOW (D_DELETE, delete_window_id [, ...])
```

Arguments

D_PLACE

Place a window. (n)

D_MOVE

Move a window. (n)

D_REMOVE

Remove a window. (n)

D_DELETE

Delete a window. (n)

window_id

The ID of the window to be placed or moved. (n)

row_placement

(optional) The row, relative to the screen body, where the window will be placed. (n)

col_placement

(optional) The column, relative to the screen body, where the window will be placed. (n)

row_movement

(optional) The number of rows up or down that the window will be moved. Default is 0. (n)

col_movement

(optional) The number of columns left or right that the window will be moved. Default is 0. (n)

remove_win_id

The ID of the window to remove or **D_ALL** (all windows), **D_GLOBAL** (all global windows), or **D_LOCAL** (all local windows). (n)

delete_win_id

The ID of the window to delete or **D_LOCAL** to delete all local windows. (**n**)

Discussion

The U_WINDOW subroutine manipulates windows within the screen body. You can combine the operations (D_PLACE, D_MOVE, etc.) in one call, with each operation followed by its arguments. See the examples below.

D_PLACE places a window relative to the screen body, not the entire screen. This means that the position (1,1) is the leftmost position immediately under the menu bar (in the first row). Placement is limited by the menu bar and information line; the window is adjusted so that it doesn't interfere with either of these screen sections. If *row_placement* or *col_placement* is not passed, the previous placement value is used. Placing a window that was previously hidden restores the window to its previous state which, by default, is normal. If *row_placement* and *col_placement* are both set to 0, the window is centered in the application window.

D_MOVE moves a window relative to the screen body (rather than the entire screen) and places the window if it is not placed. Negative values move the window up or to the left. Positive values move it down or to the right. Zero specifies the current position (row, column, or both), unless you pass both *row_movement* and *col_movement* as zero for an unplaced window. In this case, D_MOVE places the window at the top of all other placed windows. The position is automatically adjusted if necessary to prevent the window from occluding the menu bar or information line.

D_REMOVE removes one or more windows from the screen. You can specify qualifiers to remove all windows, all global windows, or only windows that belong to the current environment.

D_DELETE deletes one or more windows. Specify D_LOCAL to delete all windows that belong to the current environment. You can only delete windows placed at the current display level or windows "unplaced" in any environment. A fatal error occurs if you try to delete a window placed at a prior display level. If you try to delete an unallocated window ID, the deletion will be ignored.

Examples

The following example places the window whose ID is stored in **custdata** at position (1,1) of the screen body.

```
xcall u_window(D_PLACE, custdata, 1, 1)
```

In this example, the window whose ID is stored in **custdata** is moved to the top of the window pile.

```
xcall u_window(D_REMOVE, custdata, D_MOVE, custdata, 0, 0)
```

The following example removes all local windows.

```
xcall u_window(D_REMOVE, D_LOCAL)
```

The following example deletes the window whose ID is stored in **passid**.

```
xcall u_window(D_DELETE, passid)
```

The following example moves the window whose ID is stored in **custdata** 20 places down and 40 places to the right, places the window whose ID is stored in **empdata** at position (1,1) of the screen body, and removes the windows whose IDs are stored in **taxinfo** and **date**.

```
xcall u_window(D_MOVE, custdata, 20, 40, D_PLACE, empdata, 1, 1,
&      D_REMOVE, taxinfo, date)
```

%U_WINHELP – Invoke WinHelp Help



```
status = %U_WINHELP ([window_id], filename, command, data)
```

or

```
xcall U_WINHELP ([window_id], filename, command, data)
```

Return value

status

On Windows, an integer data type. (^VAL)

TRUE If successful.

FALSE If unsuccessful.

On UNIX and OpenVMS, always returns false.

Arguments

window_id

(optional) The ID of the window requesting Help. (**n**)

filename

The name of the Help file to invoke. The extension must be **.hlp**. (**a**)

command

The command code that specifies the type of help requested. See the “[Command Codes and Descriptions](#)” table on page 16-163 for codes and descriptions. (**n**)

data

The specific data for the command code. (**a** or **n**)

Discussion

%U_WINHELP starts the Windows Help application (**winhlp32.exe**) and passes additional data indicating the nature of the help requested. The API is nearly identical to the Windows SDK “WinHelp” function, with two exceptions:

- ▶ Instead of passing a window handle, you pass a window ID which is translated to a handle. If you don’t pass *window_id* or if it is zero, the handle of the application window will be used.
- ▶ If you pass a numeric value in *data*, that value is passed directly to WinHelp. If you pass an alphanumeric value, %U_WINHELP copies and null-terminates *data*’s contents, and the address of the copied string is passed instead. This enables either alphanumeric strings or data structures to be passed in a native Synergy manner.

%U_WINHELP deactivates your application when it invokes WinHelp Help. Your application will remain suspended until reactivated, which will happen when either help is closed or the user returns to the application (by clicking on the application window with the mouse or by using ALT+TAB to switch back to the application).

If the user reactivates the application without closing Help, the Help will continue running (in a deactivated state) until %U_MSGBOX is called with the **WHELP_QUIT** command option or until the user closes Help. The user can keep Help displayed indefinitely while continuing to use the application.

%U_WINHELP uses *window_id* to keep track of which applications have requested Help.

You can display the help topic in a secondary window instead of the primary window by using the following syntax for *filename*:

filename>secondary_windowname

where *secondary_windowname* is the name of the window for displaying the topic. To use the secondary window specified, you must first define its name in the [WINDOWS] section of the Help project (**.hbj**) file.

The *data* value used depends on the value of the *command* argument. The following table shows the possible values for the *command* argument and the corresponding formats of *data*.

Command Codes and Descriptions		
Command	Data value	Action
WHELP_COMMAND	Macro name (a)	Executes a help macro.
WHELP_CONTENTS	Zero (0)	Displays Help for the topic specified by the CONTENTS setting in the .hbj file.
WHELP_CONTEXT	Context number for the topic (n)	Displays Help for the topic identified by a context number that has been defined in the [MAP] section of the .hbj file.
WHELP_CONTEXTPOPUP	Context number for a topic (n)	Displays Help in a pop-up window for the topic identified by a context number that has been defined in the [MAP] section of the .hbj file.
WHELP_FORCEFILE	Zero (0)	Ensures that WinHelp is displaying the correct Help file. If the correct Help file is currently displayed, there is no action. If the incorrect Help file is displayed, WinHelp opens the correct file.
WHELP_HELPONHELP	Zero (0)	Displays the Help Topics dialog box for Windows Help.

Command Codes and Descriptions (Continued)		
Command	Data value	Action
WHELP_INDEX	Zero (0)	Displays the Help Contents topic. (See WHELP_CONTENTS .)
WHELP_KEY	Keyword (a)	Displays the Index tab of the Help Topics dialog box with the specified keyword in the “Type the first few letters of the word you’re looking for” field.
WHELP_MULTIKEY	<i>Multikey</i> (a) (see below)	Displays the structure identified by a keyword in an alternative keyword table.
WHELP_PARTIALKEY	Keyword (a)	If there is one exact match, displays the topic found in the keyword list that matches the specified keyword. If there is more than one match, displays the Search dialog box with the topics listed in the Go To list box. If there is no match, displays the Search dialog box. To bring up the Search dialog box without passing a keyword (the third result), use a null string (“ ”).
WHELP_QUIT	Zero (0)	Informs the Help application that Help is no longer needed. If no other applications have asked for Help, Windows closes the Help application.
WHELP_SETCONTENTS	Context number for the topic to be designated as the Index topic (n)	Determines the Contents topic that Help is to display when a user presses the F1 key.
WHELP_SETINDEX	Context number for the topic (n)	Specifies the Contents topic.
WHELP_SETWINPOS	<i>Helpwininfo</i> (a) (See below)	Displays the Help structure, if it is minimized or in memory, and positions it according to the data passed.

The *multikey* record definition must be defined as follows:

```
record multikey
    mksize      ,i4      ;Length, in bytes, of this structure
    mkkeylist   ,a1      ;Identifies the keyword table to search
    keyphase    ,a*      ;Contains the keyword to be located
```

The *helpwininfo* record definition must be defined as follows:

```
record helpwininfo ;Length, in bytes,
    structsize ,i4 ;of this structure
```


x	,i4	;Pixel x-coordinate of upper-left corner
y	,i4	;Pixel y-coordinate of upper-left corner
dx	,i4	;Width, in pixels
dy	,i4	;Height, in pixels
wMax	,i4	;Specifies how the window is to be shown
rgchmember	,a*	;Unicode window name

%U_WININFO – Return Windows-specific information

WT			

value = %U_WININFO (*info* [, *arg*, ...])

or

xcall U_WININFO (*info* [, *arg*, ...])

Return value

value

On Windows, a value that is specific to the *info* flag. On UNIX, always returns false.
On OpenVMS, always returns true. (^VAL)

Arguments

info

The information to return: (n)

D_CURSORPOS Return the current pixel coordinates for the mouse pointer.

D_HINSTANCE Return the instance handle for the Synergy runtime.

D_VKTOASCII Return the ASCII character that corresponds to a virtual key code.

arg

(optional) Up to two arguments if required for *info*.

Discussion

The %U_WININFO function returns Windows-specific information. See the individual subfunctions for details.

If a value other than those listed above is passed for *info*, a fatal error will occur.

See also

[%U_WINMETRICS](#) on page 16-170 for Windows-specific metrics information.

D_CURSORPOS

value = %U_WININFO (D_CURSORPOS, *x_coordinate*, *y_coordinate*)

or

xcall U_WININFO (D_CURSORPOS, *x_coordinate*, *y_coordinate*)

Return value

value

On Windows, true. (^VAL)

Arguments

x_coordinate

The returned x-axis pixel coordinate for the mouse cursor. (**n**)

y_coordinate

The returned y-axis pixel coordinate for the mouse cursor. (**n**)

Discussion

The D_CURSORPOS subfunction returns the x-axis and y-axis pixel coordinates for the current location of the mouse cursor. Note that these coordinates are relative to the *screen*, not a window.

Examples

The following example returns in **xpos** and **ypos** (both **i4**) the current pixel coordinates of the mouse cursor relative to the screen.

```
success = %u_wininfo(D_CURSORPOS, xpos, ypos)
```

Utility Routines

%U_WININFO

D_HINSTANCE

value = %U_WININFO (D_HINSTANCE)

Return value

value

On Windows, the instance handle (D_ADDR) for the Synergy runtime (**dbr**). (^VAL)

Discussion

The D_HINSTANCE subfunction returns the instance handle for the Synergy runtime (**dbr**). This could be passed, for example, to a DLL routine to create a window. (The instance handle is one of the required arguments to CreateWindow and CreateWindowEx.)

Examples

The following example returns in **hinstance** the instance handle of the Synergy runtime (**dbr**).

```
hinstance = %u_wininfo (D_HINSTANCE)
```

D_VKTOASCII

value = %U_WININFO (D_VKTOASCII, *vk_code*[, *key_state*])

Return value

value

On Windows, the ASCII character that corresponds to *vk_code* if one exists; otherwise returns 0. (^**VAL**)

Arguments

vk_code

The virtual key code (numeric value) of any alphanumeric character or any function key as defined in **tools.def** (D_VKINSERT, D_VK1, and so forth). (**n**)

key_state

(optional) The key state flags as defined in **tools.def**: D_KEY_SHIFT, D_KEY_CONTROL, D_KEY_ALT, or any combination of these combined using .BOR. or the vertical bar (|) operator. (**n**)

Discussion

The D_VKTOASCII subfunction returns the ASCII character that corresponds to *vk_code* or *vk_code* plus *key_state*. If this doesn't result in an ASCII value, D_VKTOASCII returns zero. Note that the numeric values of most printable characters correspond to the ASCII values of the characters. The exceptions are a-z, which use their upper-case values (65-90).

Examples

The following example returns in **ascii (i4)** the value that corresponds to the letter “a” (lower-case).

```
ascii = %U_WININFO(D_VKTOASCII, %decml('A'))
```

The following example returns the value that corresponds to the letter “D” (upper-case).

```
ascii = %U_WININFO(D_VKTOASCII, %decml('D'), D_KEY_SHIFT)
```

The next example returns the value that corresponds to the F6 key with the ALT and CTRL keys.

```
ascii = %U_WININFO(D_VKTOASCII, D_VKF6, D_KEY_ALT .bor. D_KEY_CONTROL)
```

%U_WINMETRICS – Return information about a window

WT			

return = %U_WINMETRICS (*metric*[, *arg*, ...])

or

xcall U_WINMETRICS (*metric*[, *arg*, ...])

Return value

return

On Windows, a value that is specific to the requested metric. On UNIX, always returns false.
On OpenVMS, always returns true. (^**VAL**)

Arguments

metric

The information to return: (**n**)

D_CHARHEIGHT	Return height of characters.
D_CHARWIDTH	Return width of characters.
D_CHECKBOX_X	Return width of a check box or radio button.
D_CLIENT_RECT	Return coordinates for client area of a window.
D_COLTOX	Return width for a specified number of columns, or return the coordinate for the right side of a specified column.
D_EDITEXTRA_X	Return extra width required for an edit control.
D_HSCROLL_Y	Return height of a horizontal scroll bar.
D_MAPPOINTS	Convert points from the coordinate system of one window to the coordinate system for another window.
D_ROWHEIGHT	Return the overall height of a row.
D_ROWLEADING	Return the space between rows.
D_ROWTOY	Return height for the specified number of rows, or return the coordinate for the bottom of a specified row.
D_SCREEN_X	Return screen width.
D_SCREEN_Y	Return screen height.
D_TABHEIGHT	Return the height of a tab.
D_TABWIDTH	Return the width of a tab.

D_TABXOFFSET	Return the horizontal offset for a tab set.
D_TABYOFFSET	Return the vertical offset for a tab set.
D_VISIBLE_RECT	Return coordinates for client area minus any area occupied by standard buttons.
D_VSCROLL_X	Return width of a vertical scroll bar.
D_WINDOW_RECT	Return coordinates for screen rectangle for a window.
D_XBORDER	Return width of a vertical side of a thin border.
D_XFRAME	Return width of a vertical side of a thick window frame.
D_XORIGIN	Return offset of first visible column from the left side of the client area.
D_XTOCOL	Return number for the column that contains pixels with a given x-axis coordinate.
D_YBORDER	Return thickness (height) of a horizontal side of a thin border.
D_YCAPTION	Return height of a caption bar.
D_YFRAME	Return thickness (height) of a horizontal side of a thick window frame.
D_YORIGIN	Return offset of first visible row from the top of the client area.
D_YTOROW	Return number for the row that contains pixels with a given y-axis coordinate.

arg

(optional) One to three arguments as required for the metric.

Discussion

The %U_WINMETRICS function returns information about a window or the contents of a window. See the individual subfunctions for details.

If a value other than those listed above is passed for *metric*, a fatal error will occur.

See also

You can retrieve other types of information about a window by using the W_INFO and %W_INFO Synergy DBL routines. See [W_INFO](#) and [%W_INFO](#) in the “Synergy Windowing API” chapter of the *Synergy DBL Language Reference Manual*.

Examples

You can combine %U_WINMETRICS subfunctions to perform additional calculations. For example, the following code displays the size of the screen in characters. It does this by dividing the pixel size of the screen by the size of a character. (Note that this calculation doesn't account for toolbars or scrollbars.)

```
xcall U_MESSAGE("width " + %string(%U_WINMETRICS(D_SCREEN_X)/  
& %U_WINMETRICS(D_CHARWIDTH, my_window))) ;Screen width in characters
```

```
xcall U_MESSAGE("height " + %string(%U_WINMETRICS(D_SCREEN_Y)/  
& %U_WINMETRICS(D_CHARHEIGHT, my_window))) ;Screen height in characters
```

For a more complete example, see [GetWindowSize.zip](#), available from the Synergy CodeExchange in the Resource Center on the Synergex web site. This example calculates the available area in a window (with some exceptions) when that window is fully expanded.

D_CHARHEIGHT

return = %U_WINMETRICS (D_CHARHEIGHT, *window_id*)

Return value

return

On Windows, the height in pixels of the sizing character for the specified window. (^VAL)

Arguments

window_id

The window ID or one of the following: (**n**)

WID_APP The application window.

WID_CONTAINER The container window for the Synergy application.

Discussion

The D_CHARHEIGHT subfunction returns the height in pixels of the sizing character for the window specified by *window_id*. For a definition of sizing character, see the [Glossary](#) in this manual and “[Using Fonts on Windows](#)” on page 18-40.

Examples

The following example returns in **char_height (i4)** the pixel height of the sizing character associated with **wndid**.

```
char_height = %U_WINMETRICS (D_CHARHEIGHT, wndid)
```

The next example returns the pixel height of the sizing character associated with the application window.

```
char_height = %U_WINMETRICS (D_CHARHEIGHT, WID_APP)
```

See also the %U_WINMETRICS [Examples on page 16-172](#) for an example that uses D_CHARHEIGHT as part of a calculation that uses multiple %U_WINMETRICS subfunctions.

D_CHARWIDTH

return = %U_WINMETRICS (D_CHARWIDTH, *window_id*)

Return value

return

On Windows, the width in pixels of the sizing character for the specified window. (^VAL)

Arguments

window_id

The window ID or one of the following: (**n**)

WID_APP

The application window (excluding border and caption).

WID_CONTAINER

The container window for the Synergy application (including the border and caption).

Discussion

The D_CHARWIDTH subfunction returns the width in pixels of the sizing character for the window specified by *window_id*. For a definition of sizing character, see the [Glossary](#) in this manual and “[Using Fonts on Windows](#)” on page 18-40.

Examples

The following example returns in **char_width (i4)** the pixel width of the sizing character associated with **wndid**.

```
char_width = %U_WINMETRICS (D_CHARWIDTH, wndid)
```

The next example returns the pixel width of the sizing character associated with the application window.

```
char_width = %U_WINMETRICS (D_CHARWIDTH, WID_APP)
```

D_CHECKBOX_X

return = %U_WINMETRICS (D_CHECKBOX_X)

Return value

return

On Windows, the width in pixels of a check box or radio button. (^VAL)

Discussion

The D_CHECKBOX_X subfunction returns the width of a check box or radio button in pixels.

Examples

The following example returns in **pixel_width (i4)** the width of a check box in pixels.

```
pixel_width = %U_WINMETRICS (D_DCHECKBOX_X)
```

D_CLIENT_RECT

```
return = %U_WINMETRICS (D_CLIENT_RECT, h_wnd, rect)
```

or

```
xcall U_WINMETRICS (D_CLIENT_RECT, h_wnd, rect)
```

Return value

return

On Windows, true if successful, or false if not. (^VAL)

Arguments

h_wnd

The handle for the window whose client area coordinates you want to retrieve. This doesn't need to be a Synergy window. For example, this could be the handle for an ActiveX control, a window created by a DLL, or a window created by a different process. (n)

rect

The returned rectangle coordinates for the client area of the window. *Rect* is returned as a group of four **i4** values, which are coordinates for the left, top, right, and bottom points (respectively) of the client area. (n)

Discussion

The D_CLIENT_RECT subfunction returns coordinates for the client area of the *h_wnd* window. (The client area is the inside area of a window minus the frame, captions, information line, and footer.) These coordinates are returned in the *rect* argument, which is a group of **i4** values for the left, top, right, and bottom points of the client area. The left and top coordinates are always zero.

D_CLIENT_RECT and D_VISIBLE_RECT will always be equal for the application window (WID_APP), and they will always be equal for the container window (ID_CONTAINTER).

For the application and container windows, however, D_CLIENT_RECT may not be equal if, for example, a toolbar is placed. This area would be excluded from D_CLIENT_RECT for the application window.

For any other Synergy window, D_CLIENT_RECT and D_VISIBLE_RECT will be equal if the window does not include buttons.

Examples

The following example gets the client rectangle for the application window in **rect**.

```
success = %U_WINMETRICS (D_CLIENT_RECT, %W_INFO(WIF_HWND, WID_APP), rect)
```

D_COLTOX

return = %U_WINMETRICS (D_COLTOX, *window_id*, *col_num*[, D_SIZE | D_POSITION])

Return value

return

Arguments

window_id

The window ID or one of the following: (n)

WID_APP

The application window (excluding border and caption).

WID_CONTAINER

The container window for the Synergy application (including the border and caption).

col_num

If D_SIZE is passed (or neither D_SIZE nor D_POSITION is passed), *col_num* specifies the number of columns whose width you want to measure. If D_POSITION is passed, *col_num* specifies a column number. (n)

D_SIZE

(optional) Return the pixel width of the number of columns specified in *col_num*. (default) (n)

D_POSITION

(optional) Return the x-axis pixel coordinate for the right side of the column specified by *col_num*. (n)

Discussion

The D_COLTOX subfunction returns either a width or an x-axis pixel coordinate.

- ▶ If D_SIZE is passed (or neither D_SIZE nor D_POSITION is passed), D_COLTOX returns the width in pixels of the number of columns specified in *col_num*.
- ▶ If D_POSITION is passed, D_COLTOX returns an x-axis (horizontal) pixel coordinate for the right side of the column specified by *col_num*. This value is based on the display area origin, a coordinate system in which the 0, 0 pixel is at the top left of the display area.

Examples

This example returns in **pixel_width (i4)** the number of pixels that equates to six columns for **wndid**.

```
pixel_width = %U_WINMETRICS (D_COLTOX, wndid, 6, D_SIZE)
```

This example returns in **pixel_width** the right-most pixel of column 25 in the application window.

```
pixel_width = %U_WINMETRICS (D_COLTOX, WID_APP, 25, D_POSITION)
```

D_EDITEXTRA_X

return = %U_WINMETRICS (D_EDITEXTRA_X)

Return value

return

On Windows, the extra pixel width required for an edit control. (^VAL)

Discussion

The D_EDITEXTRA_X subfunction returns the extra pixel width required for an edit control to be placed over a specific set of characters. This is in addition to the number of characters times the character width.

Examples

The following example returns in **pixel_width (i4)** the number of extra pixels required for an edit control.

```
pixel_width = %U_WINMETRICS (D_EDITEXTRA_X)
```

D_HSCROLL_Y

return = %U_WINMETRICS (D_HSCROLL_Y)

Return value

return

On Windows, the height of a horizontal scroll bar in pixels. (^VAL)

Discussion

The D_HSCROLL_Y subfunction returns the height of a horizontal scroll bar in pixels.

Examples

The following example returns in **pixel_height (i4)** the height of a horizontal scroll bar in pixels.

```
pixel_height = %U_WINMETRICS (D_HSCROLL_Y)
```

D_MAPPOINTS

```
return = %U_WINMETRICS (D_MAPPOINTS, h_from, h_to, x_y_pair)
```

or

```
xcall U_WINMETRICS (D_MAPPOINTS, h_from, h_to, x_y_pair)
```

Return value

return

On Windows, a 32-bit value that contains the addends used to convert coordinates. The low-order 16 bits contain the x-axis addend, and the high-order 16 bits contain the y-axis addend. Each value can be a negative number. (See the [Discussion](#) below for information on extracting these values.) (^VAL)

Arguments

h_from

A handle for the window whose coordinate system you want *x_y_pair* converted from. (n)

h_to

A handle for the window whose coordinate system you want *x_y_pair* converted to. (n)

x_y_pair

A set (or sets) of coordinates (*x*, *y*) for a point (or points) in the *h_from* window. Each 8 bytes of this argument is interpreted as a pair of **i4** values that represent x- and y- axis coordinates, so the value you pass must have a size that's a multiple of 8 bytes (8 bytes for a single point, 16 bytes for two points, and so forth). The converted coordinates are returned in this argument, so it must be writable. (n)

Discussion

The D_MAPPOINTS function converts the *x_y_pair* set (or sets) of coordinates from the coordinate system of the *h_from* window to the coordinate system of the *h_to* window. In other words, it converts coordinates that are relative to the upper-left corner of the *h_from* window to coordinates that are relative to the upper-left corner of the *h_to* window. (Note that these windows can be in different processes.) This could be useful, for example, if you have an ActiveX control that has no border and acts as a background or pointer within a larger window. When a user clicks on the control, your event routine may need to convert the coordinates of the click location to coordinates in the same location in the parent window.

The converted coordinates are returned in the *x_y_pair* argument in the same order they were passed.

The addends used to calculate the new set(s) of coordinates are returned in *return*, which is a 32-bit value. The low-order 16 bits contain the x-axis addend, and the high-order 16 bits contain the y-axis addend. For example, if *h_from* is a window positioned at 100, 200 and *h_to* is a window positioned

at 500, 50, *return* will contain the x-axis addend (decimal 400, hex190) in the low-order word and the y-axis addend (decimal -150, hex FF6A in 16-bit twos complement) in the high-order word. In this case, *return* will be returned as FF6A0190.

To extract the x- and y-axis addends from *return*, use the following:

```
cx = rval .band. ^x(FFFF)
cy = (rval .band. ^x(FFFF0000)) / ^x(1000)
```

where *cx* (**i2** or **i4**) is the addend used to calculate the x-axis coordinates, *cy* (**i2** or **i4**) is the addend used to calculate the y-axis coordinates, and *rval* (**i4**) is the returned value (*return*). (*Rval* can be a group of *cx*, **i2** and *cy*, **i2** values on little endian architectures.)

Examples

The following example first gets the visible rectangle for **wndid**, and then converts these points to their corresponding values in the application window.

```
if (%U_WINMETRICS(D_VISIBLE_RECT, wndid, rect)) then
    pts = %U_WINMETRICS(D_MAPPOINTS, %W_INFO(WIF_HWND, wndid),
    &      %W_INFO(WIF_HWND, WID_APP), rect)
```

D_ROWHEIGHT

return = %U_WINMETRICS (D_ROWHEIGHT, *window_id*)

Return value

return

On Windows, the overall height in pixels of a row in the specified window. (^VAL)

Arguments

window_id

The window ID or one of the following: (**n**)

WID_APP

The application window (excluding border and caption).

WID_CONTAINER

The container window for the Synergy application (including the border and caption).

Discussion

The D_ROWHEIGHT subfunction returns the overall height (character height plus leading) in pixels of a row in the window specified by *window_id*.

Examples

The following example returns in **row_height (i4)** the pixel height of a row in **wndid**.

```
row_height = %U_WINMETRICS (D_ROWHEIGHT, wndid)
```

D_ROWLEADING

```
return = %U_WINMETRICS (D_ROWLEADING, window_id)
```

Return value

return

On Windows, the space between rows in pixels for the specified window. (^VAL)

Arguments

window_id

The window ID or one of the following: (n)

WID_APP

The application window (excluding border and caption).

WID_CONTAINER

The container window for the Synergy application (including the border and caption).

Discussion

The D_ROWLEADING subfunction returns the leading (the space between rows) in pixels for the window specified by *window_id*.

See also

- ▶ [%U_WNDSTYLE](#) on page 16-232.
- ▶ [MINIMIZE_LEADING](#) in the “Environment Variables” chapter of *Environment Variables & System Options*.

Examples

The following example returns in **row_leading (i4)** the number of pixels between rows for **wndid**.

```
row_leading = %U_WINMETRICS (D_ROWLEADING, wndid)
```

D_ROWTOY

return = %U_WINMETRICS (D_ROWTOY, *window_id*, *row_num* [, D_SIZE | D_POSITION])

Return value

return

Varies depending on syntax. See Discussion below. (^VAL)

Arguments

window_id

The window ID or one of the following: (n)

WID_APP

The application window (excluding border and caption).

WID_CONTAINER

The container window for the Synergy application (including the border and caption).

row_num

If D_SIZE is passed (or neither D_SIZE nor D_POSITION is passed), *row_num* specifies the number of rows whose height you want to measure. If D_POSITION is passed, *row_num* specifies a row number. (n)

D_SIZE

(optional) Return the pixel height of the number of rows specified in *row_num*. (default) (n)

D_POSITION

(optional) Return y-axis (vertical) pixel coordinate for the bottom of the row specified by *row_num*. (n)

Discussion

The D_ROWTOY subfunction returns either a height or a y-axis pixel coordinate.

- ▶ If D_SIZE is passed (or neither D_SIZE nor D_POSITION is passed), D_ROWTOY returns the height of the number of rows specified in *row_num*.
- ▶ If D_POSITION is passed, D_ROWTOY returns a y-axis pixel coordinate for the bottom of the row specified by *row_num*.

Examples

This example returns in **pixel_height (i4)** the number of pixels that equates to six rows for **wndid**.

```
pixel_height = %U_WINMETRICS (D_ROWTOY, wndid, 6, D_SIZE)
```

The next example returns in **pixel_height** the bottom pixel of row 17 in the application window.

```
pixel_height = %U_WINMETRICS (D_ROWTOY, WID_APP, 17, D_POSITION)
```

D_SCREEN_X

return = %U_WINMETRICS (D_SCREEN_X)

Return value

return

On Windows, the width of the screen in pixels. (^VAL)

Discussion

The D_SCREEN_X subfunction returns the width of the screen in pixels.

Examples

The following example returns in **pixel_width (i4)** the width of the screen (the monitor's display area) in pixels.

```
pixel_width = %U_WINMETRICS (D_SCREEN_X)
```

See the %U_WINMETRICS [Examples on page 16-172](#) for an example that uses D_SCREEN_X as part of a calculation that uses multiple %U_WINMETRICS subfunctions.

Utility Routines

%U_WINMETRICS

D_SCREEN_Y

return = %U_WINMETRICS (D_SCREEN_Y)

Return value

return

On Windows, the height of the screen in pixels. (^**VAL**)

Discussion

The D_SCREEN_Y subfunction returns the height of the screen in pixels.

Examples

The following example returns in **pixel_height (i4)** the height of the screen (the monitor's display area) in pixels.

```
pixel_height = %U_WINMETRICS (D_SCREEN_Y)
```

See the %U_WINMETRICS [Examples on page 16-172](#) for an example that uses D_SCREEN_Y as part of a calculation that uses multiple %U_WINMETRICS subfunctions.

D_TABHEIGHT

height = %U_WINMETRICS (D_TABHEIGHT, *tabset_id*, *index*)

Return value

height

On Windows, the height in pixels of a tab. (^VAL)

Arguments

tabset_id

The ID of a tab set container window (the ID returned by the DTS_CREATE subfunction to %TS_TABSET). (n)

index

The index of the tab whose height you want returned. (n)

Discussion

The D_TABHEIGHT subfunction returns the height in pixels of the tab specified by the *tabset_id* and *index* arguments. *Height* is the y-axis measurement of the tab (so *height* will vary depending on whether the tab is part of a horizontal or vertical tab set).

If *index* is less than 1 or greater than the number of tabs in the specified tab set (*tabset_id*), Toolkit generates an error.

Examples

The following example returns the number of pixels that make up the height of the tab specified by **tabset_id** and **tab_index**.

```
tab_height = %U_WINMETRICS (D_TABHEIGHT, tabset_id, tab_index)
```

D_TABWIDTH

width = %U_WINMETRICS (D_TABWIDTH, *tabset_id*, *index*)

Return value

width

On Windows, the width in pixels of a tab. (^**VAL**)

Arguments

tabset_id

The ID of a tab set container window (the ID returned by the DTS_CREATE subfunction to %TS_TABSET). (**n**)

index

The index of the tab whose width you want returned. (**n**)

Discussion

The D_TABWIDTH subfunction returns the width in pixels of the tab specified by the *tabset_id* and *index* arguments. *Width* is the x-axis measurement of the tab (so *width* will vary depending on whether the tab is part of a horizontal or vertical tab set).

If *index* is less than 1 or greater than the number of tabs in the specified tab set (*tabset_id*), Toolkit generates an error.

Examples

The following example returns the number of pixels that make up the width of the tab specified by **tabset_id** and **tab_index**.

```
tab_width = %U_WINMETRICS (D_TABWIDTH, tabset_id, tab_index)
```


D_TABXOFFSET

```
x_offset = %U_WINMETRICS (D_TABXOFFSET, tabset_id)
```

Return value

x_offset

On Windows, a tab set's horizontal offset in pixels. (^VAL)

Arguments

tabset_id

The ID of a tab set container window (the ID returned by the DTS_CREATE subfunction to %TS_TABSET). (n)

Discussion

The D_TABXOFFSET subfunction returns the distance in pixels from the left side of the client area of the tab set container window to the left side of the windows on the tabs. This distance includes the edge around the tab area and, if tabs are on the left, the width (x-axis measurement) of tabs.

Examples

The following example returns in **x_offset (i4)** the horizontal offset for the tab set specified by **tabset_id**.

```
x_offset = %U_WINMETRICS (D_TABXOFFSET, tabset_id)
```

D_TABYOFFSET

```
y_offset = %U_WINMETRICS (D_TABYOFFSET, tabset_id)
```

Return value

y_offset

On Windows, a tab set's vertical offset in pixels. (^**VAL**)

Arguments

tabset_id

The ID of a tab set container window (the ID returned by the DTS_CREATE subfunction to %TS_TABSET). (**n**)

Discussion

The D_TABYOFFSET subfunction returns the distance in pixels from the top of the client area of the tab set container window to the top of the windows on the tabs. This distance includes the edge around the tab area and, if tabs are on the top, the height (y-axis measurement) of tabs.

Examples

The following example returns in **y_offset (i4)** the vertical offset for the tab set specified by **tabset_id**.

```
y_offset = %U_WINMETRICS (D_TABYOFFSET, tabset_id)
```

D_VISIBLE_RECT

return = %U_WINMETRICS (D_VISIBLE_RECT, *window_id*, *rect*)

or

xcall U_WINMETRICS (D_VISIBLE_RECT, *window_id*, *rect*)

Return value

return

On Windows, true if successful, or false if not. (^VAL)

Arguments

window_id

The window ID or one of the following: (n)

WID_APP

The application window (excluding border and caption).

WID_CONTAINER

The container window for the Synergy application (including the border and caption).

rect

The returned rectangle coordinates for the points of the client area of the window, minus any area occupied by standard buttons. *Rect* is returned as a group of four **i4** values, which are coordinates for the left, top, right, and bottom points (respectively) of the client area. (n)

Discussion

The D_VISIBLE_RECT subfunction returns coordinates for the client area of a window, minus any area occupied by standard buttons. (The client area is the inside area of a window minus the frame, captions, information line, and footer.) These coordinates are returned in the *rect* argument, which is a group of **i4** values for the left, top, right, and bottom points of the client area.

See the D_CLIENT_RECT [Discussion on page 16-176](#) for information on when D_CLIENT_RECT and D_VISIBLE_RECT must agree or differ.

Examples

The following example gets the visible rectangle for *wndid* in **rect**.

```
success = %U_WINMETRICS (D_VISIBLE_RECT, wndid, rect)
```

Utility Routines

%U_WINMETRICS

D_VSCROLL_X

return = %U_WINMETRICS (D_VSCROLL_X)

Return value

return

On Windows, the width of a vertical scroll bar in pixels. (**^VAL**)

Discussion

The D_VSCROLL_X subfunction returns the width of a vertical scroll bar in pixels.

Examples

The following example returns in **pixel_width (i4)** the width of a vertical scroll bar in pixels.

```
pixel_width = %U_WINMETRICS (D_VSCROLL_X)
```

D_WINDOW_RECT

```
return = %U_WINMETRICS (D_WINDOW_RECT, h_wnd, rect[, rel_pos])
```

or

```
xcall U_WINMETRICS (D_WINDOW_RECT, h_wnd, rect[, rel_pos])
```

Return value

return

On Windows, true if successful, or false if not. (^VAL)

Arguments

h_wnd

The handle for the window whose screen rectangle you want to retrieve. This doesn't need to be a Synergy window. For example, this could be the handle for an ActiveX control, a window created by a DLL, or a window created by a different process. (n)

rect

The returned screen coordinates for the window. *Rect* is returned as a group of four **i4** values, which contain the coordinates for the left, top, right, and bottom points (respectively) of the window. (n)

rel_pos

(optional) Determines whether the *rect* coordinates are relative to the screen or to the client area of the parent window. (n)

TRUE

Coordinates are relative to the upper-left corner of the screen.

FALSE

Coordinates are relative to the upper-left corner of the client area of the parent window. (default)

Discussion

The D_WINDOW_RECT subfunction returns coordinates for the outside edges of the screen rectangle for the *h_wnd* window. This includes frames, borders, captions, and so forth. If *rel_pos* is passed and true, these coordinates are relative to the upper-left corner of the screen. If *rel_pos* is not passed or is false, these coordinates are relative to the upper-left corner of the client area of the parent window.

Examples

The following example returns the screen rectangle for **wndid** in *rect*. The returned coordinates will be relative to the screen.

```
success = %U_WINMETRICS (D_WINDOW_RECT, %w_info(WIF_HWND, wndid),  
& rect, TRUE)
```

Utility Routines

%U_WINMETRICS

D_XBORDER

```
return = %U_WINMETRICS (D_XBORDER)
```

Return value

return

On Windows, the width in pixels of a vertical thin border. (**^VAL**)

Discussion

The D_XBORDER subfunction returns the width in pixels of a vertical thin border.

Examples

The following example returns in **pixel_width (i4)** the width of the vertical side of a thin border.

```
pixel_width = %U_WINMETRICS (D_XBORDER)
```

D_XFRAME

return = %U_WINMETRICS (D_XFRAME)

Return value

return

On Windows, the width in pixels of a vertical side of a thick window frame. (^VAL)

Discussion

The D_XFRAME subfunction returns the width, in pixels, of a vertical side of a thick window frame.

Examples

The following example returns in **pixel_width (i4)** the width of the vertical side of a thick window frame.

```
pixel_width = %U_WINMETRICS (D_XFRAME)
```

D_XORIGIN

return = %U_WINMETRICS (D_XORIGIN, *window_id*)

Return value

return

On Windows, the offset in pixels of the first visible column from the left side of the client area for the specified window. (^VAL)

Arguments

window_id

The window ID or one of the following: (**n**)

WID_APP

The application window (excluding border and caption).

WID_CONTAINER

The container window for the Synergy application (including the border and caption).

Discussion

The D_XORIGIN subfunction returns the offset in pixels of the first visible column from the left side of the client area of the window specified by *window_id*.

Examples

The following example returns in **offset (i4)** the number of pixels between the first visible column and the left side of the client area for **wndid**.

```
offset = %U_WINMETRICS (D_XORIGIN, wndid)
```


D_XTOCOL

return = %U_WINMETRICS (D_XTOCOL, *window_id*, *x_value*)

Return value

return

On Windows, the number of the column that contains a given x-axis (horizontal) coordinate for a pixel. (^VAL)

Arguments

window_id

The window ID or one of the following: (n)

WID_APP

The application window (excluding border and caption).

WID_CONTAINER

The container window for the Synergy application (including the border and caption).

x_value

The x-axis (horizontal) coordinate for a pixel. This number must be based on the display area origin (where the 0, 0 pixel is at the top left of the display area). (n)

Discussion

The D_XTOCOL subfunction returns the number of the column that contains the pixel whose x-axis coordinate is *x_value* in the window specified by *window_id*. This subfunction assumes *x_value* is based on the display area origin, a coordinate system in which the 0, 0 pixel is at the top left of the display area.

When using proportional fonts in conjunction with D_XTOCOL, the returned column value may not be correct. This value is based on the sizing character for the window, not any actual text.

Examples

This example returns in **column_num** (i4) the number of the column with the specified pixel.

```
column_num = %U_WINMETRICS (D_XTOCOL, wndid, xpos)
```

Utility Routines

%U_WINMETRICS

D_YBORDER

```
return = %U_WINMETRICS (D_YBORDER)
```

Return value

return

On Windows, the height (thickness) in pixels of a horizontal thin border. (^**VAL**)

Discussion

The D_YBORDER subfunction returns the height (thickness) in pixels of a horizontal thin border.

Examples

The following example returns in **pixel_height (i4)** the height of the horizontal side of a thin border.

```
pixel_height = %U_WINMETRICS (D_YBORDER)
```

D_YCAPTION

return = %U_WINMETRICS (D_YCAPTION)

Return value

return

On Windows, the height in pixels of a caption bar. This includes the height of the upper frame. (**^VAL**)

Discussion

The D_YCAPTION subfunction returns the height in pixels of a caption bar. This height includes the height of the upper frame.

Examples

The following example returns in **pixel_height (i4)** the height of the caption bar.

```
pixel_height = %U_WINMETRICS (D_YCAPTION)
```

Utility Routines

%U_WINMETRICS

D_YFRAME

```
return = %U_WINMETRICS (D_YFRAME)
```

Return value

return

On Windows, the thickness (height) in pixels of a horizontal side of a thick window frame. (**^VAL**)

Discussion

The D_YFRAME subfunction returns the thickness (height) in pixels of a horizontal side of a thick window frame.

Examples

The following example returns in **pixel_height (i4)** the height of the horizontal side of a thick window frame.

```
pixel_height = %U_WINMETRICS (D_YFRAME)
```

D_YORIGIN

return = %U_WINMETRICS (D_YORIGIN, *window_id*)

Return value

return

On Windows, the offset in pixels of the first visible row from the top of the client area for the specified window. (^VAL)

Arguments

window_id

The window ID or one of the following: (**n**)

WID_APP

The application window (excluding border and caption).

WID_CONTAINER

The container window for the Synergy application (including the border and caption).

Discussion

The D_YORIGIN subfunction returns the offset in pixels of the first visible row from the top of the client area of the window specified by *window_id*.

Examples

The following example returns in **offset (i4)** the number of pixels between the first visible row and the top of the client area for **wndid**.

```
offset = %U_WINMETRICS (D_YORIGIN, wndid)
```

D_YTOROW

return = %U_WINMETRICS (D_YTOROW, *window_id*, *y_value*)

Return value

return

On Windows, the number of the row that contains a given y-axis (vertical) coordinate for a pixel. (^VAL)

Arguments

window_id

The window ID or one of the following: (n)

WID_APP

The application window (excluding border and caption).

WID_CONTAINER

The container window for the Synergy application (including the border and caption).

y_value

The y-axis (vertical) coordinate for a pixel. This number must be based on the display area origin (where the 0,0 pixel is at the top left of the display area). (n)

Discussion

The D_YTOROW subfunction returns the number of the row that contains the pixel whose y-axis coordinate is *y_value* in the window specified by *window_id*. This subfunction assumes *y_value* is based on the display area origin, a coordinate system in which the 0, 0 pixel is at the top left of the display area.

Examples

The following example returns in **row_num** (i4) the row number that contains the specified pixel for **wndid**.

```
row_num = %U_WINMETRICS (D_YTOROW, wndid, ypos)
```

%U_WNDEVENTS – Associate methods with mouse events

WT			

```
return = %U_WNDEVENTS (subfunction[, arg, ...])
```

or

```
xcall U_WNDEVENTS (subfunction[, arg, ...])
```

Return value

return

On Windows, a value specific to the subfunction. On UNIX and OpenVMS, always returns false. (^VAL)

Arguments

subfunction

The function to perform: (n)

D_CREATE	Create a new method set.
D_ASSIGN	Assign a method set to one or more windows.
D_SIGNAL	Call a routine registered as part of a method set for a Toolkit window.
D_GETEVENT	Return the address of a method routine for a specified event.
D_GETEVENTS	Determine whether a method set has been registered for a Toolkit window and, if it has, return the method set ID.

arg

One or more arguments, depending on *subfunction*.

Discussion

The %U_WNDEVENTS function can be used to create an event method set, to assign an event method set to one or more windows for handling various mouse events, and to trigger a routine that is registered as part of a method set for a Toolkit window. See the individual subfunctions for details.



Don't use this function for event methods for ActiveX controls or for composite windows. Do the following instead:

- ▶ For ActiveX controls, use the *prefix* argument for %AX_TKSINGLE and %GRID_CREATE to register extensions to the event method set that Toolkit automatically uses for these.
 - ▶ For composite windows, use the DC_EVENT subfunction for %C_CONTAINER to register extensions to the event method set that Toolkit automatically uses for these windows.
-

See also

[“Appendix D: Methods”](#).

D_CREATE

```
method_set_id = %U_WNDEVENTS (D_CREATE, [elb], event, event_method,
& [event, event_method][, ...])
```

or

```
xcall U_WNDEVENTS (D_CREATE, [elb], event, event_method, [event, event_method][, ...])
```

Return value

method_set_id

A method set ID if successful, or false (0) if not. (^VAL)

Arguments

elb

(optional) The name of the library from which to draw the *event_method* routines. (a)

event

The ID of the event: (n)

D_EVENT_LEFT_CLICK	Left mouse button event.
D_EVENT_MIDDLE_CLICK	Middle mouse button event.
D_EVENT_RIGHT_CLICK	Right mouse button event.
D_EVENT_LEFT_DBLCLK	Double-click left mouse button.
D_EVENT_MIDDLE_DBLCLK	Double-click middle mouse button.
D_EVENT_RIGHT_DBLCLK	Double-click right mouse button.
D_EVENT_MOVE	Move window event.
D_EVENT_SIZE	Size window event.
D_EVENT_MINIMIZE	Minimize window event.
D_EVENT_MAXIMIZE	Maximize window event.
D_EVENT_RESTORE	Restore window event.
D_EVENT_SCROLL	Scroll window event.
D_EVENT_CLOSE	Close window event.

event_method

The name of the %UWNDEVENTS_METHOD function that will handle the event. (a)

Discussion

The `D_CREATE` subfunction of `%U_WNDEVENTS` is used to define which `%UWNDEVENTS_METHOD` should be invoked for a specific event. You can specify the same `%UWNDEVENTS_METHOD` for more than one event, or for all events.

Event and *event_method* must be specified as a pair. You can specify more than one event pair to `%U_WNDEVENTS`. All event pairs specified within one `D_CREATE` call to `%U_WNDEVENTS` are referred to as a method set and will be identified by a method set ID, which is returned by this function.

Only the events specified in the method set will invoke a call to `%UWNDEVENTS_METHOD`. The method set must also be associated with a window. See [D_ASSIGN on page 16-209](#) for information on associating methods with windows.



Once defined, method sets continue to exist for the life of your program. Each method set requires a small amount of memory. Be careful to avoid creating more method sets than you need, and try to reuse each method set for multiple windows. Creating too many method sets could result in performance degradation or program termination.

If `D_EVENT_SIZE` is specified, the frame of the window is changed to one that the user can “grab” by clicking and dragging the border to designate the new size. Also, a “Size” menu entry is placed on the window’s system menu to allow for the use of the keyboard when resizing. If this method is removed from the window, its corresponding menu entry is also removed. *Argument_5* and *argument_6* refer to the width and height of the logical window. The logical window is defined as the rows and columns used by Synergy/DE®. *Argument_7* and *argument_8* refer to the width and height of the client area. The client area is defined as the visible area within the window (inside the frame and scroll bars) in which client objects can be placed.

Resizing is snapped to the nearest row/column boundary. A window can be resized no smaller than the minimum size required for buttons plus one row and column of window space. When a window is resized, the display area of the window is modified, rather than the actual window size. If the window has a display area smaller than the total window area and scroll bars are enabled, scroll bars are displayed. When the scroll bars are moved, the display area origin changes. If scroll bars are disabled, they are not displayed. In certain instances, scroll bars may not be desirable, such as when the resizable window contains only one resizable component that shows scroll bars. Scroll bars are enabled and disabled using the `WB_SBON` and `WB_SBOFF` subfunctions of `W_BRDR`.

When a tab set container is resized smaller, the tabbed dialogs inside the container are also resized. If a window on a tab is resizable and scroll bars have not been disabled for that window, then scroll bars will be shown on the contained window if the tab is no longer large enough to display the entire window. Because the border of a contained window is turned off when it is placed on a tab, you must assign the window method set (with `D_ASSIGN`) prior to placing the window in the tab set. If a contained window does not show scroll bars (either because it is not resizable or because

scroll bars have been disabled with `W_BRDR`), then resizing the tab set smaller may clip the contained window so that it cannot all be viewed by the user. If a tab set container is sized larger than its logical window size, the tab set is still only as large as the logical window size.

For a list, horizontal resizing is limited to between 1 and the number of columns in the associated input window, and vertical resizing is limited at 1 and has no logical upper limit.

With the “full drag mode” enabled in Windows, resize events may occur rapidly. Therefore, actions performed in the resize method should be minimal and quick, otherwise the message queue may fill up.

If `D_EVENT_MINIMIZE` is specified, a minimize button is added to the right side of the window’s frame, as well as a “Minimize” menu entry on the window’s system menu. A “Restore” menu entry is also added. The minimize button changes to a restore button when the window is minimized. A `%UWNDEVENTS_METHOD` must be written to enable this functionality; however, the method need not actually do anything. If this method is removed from the window, its corresponding button and menu entry are also removed.

When a window is minimized, a minimal title bar is displayed along the bottom border of the application window. The display area of a minimized window remains unaffected.

If `D_EVENT_MAXIMIZE` is specified, a maximize button is added to the right side of the window’s frame, as well as a “Maximize” menu entry on the window’s system menu. A “Restore” menu entry is also added. The maximize button changes to a restore button when the window is maximized. A `%UWNDEVENTS_METHOD` must be written to enable this functionality; however, the method need not actually do anything. If this method is removed from the window, its corresponding button and menu entry are also removed.

When a maximize event occurs, a resize event also occurs so that the application can respond to the change in size.

If a window is maximized, the display area is set to the number of visible rows and columns, or the maximum number of rows and columns, whichever is smaller. The frame of the window occupies the entire desktop, even if part of the contained area lies outside of the window’s logical size. In this case, the excess area is painted with the application background color defined by Windows.

When a window is restored, the display area is set to the number of rows and columns that fit into the restored rectangle. Note that when a restore event occurs, a resize event also occurs so that the application can respond to the change in size.

The `D_EVENT_MOVE`, `D_EVENT_SIZE`, `D_EVENT_MINIMIZE`, `D_EVENT_MAXIMIZE`, `D_EVENT_RESTORE`, `D_EVENT_SCROLL`, and `D_EVENT_CLOSE` events occur in response to both user action and programmatic changes to the window. For instance, a window can be programmatically maximized, minimized, or restored even if it doesn’t have a corresponding

method, by using the `W_PROC(WP_STATE)` subroutine. The application should not assume that an event is always fired in response to a user's action. For instance, if a resize method resizes the window, an infinite loop of method invocation can occur.



Selection windows do not support the minimize, maximize, or restore buttons, and they don't support the sizing arrow you see when you drag the edge or corner of a window. Because of this, users aren't able to minimize, maximize, restore, or size selection windows. However, you can do these things programmatically by using `W_PROC`. If you use `W_PROC`, events associated with the selection window will be called.

D_ASSIGN

status = %U_WNDEVENTS (D_ASSIGN, *method_set_id*, *window_id*[, ...])

or

xcall U_WNDEVENTS (D_ASSIGN, *method_set_id*, *window_id*[, ...])

Return value

status

True or false depending upon success of the assignment. (^VAL)

Arguments

method_set_id

The method set ID returned from the D_CREATE subfunction. (n)

window_id

The window ID that will produce those events. (n)

Discussion

The method set ID obtained from the D_CREATE subfunction of %U_WNDEVENTS must be associated with at least one window before the methods associated with mouse events are ever called. The D_ASSIGN subfunction associates the method set ID with one or more windows.

If you have at least one method set created, you can start handling mouse events specified in the method set by associating the method set ID with windows. The same method set may be associated with one or more windows in your Synergy application. Or, you could define and assign a different method set to each window.

Once you associate your application windows with a method set ID, event methods will start receiving calls as mouse events occur.

Only one method set may be assigned to a window; therefore if you call %U_WNDEVENTS with the D_ASSIGN subfunction more than once for the same window, the last method set assigned is the one that will be retained. Any previous one will be ignored.

%U_WNDEVENTS cannot be used to associate method sets with lists. To associate a method set with a list, use the D_LWNDEVENT function in L_METHOD.

D_SIGNAL

rval = %U_WNDEVENTS (D_SIGNAL, *window_id*, *event*[, *arg*, ...])

or

`xcall` U_WNDEVENTS (D_SIGNAL, *window_id*, *event*[, *arg*, ...])

Return value

rval

The value returned by the triggered event routine. (^VAL)

Arguments

window_id

The ID of the Toolkit window whose method set contains the routine you want to trigger. (n)

event

The %UWNDEVENTS_METHOD event code for the routine to be triggered. (n)

arg

(optional) One or more arguments required for the routine. (n)

Discussion

The D_SIGNAL subfunction triggers a routine that is registered as part of a method set for a Toolkit window. You specify the routine by passing the %UWNDEVENTS_METHOD event code for the routine as the *event* argument. For example, to trigger the D_EVENT_SIZE method for a window, pass D_EVENT_SIZE as *event*. Note that if no method is registered for the window for the specified event, the signal is ignored. For a list of these method codes (which are defined in **tools.def**), see [%UWNDEVENTS_METHOD on page 16-237](#).

D_GETEVENT

method_addr = %U_WNDEVENTS (D_GETEVENT, *method_set*, *event*)

Return value

method_addr

The address (D_ADDR) for the method for the event passed as *event*, or 0 if no routine is registered for this event. (^VAL)

Arguments

method_set

The method set ID. (n)

event

The event code for the routine whose address you want to retrieve. (n)

Discussion

The D_GETEVENT subfunction enables you to determine if a method is registered for a given event in a method set and, if so, what the address for the method routine is. For example, the following returns the address of the method for D_EVENT_LEFT_CLICK, or it returns 0 if no method is registered for D_EVENT_LEFT_CLICK. (Note how D_GETEVENTS enables us to use the window ID to get this information.)

```
method_address=%u_wndevents(D_GETEVENT, %u_wndevents(D_GETEVENTS, wndid),  
& D_EVENT_LEFT_CLICK)
```

For a list of the method codes you can pass as *event*, see [%UWNDEVENTS_METHOD](#) on [page 16-237](#).

Utility Routines

%U_WNDEVENTS

D_GETEVENTS

method_set = %U_WNDEVENTS (D_GETEVENTS, *window_id*)

Return value

method_set

The method set ID, or 0 if there is no method set associated with the specified window. (^**VAL**)

Arguments

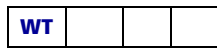
window_id

The ID of the window whose method set ID you want to retrieve. (**n**)

Discussion

The D_GETEVENTS subfunction enables you to determine if a method set has been registered for a UI Toolkit window and, if it has, what the ID for the method set is.

%U_WNDFONT – Set/retrieve font information on Windows



```
return = %U_WNDFONT (subfunction, arg[, ...])
```

or

```
xcall U_WNDFONT (subfunction, arg[, ...])
```

Return value

return

On Windows, a value specific to the subfunction. On UNIX and OpenVMS, always returns false, except for D_SETWNDFONT, which returns true if successful. (^VAL)

Arguments

subfunction

The specific font function to perform: (n)

D_ALTFONT132	Enable/disable automatic font switching.
D_CHOOSEFONT	Display font selection dialog.
D_ENUMFONTS	Enumerate font palette entries.
D_FONTHANDLE	Retrieve the font handle for a font palette entry.
D_FONTNAME	Retrieve the font palette entry for a font handle.
D_FONTSIZE	Retrieve font size information.
D_GETFONT	Retrieve font palette information.
D_GETSCALE	Retrieve the application scaling percentage.
D_GETWNDFONT	Retrieve the font palette entry for a window.
D_SAVEFONT	Save font palette information in synergy.ini or synuser.ini .
D_SETFONT	Set font palette information.
D_SETSCALE	Set the application scaling percentage.
D_SETWNDFONT	Set the font palette entry for a window.

arg

One or more arguments, depending on *subfunction*.

Discussion

The %U_WNDFONT function sets or retrieves font information in a Windows environment. It also enables you to display a standard Microsoft Font dialog box. See the individual subfunctions for details.

The Synergy runtime manages font information by maintaining a font palette. A font palette consists of font palette entries, each of which associates a font name with a typeface, point size, and sizing character combination. Font palette entries can be added or modified using %U_WNDFONT.

%U_WNDFONT can be used to change the font for an application by assigning a different font palette entry name to the global and/or alternate fonts.

An application window defined as less than 132 columns uses the “global font.” An application window defined as 132 columns or more, or one changed to this size, uses the “alternate font.” This enables an application window having a greater number of columns to use a different font and, consequently, still fit the entire window on the screen.

%U_WNDFONT can also be used to change the font for a specific window.

Each time the font is changed, the application (or window) is completely updated to use the new font.

See also

- ▶ [“Using Fonts on Windows” on page 18-40](#) for information on fonts and font palettes.
- ▶ [I_FLDMOD on page 8-25](#) and [IB_FIELD on page 8-95](#) for information on changing a field or prompt font.

D_ALTFONT132

prev_flag = %U_WNDFONT (D_ALTFONT132, *alt_font_flag*)

or

xcall U_WNDFONT (D_ALTFONT132, *alt_font_flag*)

Return value

prev_flag

The previous value of *alt_font_flag*. (^VAL)

Arguments

alt_font_flag

A flag that specifies if automatic switching to/from the alternate font is enabled. (n)

TRUE Enable automatic font switching. (default)

FALSE Disable automatic font switching.

Discussion

The D_ALTFONT132 subfunction of %U_WNDFONT enables/disables automatic font switching.

If *alt_font_flag* is true, then whenever the number of columns in the application window crosses the 132 column boundary, the font will automatically switch to or from the “alternate” font. If *alt_font_flag* is false, automatic font switching will not occur.

See also

[“Using Fonts on Windows” on page 18-40.](#)

D_CHOOSEFONT

```
status = %U_WNDFONT (D_CHOOSEFONT, face_name, point_size, [sizing_char], [include_prop],  
& [help_filename], [help_context_id][, title])
```

or

```
xcall U_WNDFONT (D_CHOOSEFONT, face_name, point_size, [sizing_char], [include_prop],  
& [help_filename], [help_context_id][, title])
```

Return value

status

True if the user selected a font; otherwise false. (**^VAL**)

Arguments

face_name

The name of the Windows font. (**a**)

point_size

The size of the Windows font. (**n**)

sizing_char

(optional) The sizing character. (**a**)

include_prop

(optional) The flag to include proportional fonts. (**n**)

help_filename

(optional) The name of the help file. (**a**)

help_context_id

(optional) The context identifier for help. Required if *help_filename* is passed. (**n**)

title

(optional) The title for the dialog box. (**a**)

Discussion

The D_CHOOSEFONT subfunction of %U_WNDFONT displays a Microsoft Windows Font dialog box. Any values passed in *face_name*, *point_size*, and *sizing_char* are used as the typeface, point size, and sizing character initially displayed in the dialog box. *Face_name* and *sizing_char* are case sensitive.

If the user makes a selection, *face_name*, *point_size*, and *sizing_char* are returned with the typeface name, size, and sizing character values.

D_CHOOSEFONT does *not* set the new font or update **synergy.ini** or **synuser.ini**. To set the new font, use D_SETFONT. To explicitly update **synergy.ini** or **synuser.ini**, use D_SAVEFONT.

If *sizing_char* is not present, the sizing character cannot be modified and defaults to an uppercase “A”. The sizing character is a single letter used to determine the cell size for proportional fonts within a visual object for the purposes of positioning. When you specify *sizing_char* to be “W”, for example, the object will be significantly wider than an object with a sizing character of “i”. The sizing character is case sensitive.

If *include_prop* is true, proportional fonts are included in the list of available typefaces for the dialog box. The default *include_prop* value is false.

If *help_filename* is passed, the dialog will have a Help button. *Help_filename* can be the name of either a WinHelp file (**.hlp**) or an HTML Help file (**.chm**):

- ▶ If the file extension is **.hlp**, *help_context_id* is passed as a context number to the HELP_CONTEXT command for the WinHelp API. See the Microsoft documentation for WinHelp for more information on this command.
- ▶ If the file extension is **.chm**, the HtmlHelp API is used, and *help_context_id* is passed on to the HTML Help API command HH_HELP_CONTEXT. See the Microsoft documentation for the HTML Help API for more information on this command.

If *title* is not passed, the default title for the dialog box is “Font.”

Examples

The following example displays a dialog box, entitled “Select Font,” with the font name **typeface**, the font size **size**, and sizing character **char**, specified.

```
success = %u_wndfont(D_CHOOSEFONT, typeface, size, char, TRUE,,
    & "Select Font")
```

Compare the previous example with the following example, which neither includes proportional fonts nor prompts for a sizing character, but includes a help button on the dialog box.

```
success = %u_wndfont(D_CHOOSEFONT, font, size,,, "helpfile.hlp",
    & select_font_id, "Select Font")
```

D_ENUMFONTS

```
num_palette = %U_WNDFONT (D_ENUMFONTS, [DF_BY_HANDLE | DF_BY_NAME],  
& [maximum], [hndl_array] [, name_array])
```

or

```
xcall U_WNDFONT (D_ENUMFONTS, [order], [maximum], [hndl_array] [, name_array])
```

Return value

num_palette

The total number of defined font palette entries. (^VAL)

Arguments

DF_BY_HANDLE

(optional) Order by handle. (default) (n)

DF_BY_NAME

(optional) Order by name. (n)

maximum

(optional) The maximum number of fonts to return in the arrays. (n)

hndl_array

(optional) First element of the returned array of font handles. (n)

name_array

(optional) First element of the returned array of font palette entry names. (n)

Discussion

The D_ENUMFONTS subfunction of %U_WNDFONT returns font information for the fonts that are currently defined.

If *maximum* is not passed, it defaults to zero. This returns the number of font palette entries, and any arrays that might be passed are ignored. Because the number of defined fonts has no logical limit, D_ENUMFONTS can be first called with *maximum* not passed (or 0) to retrieve the number of font palette entries. and then the desired array(s) can be allocated accordingly.

The return value is always the total number of defined fonts, even if *maximum* is less than this value. The number of items returned in the arrays is the lesser of the return value or *maximum*.

We recommend that the size of each array entry in *hndl_array* be an **i4** for efficiency and to guarantee that the full handle value will be returned.

If the size of each array entry in *name_array* is less than 60 characters, some font palette entry names may be truncated.

Examples

The following example returns the total number of fonts defined in **numfonts** and the first 100 palette entry names in **font_array**.

```
numfonts = %u_wndfont(D_ENUMFONTS, DF_BY_NAME, 100,, font_array)
```

Utility Routines

%U_WNDFONT

D_FONTHANDLE

font_hndl = %U_WNDFONT (D_FONTHANDLE, *palette_name*)

Return value

font_hndl

The handle to the specified font. (^**VAL**)

Arguments

palette_name

The name of a font palette entry or one of the following: (**a** or **n**)

DF_CURRENT Current font.

DF_DEFAULT Global font.

DF_ALTERNATE Alternate global font.

DF_DEBUGGER Debugger font.

Discussion

The D_FONTHANDLE subfunction of %U_WNDFONT returns the font handle for a specified font palette entry name. Using handles is more efficient than using font palette entry names, however, font names are more persistent.

See also

[D_FONTNAME](#) on page 16-221 for more information on retrieving a font palette entry name.

Examples

The following example returns the font handle in **hndl** for the font “font_name.”

```
hndl = %u_wndfont (D_FONTHANDLE, "font_name")
```


D_FONTNAME

```
status = %U_WNDFONT (D_FONTNAME, font_hndl, palette_name)
```

or

```
xcall U_WNDFONT (D_FONTNAME, font_hndl, palette_name)
```

Return value

status

True if successful, or false if not. (^VAL)

Arguments

font_hndl

The handle to the font. (**n**)

palette_name

The returned font palette entry name. (**a**)

Discussion

The D_FONTNAME subfunction of %U_WNDFONT retrieves the font palette entry name for a specified font handle. Font palette entry names are more persistent than font handles, however, using font handles is more efficient.

See also

[D_FONTHANDLE on page 16-220](#) for more information on returning a font handle.

Examples

The following example returns the name of the font in **name** that is specified by the font handle **hndl**.

```
success = %u_wndfont (D_FONTNAME, hndl, name)
```

D_FONTSIZE

status = %U_WNDFONT (D_FONTSIZE, *font_name*, *width*, *height*)

or

xcall U_WNDFONT (D_FONTSIZE, *font_name*, *width*, *height*)

Return value

status

On Windows, true. On UNIX and OpenVMS, false. (^VAL)

Arguments

font_name

The name of a font palette entry, a font handle, or one of the following: (**a** or **n**)

DF_CURRENT Current font.

DF_DEFAULT Global font.

DF_ALTERNATE Alternate global font.

DF_DEBUGGER Debugger font.

width

The returned width of the sizing character for the font palette entry. (**n**)

height

The returned height of the sizing character for the font palette entry. (**n**)

Discussion

The D_FONTSIZE subfunction returns font size information (height and width) for a given Synergy font palette entry. This size is given as the number of pixels used on the screen. Note that this may differ from the same font's size when sent to a printer.

Examples

The following example returns the width and height of the font palette entry specified by **font_entry**.

```
success = %u_wndfont(D_FONTSIZE, font_entry, font_width, font_height)
```

D_GETFONT

```
font_hndl = %U_WNDFONT (D_GETFONT, font_id, [face_name], [point_size], [sizing_char]  
&      [, palette_name])
```

or

```
xcall U_WNDFONT (D_GETFONT, font_id, [face_name], [point_size], [sizing_char]  
&      [, palette_name])
```

Return value

font_hndl

The font handle if the font request was successful, or false if it was not. (^VAL)

Arguments

font_id

The name of a font palette entry, a font handle, or one of the following: (**a** or **n**)

DF_CURRENT Current font.

DF_DEFAULT Global font.

DF_ALTERNATE Alternate global font.

DF_DEBUGGER Debugger font.

face_name

(optional) The name of the Windows font. (**a**)

point_size

(optional) The size of the Windows font. (**n**)

sizing_char

(optional) The sizing character. (**a**)

palette_name

(optional) The returned font palette entry name. (**a**)

Discussion

The D_GETFONT subfunction of %U_WNDFONT retrieves the current typeface, point size, and sizing character of a specific font palette entry or of the current, global, alternate, or debugger font.

Note that D_GETFONT does not use the application scaling percentage (see [D_GETSCALE on page 16-225](#)). To get the approximate visual point size for a font used in an application, multiply the point size (returned by D_GETFONT) by the application scaling percentage (returned by D_GETSCALE) and then divide by 100. For example if the size returned by D_GETFONT is 12, and the application scaling percentage is 150, the visual point size is 18 (that is, $12 * 150 / 100 = 18$).

See also

[D_SETFONT](#) on page 16-228 for a description of *sizing_char*.

Examples

The following example returns the name and size of the font specified by **fontid** in **face_name** and **point_size**. The font palette entry name is returned in **palette_name**, and the default sizing character is returned in **size_char**.

```
sts = %u_wndfont(D_GETFONT, fontid, face_name, point_size, size_char,  
& palette_name)
```

D_GETSCALE

scaling_percentage = %U_WNDFONT (D_GETSCALE)

Return value

scaling_percentage

The application scaling percentage. (^VAL)

Discussion

The D_GETSCALE subfunction returns the application scaling percentage, which is a percentage of the default size of the application. See [D_SETSCALE on page 16-230](#) for more information.

Note that when using a proportionally spaced font for the application window, D_GETSCALE may not return 100 initially when SYN_RESIZE_SCALE is set.

D_GETWNDFONT

status = %U_WNDFONT (D_GETWNDFONT, *window_id*[, *palette_name*])

or

xcall U_WNDFONT (D_GETWNDFONT, *window_id*[, *palette_name*])

Return value

status

On Windows, the handle of the font requested if successful, or false if not. On UNIX and OpenVMS, always returns false (0). (**^VAL**)

Arguments

window_id

The ID of the window whose font is being retrieved. (**n**)

palette_name

(optional) The returned font palette entry name. (**a**)

Discussion

The D_GETWNDFONT subfunction of %U_WNDFONT retrieves the name of the font palette entry applied to a general window or input window.

On UNIX and OpenVMS, D_GETWNDFONT, *palette_name* returns the name of the font palette entry if it is defined.

See also

[“Using Fonts on Windows” on page 18-40.](#)

Examples

The following example returns the font handle and font palette entry name for the window specified by **wndid**.

```
hndl = %u_wndfont (D_GETWNDFONT, wndid, name)
```

D_SAVEFONT

```
status = %U_WNDFONT (D_SAVEFONT, [font_id][, all_users])
```

or

```
xcall U_WNDFONT (D_SAVEFONT, [font_id][, all_users])
```

Return value

status

On Windows, true if the function is successful, or false if it was not. On UNIX and OpenVMS, always returns false. (^**VAL**)

Arguments

font_id

(optional) The name of a font palette entry, a font handle, or one of the following: (**a** or **n**)

DF_CURRENT Current font.

DF_DEFAULT Global font.

DF_ALTERNATE Alternate global font.

DF_DEBUGGER Debugger font.

all_users

(optional) One of the following: (**n**)

TRUE Save font palette entry information to **synergy.ini**.

FALSE Save font palette entry information to **synuser.ini**. (default)

Discussion

The D_SAVEFONT subfunction of %U_WNDFONT saves the current font palette entry information for the specified *font_id* by writing it to the [fonts] section of **synergy.ini** or **synuser.ini**. If the function fails, which causes *status* to be returned as false, you can use %SYSERR to retrieve the system-specific error code.

If *font_id* is not passed, all font palette entries are saved.

If automatic font switching has been turned off using the D_ALTFONT132 subfunction, then DF_CURRENT is the same as DF_DEFAULT.

D_SETFONT

```
font_hndl = %U_WNDFONT (D_SETFONT, font_id, [face_name], [point_size][, sizing_char])
```

or

```
xcall U_WNDFONT (D_SETFONT, font_id, [face_name], [point_size][, sizing_char])
```

Return value

font_hndl

The font handle if the font request was successful, or false if it was not. (^VAL)

Arguments

font_id

The name of a font palette entry, a font handle, or one of the following: (**a** or **n**)

DF_CURRENT Current font.

DF_DEFAULT Global font.

DF_ALTERNATE Alternate global font.

DF_DEBUGGER Debugger font.

face_name

(optional) The name of the Windows font. (**a**)

point_size

(optional) The size of the Windows font. If omitted or passed as 0, defaults to 10. (**n**)

sizing_char

(optional) The sizing character. (**a**)

Discussion

The D_SETFONT subfunction of %U_WNDFONT defines or modifies a font palette entry at runtime. This font information is not saved in **synergy.ini** until the %U_WNDFONT subfunction, D_SAVEFONT, is called.

When defining a new font palette entry, specify *font_id* as either the font palette entry name or one of the literals listed above. You are returned the font handle. When modifying an existing font palette entry, you can refer to it (using *font_id*) by font handle.

Face_name is case sensitive. If it is omitted or is passed as blank, it defaults to the name of the current default font. If a proportional font is passed as *face_name*, *sizing_char* (a single letter) is used to determine the cell size within a visual object for the purposes of positioning. For example, “W” results in a cell size that is significantly wider than with “i”. The sizing character is case sensitive and, if omitted or passed as %char(0), defaults to “A”.

Examples

The following examples set the font palette entry specified by “my_font” to 10-point Courier typeface.

```
sts = %u_wndfont(D_SETFONT, "my_font", Courier, 10)
```

```
sts = %u_wndfont(D_SETFONT, "my_font", Courier)
```

The following sets the current font to 12-point Arial typeface with a “q” sizing character.

```
sts = %u_wndfont(D_SETFONT, DF_CURRENT, "Arial", 12, "q")
```

D_SETSCALE

status = %U_WNDFONT (D_SETSCALE, *scaling_percentage*)

or

xcall U_WNDFONT (D_SETSCALE, *scaling_percentage*)

Return value

status

On Windows, true. On UNIX and OpenVMS, false. (^VAL)

Arguments

scaling_percentage

The application scaling percentage. This can be specified as a whole number or with fractional precision (e.g., 176 or 50.95). Passing 100 sets the scaling percentage to 100 (no scaling). (n)

Discussion

The D_SETSCALE subfunction changes the size of the application display screen, windows for the application, and characters in those windows. It changes the height and the width of these items by the percentage passed as *scaling_percentage*. For example, passing 200 increases the height and width of these items by 200%. Note, however, that characters don't always scale evenly; they are sized as large as possible without exceeding the specified percentage in either direction.



For best results, use OpenType or TrueType fonts. D_SETSCALE's effectiveness is dependent on how well the font used for windows in the application can be scaled. If the font doesn't scale well, D_SETSCALE will not be able to correctly scale the display screen, windows for the application, or characters on those windows. For example, Fixedsys (which is the default font for Toolkit applications on Windows in traditional Synergy) doesn't scale well and will prevent D_SETSCALE from working correctly.

The following will not scale: message boxes, menu bars, toolbars, buttons, ActiveX controls, and Windows common controls (e.g., the Color, Font, Open, and Print Preview dialog boxes).

See also

- ▶ [APP_SCALE](#) in the “Environment Variables” chapter of *Environment Variables & System Options* for information on the default scaling percentage.
- ▶ [D_GETSCALE](#) on page 16-225.
- ▶ “Resizing the application” on page 1-20

D_SETWNDFONT

```
status = %U_WNDFONT (D_SETWNDFONT, window_id, font_id)
```

or

```
xcall U_WNDFONT (D_SETWNDFONT, window_id, font_id)
```

Return value

status

True if the font request was successful, or false if it was not. (^VAL)

Arguments

window_id

The ID of the window whose font is being set. (**n**)

font_id

The name of a font palette entry, a font handle, or one of the following: (**a** or **n**)

DF_CURRENT Current font.

DF_DEFAULT Global font.

DF_ALTERNATE Alternate global font.

DF_DEBUGGER Debugger font.

Discussion

The D_SETWNDFONT subfunction of %U_WNDFONT modifies the font for a general window or input window at runtime. If a proportional font is specified, objects within that window that do not support proportional fonts may not display as expected.

If *font_id* is invalid, the behavior of D_SETWNDFONT depends on the argument type. If *font_id* is numeric, an “invalid font id” error is generated. If *font_id* is alphanumeric, no error is generated, and the application window’s font is used.

While it does not visually render the font on UNIX or OpenVMS systems, D_SETWNDFONT does preserve the font information in the window (when the window is saved with U_SAVE) so that it can be used later on Windows.

See also

[“Using Fonts on Windows” on page 18-40.](#)

Examples

The following example sets the font for the window specified by **wndid** to be “my_input_font.”

```
success = %u_wndfont (D_SETWNDFONT, wndid, "my_input_font")
```

%U_WNDSTYLE – Change the style of a list or window



```
status = %U_WNDSTYLE (D_LST_STYLE | D_WND_STYLE, id
&      [, D_LEADING | D_NO_LEADING])
```

or

```
xcall U_WNDSTYLE (D_LST_STYLE | D_WND_STYLE, id
&      [, D_LEADING | D_NO_LEADING])
```

Return value

status

True if successful; false if *id* does not exist. On UNIX, always returns false. On OpenVMS, always returns true. (^VAL)

Arguments

D_LST_STYLE

Change a list style. (n)

D_WND_STYLE

Change a window style. (n)

id

The ID of the list or window. (n)

D_LEADING

(optional) Windows are spaced to allow edit control frames. (default) (n)

D_NO_LEADING

(optional) Windows are spaced according to font height. (n)

Discussion

The %U_WNDSTYLE function changes vertical spacing (leading) on a window-by-window and list-by-list basis. By default, leading is added to all windows, lists, and the application window, to allow for input windows and lists to incorporate edit controls with frames without clipping text. Calling %U_WNDSTYLE with D_NO_LEADING eliminates this extra vertical space. We do not recommend using this style on windows processed using I_INPUT, as any fields that are vertically adjacent will occlude each other.

%U_WNDTYPE – Return a window’s type



window_type = %U_WNDTYPE (*window_id*)

Return value

window_type

The type of the window: (^VAL)

D_WTYPE_ACTIVEX	Container window for an ActiveX control (i.e., window created by calling %AX_TKWIN or %AX_TKSINGLE).
D_WTYPE_COLUMN	Menu column (i.e., window created by the .COLUMN script command or by calling MB_END).
D_WTYPE_CONTAINER	Composite container window (i.e., window created by calling %C_CONTAINER with DC_CREATE).
D_WTYPE_DOTNET	Container window for a .NET form, .NET control, or WPF element (i.e., window created by %DOTNET_TKWIN).
D_WTYPE_INPUT	Input window (i.e., window created by the .INPUT script command or by calling the IB_END routine).
D_WTYPE_LISTCLASS	List class (i.e., window created by the .LISTCLASS script command or by calling L_CLASS).
D_WTYPE_SELECT	Selection window (i.e., window created by the .SELECT script command, by calling S_SELBLD, or by calling I_LDINP for a .FIELD script command with a SELECT qualifier).
D_WTYPE_TABS	Tabbed window (i.e., window created by %TS_TABSET).
D_WTYPE_TEXT	Text window (i.e., window for which T_SETUP has been called—there’s no other distinction in Toolkit between a text window and a general window).
D_WTYPE_OTHER	None of the above.

Arguments

window_id

The window ID whose type you want to retrieve. (n)

Discussion

The %U_WNDTYPE function returns a window's type, regardless of the routine used to load the window. (The D_WINDOW_TYPE subfunction of %E_INFO also returns a window's type. However, if U_DLWND was used to load the window, D_WINDOW_TYPE returns D_WTYPE_OTHER, so it's useful only for windows loaded with a type-specific routine—for example, I_LDINP for an input window.)

Note that in the future more window types may be added to the list of possible window types, so D_WTYPE_OTHER could at some point encompass different types of windows and should only be used as an “else” case.

Examples

The following example uses %U_WNDTYPE to determine how to process the windows in a window library.

```
count = %u_enumwnds(channel, max, window)
for ndx from 1 thru count
  begin
    xcall u_ldwnd(id, channel, window(ndx))
    using %u_wndtype(id) select
      (D_WTYPE_LISTCLASS),    call process_listclass
      (D_WTYPE_INPUT),       call process_input
    .
    .
    .
  endusing
end
```

The Utility Methods

This section discusses the syntax and arguments for the utility method subroutines. These method subroutines are written by you and their names specified as arguments to U_START and U_WNDEVENTS. UI Toolkit then calls them at the appropriate time.

U_START_METHOD – Perform application start-up processing

“%UWNDEVENTS_METHOD – Perform window event processing”

USTART_METHOD – Perform application start-up processing



```
subroutine USTART_METHOD
```

Discussion

USTART_METHOD is a subroutine you write (and name) when you want your program to perform application-specific start-up processing.

This subroutine is called by U_START (just prior to returning to the calling routine) if a start method was passed.

See also

- ▶ [U_START on page 16-136](#) for information about application start-up.
- ▶ [“Overloading UI Toolkit Routines” on page 18-8](#) for more information on creating this subroutine.

%UWNDEVENTS_METHOD – Perform window event processing

WT			

```
function UWNDEVENTS_METHOD, ^val
{reserved1}          , a
{reserved2}          , a
^val (window_id)     , i
^val (event)         , i
[arguments]          , type
```

Return value

False when a mouse click is not passed to the target window on a mouse click event; otherwise returns true. (n)

Arguments

{reserved1}
{reserved2}

Reserved for internal use. (a)

window_id

The ID of the window receiving the event. This must be declared as ^VAL(*window_id*). (i)

event

The ID of the event that occurred. This must be declared as ^VAL(*event*). (i)

D_EVENT_LEFT_CLICK	Left mouse button event.
D_EVENT_MIDDLE_CLICK	Middle mouse button event.
D_EVENT_RIGHT_CLICK	Right mouse button event.
D_EVENT_LEFT_DBLCLK	Double-click left mouse button event.
D_EVENT_MIDDLE_DBLCLK	Double-click middle mouse button event.
D_EVENT_RIGHT_DBLCLK	Double-click right mouse button event.
D_EVENT_MOVE	Move window event.
D_EVENT_SIZE	Size window event.
D_EVENT_MINIMIZE	Minimize window event.
D_EVENT_MAXIMIZE	Maximize window event.
D_EVENT_RESTORE	Restore window event.

D_EVENT_SCROLL Scroll window event.

D_EVENT_CLOSE Close window event.

arguments

One or more arguments specific to *event*. The MINIMIZE, MAXIMIZE, RESTORE, and CLOSE events have no additional arguments.

- All of the CLICK and DBLCLK events have the following arguments (but see Discussion for variations when using a list):

^VAL (arg_5) Position in x-direction (column in cells).

^VAL (arg_6) Position in y-direction (row in cells).

^VAL (arg_7) Position in x-direction (column in pixels).

^VAL (arg_8) Position in y-direction (row in pixels).

^VAL (arg_9) Additional mouse click information. One of the following:

D_AREA_CLIENT Click was in client area.

D_AREA_NONCLIENT Click was in non-client area.

D_BUTTON_LEFT Left mouse button was down during click.

D_BUTTON_MIDDLE Middle mouse button was down during click.

D_BUTTON_RIGHT Right mouse button was down during click.

D_KEY_CONTROL Control key was down during click.

D_KEY_SHIFT Shift key was down during click.

D_LSTCLK_FOOTER Click was in footer section.

D_LSTCLK_HEADER Click was in header section.

D_LSTCLK_ONLINE Click was on a line.

- **D_EVENT_MOVE** has the following arguments:

^VAL (arg_5) Position in x-direction (column in cells).

^VAL (arg_6) Position in y-direction (row in cells).

^VAL (arg_7) Position in x-direction (column in pixels).

^VAL (arg_8) Position in y-direction (row in pixels).

- **D_EVENT_SIZE** has the following arguments:

^VAL (arg_5) Width of logical window (number of columns).

^VAL (arg_6) Height of logical window (number of rows) or the number of visible list items when using a list.

- ^VAL (*arg_7*)** Width of client area (number of pixels).
- ^VAL (*arg_8*)** Height of client area (number of pixels).
- **D_EVENT_SCROLL** has the following arguments:
 - ^VAL (*arg_5*)** Display area position in x-direction (column in cells).
 - ^VAL (*arg_6*)** Display area position in y-direction (row in cells) or item number of the display area position when using a list.
 - ^VAL (*arg_7*)** Offset between window position and display area position in x-direction (number of pixels).
 - ^VAL (*arg_8*)** Offset between window position and display area position in y-direction (number of pixels). Always 0 when using a list.

Discussion

%UWNDEVENTS_METHOD is a function that you write and name. You do not call it. Instead UI Toolkit will call it when a mouse event that you have assigned to the method occurs.

To suppress a mouse click from being passed through to the target window, return a value of false for the %UWNDEVENTS_METHOD; otherwise always return true.



It is vital that the argument types be declared exactly as documented. Incorrect definition of arguments to your event method routine will cause undefined results.

When using a list with any of the CLICK and DBLCLK events, *arg_5* returns the column number clicked, and *arg_6* returns the row number clicked. For ActiveX Toolkit lists, the row number specifies the item's position in the currently loaded rows, both visible rows and those that don't fit in the window. For standard Toolkit lists, the row number specifies the item's position in the currently visible set of rows. However, note the following:

- When clicking on the header or footer in an ActiveX Toolkit list, *arg_5* returns the column number clicked, unless the header or footer is static, in which case it returns 1. *Arg_6* returns zero.
- When clicking on the header or footer for a standard Toolkit list, *arg_5* and *arg_6* return zero.
- When clicking on a line in an ActiveX Toolkit list, *arg_5* returns the column number to the left of the line that was clicked.



When using proportional fonts in conjunction with %UWNDEVENTS_METHOD, the column (x-direction) argument may not be correct because it is based on the sizing character for the window, and not the actual text.

See also

- ▶ [“Overloading UI Toolkit Routines” on page 18-8](#) for more information on creating this function.
- ▶ [“Appendix D: Methods”](#).

17

Debugger

What is the Window Debugger? 17-2

Explains what the debugger enables you to do, lists the debugger commands, and tells you how to access online help for the debugger.

Using the Debugger 17-3

Gives you the proper format for each debugger command and discusses what each command does.

What is the Window Debugger?

The window debugger is a feature of UI Toolkit that enables you to look into the windowing environment and determine what is really happening in the windows you’ve created.

The debugger supports the following commands:

Command	Purpose
CLEAR or CLS	Clear the display screen.
DISPLAY	Display windows or lists.
EXAMINE or SHOW	Provide status information.
QUIT	Exit the debugger.
REDRAW	Redisplay debugger system output lines.
WINDOW	Display information about a window.
LIST	Display information about a list.
FILESTACK	Display information about a file stack.

The commands and their arguments are not case sensitive and can be abbreviated.

To place more than one command on a line, separate them with commas.

For online help on a particular command, type **help** followed by the command name. For example:

```
help examine
```

Using the Debugger

Before you can run the debugger, you must enable it. There are three ways to do this:

- ▶ Set the logical or environment variable DTKDBG to any value.
If UI Toolkit generates an error while this environment variable is set, a window is displayed that asks if you want to examine the window system. If you answer **Y**, the UI Toolkit debugger opens. Note that when DTKDBG is set, the debugger checks for any %M_SIGNAL call that supersedes a pending menu entry, and it makes sure that a method passed to U_START can be invoked. For more information, see [DTKDBG](#) in the “Environment Variables” chapter of *Environment Variables & System Options*.

When you’re running a Toolkit program, you can explicitly enter debug mode by pressing CTRL+R while performing any Toolkit input. Neither the environment nor window system resources are affected while you are in debug mode. If DTKDBG is not in the logical table or environment, CTRL+R will never enter the debugger.
- ▶ Explicitly invoke the debugger without setting DTKDBG by using the U_DEBUG subroutine. See [U_DEBUG on page 16-31](#) for more information.
- ▶ Call the UI Toolkit window debugger from the Synergy debugger prompt.
At the debug prompt, type “windbg” to invoke the UI Toolkit window debugger. If UI Toolkit is not initialized (for example, before U_START or after U_FINISH), the command simply returns without doing anything. If the program was not linked with UI Toolkit, a “Cannot access external routine” error is generated. Otherwise, the UI Toolkit debugger takes over input until a QUIT command is issued. At that point, the Synergy debugger resumes control with a new prompt.

VMS

If your application is linked against the UI Toolkit shared image (**TKLIB.SH.EXE**), you must invoke OPENELB for the shared image in your application before this feature is enabled. Execute this statement prior to using this feature:

```
XCALL OPENELB ("TKLIB_SH")
```

where “**TKLIB_SH**” is the name of a logical symbol assigned to Toolkit’s shared image file. We do not perform this OPENELB automatically because it slows every XSUBR call. Your program should include it only if needed.

If you need to repaint the screen while you are in debug mode, press CTRL+R, CTRL+D. (On OpenVMS, press CTRL+R, CTRL+Z.)

On Windows, the debugger runs in a separate application window. The initial size and location of this window is user-definable. See environment variables [DBG_X](#), [DBG_Y](#), and [DBG_WIDTH](#) in “Environment Variables” chapter of *Environment Variables & System Options* for more information.

CLEAR or CLS

The CLEAR or CLS command clears the display screen and redisplay the debugger command prompt.

DISPLAY

The DISPLAY command shows the screen displays for components of the UI Toolkit environment.

```
display environment [level]
```

```
display windows [n, ...]
```

```
display lists n [, n,...]
```

where *level* is the environment to be displayed and *n* is the ID of a window or list to be displayed.

DISPLAY ENVIRONMENT shows the screen display of the specified environment. Displaying an environment level re-creates that level's display. If *level* is not specified, the current environment level is displayed.

DISPLAY WINDOWS shows the screen display of the specified window(s), placed on the screen in the order that you've specified them. If no windows are specified, the current list of windows is redisplayed.

DISPLAY LISTS shows the screen display of the specified list(s), placed on the screen in the order you've specified them. You must specify at least one list ID for this command.

Once a display of any kind has been generated, you can press any key to continue. All windows are removed and the debugger command prompt displays.

EXAMINE or SHOW

The EXAMINE or SHOW command provides status information about components of the UI Toolkit environment. Information in this section applies to both EXAMINE and SHOW.

```
examine defines
```

```
examine environment [level]
```

```
examine system
```

```
examine window n [fields] [user]
```

```
examine list n
```

where *level* is the environment level to be examined and *n* is the ID of the window or list to be examined.

EXAMINE DEFINES outputs status information about the current system definitions. The definitions are the following:

- ▶ Toolkit version being used
- ▶ Maximum allowable number of the following items:
 - ▶ Channels
 - ▶ Windows
 - ▶ Header and footer lines
 - ▶ Lists
 - ▶ Environment levels
 - ▶ Placed windows in all environments
 - ▶ Entries per column
 - ▶ Fields per input window
 - ▶ Sets per input window
 - ▶ Information text per window field
 - ▶ Characters per text row

EXAMINE ENVIRONMENT outputs status information about the specified environment level. If *level* is not specified, the debugger examines the current environment level. The environment status includes the following:

- ▶ Lists of local channels
- ▶ Windows, lists, and columns
- ▶ Placed windows, lists, and columns
- ▶ Terminal states (echo, cursor, lowercase, interrupt)
- ▶ Program states (wait on lock, wait on message)

EXAMINE SYSTEM outputs status information about the current system state. The system-wide information includes a list of all loaded columns and windows, channels in use, all lists, and all file stacks.

EXAMINE WINDOW (which is identical to [WINDOW](#)) outputs status information about the specified window. If *fields* is specified, the “raw” information about the window’s fields is also be output. If *user* is specified, the contents of the window’s user data set are also output. .

EXAMINE LIST (which is identical to [LIST](#)) outputs status information about the specified list.

EXAMINE FILESTACK (which is identical to [FILESTACK](#)) outputs status information about a file stack.

QUIT

The QUIT command exits the window debugger. If you entered the debugger from a screen input field, that field's input will be redisplayed. If you entered the debugger after a fatal error was generated, the program will terminate.

REDRAW

On Windows, by default you can access 300 lines of output by scrolling. You can change the number of available lines by setting the `DBG_BUFFER` environment variable in **synergy.ini** or **synuser.ini** to a different value.

On UNIX and OpenVMS, the REDRAW command redisplay the last full screen (24 lines) of output produced by the debugger system.

WINDOW

The WINDOW command displays detailed windowing system information for a specific window.

`window n [environment level] [fields] [user]`

where *level* is the environment level at which the window should be examined and *n* is the ID of the window for which information should be displayed. If *level* is not specified, the window is examined at the current level. *Fields* specifies that the parameters of all window fields are to be displayed. *User* specifies that information in any user data set is to be displayed.

The WINDOW command always displays the following information about the window:

- ▶ Name
- ▶ Size
- ▶ Placement
- ▶ Title
- ▶ Cursor position
- ▶ Display area
- ▶ Processing area
- ▶ Characteristics (whether or not it is placed, occluded, border is on or off, title is displayed)
- ▶ Renditions (for window, border, and title)
- ▶ Number of fields
- ▶ Current field
- ▶ Number of user-data entries
- ▶ Size of user-data entries
- ▶ Current user-data entry

If *fields* is specified, the following is displayed for each field:

fld = (*row* : *col* : *len*)

where *fld* is the field number, *row* and *col* are the window-relative placement coordinates for the field, and *len* is the field's length.

If *user* is specified, for each user-data entry, the entry number followed by the data itself is displayed. Up to 50 characters per line are displayed and, if needed, continued on subsequent lines. Any non-printable character in the data is replaced by a period (.).

LIST

The LIST command displays information for a specific list.

`list n`

where *n* is the ID of the list for which you want information. This command displays the following:

- ▶ Current item number
- ▶ Last item number (highest loaded)
- ▶ Current row (ordinal item number in visible portion of the list)
- ▶ Current item status (whether or not it is null or if it is at the top or bottom of list)
- ▶ Associated input window ID
- ▶ List status (enabled or disabled)
- ▶ Current find parameters (case sensitivity, direction, search type, area of the optional data to be searched, the search string, and the length of the string)
- ▶ Find status (whether the last find was successful)

FILESTACK

The FILESTACK command displays information for a file stack.

`filestack [n] [level level] [record record]`

where *n* is the file-stack channel, *level* is the level of the file stack, and *record* is a record within that level. If the channel (*n*) is not specified, the default file stack is accessed. If *level* is not specified, the highest level on the file stack is used. If *record* is not specified, none is assumed.

The following information is always displayed: file-stack channel, level, record size, highest record written at that level.

If *record* is specified, the contents of that record are also displayed. Any non-printable character is replaced by a period (.).

18

Customizing UI Toolkit

Customizing Configuration Fields 18-2

Discusses the `tkctl.def` and `tools.def` fields you can change.

Overloading UI Toolkit Routines 18-8

Defines user-overloadable routines and explains how to invoke them.

Customizing Key Mapping for Menu Shortcuts 18-11

Explains what key mapping is, how key mapping works with your system, and how and why you might want to change the key maps.

Customizing the Look of Your Application 18-24

Explains renditions and rendition schemes, and explains how they work with the color palette and, on Windows, Synergy colors.

Using Fonts on Windows 18-40

Explains how to define font palette entries, set the font for an application, set the font for an object, and use proportional fonts on Windows.

Customizing Configuration Fields

The **tkctl.def** and **tools.def** files contain a number of configuration fields that you can modify from within your program to change operational characteristics of UI Toolkit. These fields are listed below. In the sections below, “false” indicates a value of zero; “true” indicates a nonzero value.



If you want to modify any of these fields, you must first **.INCLUDE** the corresponding file(s) in your program.

When using Toolkit code in Synergy .NET, there are some limitations to the ways you can use these configuration fields. See [“Restrictions for global fields” on page 1-22](#).

tkctl.def

Tkctl.def includes the following fields, which are initialized by U_START.

g_ActiveX_list

Specifies whether Toolkit will use the ActiveX list control for lists (true/false). See [“Specifying and determining the list type” on page 9-5](#) for more information. This field is initialized to true if the environment variable **ACTIVEX_LIST** is set to 1. Otherwise, it is initialized to false.

g_beep

Turns on or turns off the terminal bell (on all platforms) and MessageBeep (on Windows). If **g_beep** is set to true, the terminal bell and MessageBeep are enabled. If it is set to false,

- ▶ calls to **U_BEEP** have no effect.
- ▶ runtime routines other than **%U_MSGBOX** suppress the terminal bell and MessageBeep. Note that on Windows, the **g_beep** setting does not affect beeps for message boxes or other sounds associated with message box icons.

U_START initializes this field to false if the **DTK_BEEP** environment variable is set to 0 or “off” (case-insensitive) in **synuser.ini**, **synergy.ini**, or the environment. Otherwise **U_START** initializes it to true (the default). Note that an application can set **g_beep**, and the setting will determine subsequent behavior.

g_chk_char

Specifies the check mark character to be used by **S_SELECTCB**, **S_UPDATECB**, and **U_MODREND**. Note the following:

- ▶ On Windows, this field affects only check marks for check boxes that aren’t framed (using **I_FRAMES**, **I_INPUT**, or **I_INPFLD**). Framed check boxes always use the Windows standard check mark character.
- ▶ This field is initialized to “X” on most environments.

g_date_order

Defines date input and display. See “[DATE, NODATE](#)” on page 2-42 for information. This field is initialized to 0.

g_date_sep

Specifies the character to be used to separate parts of a date when displayed. This field is initialized to a forward slash (/).

g_dtkbounds

Controls DTK_BOUNDS checking behavior after U_START is called. See [DTK_BOUNDS](#) in the “Environment Variables” chapter of *Environment Variables & System Options* for information. This field is initialized to 1 if the DTK_BOUNDS environment variable is set to 1. It is initialized to 2 if DTK_BOUNDS is set to 2. Otherwise, it is initialized to 0, which turns off bounds checking. In **tools.def**, there are defines for 1 (D_BOUNDS_OVERRUN) and 2 (D_BOUNDS_EXACT).

g_edt_fwrd_msg

Defines the edit forward direction message. This field is initialized to “Forward”. (The message string is read from the Synergy/DE text message file.)

g_edt_ins_msg

Defines the edit insert mode message. This field is initialized to “Insert”. (The message string is read from the Synergy/DE text message file.)

g_edt_ovr_msg

Defines the edit overstrike mode message. This field is initialized to “Overstrike”. (The message string is read from the Synergy/DE text message file.)

g_edt_rev_msg

Defines the edit reverse direction message. This field is initialized to “Reverse”. (The message string is read from the Synergy/DE text message file.)

g_entnamctab (Windows only)

Specifies the name of the menu entry to be activated when CTRL+TAB is pressed. This field is initialized to blank (no entry).

g_entnamcstab (Windows only)

Specifies the name of the menu entry to be activated when CTRL+SHIFT+TAB is pressed. This field is initialized to blank (no entry).

g_entnamtab (Windows only)

Specifies the name of the menu entry to be activated with TAB is pressed. This is initialized to blank (no entry).

g_entnamstab (Windows only)

Specifies the name of the menu entry to be activated when SHIFT+TAB is pressed. This is initialized to blank (no entry).

g_eof_max

To facilitate redirection of input, this field specifies the maximum number of successive end-of-file (EOF) characters to allow on input. (On UNIX, these are CTRL+D. On OpenVMS, these are CTRL+Z.) After the specified maximum number of EOF characters are encountered, the program will stop and display a message saying that too many EOF characters were encountered.

This is initialized to 100. To disable the behavior described above, set it to 0.

g_inp_trm_key

Defines the completion key code for the I_INPUT subroutine. This field is initialized to RETURN_KEY. (RETURN_KEY is defined in **inpctl.def**.)

g_inp_trm_msg

Defines the I_INPUT termination processing message. This field is initialized to “Make corrections or press RETURN to complete input”. (The message string is read from the Synergy/DE text message file.)

g_periods

Defines the maximum value for period-style dates. This field is initialized to 12.

g_plc_col_args

Determines whether Toolkit will place menu columns that are passed as arguments to Toolkit routines (true/false). When set to true, Toolkit places and removes menu columns as needed. When set to false, Toolkit ignores menu column arguments. This field is initialized to false for Windows and true for UNIX and OpenVMS.



Although you can change the value of **g_plc_col_args** on any supported platform after U_START is called, we recommend leaving it set to true for UNIX and OpenVMS because on these platforms menu columns must be placed to make their functions available to users.

g_retaincontext (Windows)

Determines how context is handled when a menu entry is signaled from a change, display, or leave method and a user action (mouse click, ENTER, TAB, SHIFT+TAB, etc.) moves input context to a subsequent field or button. Prior to UI Toolkit version 8.3, if a menu entry was signaled in this situation, Toolkit would incorrectly apply the menu entry processing to the subsequent field or button rather than to the field whose method called %M_SIGNAL. This was corrected in Toolkit version 8.3. By default, context now remains on the field whose method signaled the menu entry, which matches Toolkit behavior on UNIX and OpenVMS. You can, however, restore the previous behavior. If **g_retaincontext** is set to true, Toolkit emulates Windows behavior for Toolkit versions prior to 8.3 (it moves input context to the subsequent field or button). If **g_retaincontext** is set to false (the default), context remains on the field whose method signaled the menu entry. Note the following:

- ▶ The **g_retaincontext** field applies only when using I_INPUT. It does not apply to any menu entry signaled from outside of the input routine.
- ▶ If you set **g_retaincontext**, set it from the method that signals the menu entry, or set it before calling that method. If you don't, the setting won't apply until the next such situation (the next time a menu entry is signaled from a method and a user action moves input context to a subsequent field or button).

This field is initialized to true if the RETAIN_CONTEXT_CHANGE_ON_SIGNAL environment variable is set to 1. If RETAIN_CONTEXT_CHANGE_ON_SIGNAL is set to any other value or is undefined, **g_retaincontext** is initialized to false.

g_tabfacecolor

Determines which color palette entry will be applied to a window when the window is added to a tab set. If this field is set to 0 (or a value greater than 16), no color palette entry is applied to a window when it is added to a tab set; the window's color remains as is.

On Windows, this field is initialized to 9 if themes are enabled; otherwise, it is initialized to 0.

On UNIX and OpenVMS, this field is initialized to 0.

g_throwabort

Determines the behavior of U_ABORT. If **g_throwabort** is set to zero, U_ABORT pops up a message box and performs a STOP. If **g_throwabort** is set to a non-zero value, U_ABORT will instead call U_FINISH to shut down Toolkit and then use EXITE to throw a trappable error.

This field is initialized to true (1) if the environment variable DTK_THROW_ABORT is set to 1. Otherwise, this is initialized to false (0).



If you want U_ABORT to throw a trappable error during U_START processing (i.e., call U_FINISH and then EXITE), set **g_throwabort** before calling U_START. However, note that U_START will reset **g_throwabort** to the value of DTK_THROW_ABORT, so you may need to set **g_throwabort** again after the U_START call.

`g_time_entry`

Specifies the name of the time-out menu entry. This field is initialized to `O_TIMEOUT`.

`g_time_sep`

Defines the character used to separate parts of a time field when displayed. This field is initialized to a colon (:).

`g_wait_time`

Specifies a default time-out value. This field is initialized by `U_START` to `Q_WAIT`, which is equivalent to `D_FOREVER`, `-1`, or `FOREVER` (for `.FIELD`), but can be any one of the following:

- ▶ `D_FOREVER` (or `-1`), which instructs Toolkit to wait until I/O processing is complete (i.e., no time-out).
- ▶ `D_IMMEDIATE` (or `0`), which instructs Toolkit to immediately time out.
- ▶ *n* (a positive number), which instructs Toolkit to wait *n* seconds before timing out.

See the following for more information:

- ▶ [AX_TIMEOUT on page 4-16](#).
- ▶ [AX_INPUT on page 4-12](#).
- ▶ [WAIT](#) in the “Synergy DBL Statements” chapter of the *Synergy DBL Language Reference Manual*.

`g_wt_alrt_key`

Defines the completion key code for the alert state of the `U_WAIT` subroutine. This field is initialized to `RETURN_KEY`. (`RETURN_KEY` is defined in **inpctl.def**.)

`g_wt_alrt_msg`

Defines the display message for the alert state of the `U_WAIT` subroutine. This field is initialized to “Press RETURN to continue”. (This message string is read from the Synergy/DE text message file.)

`g_wt_err_key`

Defines the completion key code for the error state of the `U_WAIT` subroutine. This field is initialized to `CTRL_G_KEY`. (`CTRL_G_KEY` is defined in **inpctl.def**.)

`g_wt_err_msg`

Defines the display message for the error state of the `U_WAIT` subroutine. This field is initialized to “Press ^G to continue”. (The message string is read from the Synergy/DE text message file.)

tools.def

Tools.def includes the following fields.

g_netnativemsgbox

In Synergy .NET, Toolkit uses native Windows message boxes for the following routines if **g_netnativemsgbox** is set to true: U_MESSAGE, %U_MSGBOX, and U_WAIT. If this field is set to false, which is the default, Toolkit uses cell-based-style message boxes for these routines in Synergy .NET.

g_txt_rtrn

When set to false, pressing ENTER while in an input window text field will be interpreted as completion of input. If you want ENTER to insert a new line, set the variable to true. This field is initialized to false.

g_txt_softreturn

Controls the way Toolkit handles line ends on Windows when using T_EDIT or when using I_INPUT with multi-line text fields. When set to 0, Toolkit retains line ends in incoming data and uses those line ends when displaying the data. When set to a non-zero value, Toolkit assumes that line ends are the result of wrapping and discards them, rewrapping lines as needed when displaying the data. This field is initialized to 0.

g_updt_fld

When performing “DONE” processing (passing ***DONE*** to I_NEXT or selecting an I_DONE menu entry), UI Toolkit updates the window system as each field in the input set is processed. If you have a lot of input fields being set by the DONE processing, and you want to improve performance, set this variable to false. This field is initialized to true.

g_updt_set

When performing “DONE” processing (passing ***DONE*** to I_NEXT or selecting an I_DONE menu entry), UI Toolkit does not update the window system until the next input occurs (unless **g_updt_fld** is set). If you want to force a window system update after DONE processing (for instance if your program does some lengthy processing before the next input might normally occur), set this variable to true. This field is initialized to false.

Overloading UI Toolkit Routines

UI Toolkit includes many routines that you can overload. At runtime, you register the name of your customized routine, and UI Toolkit will call it at the appropriate time. If you do not register your routine, Toolkit will use the default implementation.

You can name your routines almost anything you want (following general routine naming conventions). However, do not use the prefix “TKP_”, which is used internally by UI Toolkit for its default implementation of customizable routines.

The following routines can be overloaded at runtime:

<code>%EAPPMOVE_METHOD</code>	<code>IARRIVE_METHOD</code>
<code>%EAPPSIZE_METHOD</code>	<code>%ICHANGE_METHOD</code>
<code>%EAPPSTATE_METHOD</code>	<code>%IDISPLAY_METHOD</code>
<code>%ECENTURY_METHOD</code>	<code>IDRILL_METHOD</code>
<code>ECHKFLD_METHOD</code>	<code>%IEDITFMT_METHOD</code>
<code>ECLOSE_METHOD</code>	<code>IHYPER_METHOD</code>
<code>EDSPFLD_METHOD</code>	<code>ILEAVE_METHOD</code>
<code>EEDTDSP_METHOD</code>	<code>LARRIVE_METHOD/LLEAVE_METHOD</code>
<code>EENTRST_METHOD</code>	<code>LDBLCLICK_METHOD</code>
<code>EFKEY_METHOD</code>	<code>LLOAD_METHOD</code>
<code>EHELP_METHOD</code>	<code>USTART_METHOD</code>
<code>ESCRIPTERR_METHOD</code>	<code>%UWNDEVENTS_METHOD</code>
<code>EUTILS_METHOD</code>	

There are various ways to overload these routines. In general, you need to do the following:

1. Write your customized routine.
2. Register your routine.
3. Make sure your customized routine is available to UI Toolkit. See [“Making your routine available to UI Toolkit”](#) below. Refer to the specific user-overloadable routines for more detailed information.

Making your routine available to UI Toolkit

In order for Toolkit to be able to locate your customized routine, one of the following must be true:

- ▶ The routine resides in your application's main routine.
- ▶ The routine resides in an ELB that's linked with your application, which you have previously opened with OPENELB or specified when you registered the routine.
- ▶ The routine resides in an OpenVMS shared image file you have previously opened with OPENELB or specified when you registered the routine.
- ▶ The routine is explicitly linked into your application. (Even if you do not call these routines yourself, you still need to reference them explicitly in order for them to be linked into your application. A tactic for creating an explicit reference to a customized routine is to include "dummy" calls after the XRETURN or FRETURN in a routine you know will be explicitly referenced, such as a start-up routine, or after the STOP in a main routine. This method is required if you use OpenVMS object libraries.)

Overloading examples

Example 1: Creating explicit references

In the following example, the routines **my_help** and **my_utils** are explicitly linked into the application by specifying dummy references to them within the application but outside of the code sequence. The routine **my_help** is registered as the help method, and the routine **my_utils** is the utility method. They are drawn from an ELB or shared image that's linked with the application.

```
subroutine my_startup

proc
    xcall u_start(...)

    xcall e_method(D_METH_HELP, "my_help", D_METH_UTILS, "my_utils")
    .
    .    (other application-specific start-up code)
    .
xreturn

xcall my_help           ;Dummy references to ensure availability.
xcall my_utils          ;Note that no arguments are required because these
                        ; statements are never actually invoked.

endsubroutine
```

Example 2: Specifying the library

In the example below, the routine **myhelp** is registered as the help method and will be drawn from whatever source is available (program image or most recently opened ELB). The utility method will be the routine **myutils**, and the function key mapping method will be **myfkey**, both of which are drawn from the ELB or shared image pointed to by the logical symbol USRLIB. A different library, pointed to by the logical symbol APPLIB, is used to supply the user-defined field support methods **mychkfld**, **mydspfld**, and **myedtdsp**. The routine **myclose** is used for the application close method, but no specific library will be used (as with **myhelp** above). The Toolkit is initialized with the window library **mywnds**, with one header and one footer line.

```
xcall u_start("mywnds", 1, 1,,,,,,,,, "mystart", "USRLIB:") ;Start Toolkit

subroutine mystart
;
; User start-up routine example which also registers other methods
;
.include "WND:tools.def"

proc
  xcall e_method(D_METH_HELP, "myhelp",
    &          D_METH_LIBRARY, "USRLIB:",
    &          D_METH_UTILS, "myutils",
    &          D_METH_FKEY, "myfkey",
    &          D_METH_LIBRARY, "APPLIB:",
    &          D_METH_CHKFLD, "mychkfld",
    &          D_METH_DSPFLD, "mydspfld",
    &          D_METH_EDTDSP, "myedtdsp",
    &          D_METH_LIBRARY,,
    &          D_METH_APPCLOSE, "myclose")
  xreturn
endsubroutine
```

Customizing Key Mapping for Menu Shortcuts

Toolkit enables you to create menu shortcuts by mapping keys and key sequences to menu entries. As distributed, Toolkit contains default key mappings for most terminal types, but you can create custom mappings for individual keys and key sequences, and you can create sets of mappings (called *key maps*) for terminals and for users.

When a user presses a nonstandard key or key sequence during an input routine, Toolkit consults the key map, which is a set of shortcuts for the terminal or user, to see if there's a mapping for the escape sequence it received. We call this *runtime terminal binding*. If there is, Toolkit then looks at the menu definitions created by MB_ENTRY or .ENTRY to find a match. If there is a match, Toolkit loads the menu entry name into the **g_entnam** global variable and returns control to your application. If there is no key mapping for the escape sequence or match for the key mapping, Toolkit rings the keyboard bell and ignores the escape sequence.

Because of runtime terminal binding, your application can be made to handle input from various types of terminals and keyboards. In addition, you can enable your users to create their own shortcuts by including a call to the U_EDITKEYS routine in your application.

You can create shortcuts by doing the following:

- ▶ **Specifying shortcuts in menu entry definitions.** If you use .ENTRY or MB_ENTRY and use the ^key form to specify a shortcut, your shortcut is complete. (See [.ENTRY on page 2-27](#) or [MB_ENTRY on page 10-66](#) for more information.) On UNIX and OpenVMS, however, if you use .ENTRY and specify an arrow key or use the f# form for a shortcut, you must make sure the function code is mapped to a key or key sequence in the key map(s) that will be used with the application. (On Windows, functions codes are automatically mapped to a native Windows accelerator table.)
- ▶ **Mapping keys in key maps.** A key map is a set of shortcuts for a terminal or keyboard type. Key maps are stored in a Synergy database called a key map file (which is **dtkmap.ism** by default). The key map file as it's distributed includes several key maps, but you can use the key mapping utility to add your own key maps, modify key maps, and delete key maps. See [“Customizing with the key mapping utility \(UNIX and OpenVMS\)” on page 18-15](#) for more information.
- ▶ **Defining identifiers in the key mapping script file.** Identifiers defined in the key mapping script file (which is **keymap.ctl** by default) can be used as the *shortcut* argument in .ENTRY definitions. This can make shortcut definitions in script files easier to understand. See [“Modifying shortcut identifiers for menu entries defined in a script” on page 18-19](#) for more information.

WIN

Key mappings are fixed on Windows. Only the text for a key map can be changed (the text that's displayed next to the menu entry) because function codes on Windows are mapped to a native Windows accelerator table. If you use the key mapping utility to view the key map for Windows, you'll see both the shortcuts and the shortcut text, but only the shortcut text can be changed.

Note the following:

- ▶ Some keys and key sequences are reserved. CTRL+M, for example, is reserved for carriage return, and it can't be mapped to any other function. The other reserved keys and key sequences, which are listed in [“Customizing reserved keys” on page 18-22](#) can be mapped to other functions.
- ▶ If there are conflicts, entries in key maps override entries in the key mapping script file and shortcuts specified with .ENTRY or MB_ENTRY. Shortcuts specified with .ENTRY must not conflict with entries in the key mapping script file. If they do, errors will be generated when you generate the window libraries.
- ▶ Shortcut text (text displayed next to a menu entry) can be specified only in key maps. (Shortcut text specified with MB_ENTRY or the key mapping script file is ignored.)

Files and environment variables used for key mapping

The following files are used for key mapping.

File	Purpose
inpctl.def	Defines identifiers passed as the <i>shortcut</i> argument to MB_ENTRY.
key map file (dtkmap.ism by default)	Associates keys (escape sequences) with internal function codes. To locate this file, Toolkit looks first for the file and directory specified by DTKMAPFIL if that environment variable is set. Toolkit then looks for dtkmap.ism in the directory specified by DTKMAP if that's set. If the file isn't there or can't be opened, Toolkit looks in the WND directory.
key mapping script file (keymap.ctl by default)	As distributed, this file contains a copy of keymap.msw or keymap.vt depending on the operating system or terminal type. (See “Modifying shortcut identifiers for menu entries defined in a script” on page 18-19.) To locate the key mapping script file, Toolkit looks first for the directory and filename specified by DTKKEYCTLFIL if it's set. If DTKKEYCTLFIL isn't set, Toolkit looks for keymap.ctl in the current working directory. And if the file isn't there, Toolkit looks for keymap.ctl in the WND directory. If Toolkit can't find a key mapping script file, the only shortcuts supported by scripts are “F1” through “F63”.

File	Purpose
keymap.msw	Contains shortcut identifier definitions for Windows systems. On Windows, keymap.ctl is a copy of this file.
keymap.vt	Contains shortcut identifier definitions for VT-style keyboards. On UNIX and OpenVMS, keymap.ctl is a copy of this file.
keys.dbl	Enables you to test menu shortcuts. See “Using keys.dbl to test shortcuts (UNIX and OpenVMS)” on page 18-21.
script file	Assigns shortcuts or function codes to menu entries. (You create this. See chapter 2, “Script,” for information.)
window library	Contains the shortcuts for menu entries. Proto and Script generate this from the script file and keymap.ctl .

The following environment variables are used in key mapping:

Environment variable	Purpose
DTKKEYCTLFIL	Defines the location and name of the key mapping script file. If DTKKEYCTLFIL is not set, Toolkit looks for the default key mapping script file (keymap.ctl) in the current working directory or, if it's not there, in the WND directory. To specify a different name or location, set DTKKEYCTLFIL to the full path and name of the file. See DTKKEYCTLFIL in the “Environment Variables” chapter of <i>Environment Variables & System Options</i> for more information.
DTKMAP	Defines the location of the default key map file, dtkmap.ism . Use this environment variable if you want to change the location (but not the name) of the key map file. This setting is ignored if DTKMAPFIL is set. See DTKMAP in the “Environment Variables” chapter of <i>Environment Variables & System Options</i> for more information.
DTKMAPFIL	Defines the location <i>and</i> name of the key map file. By default, the key mapping file is named dtkmap.ism and is located in the WND directory. To specify a key map file with a different name or location, set DTKMAPFIL to the path <i>and</i> name of the file. This environment variable overrides DTKMAP. See DTKMAPFIL in the “Environment Variables” chapter of <i>Environment Variables & System Options</i> for more information.

Environment variable	Purpose
DTKTERM ^a	Specifies a key map for a terminal. The value of DTKTERM is used as the database key to locate the appropriate key map in the key map file (which is a database). For this purpose, DTKTERM overrides TERM. (It doesn't override TERM as a setting for terminal capabilities on UNIX systems.) See DTKTERM in the “Environment Variables” chapter of <i>Environment Variables & System Options</i> for more information.
TERM ^a	Specifies a key map for a terminal if DTKTERM is not defined. See TERM in the “Environment Variables” chapter of <i>Environment Variables & System Options</i> for more information.
WND	Specifies the default location of dtkmap.ism and keymap.ctl , among other things. See WND in the “Environment Variables” chapter of <i>Environment Variables & System Options</i> for more information.

- a. If neither DTKTERM nor TERM is defined, Toolkit uses MSWINDOWS in Windows environments and VT100 elsewhere.

As distributed, the key map file (**dtkmap.ism**) includes the following key maps:

Key map name	Terminal type
ANSI	UNIX console
GENERIC	Unknown terminal type
HFT	IBM high-function terminal
IBMPC	PC
MSWINDOWS	Windows
VT100	VTxxx terminal
WY60	Wyse-60 and compatibles
XTERM	X-Windows terminals

Customizing with the key mapping utility (UNIX and OpenVMS)



Because the key mapping utility modifies key maps for *all* Toolkit applications on UNIX and OpenVMS, *including the key map utility itself*, we recommend saving a copy of the original default key map file (**WND:dtkmap.ism**) with a different name before you open the key map utility or modify a key map. Then, if you make unwanted changes that you can't undo, you can go back to the original key maps and start again.

Running the key mapping utility

Open the key mapping utility by doing one of the following:

- ▶ Run Proto and select General > Key mapping.
- ▶ Run the Synergy UI Toolkit Control Panel and select Preferences > Key mapping.

You can also run this utility from your application by calling U_EDITKEYS.

Adding a key map for a terminal or user

Most terminals on UNIX and OpenVMS systems can emulate VT-style keyboards. If you use a terminal that doesn't, you need to create a key map for that specific terminal type. You can also create a key map for a specific user.

You can create a key map from scratch (see below), or you can copy an existing key map, customize it, and then save it with a new name. (See [“Creating a new key map from an existing key map” on page 18-17.](#))

Creating a key map from scratch

1. Set the DTKTERM environment variable to the GENERIC key map. For example,
`SET DTKTERM=GENERIC`
2. Open the key mapping utility. (See [“Running the key mapping utility” on page 18-15.](#))



If you set DTKTERM for an environment, rather than setting it for the system, open the key mapping utility in the same environment.

3. The GENERIC (blank) key map is displayed. Select Key mapping> Add or change terminal type.
4. Enter a name (with a maximum of 10 characters) for the new key map at the “Enter New Terminal Type to Modify” prompt. For example, if you are creating a key map for the IBM 3151 terminal, you could enter IBM3151. If you are creating a key map for a specific user, you could enter something like SMITH_W_H.
5. If a key map with that name doesn't exist, enter **Y** at the “Do you wish to add it?” prompt, and press ENTER. An empty key map layout is displayed.

6. If a key map with that name already exists, it is displayed. Select General > Exit. Then select Key mapping from the General or Preferences menu and start again at [step 3](#).
7. Highlight the entry you want to modify. (Use the arrow keys to move around the key map and highlight key map entries.) Then press ENTER.
8. In the Key Definition window, enter key map information. All fields are optional.
 - ▶ If the key map is for a keyboard type that is different than the type you are using, use the Codes fields to enter the ASCII code(s) for the key or key sequence you want to associate with the function.
 - ▶ If the key map is for the type of keyboard you are now using, select Key mapping > Accept key sequence. This moves you to fields above the Codes fields. Here you can define the entry by pressing the actual key sequence (rather than by entering the ASCII code for the key sequence).
 - ▶ To clear an entry in one of the Codes fields, press the spacebar when the entry is highlighted, and then press ENTER to move to the next field.
 - ▶ To clear an entry in the Key Sequence, Alternate #1, or Alternate #2 fields, press ENTER when the entry is highlighted.
 - ▶ To cancel changes in this window, select Key mapping > Cancel current changes.

Key Sequence/Codes. If you are on the Key Sequence line (rather than Codes), press the key or key sequence you want to associate with the function. (The key mapping utility automatically fills the Codes field with the characters for the corresponding ASCII codes.) Otherwise, in the Codes field, enter the ASCII code(s) for the key or key sequence you want to associate with the function.

Alternate #1/Codes. If you are on the Alternate #1 line (rather than Codes), press the key or key sequence you want to use as the alternate for the function. (The key mapping utility fills the Codes field with the characters for the corresponding ASCII codes.) Otherwise, in the Codes field, enter the ASCII code(s) for an alternate key or key sequence you want associated with the function.

Alternate #2/Codes. If you are on the Alternate #2 line (rather than Codes), press the key or key sequence you want to use as the second alternate for the function. (The key mapping utility fills the Codes field with the characters for the corresponding ASCII codes.) Otherwise, in the Codes field, enter the ASCII code(s) for a second alternate key or key sequence you want associated with the function.

Menu Shortcut Text. Enter the text you want to display in the menu column for the shortcut.

Note that for the Submenu Flag entry, you can edit only the Menu Shortcut Text field. This is because this entry defines the characters on UNIX and OpenVMS systems that mark a menu entry as having submenus. By default, the submenu flag is set to a dash and a greater-than symbol (->). (On Windows, this setting has no effect. Windows has its own, fixed submenu indicator.)

9. Close the Key Definition window by selecting General > Exit or by pressing ENTER while you are on the Menu Shortcut Text field.

10. Continue mapping key sequences to functions by repeating [step 7](#) through [step 9](#). When you're finished assigning function codes, exit the key mapping utility by selecting General > Exit.
11. Enter Y at the "Save current changes to ...?" prompt to save the key map with the specified name, or enter N to save it with a different name. Note that you can't cancel changes at this point; you must save the key map. If you don't want the key map, go ahead and save it; then follow the instructions in ["Deleting a key map for a terminal or user" on page 18-18](#) to delete it.
12. Test the key map by setting DTKTERM to the key map name (specified in [step 4](#)) and then running your application from a machine with a keyboard that the key map was designed for.

If you created a key map for a specific user, set DTKTERM in the user's environment before you test.

Creating a new key map from an existing key map

1. Open the key mapping utility. (See ["Running the key mapping utility" on page 18-15](#).)
2. Select Key mapping > Add or change terminal type.
3. Enter a name (with a maximum of 10 characters) for the new key map at the "Enter New Terminal Type to Modify" prompt.

For example, if you are creating a key map for the IBM 3151 terminal, you could enter IBM3151. If you're creating a key map for a specific user, you could enter something like SMITH_W_H.
4. If a key map with that name doesn't exist, enter Y at the "Do you wish to add it?" prompt, and press ENTER. An empty key map layout is displayed.
5. If a key map with that name already exists, it is displayed. Unless you want to overwrite this key map, select General > Exit, and then select Key mapping from the General or Preferences menu and start again at [step 2](#).
6. Select Key mapping > Copy terminal definition.
7. At the "Enter terminal type to copy from:" prompt, enter the name of the key map you want to use as the basis for the key map you are creating.

The key map is displayed.
8. Highlight an entry you want to modify. (Use the arrow keys to move around the key map and highlight key map entries.) Then press ENTER.
9. In the Key Definition window, enter key map information. See [step 8](#) above for details on entering data in these fields.
10. Close the Key Definition window by selecting General > Exit or by pressing ENTER while you're on the Menu Shortcut Text field.
11. Continue mapping key sequences to functions by repeating [step 8](#) through [step 10](#). When you're finished assigning function codes, exit the key mapping utility by selecting General > Exit.

12. Enter N at the “Save current changes to ...?” prompt. Then, enter a name for your new key map at the “Enter name of terminal to save current changes” prompt. You can use up to 10 characters.
13. Test the key map by setting DTKTERM to the key map name (specified in [step 12](#)) and then running your application from a machine with a keyboard that the key map was designed for.

If you created a key map for a specific user, set DTKTERM in the user’s environment before you test.

Modifying an entry in a key map

1. Open the key mapping utility. (See [“Running the key mapping utility” on page 18-15.](#))
2. The default key map for your operating system or terminal is displayed. If you want to modify an entry in another key map, select Key mapping > Add or change terminal type. Then, at the “Enter New Terminal Type to Modify” prompt, enter the name of the key map you want to modify.
3. Highlight the entry you want to modify. (Use the arrow keys to move around the key map and highlight key map entries.) Then, press ENTER.
4. In the Key Definition window, enter key map information. See [step 8](#) in [“Creating a key map from scratch”](#) for details on entering data in these fields.
5. Close the Key Definition window by selecting General > Exit or by pressing ENTER while you’re on the Menu Shortcut Text field.
6. Continue mapping key sequences to functions by repeating [step 3](#) through [step 5](#). When you’re finished assigning function codes, exit the key mapping utility by selecting General > Exit.
7. Enter Y at the “Save current changes to ...?” prompt to save the key map with the specified name, or enter N to save it with a different name. You can’t cancel changes at this point; you must save the key map. If you don’t want the key map, go ahead and save it; then follow the instructions in [“Deleting a key map for a terminal or user”](#) below to delete it.
8. Test the key map by setting DTKTERM to the key map you specified in [step 3](#) and then running your application from a machine with a keyboard that the key map was designed for.

Deleting a key map for a terminal or user

1. Open the key mapping utility. (See [“Running the key mapping utility” on page 18-15.](#))
2. If you want to delete a key map other than the one that’s displayed in the Select Functions for... window, select Key mapping > Add or change terminal type. Then, at the “Enter New Terminal Type to Modify” prompt, enter the name of the key map you want to modify.
3. Select Key mapping > Delete terminal definition.
4. Enter Y at the “Delete key map for” prompt to delete the current key map or N to cancel.
5. To exit the “Enter New Terminal Type to Modify” prompt, select General > Exit. Select General > Exit a second time to exit the Key mapping utility.

Customizing the submenu flag

By default on UNIX and OpenVMS systems, a right arrow symbol (->) is displayed next to menu entries that have submenu columns. (On Windows, the submenu flag setting has no effect. Windows has its own, fixed submenu indicator.)

1. Open the key mapping utility. (See [“Running the key mapping utility” on page 18-15.](#))
2. Use the arrow keys to move to the “Submenu Flag” entry and then press ENTER.
3. In the Key Definition input window, enter the menu text you want to use for the submenu flag, and press ENTER.
4. Select General > Exit to exit the Key mapping utility.

Modifying shortcut identifiers for menu entries defined in a script

If you use script files to create menus, you can customize the way shortcuts for menu entries are defined in scripts by adding, deleting, or changing shortcut identifiers, which are defined in **keymap.ctl**. For example, to specify the CTRL+I key combination as a shortcut, you use KEY(^i) because it adheres to one of the forms for *shortcut* definitions for KEY (listed in the [.ENTRY Discussion on page 2-28](#)). But what if you wanted to use KEY(inv) instead because the shortcut was for a menu entry that opened an inventory listing? You could do this by adding an identifier for the shortcut to **keymap.ctl**. In this case, you would add ^i=inv.

Note that if an identifier is defined for a shortcut in **keymap.ctl**, you can’t use that shortcut, rather than the identifier, in a KEY(*shortcut*) definition. For example, if you add ^i=inv to **keymap.ctl**, you can’t use KEY(^i) in a script; you must use KEY(inv) instead to assign the CTRL+I key combination as a menu entry shortcut.



If you change **keymap.ctl** and leave it in the synergy\toolkit directory, be sure to make a copy of it with a different name or in a different location. UI Toolkit distributions install a **keymap.ctl** file to this directory, so installing a new version of Toolkit will overwrite any **keymap.ctl** file in this directory.

1. Open the **keymap.*** file you want to modify in a text editor. You can modify **keymap.ctl** directly, or you can modify **keymap.msw** (for Windows) or **keymap.vt** (for UNIX and OpenVMS) and then save the file as **keymap.ctl** or another name of your choosing (in [step 3](#)).
2. Add, change, or delete entries as needed. If you add or change entries, use the following format:
shortcut=identifier

where *shortcut* is the internal function code and *identifier* is the identifier to use in KEY(*shortcut*) definitions for .ENTRY. *Identifier* must be unique and have no more than five characters.

As distributed, **keymap.*** files also contain definitions with the following formats:

Deprecated format	Example
<i>shortcut=(identifier)</i>	f1=(PF1)
<i>shortcut=identifier(text)</i>	f31=alt_q("Alt-Q")

where *text* is ignored by Toolkit. (In early versions of Toolkit, *text* was used as shortcut text, which is the text that appears next to the menu entry. But now shortcut text can be defined only in key maps. For the first deprecated format above, *identifier* doubled as the shortcut text and the shortcut identifier.)

The following are true of all three formats:

- ▶ If you place multiple specifications on a single line, you must separate them with commas.
 - ▶ Blank lines and lines that begin with a semicolon are ignored.
 - ▶ If an *identifier* specification contains a blank or non-alphanumeric character, you must enclose *identifier* in quotation marks.
3. Save the file as **keymap.ctl** or another name of your choosing. If you save it as **keymap.ctl** (the default name) in the current working directory or the WND directory (the default location), Toolkit will automatically be able to find it. If you save it to another location or give it another name, you'll need to set the DTKKEYCTLFIL environment variable to its name and location.

Redefining identifiers for shortcuts specified with MB_ENTRY

You can customize the way shortcuts for menu entries are defined in MB_ENTRY calls by creating and including a **.def** file that redefines and/or adds to the shortcut identifiers defined in **inpctl.def**. For example, if you use only the distributed version of **inpctl.def** and want to specify the CTRL+I key combination as a shortcut, you must pass CTRL_I_KEY as the *shortcut* argument for MB_ENTRY, because that's the identifier defined in **inpctl.def** for this key combination. But what if you are assigning this key combination to a menu entry that opens an inventory list, for example, and for more readable code you want to instead pass "INVENTORY_KEY" as the shortcut argument? You can do this by creating a **.def** file that includes the following definition and which could, in turn, include **inpctl.def**.

```
.define INVENTORY_KEY, CTRL_I_KEY
```

Note that on UNIX and OpenVMS, identifiers defined in **inpctl.def** must be set to function values defined in a key map file (**dtkmap.ism** by default). On Windows, use only the function values used in the distributed version of **inpctl.def**. These correspond to function values defined in the accelerator table for Toolkit on Windows.

Using **keys.dbl** to test shortcuts (UNIX and OpenVMS)

To verify that your key mapping is working correctly, use the **keys.dbl** file. **Keys.dbl** enables you to test your key mapping. To create the test program,

1. Compile **keys.dbl**, and link the resulting file with **tklib.elb** (**TKLIB_SH.EXE** on OpenVMS).
2. If you made changes to **keymap.ctl**, make the same shortcut changes **vtkeys.wsc**.
3. Run Script to create a file called **keys.ism** from **vtkeys.wsc**.
4. Run the **keys** program by entering the following at the command prompt:

```
dbr keys
```

Each menu entry in this program has a different shortcut key, and all 98 shortcuts are represented. You can use this program to test each shortcut.

Example

If your application runs on VT-style terminals, your script file could assign the HELP key to the menu entry **O_HELP** as follows:

```
.entry o_help, Help, key(help)
```

Keymap.ctl defines the following function code for the HELP key:

```
f15=Help
```

When you process the script file with Proto, Script, or Composer, the menu column is stored in a window library. The menu entry **O_HELP** is associated with the function code **f15**.

Because your application runs on VT-style terminals, you've set the **DTKTERM** environment variable to **VT100**. When your application calls the **U_START** subroutine, Toolkit opens the key map file and loads the key map for VT-style keyboards.

When you press the Help key during UI Toolkit input, the escape sequence **^[28~** is generated. Toolkit searches the VT100 key map for this escape sequence. The key map associates this sequence with **f15**. Toolkit then looks through its list of placed menu entries to see if any of them use **f15** (Help) as a shortcut. The shortcut for **O_HELP** matches, and **O_HELP** is returned in **g_entnam**. [Figure 18-1 on page 18-22](#) illustrates how the Help key would be mapped.

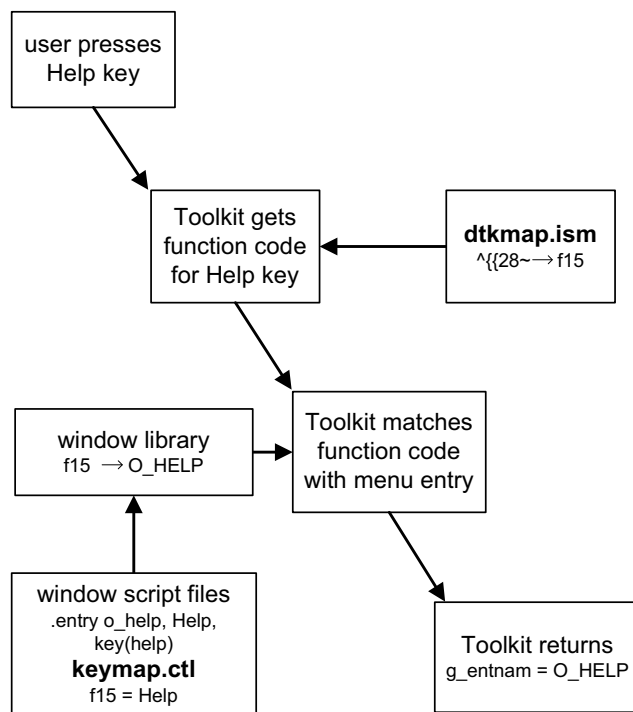


Figure 18-1. Key map processing for the HELP key.

Customizing reserved keys

This section explains how to assign reserved keys to other functions and how to assign new keys for reserved functions.

UI Toolkit's input processing uses four reserved functions. Each of these functions is tied to a specific key, as shown in the table below.

Reserved Keys	
Function	Key
Process Menu	CTRL+P or ALT (Windows)
Screen Redraw	CTRL+R
Delete Character	BACKSPACE (CTRL+H) or DELETE
End of File	CTRL+D (UNIX) or CTRL+Z (OpenVMS)

Assigning reserved keys to other functions

You can use a reserved key for some other function either by assigning it directly as a shortcut or by entering it as the sequence for the function code in the key map. However, you will need to assign a new key to the reserved function.

Assigning new keys for reserved functions

If you want to use a different key for a reserved function, you need to write your own version of the EFKEY_METHOD subroutine. (See [EFKEY_METHOD on page 7-72](#) for syntax information.) This subroutine enables you to remap any function codes, and you can name the subroutine anything you want. After you register it with the E_METHOD subroutine, UI Toolkit will call your subroutine whenever the user presses a function key or control character. (See [“Overloading UI Toolkit Routines” on page 18-8](#) for more information on overloading a user-overloadable subroutine.)

The distributed include file **inpctl.def** defines the function codes for key mapping. EFKEY_METHOD is passed the function code that resulted from any interpretation of the key map. You can modify this function code and change it to any other code defined in **inpctl.def**. At the end of the list of defined codes, you may notice definitions for DELETE_KEY, PROCESS_MENU, SCREEN_REDRAW, and EOF_KEY. To assign these functions to a different key, you need to change the usual code generated by the desired keystroke to one of these codes in your EFKEY_METHOD subroutine.

For example, to make the F10 key process the menu and CTRL+L redraw the screen, you'd write EFKEY_METHOD as follows:

```
subroutine efkey_method
    a_sts      ,n      ;Function code (incoming and returned)
    .include "WND:inpctl.def"
    proc
        case a_sts of
            begincase
F10_KEY:      a_sts = PROCESS_MENU    ;Make F10 process the menu
CTRL_L_KEY:   a_sts = SCREEN_REDRAW  ;Make ^L redraw the screen
            endcase
        xreturn
    endsubroutine
```

Customizing the Look of Your Application

To enable you to control the look of your application, UI Toolkit uses the following:

- ▶ The *color palette*, a set of color settings used to specify colors for renditions. On Windows, these settings correspond to *Synergy colors*, which are RGB triplets assigned to color numbers. On UNIX and OpenVMS, they correspond to color codes (for WNDC) or system-specific color numbers (for other routines).
- ▶ *Renditions*, combinations of colors and attributes—boldface, blinking (which is italic on Windows), reverse video, and underscore—for UI elements. You can set Toolkit renditions for types of UI elements, and in many cases, you can set renditions for a specific UI element.
- ▶ *Rendition schemes*, sets of Toolkit renditions that you can modify, create, save, and restore. You can create, change, and delete rendition schemes, and you can save them to and load them from a *rendition file*.

Synergy colors and the color palette

The color palette consists of 16 pairs of color settings called *color palette entries*, which are used to specify colors for renditions. Color palette entries are numbered 1 through 16, include a foreground setting (used for text) and a background setting. On UNIX and OpenVMS, the foreground and background settings correspond to color codes (for WNDC) or system-specific color numbers (for other routines). On Windows, however, these settings correspond to Synergy colors, which are RGB triplets assigned to color numbers.

There are three types of Synergy colors:

- ▶ *User colors* (color 0 through color 255), which are colors you can change by using `U_EDITREND`, `U_MODREND`, `U_REND`, the `COLORn` setting, or the Renditions option for Proto or Synergy UI Toolkit Control Panel (which, in both cases, call `U_EDITREND`). See [“Redefining user colors \(Windows\)”](#) below.
- ▶ *System colors* (color 256 through color 285), which correspond to Windows system colors and can be changed only through Windows mechanisms (Windows Control Panel, for example).
- ▶ *Reserved colors* (color 286 through color 511), which are reserved for future Windows system colors. These are set to black (0,0,0) and cannot be changed.

On Windows, the Synergy runtime loads default Synergy colors and a default color palette when it starts. On UNIX and OpenVMS, there is no default color palette; you must define the initial color palette with the WNDC environment variable.

Note that the color palette and Synergy color definitions, which are also used by the Synergy Windowing Subroutines, are more fully described in [“Colors and the color palette”](#) in the “Synergy Windowing API” chapter of the *Synergy DBL Language Reference Manual*. The sections below list methods you can use with Toolkit applications to change or define the color palette and to change user colors.

Defining and modifying the color palette

You can override one or more of the Synergy color palette entries by doing one of the following:

- ▶ (Windows only) Use the PALETTE setting in **synergy.ini** or **synuser.ini**. See [PALETTE](#) in the “Environment Variables” chapter of *Environment Variables & System Options*.
- ▶ Use U_EDITREND or the Renditions option for Proto or Synergy UI Toolkit Control Panel (both use U_EDITREND). See [“Using the Palette Entry window \(Windows\)”](#) on page 18-37.
- ▶ Use the WP_PALET function for W_PROC. See [W_PROC](#) in the “Synergy Windowing API” chapter of the *Synergy DBL Language Reference Manual*.
- ▶ (UNIX and OpenVMS only) Use the WNDC environment variable to define color palette entries. See [WNDC](#) in the “Environment Variables” chapter of *Environment Variables & System Options* for information.

Redefining user colors (Windows)

You can redefine Synergy color 0 through Synergy color 255, which are user colors, by doing one of the following:

- ▶ Use the D_SETCOLOR operation for the %U_WINCOLOR function. See [%U_WINCOLOR](#) on page 16-152.
- ▶ Use the COLOR n setting in the **synergy.ini** or **synuser.ini** file (see [COLOR \$n\$](#) in the “Environment Variables” chapter of *Environment Variables & System Options*). And use the %U_SAVELOG function to save a COLOR n setting (see [%U_SAVELOG](#) on page 16-128).
- ▶ Use U_EDITREND or the Renditions option for Proto or Synergy UI Toolkit Control Panel (both use U_EDITREND). See [“Using the Palette Entry window \(Windows\)”](#) on page 18-37.

Renditions

Renditions are combinations of colors and attributes (boldface, blinking, reverse video, and underscore) for UI elements, which make up the visible part of an application, such as backgrounds, scroll bars, borders, titles, and so forth. You can set renditions for a specific UI element in many cases (e.g., for a particular field on a particular window), and you can set renditions for some types of elements. For example, Input Field Display is a set of color and attribute settings that apply to input fields. For a list of renditions, see the [“Renditions”](#) table below.

You can set renditions with the following. Note that these are listed in order of precedence: settings in input methods override field-level settings, and so forth. For more information on methods for setting renditions, see [“Renditions for specific UI elements”](#) on page 18-31 and [“Rendition schemes and Toolkit renditions for types of UI elements”](#) on page 18-31.

1. Input methods (%IDISPLAY_METHOD or %IEDITFMT_METHOD) or published properties for ActiveX Toolkit lists.
2. Field-level settings. These are set with .FIELD, IB_FIELD, I_FLDMOD, or U_HELPREND.

3. `U_REND`, `U_START`, `U_MODREND`, or `U_EDITREND`. These set renditions for types of UI elements.
4. Other script settings (`.FILL`, `.TEXT`, `.BORDER`, and so forth).

You can also use the Synergy windowing routines, such as `W_DISP`, to set colors and attributes, and these changes generally have precedence for screen updates that occur immediately after they are set. But note that settings made with windowing routines can later be overridden by calls to Toolkit routines.

Note the following:

- ▶ On Windows, most renditions are controlled by Windows and cannot be set by Toolkit applications. (For example, on Windows 7 and earlier, the background and text color for buttons are controlled by the Windows Control Panel's "3D Objects" setting.)
- ▶ When a Toolkit application starts, the `U_START` subroutine sets default Toolkit renditions for types of UI elements as listed in the table below. Renditions set by `U_START` override most renditions specified in window scripts except for those marked as "No" in the "Override script?" column. (Note that the `U_INITREND` subroutine enables you to reset Toolkit renditions to these initial settings. See [U_INITREND on page 16-89](#).)
- ▶ You can instruct Toolkit to override field-level rendition settings for the active input field with the Input Field Processing renditions during `I_INPUT` and `I_INPFLD` processing. See [E_STATE on page 7-41](#) for information.

Renditions					
Toolkit rendition <code>U_REND</code> argument	<code>U_START</code> initialization			Affected platforms	Corresponding Control Panel items for Windows 7 and earlier
	Override script?	Attributes	Palette		
<code>D_RND_DLG_BDR</code> Dialog Box Border	Yes	n/a	1	UNIX, OpenVMS	Active Window Border, Inactive Window Border
<code>D_RND_DLG</code> Dialog Box Body	Yes	n/a	1	All	n/a
<code>D_RND_FTR</code> Screen Footer	Yes	Bold	5	UNIX, OpenVMS	3D Objects
<code>D_RND_HDR</code> Screen Header ^a	Yes	Bold	5	All	Active Title Bar, Inactive Title Bar
<code>D_RND_HLP_BCK</code> Help Window Body	No	n/a	n/a	All	n/a
<code>D_RND_HLP_BRDR</code> Help Window Border	No	n/a	n/a	UNIX, OpenVMS	Active Window Border, Inactive Window Border

Renditions (Continued)					
Toolkit rendition U_REND argument	U_START initialization			Affected platforms	Corresponding Control Panel items for Windows 7 and earlier
	Override script?	Attributes	Palette		
D_RND_HLP_TTL Help Window Title	No	n/a	n/a	UNIX, OpenVMS	Active Title Bar, Inactive Title Bar
D_RND_INF Information Line	Yes	Reverse	6	UNIX, OpenVMS	3D Objects
D_RND_INP_BCK Input Window Background ^b (includes prompts and other non-field text)	No	n/a	n/a	All	n/a
D_RND_INP_BDR Input Window Border	No	n/a	n/a	UNIX, OpenVMS	Active Window Border, Inactive Window Border
D_RND_INP_CLR Cleared Input Field (for non-active cleared input fields)	Yes	n/a	3	All	Window
D_RND_INP_DSA Disabled Input Field ^c	Yes	n/a	3	UNIX, OpenVMS	3D Objects (with Windows themes), Window (without Windows themes)
D_RND_INP_DSP Input Field Display (for non-active read/ write input fields)	Yes	Bold	3	All	Window
D_RND_INP_PRC Input Field Processing ^d (for active input read/ write input field)	Yes	Reverse	3	UNIX, OpenVMS	Window, Selected Items
D_RND_INP_ROD Read-Only Input Field Display ^c (for non-active read-only input fields)	Yes	Normal	3	UNIX, OpenVMS	3D Objects

Customizing UI Toolkit

Customizing the Look of Your Application

Renditions (Continued)					
Toolkit rendition U_REND argument	U_START initialization			Affected platforms	Corresponding Control Panel items for Windows 7 and earlier
	Override script?	Attributes	Palette		
D_RND_INP_ROP Read-Only Input Field Processing ^c (for active input field if it is read-only)	Yes	Reverse	3	UNIX, OpenVMS	3D Objects
D_RND_INP_TTL Input Window Title	No	n/a	n/a	UNIX, OpenVMS	Active Title Bar, Inactive Title Bar
D_RND_LST_DSA_S Disabled Selected Item	Yes	Reverse	7	UNIX, OpenVMS	Selected Items
D_RND_LST_ENA_B Enabled List Border	Yes	Bold	7	UNIX, OpenVMS	Active Window Border, Inactive Window Border
D_RND_LST_ENA_T Enabled List Title	Yes	Bold	7	UNIX, OpenVMS	Active Title Bar, Inactive Title Bar
D_RND_LST_ENA Enabled List	Yes	Bold	7	UNIX, OpenVMS	Window
D_RND_LST_HDR Enabled List Header ^e	Yes	Reverse	7	All	n/a
D_RND_MNU_DSA Disabled Menu Entry	Yes	Reverse	2	UNIX, OpenVMS	Menu
D_RND_MNU_ENA Enabled Menu Entry	Yes	Bold, reverse	2	UNIX, OpenVMS	Menu
D_RND_MNU_EQS Menu Entry Quick-Select	Yes	Bold, reverse, underline	10	UNIX, OpenVMS	n/a
D_RND_MNU_HQS Menu Header Quick-Select	Yes	Bold, reverse, underline	10	UNIX, OpenVMS	n/a
D_RND_MNU_SEL Selected Menu Entry	Yes	Bold	2	UNIX, OpenVMS	Window, Selected Items

Renditions (Continued)					
Toolkit rendition U_REND argument	U_START initialization			Affected platforms	Corresponding Control Panel items for Windows 7 and earlier
	Override script?	Attributes	Palette		
D_RND_MSG_BDR Message Window Border	Yes	Reverse	4	UNIX, OpenVMS	Message Box
D_RND_MSG Message Window Body	Yes	Reverse	4	UNIX, OpenVMS	Message Box
D_RND_SCR_BAR Scroll Bar	Yes	Reverse	8	UNIX, OpenVMS	Scrollbar
D_RND_SEL_BCK Selection Window Body	Yes	Remove reverse	n/a	UNIX, OpenVMS	Window
D_RND_SEL_BDR Selection Window Border	No	n/a	n/a	UNIX, OpenVMS	Active Window Border, Inactive Window Border
D_RND_SEL_EQS Sel Wnd Quick-Selects	Yes	Underline	10	UNIX, OpenVMS	n/a
D_RND_SEL_SEL Select Wnd Selected Item	Yes	Add reverse	n/a	UNIX, OpenVMS	Window, Selected Items
D_RND_SEL_TTL Selection Window Title	No	n/a	n/a	UNIX, OpenVMS	Active Title Bar, Inactive Title Bar
D_RND_WIN_BCK Generic Window Background	No	n/a	n/a	All	n/a
D_RND_WIN_BDR General Window Border	No	n/a	n/a	UNIX, OpenVMS	Active Window Border, Inactive Window Border
D_RND_WIN_TTL General Window Title	No	n/a	n/a	UNIX, OpenVMS	Active Title Bar, Inactive Title Bar
D-RND_LST_DSA Disabled List	Yes	n/a	7	UNIX, OpenVMS	Window

Customizing UI Toolkit

Customizing the Look of Your Application

Renditions (Continued)					
Toolkit rendition U_REND argument	U_START initialization			Affected platforms	Corresponding Control Panel items for Windows 7 and earlier
	Override script?	Attributes	Palette		
D-RND_LST_FTR Enabled List Footer ^e	Yes	Reverse	7	All	n/a
D-RND_LST_SEL Selected Item in List	Yes	Bold, reverse	7	UNIX, OpenVMS	Window, Selected Items
D-RND_SCR_IND Scroll Bar Indicators	Yes	Bold, reverse	8	UNIX, OpenVMS	3D Objects
User Rendition 1 through User Rendition 5	Yes	None	1	All	n/a

- On Windows, these renditions apply only if the header has multiple lines.
- For check boxes, these apply to the prompt only if Input Field Processing is not applied (see [E_STATE on page 7-41](#)) and no field-level renditions are set for the check boxes.
- On UNIX and OpenVMS, these can be changed only by U_REND (not by U_EDITREND or U_MODREND). On Windows, these renditions cannot be changed by Toolkit.
- On Windows, only U_REND can set these (they cannot be set by U_EDITREND or U_MODREND), and they are not used for lists (ActiveX Toolkit lists or standard Toolkit lists). Additionally, these renditions apply only if the E_STATE code D_INPPRCRND is set to on, and they override field-level settings only if the E_STATE code D_INPRNDOVER is set to on. See [E_STATE on page 7-41](#).
- These do not apply to ActiveX Toolkit lists.

Renditions for specific UI elements

There are some renditions you can set for specific UI elements (rather than types of UI elements):

- ▶ To apply help window renditions to a window that isn't a help window, use `U_HELPREND`. (The help window renditions are Help Window Body, Help Window Title, and Help Window Border.)
- ▶ To set renditions for a specific input field, use the rendition qualifiers or arguments for `.FIELD`, `IB_FIELD`, or `I_FLDMOD`, or set them in a display method or an edit format method (see [%IDISPLAY_METHOD](#) on page 8-122 and [%IEDITFMT_METHOD](#) on page 8-127).
- ▶ To set renditions for an ActiveX Toolkit list, use the published properties for ActiveX lists. See [“Published Properties for ActiveX Toolkit Lists”](#) on page 9-17.
- ▶ To set renditions for other UI elements (i.e., UI elements that are not fields), use `.BORDER`, `.BOX`, `.FILL`, `.LINE`, `.TEXT`, `.TITLE`, `.WFIELD`, `T_SETUP`, and `%TS_TABSET`. Additionally, you can use `%U_SELPALETTE` to enable users to select a color palette for a specific UI element (among other things), and you can also change color and attribute settings with the Synergy windowing routines, such as `W_DISP`. (See the [“Synergy Windowing API”](#) chapter of the *Synergy DBL Language Reference Manual* for information.)

Rendition schemes and Toolkit renditions for types of UI elements

In addition to the renditions for specific UI elements, you can set renditions for types of UI elements. For example, the Read-Only Input Field Display renditions control the look of read-only input fields (though there may be overrides for individual fields, and the Read-Only Input Field Processing renditions may be used for an active read-only input field).

On UNIX and OpenVMS, there are 47 Toolkit renditions for types of UI elements. On Windows, there are only 14. In both cases, five of these are user-assigned Toolkit renditions. (See the [“Renditions”](#) table on page 18-26.) There are more on UNIX and OpenVMS because on these platforms Toolkit controls the look of all UI elements. On Windows, however, the Windows Control Panel and Windows settings and defaults control the look of most UI elements, leaving only nine UI elements for Toolkit renditions (plus the five user-assigned Toolkit renditions).

A full set of renditions is a *rendition scheme*. You can create, change, and delete rendition schemes, and you can save them to and load them from rendition files as described in the following table.

For Windows, note the following:

- ▶ You can bring color for Toolkit renditions under the control of Windows settings by assigning Synergy *system colors* to renditions. For information on system colors and the default palette entries, see [“Colors and the color palette on Windows”](#) in the [“Synergy Windowing API”](#) chapter of the *Synergy DBL Language Reference Manual*.
- ▶ We recommend that you set the background colors for the Input Field Display renditions and the Input Window Background renditions to the same value. The default palette entries for these renditions are 3 and 1 respectively. (Incidentally, the default background settings for palette entries 3 and 1 on Windows are system colors.)

Working with Rendition Schemes		
To...	Do the following...	For more information, see...
Create a rendition scheme	Start by modifying an existing rendition scheme. Then use U_SAVREND to save it to a new rendition file by passing the name of a non-existent file as the <i>filename</i> argument.	U_SAVREND on page 16-132
Load a rendition scheme from your Toolkit application	Use U_LDREND (if you want to specify a channel on which the file is open) or U_RSTREND (if you want to specify the filename or a rendition file rather than a channel).	U_LDREND on page 16-90 U_RSTREND on page 16-125
Modify a rendition scheme from your application	Use U_REND.	U_REND on page 16-116
Save the current rendition scheme or a specified rendition scheme from your application	Use U_STOREND (if you want to specify a channel on which the rendition file is open) or U_SAVREND (if you want to specify the filename of the rendition file rather than a channel).	U_STOREND on page 16-139 U_SAVREND on page 16-132
Enable your users to modify, load, save, or delete rendition schemes	Call U_MODREND or U_EDITREND from your application or give your users access to Synergy UI Toolkit Control Panel or Proto, both of which call U_EDITREND when you select “Renditions” from their menu. (See “U_MODREND versus U_EDITREND” below.)	“Using the Renditions window (Windows)” on page 18-36 “Using the U_EDITREND screens (UNIX and OpenVMS)” on page 18-38
Manually modify, load, save, or delete rendition schemes	Open Synergy UI Toolkit Control Panel or Proto and select “Renditions” from the menu. This opens the Renditions screen, which enables you to do these things as well as modify color palette entries.	“Using the Renditions window (Windows)” on page 18-36 “Using the Palette Entry window (Windows)” on page 18-37 “Using the U_EDITREND screens (UNIX and OpenVMS)” on page 18-38

U_MODREND versus U_EDITREND

Both U_MODREND and U_EDITREND open the Renditions window, which enables users to load, change, save, and delete rendition schemes (as well as color palette entries). The difference is that while U_EDITREND automatically saves rendition changes to the loaded rendition (and can save them to a specified rendition scheme), U_MODREND just modifies the rendition scheme loaded in memory. U_MODREND doesn't save the rendition scheme unless the user clicks the Save button and supplies a rendition scheme name.

Files and environment variables used for rendition schemes

The following files and environment variables are used with rendition schemes:

dtkrnd.ism	A file that contains records defining various rendition schemes. The key for this file is the rendition scheme name. The maximum size of a rendition scheme name is 30 characters, and the scheme name is not case sensitive. A blank key is reserved for the working rendition scheme.
DTKRNDFIL	An environment variable that is used to define an alternate location <i>and</i> name for the rendition file.
DTKRND	An environment variable that is used to define an alternate location for dtkrnd.ism .

If you want to use a rendition file other than the default, **dtkrnd.ism**, set the DTKRNDFIL environment variable to the desired file. If you want to use the rendition file **dtkrnd.ism** from a directory other than the current one, set the DTKRND environment variable to the desired directory.

UI Toolkit looks for the rendition file as follows:

- ▶ If the DTKRNDFIL is defined, it determines the rendition file location and name.
- ▶ If DTKRNDFIL is not defined, UI Toolkit attempts to open the file as **DTKRND:dtkrnd.ism**.
- ▶ If Toolkit can't open **DTKRND:dtkrnd.ism**, it attempts to open the file **dtkrnd.ism** in the current directory.

Customizing user-assigned Toolkit renditions

U_EDITREND, U_MODREND, and U_REND enable you to customize the five user-assigned Toolkit renditions, which are renditions that you and your users can assign to a type of element. (See [“Why use user-assigned Toolkit renditions?”](#) below.) If you call U_EDITREND or U_MODREND, you can specify the number of user-assigned renditions to be modified and, on UNIX and OpenVMS, their descriptions in the selection window and example windows in the Sample Windows screen section.

UNIX, VMS

Note that by default

- ▶ user-assigned renditions are not displayed for editing by either `U_EDITREND` or `U_MODREND`. By passing the optional parameter for the number of user-assigned renditions, you can activate from 1 to 5 of these renditions.
 - ▶ any active user-assigned rendition receives the description “User Rendition *n*” in the Select Screen Area selection window, where *n* is the user-assigned rendition number. By passing the optional array of user-assigned rendition descriptions, you can replace the default descriptions.
 - ▶ any active user-assigned rendition is presented in the Sample Windows screen area as a single-line, unbordered window that contains only the description of the rendition. You can optionally create your own sample windows for user-assigned renditions and pass their IDs in the optional user renditions window IDs parameter. If you do this, you must also specify which areas of the window to affect when the user modifies the rendition. You can specify the window body, border, title, or any combination thereof, and you can specify a single field number to affect, or you can specify all fields. See [U_EDITREND on page 16-42](#) and [U_MODREND on page 16-99](#) for more information.
-

Why use user-assigned Toolkit renditions?

User-assigned Toolkit renditions are renditions that you can assign to types of UI elements. For example, if you’ve written routines to create and maintain a checklist window of your own invention (like the one in [figure 18-2](#)), you can use user-assigned renditions to assign unique combinations of attributes and colors to the window background and text, to the border and title, and to the check mark character.

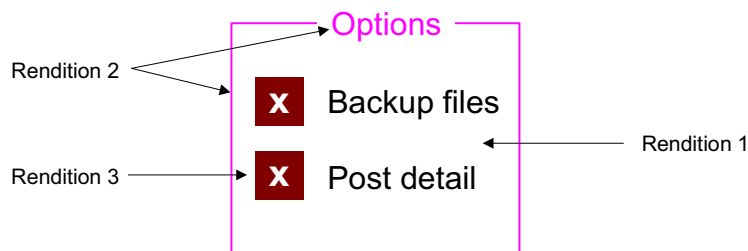


Figure 18-2. Window with user-assigned renditions.

For example, you could assign user rendition 1 as the rendition for the checklist body, user rendition 2 as the rendition for the border and title, and user rendition 3 as the rendition for the check mark character. The routines in your application that create and maintain these windows can call `U_REND` with the `D_COPY` argument to get the current settings for these renditions and then apply them to the windows.

Additionally, if you want to give your end users access to these renditions, you can tie a utility menu entry to a call to `U_EDITREND` within your application and tell `U_EDITREND` that three user renditions will be used. (A utility menu entry is one that begins with `U_` and is handled by an `EUTILS_METHOD` routine.)

UNIX, VMS

If your application runs on UNIX or OpenVMS, you can also pass a three-element array of the descriptions for these renditions—for example, “Checklist Body,” “Checklist Border,” and “Checklist Check Mark.” These descriptions will then appear in the Select Screen Area window when `U_EDITREND` or `U_MODREND` is called, enabling users to modify them as they would any other Toolkit rendition.

You can also customize the prototype for the window that will appear in the Sample Windows window when you call `U_EDITREND` or `U_MODREND`. To continue with the checklist window example, you could create a window that looks just like the checklist window that your routines create and modify. (Note that it doesn’t need to be a functioning model of the window, and note that the sample check mark characters should be window fields created with the `W_FLDS` subroutine.) You would place this window at the position in the screen where you want it to appear when the user selects one of these renditions from the Select Screen Area window. Then, when you call `U_EDITREND` or `U_MODREND`, pass the ID of this window in all three elements of the sample window ID’s array.

- ▶ In the *areas* array, you would pass `D_RND_AREA` for the first element (to affect the window’s processing area), `D_RND_BRDR+D_RND_TITLE` for the second (to affect the border and title), and 0 for the third (because this will affect fields instead of an area).
- ▶ In the *field_numbers* array, you would pass 0 for the first two elements and `WFN_ALL` for the third element, so that all of your check marks will be affected by a change to this rendition. You have now integrated the renditions presentation for your new window class with the other classes of windows in UI Toolkit.

See [U_EDITREND on page 16-42](#) for additional information.

Using the Renditions window (Windows)

When you call `U_EDITREND`, or when you select Renditions from the General menu of Proto or the Preferences menu of the Synergy UI Toolkit Control Panel, UI Toolkit opens the Renditions window (figure 18-3), which enables you to manually restore, modify, delete, and save rendition schemes. The Sample windows area shows how the current renditions will look.

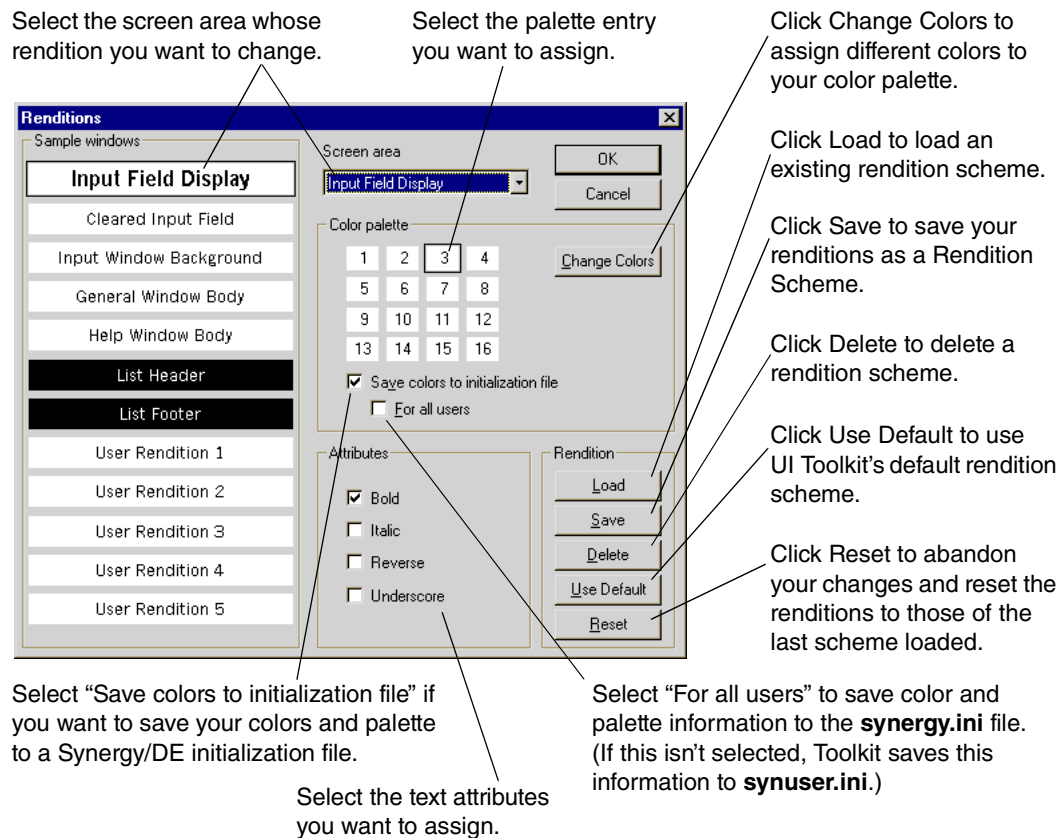


Figure 18-3. Modifying a rendition.

To load a different rendition scheme, save current settings, delete the loaded rendition scheme, or use the default renditions, click Load, Save, Delete, or Use Default. To modify one or more renditions, do the following:

1. To modify a rendition scheme other than the one that's loaded, click Load to load a rendition scheme by name, or click Use Default if you want to switch to the default rendition scheme.
2. Select a rendition by clicking one of the named squares under "Sample windows" or by selecting a rendition name from the "Screen area" list.

3. To assign a different color palette entry to the Toolkit rendition, click one of the numbered squares under “Color palette.” For more information on modifying the color palette, see [“Using the Palette Entry window \(Windows\)”](#) below.
4. Select any attributes you want to assign to the rendition.
5. Repeat [step 2](#) through [step 4](#) for each Toolkit rendition whose screen area you want to modify.
6. To save the renditions as a rendition scheme that you can access later by name, click Save and enter a name for the rendition scheme.

To reset the renditions to the last loaded rendition scheme, click Reset.

7. When you’re finished, click OK to exit the Renditions dialog box. The rendition scheme that is loaded when you click OK will be the new default.

Using the Palette Entry window (Windows)

You can use the Palette Entry Window to change the foreground (text) and background colors of any palette entry. (For other methods, see [“Defining and modifying the color palette”](#) on [page 18-25](#).)

1. In the Renditions dialog box, select the number for the palette entry you want to modify, and click the Change Colors button.

The Palette Entry window is displayed for the palette entry. (See [figure 18-4](#).) This dialog box displays colors you can apply to the foreground or background for the palette entry. System colors correspond to Windows system colors and can be changed only from Windows Control Panel. User colors are Synergy colors that you can change by clicking the Define color button or by using the `COLORn` setting in **synergy.ini** or **synuser.ini**. (See [COLORn](#) in the “Environment Variables” chapter of *Environment Variables & System Options* for more information.)

If you make a change in this dialog box, Toolkit updates the corresponding entry in the [colors] section of **synergy.ini** or **synuser.ini** if “Save colors to initialization file” is checked in the Renditions window. If there is no existing [colors] section, Toolkit will create one.

2. Click on a color in the “System colors” or “User colors” area.
3. Click “Apply background” if you want to assign the color as the background for the palette entry, or click “Apply foreground” to assign the color to the foreground (used for text). Then click OK.

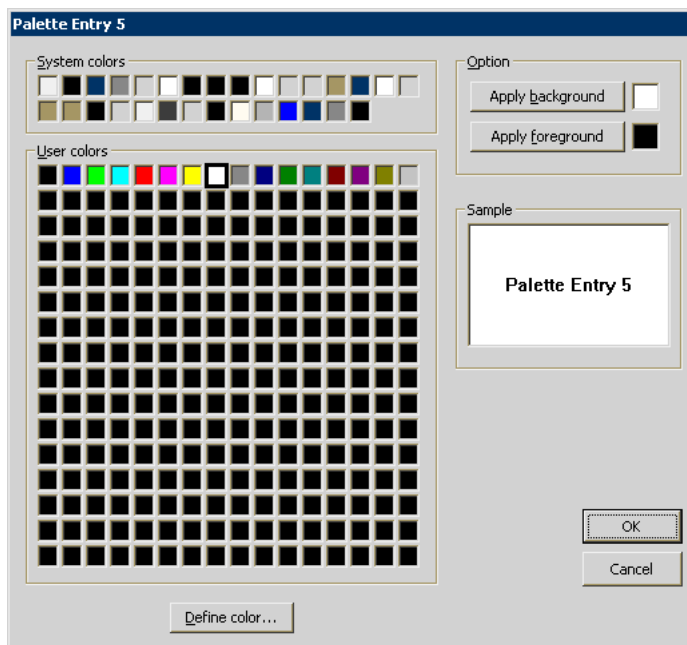


Figure 18-4. Assigning a color to a palette entry.

Using the U_EDITREND screens (UNIX and OpenVMS)

On UNIX and OpenVMS, if you call U_EDITREND or select General > Renditions in Proto or Preferences > Renditions in Synergy UI Toolkit Control Panel, Toolkit displays three windows:

- ▶ Select Screen Area
- ▶ Renditions
- ▶ Sample Windows

Modify a rendition scheme

1. In the Select Screen Area screen, use the arrow keys to highlight the rendition name for the screen area you want to change. Then press ENTER.
2. In the Renditions window, use the arrow keys to move to the rendition you want to apply to (or remove from) the rendition and press ENTER.
3. When you're finished selecting renditions for the current screen area, press the Exit shortcut.
4. Repeat [step 1](#) through [step 3](#) for each screen area you want to modify.
5. To save the renditions as a rendition scheme that you can access later by name, select Save rendition scheme from the Renditions menu and enter a name for your rendition scheme.

Loading an existing rendition scheme

1. From the Renditions menu, select Load rendition scheme.
2. In the Select Scheme window, use the arrow keys to highlight the scheme you want to load, and press ENTER.

Assigning different colors to a palette entry

If you are using a color monitor and you have enabled color on your system, you can change colors when your cursor is on one of the color codes in the Renditions window. (For information on enabling color on your system, see [WNDC](#) in the “Environment Variables” chapter of *Environment Variables & System Options*.)

Each color palette entry has a background color code and a foreground color code, each of which can be from 0 to 255. These codes and the colors that they specify are system specific. For example, some systems use basic ANSI color codes, a few of which are listed in the table below:

Color	ANSI code
Black	0
Blue	1
Green	2
Cyan	3
Red	4
Magenta	5
Yellow	6
White	7

To change the background or foreground color for a palette entry,

1. In the Renditions screen, use the arrow keys to move to the palette entry (under “Colors”) you want to modify.
2. From the Renditions menu, select Change colors. The Color Codes screen is displayed.
3. Enter the numeric codes for the colors you want applied to the background and foreground for the color palette entry.

The new background and foreground colors for the palette entry immediately go into effect, changing all screen areas that use the palette entry.

4. When you are done modifying a color palette, you can continue to change renditions for the current screen area or press the Exit shortcut to return to the Select Screen Area window.

Using Fonts on Windows

Because the fonts installed on each system may differ, we use the concept of a *font palette* to define the fonts that are actually used. A font palette consists of font palette entries, each of which associates a font name with a typeface, point size, and sizing character combination. (A sizing character determines cell width for an object using that font.) A font palette is created in memory when the Synergy runtime starts up, and it exists for the duration of that runtime (**dbr.exe**).

The runtime creates the initial font palette by reading font palette entries specified in the [fonts] section of the Synergy initialization file, **synergy.ini**, and then overrides or adds to the palette by reading **synuser.ini**. You can add new font palette entries and modify existing entries at runtime by calling the %U_WNDFONT function.

The runtime identifies font palette entries by palette entry name or font handle. When assigning fonts to objects at runtime, you can specify either. When assigning fonts to objects at script time, or through Composer or Repository, you must use palette entry names.

The font (typeface) specified in a font palette entry is either *fixed* (every character has the same width, such as Courier) or *proportional* (different characters have different widths, such as Arial). See “Using fixed fonts” on page 18-49 and “Using proportional fonts” on page 18-49 for more information.

Defining font palette entries

There are two ways to define the font palette entries used by the runtime to create the initial font palette: manually add a [fonts] section to a Synergy initialization file (**synergy.ini** or **synuser.ini**) or have Composer add a [fonts] section to **synuser.ini**. (Composer doesn’t add a [font] section or font entries to **synergy.ini**.) Additionally, you can define font palette entries at runtime by calling the %U_WNDFONT function.

Defining font palette entries in synergy.ini or synuser.ini

You can define font palette entries by manually adding them to the [fonts] section of a Synergy initialization file, **synergy.ini** or **synuser.ini**. (You can also add them at runtime by using the D_SAVEFONT subfunction for %U_WNDFONT. See [D_SAVEFONT on page 16-227](#) for information.)

Each font palette entry has the form

```
font_name=typeface;point_size[:sizing_char]
```

where *font_name* is the case-insensitive name of the font you’re defining, *typeface* is a case-sensitive string that indicates the font’s typeface (for example, Times New Roman), *point_size* is a numeric point size, and *sizing_char* is an optional single character used to determine the cell size for purposes of positioning. If *sizing_char* isn’t specified, it defaults to “A”.

Here are some example font palette entries:

```
Default=Fixedsys;9;A
Alternate=MS Sans Serif;10;W
My_input_font=Times New Roman;12
```

Four font names are reserved. If these names are not defined in **synergy.ini** or **synuser.ini**, they will be defined for you by the runtime:

- ▶ DEFAULT
- ▶ ALTERNATE
- ▶ DEBUGGER
- ▶ STATUS

For more information, see [“Setting the font for an application” on page 18-42](#).

Defining font palette entries in Composer (synuser.ini)

When a window or input field is selected in Composer’s Application window, you can add or modify font palette entries in the [fonts] section of **synuser.ini**. To do so,

1. Select a window or input field in the Application window.
2. If you selected an input field, double-click the +Font selected attribute and change the property to Yes. (You can also do this from the +Font selected attribute below the +Prompt attribute.)
3. Click the drilldown button for the Font attribute.
4. The Font Palette dialog box is displayed.
5. Click the Add button.
6. The Add Font dialog box is displayed.
7. Enter a new font name in the Font name field.

The Font dialog box is displayed. See [figure 18-5](#).

8. Enter a typeface, point size, and sizing character in the appropriate input fields.
9. Click the OK button to exit the Font dialog box.

You are returned to the Font Palette dialog box.

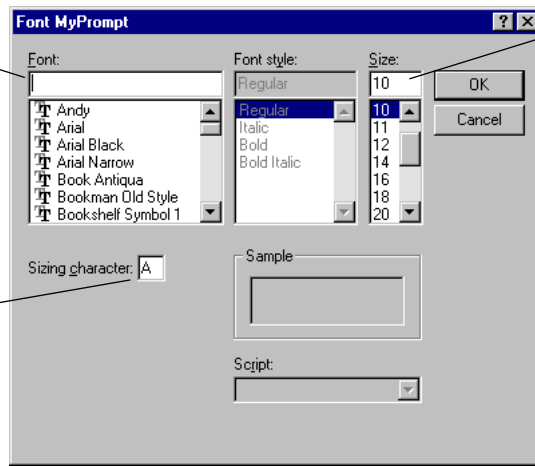
10. Click the OK button to exit the Font Palette dialog box.

You can also modify the typeface, point size, or sizing character for existing fonts by highlighting the font you want to change in the Font Palette dialog box and clicking the Modify button.

Composer updates the **synuser.ini** file to reflect your changes.

Select the typeface you want this font to have.

Enter the letter whose width will determine the cell size for any object using this font.



Select the point size for this font.

Figure 18-5. Defining a font.

Defining font palette entries at runtime

You can define or modify font palette entries at runtime using the `%U_WNDFONT(D_SETFONT)` function. `D_SETFONT` enables you to define the characteristics of a font. Use the following syntax:

```
%U_WNDFONT (D_SETFONT, font_id, face_name, point_size[, sizing_char])
```

The *font_id* argument specifies the font palette entry name or handle of the existing font you are defining. If successful, this call to `%U_WNDFONT` returns the font's handle. From this point on, you can refer to this palette entry by either palette entry name or font handle.

The `D_SETFONT` subfunction adds a font palette entry to memory or updates a font palette entry in memory. To save a font palette entry to **synergy.ini** or **synuser.ini**, use the `D_SAVEFONT` subfunction. See [D_SAVEFONT](#) on page 16-227 for information.

Setting the font for an application

The Synergy runtime supports two global fonts. An application window defined as less than 132 columns uses the "global font." If automatic font switching is enabled, an application window of 132 columns or more uses the "alternate font". If you choose the alternate font carefully, it may enable the window to fit on the screen.

To determine the initial global font when an application starts up,

- ▶ The runtime looks at the FONT_GLOBAL initialization setting, which specifies a font palette entry name. This allows the global font to be specified on a per application basis. See “[Setting application fonts in synergy.ini or synuser.ini](#)” below.
- ▶ If the above initialization setting is not found or if it references an undefined font palette entry name, the runtime looks at the font specified by the DEFAULT palette entry in the [fonts] section of **synergy.ini** and **synuser.ini**. If it’s defined in both of these files, the **synuser.ini** setting overrides the setting in **synergy.ini**.
- ▶ If the DEFAULT palette entry is not found, the runtime looks in the [fonts] section of **synergy.ini** and **synuser.ini** for the pre–version 6.3 initialization settings TYPE_FACE, FONT_HEIGHT, and FONT_WIDTH. If any of these are defined in both of these files, the **synuser.ini** settings override the settings in **synergy.ini**.
- ▶ If the TYPE_FACE, FONT_HEIGHT, and FONT_WIDTH settings are not found, the runtime uses Terminal font at the point size selected by the system as optimal (either the largest point size for which at least 80 columns of the font will fit on the screen or 10 point, whichever is smaller).

The logic described above is also used when determining the font palette entry name to use for the initial alternate font. The initialization settings involved are FONT_ALTERNATE, ALTERNATE, ALT_TYPE_FACE, ALT_FONT_HEIGHT, and ALT_FONT_WIDTH.



We recommend using the DEFAULT and ALTERNATE initialization settings in the [fonts] section of **synergy.ini** or **synuser.ini** rather than the pre–version 6.3 TYPE_FACE, FONT_HEIGHT, FONT_WIDTH, ALT_TYPE_FACE, ALT_FONT_HEIGHT, and ALT_FONT_WIDTH settings.

Setting application fonts in synergy.ini or synuser.ini

Use the FONT_GLOBAL and FONT_ALTERNATE initialization settings to specify the initial global and alternate fonts that will be used in your application. FONT_GLOBAL and FONT_ALTERNATE should be specified in the [synergy], [dbr], or [*myprog*] section of **synergy.ini** or **synuser.ini**, where *myprog* is any **.dbr** file.

FONT_GLOBAL and FONT_ALTERNATE have the syntax

FONT_GLOBAL=*palette_name*

and

FONT_ALTERNATE=*palette_name*

Use the DEFAULT and ALTERNATE initialization settings to define the characteristics of the palette entry names that will be used if FONT_GLOBAL and FONT_ALTERNATE are undefined. See “Defining font palette entries in synergy.ini or synuser.ini” on page 18-40 for the syntax.

Refer to the “Environment Variables” chapter of *Environment Variables & System Options* for more information about all of these settings.

Setting application fonts at runtime

To modify the global or alternate fonts used by your application after it has started up, use the %U_WNDFONT(D_SETFONT) function to modify the characteristics of the font IDs DF_DEFAULT and DF_ALTERNATE. Use the following syntax:

```
%U_WNDFONT (D_SETFONT, font_id, face_name, point_size[, sizing_char])
```

where *font_id* specifies which font ID to modify.

See chapter 16, “Utility Routines,” for more information on using %U_WNDFONT and its subfunctions, including D_CHOOSEFONT, which displays the Font dialog box and enables your users to define font characteristics graphically.

Setting fonts for the header, footer, and information line

The Synergy header, footer, and information line of the application window use their own fonts. The initial fonts are one of the following, in order of precedence.



For settings in **synergy.ini** and **synuser.ini**, definitions in the [*myprog*] section override definitions in [dbr], and definitions in [dbr] override definitions in [synergy]. In addition, a definition in **synuser.ini** overrides the same definition in the same section of **synergy.ini**.

- ▶ The font specified by the FONT_HEADER, FONT_FOOTER, and FONT_INFO initialization settings in the [synergy], [dbr], or [*myprog*] section of **synergy.ini** or **synuser.ini**, where *myprog* is any **.dbr** file.
- ▶ The font specified by the reserved palette entry name STATUS in the [fonts] section of **synergy.ini** or **synuser.ini**. If a palette entry named STATUS is not defined, one will be defined using Arial font, point size 10, and sizing character “A”.

Setting fonts for the Synergy debugger

The Synergy debugger window uses its own font. The initial “debugger font” is one of the following, in order of precedence.



For settings in **synergy.ini** and **synuser.ini**, definitions in the *[myprog]* section override definitions in *[dbr]*, and definitions in *[dbr]* override definitions in *[synergy]*. In addition, a definition in **synuser.ini** overrides the same definition in the same section of **synergy.ini**.

- ▶ The font specified by the FONT_DEBUG initialization setting in the *[synergy]*, *[dbr]*, or *[myprog]* section of **synergy.ini** or **synuser.ini**
- ▶ The font specified by the reserved palette entry name DEBUGGER in the *[fonts]* section of **synergy.ini** or **synuser.ini**
- ▶ The font specified by the reserved palette entry name DEFAULT in the *[fonts]* section of **synergy.ini** or **synuser.ini**
- ▶ The font specified by the TYPE_FACE, FONT_HEIGHT, and FONT_WIDTH initialization settings in the *[synergy]*, *[dbr]*, or *[myprog]* sections of **synergy.ini** or **synuser.ini**
- ▶ The terminal font

Setting the font for an object

By default, the font used for an application is the font used for all objects within that application. You can override the default font for an object by

- ▶ specifying the font for that class of object in **synergy.ini** or **synuser.ini**. (If this is defined in both files, the **synuser.ini** setting overrides the setting in **synergy.ini**.)
- ▶ specifying the font in Repository.
- ▶ specifying the font in your script file (either manually or using Composer).
- ▶ using UI Toolkit routines I_FLDMOD, IB_FIELD, or %U_WNDFONT.

In all of these cases, the font that you specify should already be defined in your font palette. See [“How fonts are applied” on page 18-48](#) for information about the defaulting hierarchy used if a referenced font is undefined.

If a font is specified for a window, it becomes the default for all objects in that window unless otherwise specified. The window is sized according to the metrics of that font, and objects are positioned according to rows and columns based upon the sizing character for that font. Likewise, if a font is specified for any other object or component, the object is sized according to the metrics for that font.



If a proportional font is specified for a window, objects in the window that do not support proportional fonts may not display as expected. See [“Using proportional fonts” on page 18-49](#) for a list of objects that support proportional fonts.

Setting fonts for a class of objects

You can define a font for a particular class of objects. This enables you to more easily migrate your application to proportional fonts.

Use one or more of the following initialization settings in the [synergy], [dbr], or [myprog] section of **synergy.ini** or **synuser.ini** to define the default font for the specified classes:

Setting	What it specifies
FONT_NUMFLD	Default font for numeric input fields (including dates and times).
FONT_ALPHAFLD	Default font for alphanumeric input fields (including user-defined).
FONT_PROMPT	Default font for input field prompts.
FONT_TEXT	Default font for text in windows.
FONT_LIST	Default font for lists.

Setting fonts for an object in Repository

Repository enables you to assign a font to a field or template. If you assign a font to a template, the font is used by all fields that use that template. You can also specify a font for a template or field prompt.

To specify fonts in Repository, enter a font palette entry name in the Font field and one in the Prompt font field on the Display tab of the Field Definition or Template Definition window. See [“Display information”](#) in the “Working with Fields” chapter of your *Repository User’s Guide* for more information.

You can also specify fonts using the Synergy Data Language. The FIELD, GROUP, and TEMPLATE statements all have FONT, NOFONT, PROMPTFONT, and NOPROMPTFONT keywords. Refer to the [“Synergy Data Language”](#) chapter of your *Repository User’s Guide* for more information about these keywords.

Setting fonts for an object in a script file

You can set fonts in your script file using the .FONT command and certain qualifiers of the .FIELD command. You can also specify fonts in Composer by setting the Font and Font selected properties for an object. Composer will then generate the script file for you.

To set a window font in a script file, use the .FONT command, which has the following syntax:

```
.FONT palette_name
```

To set a field or prompt font in a script file, use the FONT and PROMPTFONT qualifiers of the .FIELD command. These qualifiers have the syntax

[, FONT (*palette_name*)]

and

[, PROMPTFONT (*palette_name*)]

To override a font specified for a repository field, use the NOFONT and NOPROMPTFONT qualifiers. If NOFONT or NOPROMPTFONT is specified, your application will use the input window's font for the affected field or prompt.

Setting fonts for an object at runtime

You can set window fonts using the %U_WNDFONT function and input field fonts using the I_FLDMOD or IB_FIELD subroutine.

To set a window font at runtime, use the following syntax:

%U_WNDFONT (D_SETWNDFONT, *window_id*, *font_id*)

To set the font for a field or prompt at runtime, use I_FLDMOD or IB_FIELD. For either routine, use the following argument qualifiers:

D_FLD_FONT, *font_id*

and

D_FLD_PROMPTFONT, *font_id*

To tell your application to use the input window's font for fields and prompts, turn D_FLD_FONT and D_FLD_PROMPTFONT off with the D_OFF qualifier, as follows:

D_OFF, D_FLD_FONT

and

D_OFF, D_FLD_PROMPTFONT

How fonts are applied

For the application window and the objects within it, Synergy/DE determines the font to use according to the following order of precedence (from highest to lowest).



For settings in **synergy.ini** and **synuser.ini**, definitions in the *[myprog]* section override definitions in *[dbr]*, and definitions in *[dbr]* override definitions in *[synergy]*. In addition, a definition in **synuser.ini** overrides the same definition in the same section of **synergy.ini**.

- ▶ Font specified for this field by I_FLDMOD
- ▶ Font specified for this field in a script file or by IB_FIELD
- ▶ Font specified in the repository for this field
- ▶ Font specified in the repository for the template that this field is derived from
- ▶ Font specified by %U_WNDFONT(D_SETWNDFONT)
- ▶ Font specified for the window in a script file
- ▶ Font specified for this class of object by FONT_NUMFLD, FONT_ALPHAFLD, FONT_PROMPT, FONT_TEXT, or FONT_LIST in the *[synergy]*, *[dbr]*, or *[myprog]* section of **synergy.ini** or **synuser.ini**
- ▶ Font specified by FONT_GLOBAL or FONT_ALTERNATE in the *[synergy]*, *[dbr]*, or *[myprog]* section of **synergy.ini** or **synuser.ini**
- ▶ Font specified by the reserved palette entry name DEFAULT or ALTERNATE in the *[fonts]* section of **synergy.ini** or **synuser.ini**
- ▶ Font specified by TYPE_FACE, FONT_HEIGHT, and FONT_WIDTH or ALT_TYPE_FACE, ALT_HEIGHT, and ALT_WIDTH in the *[synergy]*, *[dbr]*, or *[myprog]* section of **synergy.ini** or **synuser.ini**
- ▶ Terminal font, at the point size selected by the system as optimal (either the largest point size for which at least 80 columns of the font will fit on the screen or 10 point, whichever is smaller)

For the header, footer, and information line sections within the application window, Synergy/DE determines the font to use according to the following hierarchy:

- ▶ Font specified by FONT_HEADER, FONT_FOOTER, and FONT_INFO in the *[synergy]*, *[dbr]*, or *[myprog]* section of **synergy.ini** or **synuser.ini**
- ▶ Font specified by the reserved palette entry name STATUS in the *[fonts]* section of **synergy.ini** or **synuser.ini**. If a palette entry named STATUS is not defined, one will be defined using Arial font, point size 10, and sizing character “A”

For the debugger window, Synergy/DE determines the font to use according to the following hierarchy:

- ▶ Font specified by FONT_DEBUG in the [synergy], [dbr], or [myprog] section of **synergy.ini** or **synuser.ini**
- ▶ Font specified by the reserved palette entry name DEBUGGER in the [fonts] section of **synergy.ini** or **synuser.ini**
- ▶ Font specified by the reserved palette entry name DEFAULT in the [fonts] section of **synergy.ini** or **synuser.ini**
- ▶ Font specified by TYPE_FACE, FONT_HEIGHT, and FONT_WIDTH in the [synergy], [dbr], or [myprog] section of **synergy.ini** or **synuser.ini**
- ▶ Terminal font, at the point size selected by the system as optimal (either the largest point size for which at least 80 columns of the font will fit on the screen or 10 point, whichever is smaller)

Using fixed fonts

If you are using a fixed font on Windows, the foreground color (which is the text color) will change according to the following rules: If the foreground color for the palette is in the range 0-7 and the rendition for the object includes a bold attribute, 8 is added to the number for the foreground color. If reverse is applied, the background color and the foreground color are swapped and the above rules are applied to the new foreground color. For instance, the default rendition for fields that aren't in the current input set is palette 3 and includes the bold attribute. So, if you haven't changed this setting and the foreground color is in the range 0-7, it will be increased by 8 (3 will become 11, and so forth).

Using proportional fonts

The following UI Toolkit objects support proportional fonts:

- ▶ Input windows
- ▶ Text windows
- ▶ Selection windows
- ▶ Lists
- ▶ List headers and footers
- ▶ Input window fields
- ▶ Prompts
- ▶ Application header, footer, and information line

Restrictions and recommendations for using proportional fonts

- ▶ Because Synergy/DE maintains support for UNIX and OpenVMS applications, although a proportional font may make an input prompt or input field take up less space within the input window, you cannot take advantage of that extra space by placing something there. If you do place something, it will most likely be truncated or covered by another object.
- ▶ Static text and text in a window are compressed to the left and cannot be justified.
- ▶ A sizing character that is too wide may cause prompts to be clipped. A sizing character that is too narrow may cause extra space to be displayed between the prompt and the field.
- ▶ You cannot align prompts with leading spaces. Prompts should be positioned at the location where their text will begin.
- ▶ Whenever renditions change, justification is used. For example, if this text is in your script
`This is \hbold\ -h text.`
there may be extra space between the words “is” and “bold” and “text.”
- ▶ Horizontal scrolling will occur if text exceeds the width of the edit control. However, a multi-line text field will wrap without scrolling. A multi-line text field will scroll vertically only if exceeds its frame.
- ▶ When attributes are set or cleared, the text that follows begins on a cell boundary, so there may be extra space between an attribute change and the text that precedes or follows it.
- ▶ Text objects used for field headings or prompts will not be positioned correctly and should not be used. Instead, you can use input fields that are not in the set and display the text to these input fields.
- ▶ We recommend that you use a fixed font for the following: the debugger window; when doing non-Toolkit processing; for Toolkit lists, because columns do not line up when a proportional font is used; and for text in a window other than prompts or fields, because the text may wrap in strange places.
- ▶ We recommend that you do not use proportional fonts in conjunction with the row and column arguments to `%UWNDEVENTS_METHOD`, because the column (x-direction) arguments are based on the sizing character for the window and not the actual text.
- ▶ Bold proportional fonts use an actual bold font rather than rendering bold as a color change the way fixed fonts do. Consequently, bold characters are more likely to be truncated, because Toolkit determines the width of the field or prompt using a nonbold (normal) font.
- ▶ For text fields (multi-line alpha fields), the number of characters displayed on each line is equal to the width of the arrayed field. For example, a **3a10** field will display 10 characters on each line, regardless of the font specified. This means there could be a gap between the last character of a line and the frame of the edit field.

Appendices

Appendix A: Maximums

Lists all size and number maximums permitted by UI Toolkit.

Appendix B: Reserved Menu Entries

Lists all reserved menu entries that are used by the various UI Toolkit subroutines that process input.

Appendix C: Keyboard Navigation for Lists on Windows

Lists how on Windows you can navigate to and highlight items with the arrow keys and the PAGE UP, PAGE DOWN, HOME, END, and TAB keys, as well as combinations of these keys and the CTRL, ALT, and SHIFT keys.

Appendix D: Methods

Discusses the difference between asynchronous and synchronous methods, issues you should keep in mind when using input methods, and what happens when context shifts and a method has been called as a menu entry with %M_SIGNAL.

Appendix E: File-Stack Processing

Discusses what file stacks are, how they work, and why you would want to use them.

Appendix F: Linked-List Processing

Discusses how to use linked lists, as well as the advantages of using them.

A

Maximums

Maximum Values Permitted by Toolkit	
Item	Maximum
Box name (size in bytes)	30
Button name (size in bytes)	30
Caption text on Windows (size in bytes)	78
Channels	255
Composite container window, children associated with	2047
Environment levels (set by U_START, which has a default of 16)	32
Field name (size in bytes)	30
File stack (set by FS_INIT, which has a default of 16,384)	114,688
Footer lines	4
Format string (size in bytes)	80
General window name (size in bytes)	15
Header lines	4
Input field display length (size in bytes)	65,535
Input field help identifier (size in bytes)	80
Input field information string (size in bytes)	80
Input field input length (size in bytes)	65,535
Input field name (size in bytes)	30
Input field prompt (size in bytes)	80
Input field user text string (size in bytes)	80

Maximum Values Permitted by Toolkit (Continued)	
Item	Maximum
Input field view length (size in bytes)	9,999
Input field, alpha and multi-dimensional (size in bytes)	65,535
Input field, decimal (size in bytes)	28 for decimal, including implied decimal (for implied decimal, up to 28 can follow the decimal point)
Input field, integer (size in bytes)	1, 2, 4, or 8
Input set name (size in bytes)	30
Input set, fields in	253
Input structure name (size in bytes)	30
Input structure, elements in	507
Input window name (size in bytes)	15
Input window, buttons in	127
Input window, Fields in	253
Input window, input sets in	25
Input window, total characters (and associated data) for all field names	2550
Line name (size in bytes)	30
List “find” window name (size in bytes)	15
List class name (size in bytes)	15
List, data argument associated with (size in bytes)	3,999
List, header/footer line length	32,767
List, user data associated with (size in bytes)	65,535
Loaded windows and lists (note that each list includes multiple windows)	1,024
Menu column name (size in bytes)	15

Maximum Values Permitted by Toolkit (Continued)	
Item	Maximum
Menu columns, number placed (primary)	9
Menu columns, memory available for (in bytes)	65,535
Menu entries per menu column	20
Menu entry name (size in bytes)	10
Menu entry, user-defined text string associated with (size in bytes)	80
Menu list name (size in bytes)	10
Method name (size in bytes)	30
Rendition scheme name (size in bytes)	30
Repository structure alias name (size in bytes)	30
Scroll bar window name (size in bytes)	15
Selection list defined by S_SELBLD or .SELECT, entries in	32,767
Selection list defined by the SELECT qualifier for .FIELD, entries in	99
Selection window name (size in bytes)	15
Structure (size in bytes)	9,999
Tab set window name (size in bytes)	15
Tabs in a tab set	24
Text messages (size in bytes)	200
Text object name (size in bytes)	30
Toolbars	256
Toolbars placed	4 (1 per side)
U_ABORT message line, characters in (UNIX, OpenVMS)	70
U_ABORT message, characters in (Windows)	1,920
U_ABOUT information line, characters in	76
U_DIALOG dialog box, lines in	12

Maximum Values Permitted by Toolkit (Continued)	
Item	Maximum
U_MESSAGE message line, characters in	76
U_MESSAGE window, lines in (on UNIX and OpenVMS; no limit on Windows)	5
U_OPEN, filenames used with (number of characters)	256
Window, max height	32,767
Window, max width	32,767
Window library, text entry in (size in bytes)	15
Window script, physical line in (size in bytes)	255

B

Reserved Menu Entries

The following reserved menu entries are used by the various Toolkit subroutines that process input. These include C_PROCESS, I_INPUT, I_INPFLD, L_SELECT, S_SELECT, T_EDIT, T_VIEW, some of the utility subroutines, and others. (When applicable, the Discussion sections for the subroutines list the menu entries recognized by the subroutine.) Entries that begin with

- ▶ “C_” are for composite window processing
- ▶ “E_” are for editing
- ▶ “I_” are for input
- ▶ “O_” are for key mapping and renditions
- ▶ “S_” are for selection windows and lists
- ▶ “T_” are for changing the view in text windows
- ▶ “TS_” are for tab set processing
- ▶ “U_” are for EUTILS_METHOD processing

Reserved Menu Entries	
Menu entry	Description
C_FIRST	Make the first child or button in a composite window active.
C_LAST	Make the last child or button in a composite window active.
C_NEXT	Make the next child or button in a composite window active.
C_PREV	Make the previous child or button in a composite window active.
C_RESET	Reset the context in a composite window to the currently active child or button.
C_SET <i>nnnn</i>	Make the child window or list with a tabbing index of <i>nnnn</i> active.
E_BMOV	Move to beginning of line.
E_BOTTOM	Move to bottom of window.
E_CDEL	Delete current character.

Reserved Menu Entries (Continued)	
Menu entry	Description
E_CLEAR	Clear text.
E_COPY	Copy marked text.
E_CUT	Cut marked text.
E_DIR	Toggle direction.
E_DOWN	Move down.
E_EDEL	Delete to end of line.
E_EMOV	Move to end of line.
E_HARD	Toggle between inserting hard and soft returns when user presses ENTER.
E_ILIN	Insert line.
E_JOIN	Join paragraph.
E_LCLR	Clear line.
E_LDEL	Delete line in the current direction.
E_LEFT	Move one character left.
E_LMOV	Move one line in the current direction.
E_MARK	Mark text.
E_MODE	Toggle between insert and overstrike mode.
E_PASTE	Paste text.
E_PDEL	Delete paragraph in the current direction.
E_PMOV	Move one paragraph in the current direction.
E_RIGHT	Move one character right.
E_TOP	Move to top of window.
E_UP	Move up.
E_WDEL	Delete word in the current direction.
E_WMOV	Move one word in the current direction.

Reserved Menu Entries (Continued)	
Menu entry	Description
E_XDOWN	Move down, exiting at field boundary.
E_XUP	Move up, exiting at field boundary.
I_CANCEL	Restore field to its initial state.
I_DFLT	Fill all fields that have a specified default with their default. (I_DFLT doesn't affect fields that don't have defaults.)
I_DONE	Fill all fields with defaults.
I_DRILL	Invoke drill method for field
I_EDIT	Go into edit mode.
I_FRST	Move to first field.
I_FRSTCLR	Move to first empty field.
I_HYPER	Invoke hyperlink method for field
I_LAST	Move to last field.
I_NEXT	Move to next field.
I_NEXTCTL	Move to next field or button.
I_NXTCLR	Move to next empty field.
I_OK	Fill with defaults and exit I_INPUT.
I_PREV	Move to previous field.
I_PREVCTL	Move to previous field or button.
O_ACCEPT	Accept the shortcut sequence directly.
O_CANCEL	Cancel changes.
O_CHGTERM	Change which terminal key map is being modified.
O_COPYTERM	Copy a terminal key map definition.
O_DEFAULT	Set renditions to their default values.
O_DELTERM	Delete a terminal key map definition.
O_EXIT	Exit and save all changes.

Reserved Menu Entries (Continued)	
Menu entry	Description
O_HELP	Call EHELP_METHOD and pass <i>help_id</i> .
O_LDSCHEME	Load rendition schemes from the rendition file.
O_PALETTE	Modify a color palette.
O_RESET	Reset to the working rendition scheme that was saved on entry.
O_RMSCHEME	Remove a rendition scheme from the rendition file.
O_STSCHEME	Store rendition schemes to the rendition file.
O_TIMEOUT	Signal that a time-out has occurred.
S_BOTTOM	Move to bottom of a list.
S_DOWN	Move down in a selection window.
S_FIND	Prompt for a prompt string and then find that string in a list.
S_FINDDIR	Toggle the direction of the find in a list.
S_FINDNEXT	Find the next item in a list that has the same value as the previous find.
S_LEFT	Move left in a selection window.
S_PGBOT	Move to the bottom of the current page of a list.
S_PGTOP	Move to top of the current page of a list.
S_PGUP	Move to previous page of a list.
S_RIGHT	Move right in a selection window.
S_TOP	Move to top of a list.
S_UP	Move up in a selection window.
T_BOTTOM	Move to bottom of window.
T_LINDWN	Scroll down one line.
T_LINUP	Scroll up one line.
T_LSIDE	Move to left side of window.
T_PAGDWN	Scroll down one page.

Reserved Menu Entries (Continued)	
Menu entry	Description
T_PAGUP	Scroll up one page.
T_RSIDE	Move to right side of window.
T_SCRDWN	Scroll down one half page.
T_SCRLFT	Scroll left 10 columns.
T_SCRRT	Scroll right 10 columns.
T_SCRUP	Scroll up one half page.
T_TOP	Move to top of window.
TS_TAB <i>n</i>	Switch to tab <i>n</i> .
TS_TABNEXT	Switch to next active tab, wrapping if necessary.
TS_TABPREV	Switch to previous active tab, wrapping if necessary.
U_xxx	Call the EUTILS_METHOD subroutine.

C

Keyboard Navigation for Lists on Windows

Along with using the mouse to navigate lists on Windows, users can highlight list items with the arrow keys and the PAGE UP, PAGE DOWN, HOME, END, and TAB keys, as well as combinations of these keys and the CTRL, ALT, and SHIFT keys. The table below documents what these keys and key combinations do by default. Note the following:

- ▶ Some keys and key combinations are *privatized*—that is, they are reserved for the list and can’t be assigned as menu shortcuts without calling `L_RETURNKEY` and setting their return states to true. (See [L_RETURNKEY on page 9-123](#) for more information.) These are marked as “privatized” in the “Key or key combination” column of the table below.
- ▶ Some key combinations can never be assigned as shortcuts. These are marked as “list only” in the “Key or key combination” column of the table below.
- ▶ Some key combinations do nothing; these are marked “N/A”.
- ▶ The `RowMode` published property determines whether a cell or a row is highlighted for ActiveX Toolkit lists. If `RowMode` is off, one cell is highlighted at a time. If it’s on, one row is highlighted. The table below includes columns for the two `RowMode` states.
- ▶ The `ArrowKeyAction` published property affects the way some keys and key combinations work for ActiveX Toolkit lists. With the default setting, which is 0, arrow keys move from one row to the next. If `ArrowKeyAction` is set to 1, arrow keys move from one list item (which can be more than one row) to the next. The table below includes columns for the two `ArrowKeyAction` states.
- ▶ Toolkit lists (non-ActiveX) page up by shifting the highlighted row to the bottom of the display (if there are enough items left in the list) and then highlighting the row at the top of the display. They page down by doing the converse, by moving the highlighted row to the top of the page, and so forth.

For more information on published properties for ActiveX Toolkit lists, see “[Published Properties for ActiveX Toolkit Lists](#)” on page 9-17.

Key Behavior for Lists in Windows Environments					
Key or key combination	Standard Toolkit lists (non-ActiveX)	ActiveX lists			
		ArrowKeyAction=0		ArrowKeyAction=1	
		RowMode off	RowMode on	RowMode off	RowMode on
UP ARROW (privatized)	Move up one row	Move up one row (highlights same cell in previous row)	Move up one row (highlights entire row)	Move up one list item (highlights same cell in the next list item ^a)	Move up one list item (highlights same row in previous list item)
DOWN ARROW (privatized)	Move down one row	Move down one row (highlights same cell in next row)	Move down one row (highlights entire row)	Move down one list item (highlights same cell in the previous list item ^a)	Move down one list item (highlights same row in next list item)
LEFT ARROW (privatized)	N/A	Move left one cell (no wrapping; stops at end of row)	N/A	Move left one cell (wraps up to previous row of list item; does not wrap beyond first cell of list item)	Move up one row within the list item (highlights entire row; does not move beyond first row of list item)
RIGHT ARROW (privatized)	N/A	Move right one cell (no wrapping; stops at end of row)	N/A	Move right one cell (wraps down to next row of list item; does not wrap beyond last cell of list item)	Move down one row within list item (highlights entire row; does not move beyond last row of list item)
PAGE UP (privatized)	Page up	Page up	Page up	Page up	Page up
PAGE DOWN (privatized)	Page down	Page down	Page down	Page down	Page down
HOME (privatized)	Move to first row in list	Move to first cell in row	N/A	Move to first cell in first row of list item	Move to first row in list item (highlights entire row)

Key Behavior for Lists in Windows Environments (Continued)					
Key or key combination	Standard Toolkit lists (non-ActiveX)	ActiveX lists			
		ArrowKeyAction=0		ArrowKeyAction=1	
		RowMode off	RowMode on	RowMode off	RowMode on
END (privatized)	Move to last row in list	Move to last cell in row	N/A	Move to last cell in last row of list item	Move to last row in list item (highlights entire row)
TAB	N/A ^b	Move right one cell (wraps to first cell in same row) ^b	N/A ^b	Move right one cell (wraps to first cell in next row of list item; wraps from last cell in list item to first cell in list item) ^b	N/A ^b
CTRL+ UP ARROW	Move up one row	Move up one row (highlights same cell in previous row)	Move up one row (highlights entire row)	Move up one row (highlights same cell in previous row)	Move up one row (highlights entire row)
CTRL+ DOWN ARROW	Move down one row	Move down one row (highlights same cell in next row)	Move down one row (highlights entire row)	Move down one row (highlights same cell in next row)	Move down one row (highlights entire row)
CTRL+ LEFT ARROW	N/A	Move left one cell (no wrapping; stops at end of row)	N/A	Move left one cell (no wrapping; stops at end of row)	N/A
CTRL+ RIGHT ARROW	N/A	Move right one cell (no wrapping; stops at end of row)	N/A	Move right one cell (no wrapping; stops at end of row)	N/A
CTRL+ PAGE UP	Page up	Move to first cell in row	N/A	Move to first cell in row	N/A
CTRL+ PAGE DOWN	Page down	Move to last cell in row	N/A	Move to last cell in row	N/A
CTRL+HOME	Move to first row in list	Move to first cell in list	Move to first row in list (highlights entire row)	Move to first cell in list	Move to first row in list (highlights entire row)

Key Behavior for Lists in Windows Environments (Continued)					
Key or key combination	Standard Toolkit lists (non-ActiveX)	ActiveX lists			
		ArrowKeyAction=0		ArrowKeyAction=1	
		RowMode off	RowMode on	RowMode off	RowMode on
CTRL+END	Move to last row in list	Move to last cell in list	Move to last row in list (highlights entire row)	Move to last cell in list	Move to last row in list (highlights entire row)
ALT+ UP ARROW (list only)	N/A	N/A	N/A	N/A	N/A
ALT+ DOWN ARROW (list only)	N/A	N/A	N/A	N/A	N/A
ALT+ LEFT ARROW (list only)	N/A	N/A	N/A	N/A	N/A
ALT+ RIGHT ARROW (list only)	N/A	N/A	N/A	N/A	N/A
ALT+ PAGE UP (list only)	N/A	Move to first cell in row	N/A	Move to first cell in row	N/A
ALT+ PAGE DOWN (list only)	N/A	Move to last cell in row	N/A	Move to last cell in row	N/A
ALT+HOME (list only)	N/A	N/A	N/A	N/A	N/A
ALT+END (list only)	N/A	N/A	N/A	N/A	N/A
SHIFT+ UP ARROW	Move up one row	Move up one row (highlights same cell in previous row)	Move up one row (highlights entire row)	Move up one row (highlights same cell in previous row)	Move up one row (highlights entire row)

Key Behavior for Lists in Windows Environments (Continued)					
Key or key combination	Standard Toolkit lists (non-ActiveX)	ActiveX lists			
		ArrowKeyAction=0		ArrowKeyAction=1	
		RowMode off	RowMode on	RowMode off	RowMode on
SHIFT+ DOWN ARROW	Move down one row	Move down one row (highlights same cell in next row)	Move down one row (highlights entire row)	Move down one row (highlights same cell in next row)	Move down one row (highlights entire row)
SHIFT+ LEFT ARROW	N/A	Move left one cell (no wrapping; stops at end of row)	N/A	Move left one cell (no wrapping; stops at end of row)	N/A
SHIFT+ RIGHT ARROW	N/A	Move right one cell (no wrapping; stops at end of row)	N/A	Move right one cell (no wrapping; stops at end of row)	N/A
SHIFT+ PAGE UP	Page up	Page up	Page up	Page up	Page up
SHIFT+ PAGE DOWN	Page down	Page down	Page down	Page down	Page down
SHIFT+HOME	Move to first row in list	Move to first cell in row	N/A	Move to first cell in row	N/A
SHIFT+END	Move to last row in list	Move to last cell in row	N/A	Move to last cell in row	N/A
SHIFT+TAB	N/A ^b	Move left one cell (wraps to last cell in same row) ^b	N/A ^b	Move left one cell (wraps to last cell in previous row of list item; wraps from first cell in list item to last cell in list item) ^b	N/A ^b
CTRL+SHIFT+ UP ARROW (list only)	Move up one row	Move up one row (highlights same cell in previous row)	Move up one row (highlights entire row)	Move up one row (highlights same cell in previous row)	Move up one row (highlights entire row)

Key Behavior for Lists in Windows Environments (Continued)					
Key or key combination	Standard Toolkit lists (non-ActiveX)	ActiveX lists			
		ArrowKeyAction=0		ArrowKeyAction=1	
		RowMode off	RowMode on	RowMode off	RowMode on
CTRL+SHIFT+DOWN ARROW (list only)	Move down one row	Move down one row (highlights same cell in next row)	Move down one row (highlights entire row)	Move down one row (highlights same cell in next row)	Move down one row (highlights entire row)
CTRL+SHIFT+LEFT ARROW (list only)	N/A	Move left one cell (no wrapping; stops at end of row)	N/A	Move left one cell (no wrapping; stops at end of row)	N/A
CTRL+SHIFT+RIGHT ARROW (list only)	N/A	Move right one cell (no wrapping; stops at end of row)	N/A	Move right one cell (no wrapping; stops at end of row)	N/A
CTRL+SHIFT+PAGE UP (list only)	Page up	Move to first cell in row	N/A	Move to first cell in row	N/A
CTRL+SHIFT+PAGE DOWN (list only)	Page down	Move to last cell in row	N/A	Move to last cell in row	N/A
CTRL+SHIFT+HOME (list only)	Move to first row in list	Move to first cell in list	Move to first row in list (highlights the entire row)	Move to first cell in list	Move to first row in list (highlights the entire row)
CTRL+SHIFT+END (list only)	Move to last row in list	Move to last cell in list	Move to last row in list (highlights the entire row)	Move to last cell in list	Move to last row in list (highlights the entire row)

- a. The same column is activated in the previous or next row or list item. For example, if a list item has eight columns in two rows (four columns each), and the fifth column in the list item is active (i.e., the first cell in the second row), pressing down arrow activates the fifth column in the next item in the list.
- b. If the list is part of a composite window, TAB and SHIFT + TAB move focus to the next or previous child window or enabled button on the composite container window.

D

Methods

Methods: Asynchronous vs. Synchronous D-2

Lists and describes the two kinds of methods: methods that are called as soon as the triggering event takes place and methods that are deferred until Toolkit is ready.

Notes on Input Methods D-3

Discusses issues you should keep in mind when using input methods (arrive, leave, change, edit format, and display).

Signaling a Menu Entry from a Method D-10

Discusses what happens when context shifts and a method has been called as a menu entry with %M_SIGNAL.

Methods: Asynchronous vs. Synchronous

Methods are subroutines you write, but they aren't something you call from your program. UI Toolkit calls methods automatically when certain events take place. Some methods are called immediately when a given event takes place. Others aren't called until Toolkit is ready. Methods are documented throughout this manual, but this appendix discusses issues common to types of methods.

Asynchronous methods are called as soon as their events take place. For example, if you've written an arrive method for a field, the arrive method is called immediately when the user arrives on the field. Asynchronous methods include the following:

- ▶ Arrive, leave, change, edit format, and display input methods
- ▶ Arrive, leave, double-click, and load list methods
- ▶ Toolbar button methods
- ▶ %UWNDEVENTS_METHOD methods (including AX_ methods)
- ▶ The following E_METHOD environment methods: D_METH_APPCLOSE, D_METH_CHKFLD, D_METH_DSPFLD, D_METH_EDTDSP, D_METH_ENTRST, D_METH_APPMOVE, D_METH_APPSIZ, and D_METH_APPSTATE

Within asynchronous methods on Windows, it's best to avoid doing any user-interactive processing (with the possible exception of a highly conditional message box). Asynchronous methods need to be self contained and they need to execute quickly because they are triggered by Windows events, so it is difficult to determine exactly when the method might be used in relation to your code and Toolkit code. See [“Notes on Input Methods” on page D-3](#) for more restrictions.

Synchronous methods are *not* called when their events take place, but are deferred until Toolkit is ready to call them. For example, if you write a drill method for an input field, the method will not be called immediately when the user clicks the drilldown button or selects the drilldown menu entry. Instead, if the input field is in focus, the leave method for the field is first invoked, then the drill method is called. Synchronous methods include the following:

- ▶ Hyperlink and drilldown input methods
- ▶ The following E_METHOD environment methods: D_METH_UTILS, D_METH_HELP, D_METH_FKEY, D_METH_CENTURY, and D_METH_SCRIPTERR
- ▶ Methods invoked as a result of %M_SIGNAL
- ▶ Tab set methods
- ▶ Composite window methods

Notes on Input Methods

WIN

- ▶ Avoid calling anything that forces reconstruction of the input field frameset on the window being processed. This includes `I_SETDEL`, `I_SNAPSHOT (D_LOAD)`, `I_DSPAREA`, `I_FRAMES`, `T_SETUP`, `S_SELLD` (on an associated selection window). If you do, it will result in a `U_ABORT` message.
 - ▶ Any shift of input focus is allowed and supported within `%IDISPLAY_METHOD`, `%IEDITFMT_METHOD`, and `%ICHANGE_METHOD`, including calling `I_NEXT` on the current input set.
-

- ▶ Changing the read-only state for a field from an input method for the field (by using `I_READONLY` or `I_FLDMOD`) won't affect input processing. For example, if a change method for a field sets the field to read-only, Toolkit will complete the rest of the method sequence (change method, display method, and leave method). Setting a field to read-only in an input method simply prevents the user from modifying the field any further.

Additionally, note the following for edit format methods:

- ▶ If a field is read-only when it receives focus, you can't turn off the read-only state from an edit format method. No edit format method will be invoked.
- ▶ If an edit format method sets a field to read-only, the user won't be able to modify the field. The field will contain whatever the edit format method returns.

For all asynchronous input methods, note the following:

- ▶ The contents of `g_fldnam` are undefined.
- ▶ `I_INPUT` may not exit until after processing the arrive method for another field. Therefore, you cannot use `g_fldnam` to get the field's name. Instead, use `%I_GETSTRING(inp_wndid, inp_fldnam)`.
- ▶ Except for `E_CUT`, `E_COPY`, `E_PASTE`, `E_CLEAR`, and `E_MARK`, "E_" menu entries will be ignored if signaled. All other menu entries (including "I_") will be dispatched as if signaled while performing input to the field.
- ▶ `I_ENABLE` and `I_DISABLE` can be used to change the enabled state of this or other fields.

UNIX, VMS

- ▶ Using `I_ENABLE` or `I_DISABLE` to enable or disable the next field that would be accessed changes the course of input.
- ▶ You can use `I_SETDEL` and `I_SNAPSHOT` to modify input set context and/or contents. The behavior is exactly as if the action were performed after a break. Thus, using `I_SETDEL` to delete the next field to be accessed alters the context, and using `I_SNAPSHOT` with `D_LOAD` restores the context for the next input. Using `I_SNAPSHOT` with `D_COPY` saves the context that points to the next field to be accessed (if `I_NEXT` was not also called) because the change in context that causes the leave has not yet occurred.

-
- ▶ Using `I_DISPLAY` or `I_DSPFLD` marks fields as “not empty.” The behavior is exactly as if the action were performed after a break. Subsequent input set processing depends upon the previous context.
 - ▶ Using `I_INIT` for a field marks the field as “empty.” Using `I_INIT` for a set resets the context to the first field in the set, as well as marks all fields in the set as “empty.”
 - ▶ Performing input to the current set using `I_INPUT` or fields within the current set using `I_INPFLD` is possible but not recommended because it may cause confusion: input to the current set will alter the input context for subsequent input. Input to fields in the current set will alter their “empty” state.

For leave methods, note the following:

- ▶ Accessing (performing input to or displaying) other windows, sets, or fields, within the leave method will not affect the current input process. Accessing the current window, set, or field will affect the input process as noted above.
- ▶ If you modify the information line using `E_SECT` in your leave method, this will occur after restoring any information line superseded by the `INFORMATION` field qualifier. That is, your new information line will persist within the current environment. This is symmetrical with the `arrive` method, which occurs prior to overloading the information line for the field.
- ▶ The `a_inpinfo` record contains the following information:

```
inp_entered   ,i4   ;TRUE if data was modified in this field
inp_return    ,i4   ;TRUE if the RETURN key was pressed on this field
```

Either of these fields may be true or false, as shown in the following table.

inp_entered	inp_return	Meaning
False	False	User has neither modified the field nor pressed RETURN (ENTER).
True	False	User has modified field but has not pressed RETURN (ENTER).
False	True	Field was not previously empty, and user only pressed RETURN (ENTER).
True	True	The user pressed RETURN (ENTER) and either the field was previously empty and/or the user modified the field.

Method sequences

Depending on the user's action, Toolkit calls methods in one the sequences listed below. These sequences start when the user moves to a new field. Note that these are the basic sequences; additional method calls from these methods will complicate the sequences.

Method Sequences	
User activity	Method sequence
Enter or edit data in a field and then move to another field.	change display leave break arrive edit format
Signal I_OK. ▶ The first four methods (the first change through the first break) are for the current field if that field has been modified or is empty—even if it has a default value. ▶ The last six methods are called for every subsequent empty field.	change display leave break arrive edit format change display leave break

Method Sequences (Continued)	
User activity	Method sequence
Click a drilldown button or hyperlink prompt for the current field (a drilldown button or hyperlink prompt whose method uses I_FORCE to modify the field's data).	leave drill arrive change display break arrive edit format
Enter or edit data in a field and then click a drilldown button or hyperlink prompt for a different field (a drilldown button or hyperlink prompt whose method uses I_FORCE to modify the second field's data). ▶ The first five methods (the first “change” through the first “arrive”) are for the field the user edited. ▶ The next eight methods (the first “edit format” through the last “break”) are for the field whose drilldown button or hyperlink prompt was clicked. ▶ The final two methods are for the next field in the input sequence, the field that receives focus after the field with the drilldown button or hyperlink prompt.	change display leave break arrive edit format leave drill arrive change display leave break arrive edit format

To be consistent with Toolkit’s handling of other internally supported menu entries (I_NEXT, I_PREV, and so on), place drill and hyperlink events after the leave event. This enables you to validate the field (or to avoid validation by testing `g_select/g_entnam`) in the leave method without having to duplicate it in the drill or hyperlink method. Furthermore, it provides the same sequence of events for both non-Windows and Windows platforms.

WIN

If the user clicks on the hyperlink prompt or drilldown button for a field other than the one currently in focus, several things could happen:

- ▶ If any input in the currently focused field has not yet been validated, it will be. This could cause a validation message to pop up and focus to be forced back to the current field.
- ▶ If the current field is a break field and the field has been modified or the field is a “break always” field, when the user attempts to click the hyperlink prompt or drilldown button of another field, a break occurs. This effectively cancels the click, although focus naturally

transfers to the new field, unless your break logic alters the input set context. Thus, in some cases with break fields, two clicks may be required to access the hyperlink for a field other than the one currently being processed.

- ▶ If the new field has an arrive method, this method is invoked before the hyperlink or drill method. If the arrive method signals a menu entry or alters the input set context, this may effectively cancel the hyperlink or drilldown click.
-

Using hyperlink and drill methods

Both hyperlink and drill methods can be used for a number of things. Primarily, they are used for some form of lookup or presentation of additional information. Your subroutine can use `I_FORCE` or `I_DSPFLD` to return data back into the field being processed. If the default type for the field is `COPY` and the field is empty, any change you make to the data area will affect the default value for the field. You can also alter other fields or other windows. You can perform input to another window, a list, or a selection window. You can alter the input set context on the current window using `I_NEXT`. You can signal a menu entry by calling `%M_SIGNAL`. If you signal a menu entry in your drill or hyperlink method, it will be dispatched appropriately. Menu entries beginning with “`I_`” are handled by the input processor. “`U_`” menu entries invoke any `EUTILS_METHOD`. “`O_HELP`” invokes any `EHELP_METHOD`. Other menu entries are returned to the calling routine outside `I_INPUT`.

See “[Method sequences](#)” above for information on the sequence of events that occur when a user clicks a hyperlink prompt or a drilldown button.

Changing focus in an input method on Windows

Although you can call `I_NEXT` to change the current field within the current input set, if you have a change method and `%ICHANGE_METHOD` returns a validation error, the context for the current input set will be forced back to the current field, even if `I_NEXT` has been called.

Note that if a change method signals a menu entry and the response to that menu entry causes the field to be modified (for instance, with `I_FORCE`), an infinite loop may occur.

In leave methods, the problems can be similar though more complex. When the leave method is called, focus has already been established on the new field. If the leave method steals focus (with `U_MESSAGE`, for example), focus is restored to the new field. This works well, unless the leave method also calls `I_NEXT` to force the input context to a different field. In this case, the original new field loses focus and invokes its leave method before arriving at the field specified by `I_NEXT`. Additionally, the field referenced by `*NEXT*` and `*PREV*` will be different depending on whether or not the leave method stole focus.

To work around these problems, we recommend that you avoid stealing focus or calling `I_NEXT` in either arrive or leave methods. You can also use `%M_SIGNAL` in the method to signal a menu entry (which does not have to be on a menu) that tells your program which action to take. This forces `I_INPUT` to exit with `g_select` set to true and `g_entnam` set to the entry name you signaled. You can then take the desired action in the calling routine and loop back into `I_INPUT` to continue.

To do this efficiently, we recommend creating a “wrapper” for `I_INPUT` that handles the problem—assuming you can sufficiently abstract the problem to create a generalized solution. For instance, let’s say you want to display a message box and/or advance to a specific field from either an arrive or leave input method. To do this, you would create the following fields in either a common or global data section. For the following example, assume these fields are defined in an include file that’s named **myinput.def**.

```
g_break_msg    ,a256
g_break_next   ,a30
```

Next, you would create a wrapper for `I_INPUT`. For example:

```
subroutine my_input, reentrant
;
; Description: Wrapper for I_INPUT
;
; Arguments:
;
    a_inpid      ,n      ;Input window ID
    a_inpset     ,a      ;(Optional) set name
    a_record     ,a      ;Data record area
    .
    . (etc., all the rest of I_INPUT's arguments)
    .
.include "myinput.def"      ;References the global fields we need
proc
    repeat
        begin
            xcall i_input(a_inpid, a_inpset, a_record, ...)
            if (%m_signal(entnam, D_REMOVE)) then
                using entnam select
                ("MY_BREAK "), begin
                    if (g_break_msg)
                        begin
                            xcall u_message(%atrim(g_break_msg))
                            clear g_break_msg
                        end
                    if (g_break_next)
                        begin
                            xcall i_next(a_inpid, a_inpset, g_break_next)
                            clear g_break_next
                        end
                end
            end
        end
    end
```



```

        ( ),      begin
                    xcall m_signal(entnam, D_SIGNAL) ;Unrecognized--resignal
                    exitloop
                end
            endusing
        else
            exitloop
        end
    xreturn
endsubroutine

```

With the above example, you can generate a message and/or call I_NEXT from an arrive or leave input method. In the method, you can display **g_break_msg** as the message and/or set **g_break_next** to access the next field. You would then call %M_SIGNAL with the MY_BREAK argument to force a break out of I_INPUT and into the above logic. The following makes this step even simpler:

```

subroutine my_break, reentrant
;
; Description: Display message and/or call I_NEXT for a method
;
; Arguments:
;
    a_msg      ,a      ;(Optional) Message to be displayed
    a_next     ,a      ;(Optional) Next field to access
.include "myinput.def"
proc
    if (^passed(a_msg)) then
        g_break_msg = a_msg
    else
        clear g_break_msg
    if (^passed(a_next)) then
        g_break_next = a_next
    else
        clear g_break_next
    xcall m_signal("MY_BREAK")
    xreturn
endsubroutine

```

Signaling a Menu Entry from a Method

In method routines, a twilight zone exists from the moment you use `%M_SIGNAL` to signal a menu entry to the moment control returns from the method routine. An `%M_SIGNAL (D_SIGNAL)` call in a method routine simultaneously sets `g_select` and `g_entnam` and instructs the Synergy runtime to signal the menu entry when control returns from the method routine. So if you clear `g_select` or `g_entnam` or set them to something different during this time, your changes will be ignored. As soon as control returns from the method routine, they will be reset to the values set by the `%M_SIGNAL` call because the Synergy runtime has already received its instructions. We recommend, however, that you return immediately after the `%M_SIGNAL` call, and we recommend that you avoid modifying `g_select` or `g_entnam` after the `%M_SIGNAL` call. (`%M_SIGNAL` can also be used to cancel a signaled menu entry. See [%M_SIGNAL on page 10-52](#) for information.)

Note the following:

- ▶ If you signal a menu entry (using `%M_SIGNAL`) twice without dispatching the entry between the calls, the first call is discarded. There is no event queue.
- ▶ If you use `%M_SIGNAL` to signal a menu entry when not in a method routine and then call a Toolkit input routine, such as `I_INPUT`, the event is forgotten. Again, this is because `%M_SIGNAL` signals the menu entry by setting `g_select` and `g_entnam`. `I_INPUT` resets these upon entry, as do the other Toolkit input routines. Note, however, that if the `%M_SIGNAL` call causes an input field leave method to be invoked, and if the method modifies `g_select` or `g_entnam`, they remain modified. This causes the `%M_SIGNAL` call to behave just like a menu entry selected by the user. It also means that a leave method can effectively cancel a signaled menu entry or signal a different menu entry.
- ▶ On Windows, by the time an input field change or leave method signals a menu entry (by using `%M_SIGNAL`), the input context has already begun to shift to the field the user is trying to access. On UNIX and OpenVMS systems, however, the context does not shift to the next input field until menu entries are processed (`I_NEXT`, `I_PREV`, etc.). So signaling a menu entry from a change or leave method will typically cancel the movement to the next field because the new menu entry overwrites the menu entry the user selected. Even if the user left the field by pressing RETURN, signaling a menu entry on UNIX or OpenVMS implies that focus should not change unless the program explicitly moves the focus. The same thing occurs on Windows if the user uses a menu entry (`I_NEXT`, `I_PREV`, and so forth) for movement, but does not occur if the user used TAB/SHIFT+TAB, ENTER or the mouse to leave the field.
- ▶ If a change method signals a menu entry and the response to that menu entry causes the field to be modified (for instance, with `I_FORCE`), an infinite loop may occur.

Combinations of these subtleties can create order-of-magnitude subtleties. For instance, if a method routine signals a menu entry with `%M_SIGNAL` and then calls a Toolkit input routine such as `I_INPUT`, `I_INPUT` clears `g_select` and `g_entnam` and may signal another menu entry. But unless `%M_SIGNAL` is called again before returning from the method, the lost menu entry will be resurrected when the method returns. In general, it is a good idea to avoid these situations, and it's best to avoid nesting methods. You can do this by signaling the menu entry and then returning and

letting the outer-level routine dispatch the signaled menu entry. This approach results in a somewhat flatter application design, which more closely matches Toolkit's event paradigm, where there is no event stack or queue.

There is one subtle effect of `%M_SIGNAL` that makes things easier: `%M_SIGNAL` helps you to synchronize events. On Windows platforms, there are two different types of event processing: the UI Toolkit processing of the menu entry and Windows messaging. Because Windows messages occur quickly—you can get hundreds of them by simply moving the mouse across a window—the Synergy runtime dispatches these messages between lines of Synergy code. And because asynchronous methods are invoked immediately by their associated Windows events, a toolbar method, for example, can be called between statements of another Synergy routine. If the toolbar method and the routine it interrupts access shared data, the result can be extremely confusing. If, however, you limit the toolbar routine to calling `%M_SIGNAL` with an event that will be dispatched later, you have ensured that the associated code will not be executed asynchronously.

E

File-Stack Processing

Introduction to File-Stack Processing E-2

Discusses what file stacks are, how they work, and why you would want to use them.

The File-Stack Subroutines E-4

FS_APPEND – Append a record to the current logical file	E-5
FS_CLOSE – Close one or more logical files	E-6
FS_INIT – Initialize a file-stack system	E-7
FS_OPEN – Add a logical file to the top of the stack	E-8
FS_PARAM – Get a logical file’s parameters	E-9
FS_QUIT – Terminate a file-stack system	E-11
FS_READ – Read an existing record from a logical file	E-12
FS_WRITE – Write a record to a logical file	E-13

Introduction to File-Stack Processing

What is a file stack?

A *file stack* is multiple sets of fixed-length, logical records, stored in a single, physical scratch file. These logical record sets are organized in the file in a last-in, first-out (LIFO) stack manner.

Why use file stacks?

Let's say that your application does some processing that requires the use of a temporary file. For instance, it presents the user with a list of choices, and records each one selected into the temporary file. However, the user might use a hot entry to pop up a different function, also using a temporary file. Traditionally, even though the first process is suspended, you had to open a separate temporary file for the second process.

Instead, you would like to extend your already-opened temporary file for use by the hot-entry process, and then cut back to your original use of the file when the hot-entry process returns.

When you use the file stack, you have several advantages over creating and maintaining your own temporary files. The main advantages are the following:

- ▶ You do not have to come up with unique names or manage the channels for separate files.
- ▶ You minimize the number of open channels.
- ▶ You localize the mechanism for managing temporary file space to common routines.
- ▶ You tune your file-stack system for high-speed access of temporary work space.

How file stacks work

The file stack enables you to “stack” temporary files in one file. Each of these temporary files, also called logical files, may have a different record length. As the logical files are stacked, the one most recently stacked becomes the “current” file. When that one is closed, the file that was stacked immediately prior to it becomes the current file.

Typically, processing occurs on the current file, which is also the current level, though you may read from and write to records on any level.

Only the current file can be extended. If you need to extend more than one logical file at a time, you can initiate multiple file stacks. For instance, in the distributed version of the list-processing routines, each list has its own private file stack.

How much memory do the file stacks allocate?

The amount of memory allocated by the file stack will be the largest specification among the following:

- ▶ The default size 16,384.
- ▶ The size specified by environment variable DTKFSWINSIZ.
- ▶ The size specified by FS_INIT.
- ▶ The size needed by the file stack routines.

Any value specified is rounded up to the nearest multiple of 16,384 but not more than 114,688.

Note, however, that file stack processing caches data so that data is saved to disk only if the size of the cache (which is just over 100K is exceeded).

The File-Stack Subroutines

UI Toolkit file-stack subroutines can maintain one or more file stacks. These routines are used by Toolkit's list processor, and can be used from your applications as well.

Multiple file stacks are supported by passing an optional channel number, which is synchronized with the current file stack number. If the channel number is not passed, the default file stack will be used. The default file stack is separate from any file stack accessed by a channel number.

The file-stack subroutines allocate a private memory cache to minimize disk I/O. The scratch file is, therefore, not opened until the cache overflows for the first time. Thus, the channel number passed to these routines is not a true Synergy I/O channel, but rather a handle (virtual channel).

UI Toolkit supports the following file-stack subroutines:

FS_APPEND – Append a record to the current logical file

FS_CLOSE – Close one or more logical files

FS_INIT – Initialize a file-stack system

FS_OPEN – Add a logical file to the top of the stack

FS_PARAM – Get a logical file's parameters

FS_QUIT – Terminate a file-stack system

FS_READ – Read an existing record from a logical file

FS_WRITE – Write a record to a logical file

FS_APPEND – Append a record to the current logical file



```
xcall FS_APPEND (record, [channel][, error])
```

Arguments

record

The record to write. **(a)**

channel

(optional) The file-stack channel. If not passed, the default file-stack channel is used. **(n)**

error

(optional) Returned with one of the following error flags: **(n)**

0 No error occurred.

1 Not a file-stack channel.

Discussion

The FS_APPEND subroutine appends a record to the current logical file.

It is assumed that the size of the data transfer is the size of the accessed logical file's record length, as specified in FS_OPEN.

This subroutine only accesses the current (highest) level of the file stack. The record number of the appended *record* will be one greater than the highest record number written to this level. A new EOF point will be set for the level.

If *error* is not passed, and an error occurs, a fatal program error is generated.

Examples

The following example appends the data in **page** to the end of the current (highest) level in the default file stack.

```
xcall fs_append(page,, sts)
```

FS_CLOSE – Close one or more logical files



```
xcall FS_CLOSE ([level][, channel])
```

Arguments

level

(optional) The level through which to close. (**n**)

channel

(optional) The file-stack channel. If not passed, the default file-stack system is used. (**n**)

Discussion

The FS_CLOSE subroutine will close one or more file stack levels, always starting with the current one, and ending with the *level* specified. The highest level number is always the current one.

Examples

The following example will close file stack levels 3 and higher (through the current level).

```
xcall fs_close(3)
```

The second example will close the current file-stack level in the file stack whose channel is stored in **filchn2**.

```
xcall fs_close(, filchn2)
```

FS_INIT – Initialize a file-stack system



```
xcall FS_INIT([filename], [channel], [size])
```

Arguments

filename

(optional) The name of the physical scratch file to be created and used for the file stack. **(a)**

channel

(optional) Returned with the file-stack channel. **(n)**

size

(optional) The size (in bytes) of the memory to be allocated for the file-stack cache. If not specified, the default is 16,384. **(n)**

Discussion

The FS_INIT subroutine initializes a file-stack system.

If *filename* is not passed, a filename in the form **DTKTMP:XFSnnn.SCR** will be used, where *nnn* is the value returned from the %JBNO routine.

The scratch file will be opened in output mode and the resulting Toolkit channel will be promoted to global status.

If *channel* is not passed, the default file stack will be initialized. If the default file stack has already been initialized, a fatal error occurs.

The scratch file will be opened with the TEMPFILE OPEN qualifier to ensure a unique scratch file for each call to the FS_INIT subroutine. The DTKTMP environment variable can be assigned to indicate where the file should be created. The file will be only created when the internal memory cache overflows.

The default file-stack cache size can be specified with the environment variable DTKFSWINSIZ. This size specifies the amount of memory allocated by the file-stack if no larger size is specified to FS_INIT, nor is subsequently needed by the file-stack routines. Any value specified will be rounded up to the nearest multiple of 16,384 but not more than 114,688. Increasing the size may result in improved performance for applications that traverse a large number of records on the file stack (for example, large lists).

Examples

The following example will initialize a file-stack system by creating a scratch file named **pages.scr**, and will return the virtual channel number in **pgchn**.

```
xcall fs_init("pages.scr", pgchn)
```

FS_OPEN – Add a logical file to the top of the stack



```
xcall FS_OPEN(record_size[, channel])
```

Arguments

record_size

The size of the fixed-length records to be stored in the new logical file. **(n)**

channel

(optional) The file-stack channel. If not passed, the default file-stack system is used. **(n)**

Discussion

The FS_OPEN subroutine pushes the current logical file on the file stack and opens a new one.

When a new logical file is opened on the stack, all I/O at that stack level will use data of the size specified in the FS_OPEN call, regardless of the size of the field that is passed. If a shorter field is used, subsequent data will be accessed.

If *channel* is not passed and there is no default file stack, or if *channel* is zero, Toolkit automatically calls FS_INIT to activate a new file stack.

Examples

The following example will add a new logical file to the file stack whose channel is stored in **pgchn**. The size of the records for this new logical file will be 512 bytes.

```
xcall fs_open(512, pgchn)
```

FS_PARAM – Get a logical file’s parameters

WT	WN	U	V

```
xcall FS_PARAM ([rtrn_level], [record_size], [last_rec], [rtrn_channel], [which_level],
&               [which_channel][, error])
```

Arguments

rtrn_level

(optional) Returned with the file-stack level. **(n)**

record_size

(optional) Returned with the record size of the logical file. **(n)**

last_rec

(optional) Returned with the last record written to the logical file. **(n)**

rtrn_channel

(optional) Returned with the file-stack channel. **(n)**

which_level

(optional) The file-stack level to access. **(n)**

which_channel

(optional) The file-stack channel to use. **(n)**

error

(optional) Returned with one of the following error flags: **(n)**

- 0** No error occurred.
- 1** Not a file-stack channel.
- 2** File stack not active.
- 4** No logical file stacked.
- 6** Invalid file-stack level.

Discussion

The FS_PARAM subroutine returns information about a given logical file for a given file-stack system.

If *which_level* is not passed or is passed and zero, the current (highest) logical file (level) will be used. If *which_level* is not specified and no logical file is currently active, *rtrn_level* will be set to zero and *record_size* and *last_rec* will be undefined. If *which_channel* is not passed, the default file stack will be used.

Examples

This example returns the current (highest) file-stack level in **level**, the record size of the current logical file in **resize**, the last record written in **lastrec**, and any error in **sts**.

```
xcall fs_param(level, resize, lastrec,,, sts)
```

FS_QUIT – Terminate a file-stack system



```
xcall FS_QUIT([channel])
```

Arguments

channel

(optional) The file-stack channel. If not passed, the default file-stack channel is used. **(n)**

Discussion

The FS_QUIT subroutine terminates the given file-stack system.

Examples

The following example terminates the file-stack system whose channel is stored in **pgchn**.

```
xcall fs_quit(pgchn)
```

FS_READ – Read an existing record from a logical file

WT	WN	U	V

```
xcall FS_READ (record, number, [level], [channel][, error])
```

Arguments

record

Returned with the record read. (**a**)

number

The number of the record to read. (**n**)

level

(optional) The file-stack level from which to read. (**n**)

channel

(optional) The file-stack channel. If not passed, the default file-stack channel is used. (**n**)

error

(optional) Returned with one of the following error flags: (**n**)

- 0** No error occurred.
- 1** Not a file-stack channel.
- 2** File stack not active.
- 4** No logical file stacked.
- 5** Illegal record number.
- 6** Invalid file-stack level.

Discussion

The FS_READ subroutine reads a record that exists in a logical file.

If *error* is not passed and an error occurs, a fatal program error is generated.

It is assumed that the size of the data transfer is the size of the accessed logical file's record length, as specified in FS_OPEN.

Examples

This example reads the data stored in record 3 of the current level of the default file-stack into **page**. Any error will be returned in **sts**.

```
xcall fs_read(page, 3, , sts)
```


FS_WRITE – Write a record to a logical file



```
xcall FS_WRITE (record, number, [level], [channel][, error])
```

Arguments

record

The record to write. (**a**)

number

The number of the record to write. (**n**)

level

(optional) The file-stack level to which to write. (**n**)

channel

(optional) The file-stack channel. If not passed, the default file-stack channel is used. (**n**)

error

(optional) Returned with one of the following error flags: (**n**)

- 0** No error occurred.
- 1** Not a file-stack channel.
- 2** File stack not active.
- 4** No logical file stacked.
- 5** Illegal record number.
- 6** Invalid file-stack level.

Discussion

The FS_WRITE subroutine writes a record to a logical file.

It is assumed that the size of the data transfer is the size of the accessed logical file's record length, as specified in FS_OPEN.

If the current (highest) level is being accessed, and number is greater than the highest record number written to this level, a new EOF point will be set for the level.

Only the current (highest) level can extend the logical file length. Lower levels (logical files) can only update existing records.

If *error* is not passed, and an error occurs, a fatal program error is generated.

Examples

The following example writes the data in **page** to record number 5 in file-stack level **page_level**. If **page_level** is not the current (highest) level, record number 5 must be an existing record, since you can only extend the logical file length of the highest file-stack level.

```
xcall fs_write(page, 5, page_level,, sts)
```

F

Linked-List Processing

Using Linked Lists F-2

Discusses how to use linked lists, the advantages of using them, and gives the specifications for the linked list subroutines.

LL_CLOSE – Close a linked list on the file stack.....	F-3
LL_OPEN – Open a linked list on the file stack	F-4
%LL_PROCESS – Process a linked list.....	F-5
LL_SYNCH – Synchronize duplicate linked-list control structures	F-8

Using Linked Lists

The linked list subroutines enable you to maintain a linked list on the file stack. Because they use the file stack routines (which have a built-in cache), these routines are ideal if you need a mechanism for managing an ordered linked list of items of the same size without a restriction on the number of items, and with automatic paging to disk as needed. (See “[Appendix E: File-Stack Processing](#)” for information on the file stack routines.) With the linked list subroutines you can insert or delete records anywhere in the list, and you can retrieve the first, last, previous, next or previously marked record. You can also update the current record and swap the position of adjacent records.

You pass these routines a control structure, defined in **lnklst.def**, to indicate which linked list on which file stack you are accessing, as well as the current state of that list. The data in the control structure is read-only to calling routines.

[LL_CLOSE](#) – Close a linked list on the file stack

[LL_OPEN](#) – Open a linked list on the file stack

[%LL_PROCESS](#) – Process a linked list

[LL_SYNCH](#) – Synchronize duplicate linked-list control structures

LL_CLOSE – Close a linked list on the file stack



```
xcall LL_CLOSE(control)
```

Arguments

control

The linked list control structure. (a)

Discussion

The LL_CLOSE subroutine closes a linked list on the file stack.

This subroutine will close the specified linked list, but the file-stack level (logical file) will only be closed if there are no higher levels active.

Examples

The following example closes the linked list identified by **lnkctl**.

```
xcall ll_close(lnkctl)
```

LL_OPEN – Open a linked list on the file stack



```
xcall LL_OPEN(control, record_size[, channel])
```

Arguments

control

Returned with the linked-list control structure. **(a)**

record_size

The size of the linked-list records. **(n)**

channel

(optional) The file-stack channel to use. **(n)**

Discussion

The LL_OPEN subroutine opens a new linked list on a new file-stack level.

If *channel* is not passed, the default file stack will be used. If *channel* is not passed and there is no default file stack, or if *channel* is zero, a new file stack will be activated.

D_LL_CTLSIZE (defined in **tools.def**) should be used in declaring a control structure for the linked-list subroutine. Alternatively, use the structure defined in **lnklst.def**.

Examples

The following example will create a linked list on a new level of the default file stack. The size of the linked-list records will be 50, and the control structure will be returned in **lnkctl**.

```
xcall ll_open(lnkctl, 50)
```

%LL_PROCESS – Process a linked list



rec = %LL_PROCESS (*control*, *operation*, *data*, [*error*][, *mark*])

or

xcall LL_PROCESS (*control*, *operation*, *data*, [*error*][, *mark*])

Return value

rec

For operations other than D_LL_DELETE, this function returns the internal record number of the current record if the operation is successful. For D_LL_DELETE, this function returns –1 if successful. If unsuccessful, %LL_PROCESS returns 0. (^VAL)

Arguments

control

The linked list control structure. (**a**)

operation

The operation to perform: (**n**)

D_LL_FIRST	Retrieve first record.
D_LL_LAST	Retrieve last record.
D_LL_PREV	Retrieve previous record.
D_LL_NEXT	Retrieve next record.
D_LL_CURR	Read current record.
D_LL_WRITE	Rewrite current record.
D_LL_INSERT	Insert before current record.
D_LL_APPEND	Insert after current record.
D_LL_DELETE	Delete current record.
D_LL_SWAPPREV	Swap current and previous record.
D_LL_SWAPNEXT	Swap current and next record.
D_LL_MARK	Mark the current record.
D_LL_JUMP	Retrieve previously marked record.

data

The data for the operation. (**a**)

error

(optional) Returned with the error status. If not passed all errors are fatal. (**n**)

D_LL_NOCURR No current record.

D_LL_NOTFND Requested record not found.

D_LL_FSERR File-stack error.

D_LL_NOTTOP Not at top level of file stack.

mark

(optional) The internal record number of the record you want to retrieve (for **D_LL_JUMP** only; all other operations ignore *mark*). (**n**)

Discussion

The **%LL_PROCESS** function processes a linked list on the file stack.

The *data* argument is returned with the data for the current record when using **D_LL_JUMP** and the **D_LL_FIRST** through **D_LL_CURR** operations. For **D_LL_WRITE** through **D_LL_SWAPNEXT** (except **D_LL_DELETE**), the data passed as *data* is written to the record. The *data* argument is ignored when using the **D_LL_MARK** operation.

After a **D_LL_DELETE** operation, the content of *data* is undefined and there is no current record (*rec* returns -1), but the next and previous pointers remain unchanged.

After a **D_LL_SWAPPREV** or **D_LL_SWAPNEXT** operation, the next and previous pointers are changed, but the current record pointer remains unchanged. If there is no previous record for a **D_LL_SWAPPREV** operation, or no next record for a **D_LL_SWAPNEXT** operation, an error (**D_LL_NOTFND**) occurs.

D_LL_MARK marks the current record for a subsequent **D_LL_JUMP** operation. If there is no current record, this operation will cause an error (**D_LL_NOCURR**).

D_LL_JUMP retrieves either the record specified by *mark* or the last record marked by **D_LL_MARK**. The last record specified by **D_LL_MARK** is retrieved only if *mark* is not passed or is passed as 0. Note the following:

- If **D_LL_JUMP** is successful, the retrieved record becomes the new current record. *Rec* is returned with the internal record number for that record, *data* is returned with the data for the record, and *error*, if passed, is returned with 0.

- ▶ If no record has been marked by a previous call to `D_LL_MARK`, and *mark* is not passed or is passed as 0, `D_LL_JUMP` fails, and *error* (if passed) is returned with `D_LL_NOTFND`.
- ▶ If *mark* is passed with a non-zero value (including -1, the return value for `D_LL_DELETE`) and the specified record does not exist, `D_LL_JUMP` fails, and *error* (if passed) is returned with `D_LL_NOTFND`.

Examples

The following example inserts a new record into the linked list. If the operation is successful, the record is inserted before the current record, the new record contains the data passed in **new_rec**, and the internal number for the record is returned in **rec_num**. If an error occurs, the error status is returned in **sts**.

```
rec_num = %ll_process(lnkctl, D_LL_INSERT, new_rec, sts)
```

The next example reads the first record in the linked list identified by **lnkctl** and returns the data for the first record in **list_rec**. If the operation is successful, **rec_num** is returned with the internal record number of the first record in the linked list. If an error occurs, the error status is returned in **sts**.

```
rec_num = %ll_process(lnkctl, D_LL_FIRST, list_rec, sts)
```

The next example retrieves the record specified by **rec_num**. If the operation is successful, the data for the record is returned in **list_rec**.

```
xcall ll_process(lnkctl, D_LL_JUMP, list_rec, sts, rec_num)
```

LL_SYNC – Synchronize duplicate linked-list control structures

WT	WN	U	V

```
xcall LL_SYNC (mentor, protege[, deleted])
```

Arguments

mentor

The linked-list control structure that contains the most up-to-date information. **(a)**

protege

Another linked-list control structure for the same linked list. **(a)**

deleted

(optional) Returned true if the protege's current record has been deleted. **(n)**

Discussion

The LL_SYNC subroutine synchronizes two control structures for the same linked list.

It is possible to maintain more than one context in a linked list simply by copying the control structure and using one or the other copy in calls to %LL_PROCESS. The only time this can cause a problem is if records are inserted, appended, deleted, or moved in the list. In any of these cases, the secondary control structure will become outdated and will need to be updated.

This subroutine updates one control structure from another. The *mentor* control structure is assumed to possess the most up-to-date information about the linked list, and the *protege* control structure will learn from it, while maintaining its own current context in the linked list.

If the control structures do not refer to the same linked list, no action will be taken.

If *protege*'s current item has been deleted, *deleted* may optionally be returned true. In this case, the previous and next records will also be checked for deletion, and if they are deleted their pointers will be erased. It is therefore recommended that you re-sync control structures after each such operation, or else discard the *protege* control structure altogether.

Any "marked" record (through D_LL_MARK passed to %LL_PROCESS) will also be checked for deletion. If it has been deleted, there will be no marked record for the *protege* control structure.

Examples

The example below updates the information in **lnkctl2** from the information in **lnkctl1**, and returns **delflg** set to true if the current item in **lnkctl2** has been deleted.

```
xcall ll_sync(lnkctl1, lnkctl2, delflg)
```

Glossary

A

arrive method

For an input field, the method (subroutine) that Toolkit calls before processing the field. For a list, a method (subroutine) that Toolkit calls when highlighting an item.

application container

The outermost window for a Toolkit application. Along with the application title bar (header), frame, menu bar, and footer, the application window owns all windows created for the application, including the application window, the information line, and any toolbars for the application. When the application container is moved or resized, the application window is automatically moved or resized.

application window

The window that owns all Synergy windows and lists created for the application. It is the area below the title bar and menu bar, above the information line, and inside of the outermost window border for the application and any toolbars for the application.

B

body

The portion of the display screen between the menu bar and the information line. The screen body will contain all the application's placed windows.

C

callback routine

An external routine that, when called, appropriates all processing control, including error processing. The calling program regains control only when the callback routine explicitly abdicates it. Note that callback routines are called asynchronously, which means that they can be called between statements in another routine.

caption

A window's caption is synonymous with its title. In a Windows environment, the application as a whole also has a caption, which can be modified with E_SECT.

cascading menu

Another name for a submenu. *Cascading* describes the effect that submenus create on your screen.

change method	The method (subroutine) that Toolkit calls after validating an input field.
child list	A list in a composite window. Child lists can be either ActiveX Toolkit lists or standard Toolkit lists.
child window	A window within a composite window. Child windows can be any type of Toolkit window: an input window, an ActiveX container, another composite window, and so forth.
client area	The visible area within a window. This area does not include scroll bars, the information line, the footer, the title bar (header), or menu bars.
color palette	There are up to 16 combinations of foreground/background color that can be assigned to any Toolkit rendition. The entire set is called a palette, and an individual combination is called a palette code. You can change the colors assigned to a given palette code by setting the WND environment variable, or through the U_REND, U_MODREND, or U_EDITREND subroutines.
container	A window that contains another window or control. For example, a container for a button is the window (e.g., a tab set) that contains the button.
Composer	An interactive tool (on Windows) for creating and modifying window scripts.
composite container window	The parent window for a composite window.
composite window	A window that consists of a parent window (a composite container window) and child windows and lists, but functions and appears to the user as a single window.
configuration field	A field you can modify within your program to change operational characteristics of UI Toolkit.
context	The next field that will be processed in an input set. You can change the context with the I_NEXT subroutine.
D	
data control commands	The commands that affect the position and display of text within text (general) windows and input windows.
debugger	A UI Toolkit utility that enables you to look into the windowing environment and evaluate the state of the Toolkit system.

demand-loading	A method used by the list routines to load list data. List items are loaded as needed for display purposes, rather than all at once at the beginning. This can significantly improve the initial performance of processing a large list.
disabled	Refers to a menu column or entry that the user cannot currently select. See also enabled .
dispatch	To send an event to the routine that is appropriate for handling it. Typically, this involves calling a method (subroutine), or returning the event to the parent routine.
display	The display of a program consists of the menu bar and screen body. It also owns menu column and window placement (although the menu columns and windows themselves belong to the environment).
display area	The part of a window that shows on the screen. It is a subset of the total window area (and can be equal to the total window area). See also view .
display attributes	The attributes of each display cell on the screen can be any combination of reverse video, blinking, highlighted (bold), and underlined.
display length	The maximum number of characters that can be displayed in a field. This includes any formatting characters and the negative sign if negative numbers are allowed for the field.
display rendition	A combination of the display attributes and color of the screen. Each screen cell has its own display rendition.
display screen	The screen layout used by Toolkit. The screen is divided into five sections: header, menu bar, body, information line, and footer.
drag bar	In a Windows environment, if DRAGBAR is specified as ON (using the .BORDER command), the window's title bar functions also as a drag bar. The user can move such a window by clicking on the window's title (or caption) bar and, while continuing to press down on the mouse button, dragging the window to a new location within the application.
dtkmap.ism	See dtkmap.ism .
dtkrnd.ism	A library that contains the renditions for each defined rendition scheme. Each rendition scheme is stored as a record in dtkrnd.ism . See also rendition scheme .

E

EHELP_METHOD

A programmer-defined subroutine that Toolkit calls when the user selects a menu entry whose internal name is “O_HELP;” if the method was registered with E_METHOD.

empty field

A field that contains absolutely no data (not even blanks for alpha fields or zeros for decimal or implied-decimal fields). An empty field is also a field that has been neither displayed nor entered since it was last initialized. See also [full field](#).

enabled

Refers to a menu column or entry that the user is able to select. See also [disabled](#).

encapsulation

A program design philosophy in which a routine is called to perform a specified function, but how it performs that function need not be known to the caller. Private data and methods for the encapsulated routine are not accessible to other routines. Also referred to as “black-box processing”. See also [privatized](#).

endpoints

Special optional characters displayed at either end of a scroll bar on a UNIX or OpenVMS system, which may optionally indicate whether or not further progress in that direction is possible.

entry

A possible selection in a menu.

environment

A program state that consists of the current definition of the screen, any terminal settings and processing options, and open channels. When you create (enter) a new environment, UI Toolkit saves the old environment and copies its attributes to the current one. Any change to an environment item supersedes the previous contents.

environment level

Each creation of a new environment. Environments are nested as you go from one to another. A program can have up to 32 environment levels with 16 being the default. An application moves from one environment to another with the E_ENTER subroutine, and leaves an environment level using the E_EXIT subroutine. Upon exit, the current environment is deleted, including all of its local items, and the previous environment is restored. See also [shared display](#).

EUTILS_METHOD

A programmer-defined subroutine that Toolkit calls when the user selects a menu entry whose internal name begins with “U_” if the method was registered with E_METHOD. It must dispatch the menu entry to the appropriate method. See also [dispatch](#).

event	An occurrence to which a routine must respond. Events can be user-generated (for example, by selecting a menu entry) or program-driven (for example, a list needs to load a new item). Events can be dispatched by Toolkit in two ways: (1) They can be returned to the caller for dispatching (for example, menu selections); or (2) They can be dispatched automatically to a specified method.
event-driven programming	A programming style that allows Toolkit to control your application after you have called a Toolkit subroutine. After the UI Toolkit subroutine has finished its task, it returns control to your program, potentially signaling that an event has occurred that your program must now dispatch. Also known as “event-style programming”. See also dispatch and encapsulation .
F	
fatal utility errors	A problem that results in immediate, unconditional termination of your program, unless trapped by the FATAL subroutine.
file stack	A single physical file that contains multiple logical scratch files accessed in a stack manner. Only the most recently opened logical file can be extended, though prior files can be accessed. Logical files contain only fixed-length records. The file stack is managed by the file-stack routines (FS_xxx).
file-stack level	Identifies a logical file within a file stack.
fixed font	A font in which every character has the same width. Also known as a monospaced font. See also proportional font .
font	A design for a set of characters, comprised of typeface, point size, width, and spacing. For example, within the Helvetica typeface there are many different fonts: 10-point italic, 12-point bold, and so forth.
font handle	An identifier associated with a font. Be aware that this identifier is not a memory handle, and, therefore, the font handle’s value can coincidentally be the same as an existing memory handle.
font palette	A group of font entries, each associating a font name with a typeface, point size, and sizing character combination. You can change a font palette entry by explicitly modifying it in a Synergy initialization file, modifying it with Composer, or by calling the %U_WNDFONT function.

footer An optional trailer for the UI Toolkit screen layout. The footer can be up to four lines long. You can place information into the footer with the E_SECT subroutine.

force buffer The location to which pending input is copied if the user selects a menu entry (for example, a search function) before pressing RETURN to enter the data. You must use the I_FORCE subroutine to get that data from the force buffer into your program.

full field A field that contains data. Note that a field can be full of blanks and still be considered full. A full field has been either displayed or entered since its last initialization. See also [empty field](#).

G

general window A basic window that is not an input window, selection window, list, or menu column. General windows are usually used to convey information, such as error or information messages. Text windows and Message windows are general windows. See also [text window](#).

global Refers to an element that is owned by the entire environment system rather than by a specific environment level. Global elements can be defined or deleted at any environment level, but they are not automatically deleted when exiting an environment. Global elements include Synergy channels, columns, windows, and lists. See also [local](#).

H

header An optional banner for the UI Toolkit screen layout. The header can be up to four lines long. You can place information into the header with the E_SECT subroutine.

hot entry A routine that is not tied to any particular program and is accessible from anywhere in your application. Hot-entry processing is performed by the EHELP_METHOD and EUTILS_METHOD subroutines, which you write yourself but which are called by one of the UI Toolkit general input routines (such as I_INPUT, I_INPFLD, T_EDIT, or T_VIEW).

I

include file	A file containing data definitions that is included by a routine using the <code>.INCLUDE</code> compiler directive. Also called a definition file.
information line	A single line at the bottom of the screen body (above the footer) that is used by Toolkit to display messages and general information.
inheritance	When one class of object is derived from another, it may retain characteristics (properties) of its parent class. This is termed inheritance. In Toolkit, fields derived from the Repository inherit all the properties defined therein, but may overload some of them.
input field	A window field associated with a set of characteristics that define how terminal input and display are to occur.
input length	The maximum number of characters that can be entered into an input field.
input set	A list of one or more fields in an input window. An input set defines the default order for processing fields.
input window	A window that can contain text, input fields, and buttons.

K

key map	A record in the key map file that defines the correspondence between the function keys and other special keys on the user's keyboard and the functions they perform. A user's terminal type is stored in the logical DTKTERM (or TERM), which UI Toolkit uses as the key to locate the appropriate record in the key map file. When the window environment is initialized, Toolkit loads the key map defined in DTKTERM (or TERM).
key map file	A file (<code>dtkmap.ism</code> by default) that associates keys (escape sequences) with internal function codes to create shortcuts.
key mapping script file	A file (<code>keymap.ctl</code> by default) that contains function codes for shortcuts for a given environment or terminal type. When you generate a window library, shortcuts defined in the key mapping script file are combined with shortcuts specified with <code>.ENTRY</code> .
keymap.ctl	See key mapping script file .

L

leading	The spacing between lines of text.
leave method	For an input field, the method (subroutine) that Toolkit calls after processing the field. For a list, the method (subroutine) that Toolkit calls when it is preparing to change which item is highlighted.
linked list	A list of records in a logical file within a file stack. Linked lists are maintained with the linked-list routines (LL_xxx). Records can be inserted or deleted at any point, and the ordering of logically adjacent records can be interchanged. First, last, previous, next, and previously marked records can be retrieved.
list	A set of data and windows managed by the list-processing routines (L_xxx) to maintain a visible list of items. List items are loaded on demand. New items can be inserted at any point, and existing items can be deleted. Adjacent items can be interchanged. The list pointer can be positioned up or down one item, one page of items, to the top or bottom of the list or of the current page. The horizontal display area can be modified, and is automatically tied to the display area of the list's input window. Searching for a specific alphanumeric string is also supported. See also list class and null list .
list class	A set of specifications for creation of a list. These include default placement, number of visible items, headers, footers, whether or not the list may be null, and other options. You can define list classes in scripts and store them in window libraries (with the .LISTCLASS command) or you can create them at runtime (with the L_CREATE subroutine).
load method	The method (subroutine) that the UI Toolkit's list-processing routine calls when it needs to load items in a list.
local	Refers to an element owned by the defining environment level. Only the defining environment level can delete a local element. Local items include channels, columns, and windows. All local columns and windows are deleted and all local channels are purged when an environment level is exited. See also global .

logging	A way to associate Synergy channels and window IDs with the internal control structures of Toolkit. All channels and windows are automatically logged if they are opened or loaded with a Toolkit subroutine. If a program creates, loads or restores a window without using the U_LDWND, M_LDCOL, or I_LDINP subroutine, the window ID must be logged into the utility system by the U_LOGWND subroutine. When the window is logged, it is marked as global or local. If it is local, the current environment is the defining environment level.
logical file	A set of fixed-length records grouped together and maintained in a file stack. A logical file is associated with a single level.
M	
margin	The number of spaces from the left and right edge of the window that will not be occupied by text.
menu	A menu consists of menu columns associated with the menu bar.
menu bar	A one-line bar across the screen that lists the menu columns available to the current environment level. The menu bar follows the optional header and is part of the display screen. See also process-menu key .
menu column	A window that contains entries which a user can select. There are two kinds of menu columns: primary and submenu. A primary menu column is a drop-down list of selections from the menu bar. A submenu column is a drop-down list of selections adjacent to a primary menu entry or another submenu entry. Both types of menu columns are defined in a script file and are generated either by the Script or Proto programs, or at runtime. The terms “menu column” and “column” are used interchangeably. See also entry , Script , Proto .
method	A subroutine or function that you can write, but that is called by Toolkit. There are event methods (arrive methods, leave methods, etc.), composite window processing methods, environment methods, and so forth.
modal	A term used in Windows environments to describe functions, routines, or applications that suspend all other processing until that function (routine or application) is done processing. Application-modal means that other processes <i>within</i> that application are unavailable until the modal process is complete; however, the user can use ALT+TAB or click on another application to move and do processing within other applications.

N

non-window data

In list processing, the data buffer containing additional information that you don't want to display in the list. Non-window data can be supplemental data that needs to be stored and retrieved for each list item. Although this data may duplicate input window data, it will be stored and retrieved in the format passed to the list processor, whereas the input window data is stored and retrieved as displayed.

null list

A list with no items loaded into it, or a list that has had its last item deleted. You can prohibit null lists in a list class. See also [list class](#).

numeric fields

A **d** (decimal) or **i** (integer) input field.

O

occluded window

A window that is covered, in whole or in part, by another window. No terminal input can be performed within the covered area of the window. All other processing of the windows remains unaffected by the occlusion.

overloading

Redefining the properties or methods for an object. In Toolkit, the properties of fields derived from the Repository can be redefined, or overloaded, in the script file (or by `IB_FIELD` or `I_FLDMOD`). The general help method supplied by UI Toolkit (`TKP_HELP`) can be overloaded by registering your own version with `E_METHOD`.

P

point size

The height of characters in a font. A point equals 1/72 of an inch.

pristine state

The initial state of the input window display data and the associated data field upon creation of a list (in other words, as passed to `L_CREATE`). This state is re-created each time the list processor requests load of an item, and is useful for loading initial values (prompts, static data).

privatized

Reserved for the private use of a graphic control and unavailable for public use by the calling routines.

process-menu key

The key or key sequence that the user presses to activate a menu. This key is a toggle that also deactivates the menu. As Toolkit is distributed, the process-menu key is `ALT` on Windows and `CTRL+P` on all other systems.

proportional font	A font in which different characters have different widths. See also fixed font .
Proto	A menu-driven program that creates windows from a script file, displays them on the screen, and stores them in a window library.
pseudo array	A single-dimension array of type a , d , d. , i1 , i2 , i4 , i8 , p , or p. (for example, 10a5).
Q	
quick-select character	A single character that accesses a menu, column entry, or selection window item. The default quick-select character is the first non-blank character of each entry. Quick-select characters are not case sensitive and can be defined in the menu column or selection window script file.
R	
ratio of current to maximum	The relationship between what is currently displayed in a window and the total possible.
redraw key	The key or key sequence that the user presses to redraw the screen. As Toolkit is distributed, the redraw key sequence is CTRL+R.
rendition file	See dtkrnd.ism .
rendition scheme	A record in dtkrnd.ism that defines the display attributes and colors for your screen. Each rendition scheme has a unique name that UI Toolkit uses as the key to locate the appropriate record in dtkrnd.ism . See also dtkrnd.ism and working rendition scheme .
repository	Where data definitions are stored.
Repository	The application that orders and defines your data structures, files, and attributes.
routine	A main routine, subroutine, or function.
S	
scriidl	A utility that converts and moves field information from existing script files into the Repository.
Script	A program that creates windows or columns from a script file and saves them in a window library. See also window libraries .

script file	A standard text file that contains special commands for building windows.
scroll bar	A one-dimensional window that is used to represent progress or regress by moving an indicator from one end of the window to the other. Scroll bars can optionally contain endpoints, which may indicate whether or not further progress in the given direction is possible. See also endpoints .
selection entry/field/item	The item selected from a list or menu. The terms “selection entry,” “selection field,” and “selection item” are all synonymous.
selection window	A window with a choice of one or more items that can be selected (usually with arrow keys).
shared display	A display that is accessible by more than one environment level. Any changes made to the display in one environment level are reflected in all other environment levels that share that display. See also environment level .
shortcut	A key or key sequence that is associated with a specific menu entry. It provides a quick way to access a menu item, bypassing the standard menu entry selection method. Shortcuts, except for arrow keys, can be processed at any time when the program is requesting input. During menu processing, the arrow keys are disabled as shortcuts and are used to move between menu and column entries. On Windows, some keys are privatized by certain controls (input fields, lists, and so forth), and are thus not available as shortcuts when those controls are being processed. See also privatized.
sizing character	A single character that determines the cell size within a visual object for the purposes of positioning. For example, when you specify the sizing character to be “W”, the cell size is significantly wider than with a sizing character of “i”. Additionally, the sizing character is case sensitive, so a lowercase “w” sizing character produces a smaller cell size than an uppercase “W” sizing character.
state flags	Flags marking the state of the environment. These flags can be set using the E_STATE subroutine.
storage size	The size of the data area for a field (in bytes).

structure	Defines the relative positions of fields within a data area (for example, a record). Structures for input fields can be drawn from the Repository or defined in a window script or with the IB_STRUCTURE subroutine.
submenu column	See menu column .
Synergy UI Toolkit Control Panel	A program (synctl.dbr) that enables you to customize many aspects of Toolkit, including message text, renditions, and keymapping.
T	
tab set	A group of related windows, rendered in the metaphor of tabbed file folders. The tab set processing functions in UI Toolkit enable you to associate input windows or lists with the “tabs” of a tabbed dialog. Specific input window or list processing is handled using a method that you register with Toolkit.
text entry	An entry in a window library that contains only compressed text.
text field	A multi-element alpha field that contains editable text.
text window	A special window that contains variable-length, editable text. The user can change the “view” of a text window by scrolling through it with a view-control menu column. See also general window .
tkctl.def	A file that contains runtime modifiable fields that affect the operational characteristics of Toolkit.
toolbar	A variable-size rectangle containing buttons, which are selected by clicking the mouse on the button face. The toolbar functions enable you to create and manipulate an application window toolbar. Toolbars provide an alternative method for invoking functions within an application; they are used as extensions to or replacements for the menu bar.
tools.def	A file used by Toolkit and your routines. It contains general definitions and is a required include file for programs that use the UI Toolkit subroutines. Do not modify tools.def .
true/false flags	A numeric value passed or returned to the routine. If the flag is zero (0), it is false. If the flag is any value other than zero, it is true.
typeface	A design for a set of characters which defines the shape of each character. For example, Helvetica and Times Roman are typefaces. The two general categories of typeface are serif and sans serif.

U

user data set A one-dimensional alpha array that is associated with a window (rather than your program) and is maintained within the window data area by the window system. Its parameters and contents are completely defined by you. The data is not displayed.

V

view The portion of a text or input window that is visible to the user. See also [display area](#).

view length A setting that determines the width of a field.

W

window field A portion of a window defined by row, column, length, and contents; and whose contents are visible.

window libraries A Synergy ISAM file with a specific format. Windows are saved to and restored from window libraries by Toolkit.

working rendition scheme The scheme stored with a blank key in a rendition file. It is used as the default scheme when a scheme is not specified in one of the rendition subroutine calls. See also [rendition scheme](#).

Index

A

a_inpininfo group 8-46, 8-114, D-4

abort

- button (message box) 16-102

- messages, size of A-3

- processing 16-5 to 16-7

“About” window 16-8 to 16-9

activation characters. *See* quick-select character

ActiveCellHighlightStyle published property 9-17

ActiveX API 4-3

- See also* Synergy DBL Language Reference Manual

ActiveX containers 4-4 to 4-6

- creating 4-28 to 4-30, 4-31

- for Spread control 4-36 to 4-38

- getting IDs for 4-20, 4-31

- sizing 4-60

- types 4-4

ActiveX controls 4-3 to 4-10

- composite window method for 6-47 to 6-48

- containers for. *See* ActiveX containers

- error handling for 4-8

- IDs, getting for 4-21

- input to 4-12 to 4-13

- methods

- mouse events, for 4-39 to 4-61

- native, calling 4-17 to 4-19

- properties for

- getting 4-22

- getting as integers 4-24 to 4-25

- setting 4-26

- registering 4-3, 4-9, 4-15, 4-30, 4-37

- registering keys for 4-33 to 4-35

- routines for 4-11 to 4-35

- Spread control 4-6, 4-36 to 4-38

ActiveX Diagnostic Utility 4-9

ActiveX Toolkit lists 4-6 to 4-10, 9-4

- border for 9-17

- cells in. *See* cells in ActiveX Toolkit lists

- color and 2-84, 2-86, 9-20

- columns 9-126, 9-139

- defining 2-84, 9-5

- events for 9-50

- fonts used for 9-20

- input limitations for 9-103

- methods for 9-22 to 9-49

- notes and tips for 9-15 to 9-16

- properties for 9-17 to 9-21

- renditions for 2-45, 2-63, 2-84, 8-31

- retrieving list IDs for 9-139

- Toolkit lists compared to 9-4

- See also* child windows and lists; lists

ACTIVEX_LIST environment variable 9-5

addresses

- ActiveX extension routines, for 4-37

- methods, for 7-8, 7-20, 7-36, 16-211

aliases 2-91 to 2-92, 8-107 to 8-108

Alink names, topics for 16-73 to 16-74

allow lists 2-33, 2-36, 8-95

ALLOW qualifier (.FIELD) 2-36

alpha fields

- changing type 8-29

- dimension. *See* dimension (alpha fields)

- justification of 2-55, 8-27, 8-97

- leading for multi-line 8-5

- matching case and characters 2-56

- size of, maximum 2-33

- See also* input fields

ALT_FONT_HEIGHT environment variable 18-43

ALT_FONT_WIDTH environment variable 18-43

ALT_HEIGHT setting 18-48

ALT_TYPE_FACE setting 18-43, 18-48

ALT_WIDTH setting 18-48

ALT+key activation

- for buttons 5-6 to 5-7

- for composite windows 6-7

- for lists on Windows C-1 to C-6

alternate font 16-214, 18-42

- header, footer, or information lines 7-6

- input fields 8-31, 8-100

- prompts 8-31, 8-100
 - switching to and from 16-215
- ALTERNATE palette entry 18-43, 18-44, 18-48
- ambient properties 4-4
- ANSI color codes 18-39
- application close method 7-59 to 7-61, 16-46
- application container
 - leading for 16-183
 - row height for 16-182
 - sizing character for 16-173, 16-174
- application keypad mode, setting 3-4
- application move method 7-46
- application resize method 7-47
- application scaling percentage 16-225, 16-230
- application state processing method 7-49
- application window
 - header and footer. *See* header and footer for application window
 - height 7-47, 16-130, 16-137
 - leading for, retrieving 16-183
 - row height, retrieving 16-182
 - settings, saving to initialization file 16-130
 - sizing character for, retrieving size of 16-173, 16-174
 - width 7-47, 16-130, 16-137
 - See also* dialog boxes and screens; screen
- applications
 - close methods for 7-59 to 7-61, 16-46
 - closing 16-46
 - customizing 18-1 to 18-50
 - development of, overview 1-14 to 1-15
 - examples of 1-2
 - font for 18-42 to 18-45
 - information about, displaying 16-8 to 16-9
 - linking on OpenVMS 1-14
 - move methods for 7-46
 - moving to other systems 1-23
 - requirements and considerations for 1-16 to 1-18
 - resizing 1-20, 7-47, 16-122, 16-230
 - start up processing for 16-236
 - state processing methods for 7-49
- arrive methods. *See* input methods; list methods
- ARRIVE_METHOD qualifier
 - .FIELD 2-37
 - .LISTCLASS 2-82
- arrow button (for combo boxes) 2-65
- arrow keys
 - and lists 9-17, 9-123, C-2
 - and menus 2-29, 10-2, 10-5
- ArrowKeyAction published property 9-17
- ASCII characters, retrieving for key codes 16-169
- ASCII values
 - retrieving for typed characters 9-128
 - using for key maps 18-16
- assemblies (.NET) 11-3
- associative keywords, topics for 16-73 to 16-74
- asynchronous callbacks 1-20
- asynchronous methods D-2
- attribute flags 14-13, 16-119
- attributes. *See* renditions
- automatic font switching, enabling and disabling 16-215
- AUTOMATIC qualifier (.FIELD) 2-37
- autotime fields 2-68, 8-101
- AX_INPUT routine 4-12 to 4-13, 4-33
- AX_REGISTER routine 4-15
- AX_TIMEOUT routine 4-16
- %AX_TKCALL function 4-17 to 4-19
- %AX_TKCID function 4-20
- %AX_TKCTLID function 4-21
- %AX_TKEVENT_CLOSE function 4-40
- %AX_TKEVENT_LEFT_CLICK function 4-41
- %AX_TKEVENT_LEFT_DBLCLK
 - function 4-43 to 4-44
- %AX_TKEVENT_MAXIMIZE function 4-45
- %AX_TKEVENT_MIDDLE_CLICK
 - function 4-46 to 4-47
- %AX_TKEVENT_MIDDLE_DBLCLK
 - function 4-48 to 4-49
- %AX_TKEVENT_MINIMIZE function 4-50
- %AX_TKEVENT_MOVE function 4-51 to 4-52
- %AX_TKEVENT_RESTORE function 4-53
- %AX_TKEVENT_RIGHT_CLICK function 4-54
- %AX_TKEVENT_RIGHT_DBLCLK
 - function 4-56 to 4-57
- %AX_TKEVENT_SCROLL function 4-58 to 4-59
- %AX_TKEVENT_SIZE function 4-60
- AX_TKGET routine 4-22
- %AX_TKGETINT function 4-24 to 4-25
- AX_TKSET routine 4-26
- AX_TKSINGLE subroutine 4-28 to 4-30
- %AX_TKWIN function 4-31
- AX_WANTSKEY routine 4-33 to 4-35
- axlib.elb library 4-3, 4-11

B

- B_BUTTON subroutine 5-4 to 5-13
- B_BUTTONSET subroutine 5-14
- %B_DISABLE function 5-16
- %B_ENABLE function 5-17 to 5-18

- `%B_INFO` function 5-19 to 5-34
- `BackColor` published property 9-17, 9-20
- background color
 - ActiveX lists, for 9-19
 - cells in ActiveX lists, for 2-34, 9-17, 9-23, 9-37
 - help window, for 16-70, 16-118
 - input 16-117
 - palette entries, for
 - returning 16-152, 16-157
 - setting 16-116, 18-39
 - rows in ActiveX lists, for 2-84, 2-86, 9-18, 9-19
 - windows, for 2-73, 16-118
 - See also* color
- backspace key 8-4, 18-22
- bell (terminal) 16-11
 - field input termination 2-58
 - messages and 16-95
 - popup windows and 16-109
 - selection processing and 9-132
 - turning on, off 18-2
 - unmapped keys and 14-20
 - user response, waiting for 16-148
- bitmap graphics 2-15
- `BITMAP` qualifier (`.BUTTON`) 2-14
- black-box processing 1-12 to 1-13
- blank columns, suppressing 2-95, 12-5
- blank input fields 2-38, 2-63, 8-26, 8-96
- blank lines
 - adding between paragraphs 14-11
 - in key mapping files 18-20
 - in script files 2-7
 - trimming from end of windows 2-111
- blank rows in menus, selection windows 2-8, 10-59
- `.BLANK` script command 2-4, 2-8
- `BLANK_IF_ZERO` qualifier (`.FIELD`) 2-38
- `BLANK_IFZERO` qualifier (`.FIELD`) 2-38
- blanks
 - in input fields 2-51
 - in selection window entries 2-77
 - stripping 14-9, 14-12, 14-17
- `BLINK` qualifier
 - `.BORDER` 2-9
 - `.BOX` 2-11
 - `.FIELD` 2-38
 - `.FILL` 2-73
 - `.LINE` 2-79
 - `.TITLE` 2-108
 - `.WFIELD` 2-114
- blinking. *See* renditions
- BMP graphics 2-15
- BOLD** qualifier
 - `.BORDER` 2-9
 - `.BOX` 2-11
 - `.FIELD` 2-38
 - `.FILL` 2-73
 - `.LINE` 2-79
 - `.TITLE` 2-108
 - `.WFIELD` 2-114
- boldface. *See* renditions
- `Border` published property 9-17
- `.BORDER` script command 2-4, 2-9 to 2-10
- borders
 - ActiveX Toolkit list 9-17
 - dialog box 16-32
 - list 9-54, 9-76, 9-86, 9-87
 - tab 13-30
 - window. *See* window borders
 - See also* `.BORDER` script command
- `BOTTOM` qualifier (`.BUTTON_SET`) 2-17
- bounds checking 1-19, 18-3
 - `I_DISPLAY` and 8-13
 - `I_DSPFLD` and 8-17
 - `I_GETFLD` and 8-44
 - `I_INIT` and 8-49
 - `I_INPFLD` and 8-52
 - `I_INPUT` and 8-57
 - `I_PUTFLD` and 8-71
 - `I_USER` and 8-87
 - `IB_FIELD` and 8-99
 - input windows built at runtime and 8-7
 - `L_INPFLD` and 9-99
 - `L_INPUT` and 9-102
 - Synergy `.NET` and 1-21
- `.BOX` script command 2-4, 2-11 to 2-12
- boxes
 - color for 2-11
 - combo. *See* combo boxes
 - dialog. *See* dialog boxes and screens
 - drawing 2-11 to 2-12
 - message 16-102 to 16-105
 - names for, maximum size of A-1
- break processing 8-56, D-5
 - adding to fields 2-39, 8-95, 8-99
 - changing field settings 8-26, 8-29
 - disabling field during 8-12

- enabling field during 8-20
- messages in Proto 3-6
- validation and 7-43
- BREAK qualifier (.FIELD) 2-39
- breakpoints 10-52, 16-11
- buffer overruns, detecting 1-19
- buffers
 - force 8-41 to 8-42
 - non-window data, for lists 9-7
 - status of list 9-142
- .BUTTON script command 2-4, 2-13 to 2-16
- button sets 5-2 to 5-3
 - information on, retrieving 5-19 to 5-34
 - modifying for tab sets 13-17
 - placement of, setting 2-17 to 2-19, 8-90 to 8-91, 9-54
 - See also* buttons; radio buttons; toolbar buttons
- .BUTTON_SET script command 2-4, 2-17 to 2-19
- buttons 5-2 to 5-3
 - adding 5-2
 - to lists at runtime 9-55, 9-56 to 9-57
 - to tab sets at runtime 5-5 to 5-8, 13-15
 - to windows at runtime 5-5 to 5-8, 8-88 to 8-89
 - with window scripts 2-13 to 2-16
 - color for image transparency 2-15
 - default in button set 5-3
 - for composite windows 2-18
 - for lists 9-62
 - retrieving 5-22
 - setting at runtime 5-14, 8-90
 - setting in a script 2-17
 - deleting 5-3, 5-9, 9-58
 - disabling 5-3, 5-16, 9-64 to 9-65, 15-13
 - drilldown 2-46, 9-103
 - enabled state, retrieving 5-23
 - enabling 5-3, 5-17 to 5-18, 9-64 to 9-65, 15-12
 - ENTER key and 7-41, 7-42
 - face type, retrieving 5-33
 - graphics for. *See* graphics for buttons
 - in composite windows 6-6 to 6-8, 6-25 to 6-26
 - index for, retrieving 5-24
 - information on, retrieving 5-19 to 5-34
 - input context and 8-65 to 8-68
 - input override methods and 7-70
 - input processing and 8-56
 - justification setting for, retrieving 5-25
 - maximum number of 5-29, A-2
 - message boxes, in 16-102 to 16-105
 - methods for 2-14, 5-26

- modifying 5-3
 - on a list 9-59 to 9-61
 - on a window or tab set 5-10 to 5-13
- moving to specific 8-65 to 8-68
- names for 5-28, A-1
- number of, returning 5-21
- placement of 5-3, 5-14 to 5-15, 5-30
- print dialog boxes, in 16-114
- quick-select characters for 2-13, 5-31, 8-88 to 8-89
- ToolTips for. *See* ToolTips
- See also* button sets; radio buttons; toolbar buttons
- BZRO qualifier (.FIELD) 2-38

C

- %C_CONTAINER function 6-10 to 6-41
- C_METHAX method 6-47 to 6-48
- C_METHCTR method 6-49 to 6-50
- C_METHINP method 6-51 to 6-53
 - setting input context in 8-9 to 8-10
- C_METHLST method 6-54 to 6-56
 - setting focus in 9-80 to 9-81
- C_METHNOP method 6-59 to 6-60
- C_METHSEL method 6-61 to 6-62
- C_METHTS method 6-63 to 6-64
 - setting context in 13-36
- C_METHTXT method 6-65 to 6-66
- C_PROCESS routine 6-42 to 6-45
- cache, for file stacks E-7
- callbacks 1-20
- caption bar
 - height of, retrieving 16-199
 - invisibility of, and list borders 9-54
 - number of rows in, retrieving 7-24
- captions
 - headers and 7-39, 16-137
 - justification of 7-39
 - maximum length of (Windows) A-1
 - placing 7-38
 - rows available in 7-24
 - text, returning for 7-25
- cascading menus 2-30, 10-2
- case
 - alpha field validation 2-56
 - arrow qualifiers for menu entries 2-29
 - change method specifications 2-40
 - event names (ActiveX lists) 9-50
 - filenames 16-107
 - find operations 9-88, 9-140

- font names 16-216, 16-228, 18-40
- meridian indicators (AM, PM) 2-67, 2-68, 16-30
- method names (ActiveX lists) 9-22
- property names (ActiveX lists) 9-17
- quick-select characters 10-5
- rendition scheme names 18-33
- scriidl 2-126
- script commands 2-7
- selection window information, retrieving 12-15
- sizing character 16-217, 16-228
- window debugger commands 17-2
- CASE qualifier (.FIELD) 2-56
- cells in ActiveX Toolkit lists
 - changing color 9-37
 - changing font 9-39
 - highlighting 9-19
 - retrieving color 9-23
 - retrieving font 9-25
- centering
 - button sets 5-14, 8-90, 9-62, 13-17
 - %U_GETFILENAME dialog box 16-65
 - %U_MSGBOX dialog box 16-103
 - See also* justification
- century methods 7-50 to 7-52
- change methods 8-118 to 8-121
- CHANGE_METHOD qualifier (.FIELD) 2-40
- channels 1-17
 - allocating 16-59 to 16-60
 - closing 16-18
 - current environment and 7-2
 - environment level for 7-12
 - file searches, returning for 16-55
 - file stack 17-7, E-2, E-4
 - global, promoting to 16-57
 - information about, returning 7-12, 7-13, 7-14, 16-15
 - initialized for new environment 7-3
 - maximum number 7-13, A-1
 - number logged 7-14
 - opening 16-106 to 16-107
 - purging. *See* purging channels
 - releasing 16-124
 - variables required for 1-16
- characters
 - accepting from terminal 16-16 to 16-17
 - ASCII, retrieving for key codes 16-169
 - control, using for shortcuts 10-7
 - echoing or hiding typed 2-58
 - end-of-file 18-4
 - fill, defining 2-73 to 2-74
 - matching for alpha fields 2-56
 - paint 2-59, 2-88, 8-106
 - quick select. *See* quick-select character
 - selection 2-13, 8-88 to 8-89
 - sizing. *See* sizing character
- check box style selection windows 2-77, 12-3 to 12-4, 12-13, 12-18
- check boxes
 - defining 2-40 to 2-41, 8-26, 8-96
 - renditions for 2-34
 - validating 2-41, 7-43
 - width of, returning 16-175
- check mark character 18-2
- CHECKBOX qualifier (.FIELD) 2-40 to 2-41
- child windows and lists 6-2
 - activation 6-27, 6-30 to 6-31, 6-33 to 6-34
 - adding 6-21 to 6-24
 - .NET forms and controls 11-2 to 11-5
 - determining number of 6-40
 - enabling and disabling 6-35
 - event methods for 6-9
 - focus, setting in
 - ActiveX, selection, and text windows 16-24 to 16-25
 - composite windows 6-33 to 6-34
 - input windows 8-9 to 8-10
 - lists 9-80 to 9-81
 - tab sets 13-36
 - method extensions for 6-36 to 6-37, 6-38 to 6-39
 - methods for 6-42 to 6-45, 6-46 to 6-66
 - placing, removing, deleting 6-3
 - positioning 6-8
 - preventing processing of 6-59 to 6-60
 - removing from composite windows 6-41
 - retrieving ID for container 6-29
 - tabbing index for 6-28
 - tabbing within 6-4 to 6-5
- .chm files. *See* HTML Help
- class, setting fonts for 18-48
- classes, list. *See* list classes
- click. *See* mouse events
- client area of window
 - coordinates of, retrieving 16-170, 16-176, 16-191
 - offset for, retrieving 16-196, 16-201
 - size of, retrieving 7-47
 - sizing 16-206, 16-239
 - tab set windows in, distance to 16-189, 16-190

- close events
 - associating with mouse events 16-205
 - call extension for (ActiveX) 4-40
 - creating a method for 16-237
- close methods 7-59 to 7-61, 16-46
- closing
 - channels 16-18
 - debugger 17-6
 - environment 7-5
 - file stack levels E-6
 - HTML Help 16-75
 - linked lists on file stacks F-3
 - Toolkit applications 16-46
- color
 - ActiveX Toolkit lists 9-17 to 9-21
 - background 9-19
 - cell 9-23, 9-37
 - item 2-84, 2-86
 - ANSI codes for 18-39
 - background. *See* background color
 - boxes in windows 2-11
 - button image transparency 2-15
 - Color dialog box 16-153
 - fixed fonts and 18-49
 - foreground. *See* foreground color
 - general window fields 2-114
 - hyperlink prompts 16-156
 - %IDISPLAY_METHOD and 8-123
 - %IEDITFMT_METHOD and 8-128
 - input fields 2-34, 2-41, 8-26, 8-96
 - lines 2-79
 - Palette Entry dialog box 18-37, 18-38
 - printing 16-114
 - renditions and 18-24
 - reserved colors 18-24
 - RGB values 9-23
 - saving to initialization file 16-128
 - Synergy colors. *See* Synergy colors
 - system colors 18-24
 - tabs 13-24, 13-27
 - text in windows 2-103
 - user fields 7-62
 - window borders 2-9
 - window fill character 2-73
 - window titles 2-108
 - Windows-specific operations for 16-152 to 16-158
 - See also* color palette entries; renditions
- Color dialog box 16-153
- color palette entries 18-24 to 18-25
 - assigning colors to 18-25, 18-37, 18-39
 - enabling users to select 16-134 to 16-135
 - fixed fonts and 18-49
 - returning RGB values for 16-157
 - text windows, setting for 14-12
 - See also* color; renditions
- COLOR qualifier
 - .BORDER 2-9
 - .BOX 2-11
 - .FIELD 2-41
 - .FILL 2-73
 - .LINE 2-79
 - .TITLE 2-108
 - .WFIELD 2-114
- column build array error 10-62
- .COLUMN script command 2-4, 2-20 to 2-21
 - .END and 2-26
 - MB_COLUMN compared with 10-9
- columns
 - in ActiveX Toolkit lists
 - currently highlighted 9-18
 - determine if columns have been resized 9-139
 - retrieve width of 9-27
 - set minimum width of 9-126
 - set width of 9-41
 - in display 16-177, 16-196, 16-197
 - in menus. *See* menu column
 - in selection windows
 - blank 2-95, 12-5
 - multiple 2-102, 12-6
 - number needed to span pixels 16-150
- ColumnsPerRow published property 9-17
- combo boxes 2-65
 - change methods and 2-40
 - renditions and 2-34
 - state of 7-27, 7-41
 - validating 7-43
- command line
 - compiling scripts from 2-119 to 2-120
 - searching for options in 16-19 to 16-20
- comments in script files 2-7, 2-107
- Common Controls 6.0, support for 18-5
- compiling 2-118 to 2-120
 - error information for 2-122
 - error method for 7-80 to 7-81
 - from Proto 3-5
 - into open window library 2-124

- completion key codes
 - for I_INPUT 18-4
 - for U_WAIT 16-149, 18-6
 - components of UI Toolkit 1-4 to 1-6
 - Composer 1-5
 - defining font palette entries with 18-41
 - making fields inaccessible to 2-33
 - setting font in 18-46
 - composite container windows 6-2
 - creating 6-32
 - focus button on 6-25 to 6-26
 - placing, removing, deleting 6-3
 - removing child windows, lists 6-41
 - retrieving ID for 6-29
 - composite windows 6-2 to 6-9
 - buttons in 6-6 to 6-8
 - children of. *See* child windows and lists
 - container for. *See* composite container windows
 - creating 6-2
 - method for contained containers 6-49 to 6-50
 - placing, removing, deleting 6-3
 - positioning 6-8
 - processing 6-42 to 6-45
 - tabbing, focus, and input context in 6-4 to 6-6
 - configuration fields. *See* global variables
 - Connect to Printer dialog box 16-114
 - container windows
 - for ActiveX controls. *See* ActiveX containers
 - for composite windows. *See* composite container windows
 - for .NET forms and controls 11-2, 11-12 to 11-14
 - for tab sets 13-3, 13-9
 - for Toolkit applications. *See* application container
 - for Toolkit lists 9-139
 - returning type of 7-32, 16-233 to 16-234
 - Contents tab for HTML Help, displaying 16-81
 - context
 - combo boxes, option for 7-27, 7-42
 - composite window 6-30 to 6-31, 6-33 to 6-34, 13-36
 - input set. *See* input context
 - linked list, maintaining F-8
 - See also* help: for the Color dialog box
 - context ID, displaying Help for 16-84, 16-163
 - continuation marks for script commands 2-7
 - control characters, using as shortcuts 10-7
 - Control Panel
 - UI Toolkit 1-23, 18-11, 18-32
 - Windows 18-25 to 18-30, 18-31
 - control record 8-23, 16-116, 16-120
 - control sequences, customizing 18-11
 - controls
 - ActiveX. *See* ActiveX controls
 - .NET, embedding 11-2 to 11-14
 - COPY qualifier (.FIELD) 2-41
 - ctimestamp fields 2-68, 8-101
 - CTRL+G, in error processing 16-148
 - CTRL+M as shortcut 10-7
 - CTRL+S as shortcut 10-7
 - CTRL+SHIFT+TAB menu entry 18-3
 - CTRL+TAB menu entry 18-3
 - CurrentColumn published property 9-18
 - cursor
 - displaying 7-27, 7-41
 - position of 16-167, 17-6
 - turning off 9-133, 12-12, 14-19
- D**
- DarkItemColor published property 9-18, 9-20
 - data
 - input field
 - allowable 2-33, 2-36, 8-27, 8-95
 - decrementing 2-44, 8-26
 - default 2-33, 2-44, 8-26
 - displaying 8-13 to 8-14, 8-17
 - forcing to/from 8-41 to 8-42
 - getting from 8-44
 - incrementing 2-53, 8-26
 - putting into 8-70
 - list, manipulating 9-82 to 9-84
 - searching 9-88 to 9-91, 9-128 to 9-135
 - user 2-22, 2-112, 9-144, 17-6
 - data areas
 - for input fields
 - getting data from 2-41, 8-44
 - initializing 8-49 to 8-50
 - organization of 8-3
 - position of fields in 2-99 to 2-100
 - data control commands 2-102 to 2-107
 - window definitions and 2-5
 - data dictionary. *See* repository
 - .DATA script command 2-4, 2-22
 - data types
 - enumerated 2-48
 - input fields, setting for 2-32, 8-29, 8-99
 - integers 2-33
 - user-defined 2-70

- DATE qualifier (.FIELD) 2-42
- .DATE_GENERATED script command 2-23
- dates
 - alpha, decoding into numeric
 - components 16-26 to 16-28
 - application compile, displaying for 16-8
 - century processing method 7-50 to 7-52
 - formatting for 2-33, 16-51 to 16-52
 - in input fields 2-42, 8-100
 - change processing and 8-118
 - defaults for 2-44
 - modifying at runtime 8-30
 - values allowed for 2-62, 8-27
 - lunar and Julian, converting 16-21
 - months 7-22, 7-23, 16-101
 - order of month, day, year 2-43
 - period-style, setting maximum for 2-43
 - script file generation, defining for 2-23
 - separator for 2-43, 16-52
 - user fields and 7-54, 7-63, 7-68
- DBG_ environment variables 16-31
 - See also Environment Variables & System Options*
- dblproto 11-3
- debugger 1-6, 17-2 to 17-7
 - fixed font, recommended for 18-50
 - font handle, returning for 16-220
 - font, setting for 7-6 to 7-7, 18-45, 18-49
 - invoking 16-31
 - %M_SIGNAL and 10-52
 - sizing and placement 16-31
- debugger breakpoints, setting with U_BEEP 16-11
- decimal fields
 - changing type 8-29
 - creating 2-32, 8-29, 8-99
 - decimal point in 2-57, 8-97
 - implied-decimal 2-32, 8-29, 8-36
 - maximum size of 2-33
 - negative values, allowing for 2-56
 - precision, returning for 8-36
- decimal point 2-32, 2-57, 8-97
- DECIMAL_REQUIRED qualifier (.FIELD) 2-57
- DECREMENT qualifier (.FIELD) 2-44
- decrementing values in input fields 2-44, 8-26
- DEFAULT palette entry 18-41, 18-43
- DEFAULT qualifier
 - .BUTTON_SET 2-17
 - .FIELD 2-37, 2-44
 - .ITEM 2-77, 2-78
- defaults
 - button in button set. *See* buttons: default in button set
 - button in composite window 6-7
 - input field 2-33, 2-44, 8-26
 - selection window 2-77, 2-78
- delete key, reserved function for 18-22
- dialog boxes and screens
 - Color 16-153
 - Font 16-214, 16-216 to 16-217, 18-42
 - generating and processing 16-32
 - number of lines supported for A-3
 - Open or Save As 16-62
 - Palette Entry 18-37, 18-38
 - Print Setup 16-113 to 16-114
 - Renditions window 18-36 to 18-37
 - Search 16-164
 - Select Screen Area 18-38 to 18-39
 - See also* help
- .DICTIONARY. *See* .REPOSITORY_STRUCTURE
 - script command
- dictionary. *See* repository
- dimension (alpha fields) 8-5
 - retrieving 8-21
 - setting 2-32, 8-95
- DISABLE qualifier
 - .COLUMN 2-20
 - .ENTRY 2-27, 2-28
- DISABLED qualifier (.FIELD) 2-45
- disabling
 - automatic font switching 16-215
 - buttons 5-3, 5-16, 9-64 to 9-65, 15-13
 - child windows and lists 6-35
 - fields 2-45, 8-11 to 8-12, 8-26, 8-96
 - lists 9-75, 9-86
 - menu bar 10-47
 - menu columns 2-20, 10-13 to 10-14, 10-61
 - menu entries 2-27, 10-13 to 10-14, 10-66
 - menu lists 10-13 to 10-14
 - pop-up menu entries 10-13
 - submenu columns 10-13 to 10-14
 - tabs 13-21
 - termination processing 8-55
 - toolbars 15-37
- display
 - columns
 - returning for coordinate 16-197
 - returning offset for 16-196
 - returning width of 16-177

- current 1-11, 7-2
- movable. *See* display area
- renditions. *See* renditions
- rows
 - number available in section 7-24
 - returning height of 16-184
 - returning offset for 16-201
 - returning row number for coordinate 16-202
- shared 7-2
- text, return for section 7-25
- display area
 - for general windows, setting 2-24 to 2-25
 - for input windows, setting 2-24 to 2-25, 8-15 to 8-16, 8-92, 8-104
 - for lists, shifting 9-146
 - static 2-83
- display length 2-45, 2-48, 8-5 to 8-6, 8-26, 8-96
- display level (environment), returning current 7-15
- display methods. *See* environment methods; input methods
- display screen. *See* screen
- .DISPLAY_AREA script command 2-4, 2-24 to 2-25
- DISPLAY_METHOD qualifier (.FIELD) 2-46
- Document Properties dialog box 16-114
- DOTNET_TKADDCTL subroutine 11-6 to 11-7
- DOTNET_TKFORM subroutine 11-11
- DOTNET_TKINPUT subroutine 11-8 to 11-10
- %DOTNET_TKWIN function 11-12 to 11-14
- DOUBLE_CLICK_METHOD qualifier (.LISTCLASS) 2-82
- double-click methods. *See* list methods
- down arrow key. *See* arrow keys
- drag bars 2-10, 9-54
- DRAGBAR qualifier 2-10
- DRILL_METHOD qualifier 2-46
- drilldown buttons 2-46, 9-103
- drilldown methods 8-125 to 8-126, 9-103
- DSP_LENGTH qualifier (.FIELD) 2-45
- DTK_BOUNDS environment variable 1-19, 18-3
 - Synergy .NET and 1-21
 - See also* bounds checking
- DTK_BOUNDS_LOG environment variable 1-19, 1-21
- DTK_MENU_UP environment variable 10-12
- DTK_THROW_ABORT environment
 - variable 16-5 to 16-7, 18-5
- DTKDBG environment variable 17-3
- DTKFWSWINSIZ environment variable E-3, E-7
- DTKKEYCTLFIL environment variable 18-13

- DTKMAP environment variable 18-13
- DTKMAPFIL environment variable 18-13
- dtkmap.ism (default key map file) 18-12
- DTKRND environment variable 18-33
- DTKRNDFIL environment variable 18-33
- dtkrnd.ism (default rendition file) 18-33
- DTKTERM environment variable 18-14
- DTKTMP environment variable E-7

E

- E_ENTER subroutine 7-3 to 7-4
- E_EXIT subroutine 7-5
- %E_FONT function 7-6 to 7-7
- %E_INFO function 7-8 to 7-34
- E_METHOD subroutine 7-35 to 7-37
- E_SECT subroutine 7-38 to 7-40
- E_STATE subroutine 7-41 to 7-44
- %EAPPMOVE_METHOD function 7-46
- %EAPPSIZE_METHOD function 7-47
- %EAPPSTATE_METHOD function 7-49
- %ECENTURY_METHOD function 7-50 to 7-52
- ECHKFLD_METHOD subroutine 7-53 to 7-58
 - EDSPFLD_METHOD and 7-63
 - .FIELD USER qualifier and 2-70
- echo
 - environments, default setting for 7-27, 7-41
 - runtime, setting at 8-27, 8-97
 - S_SELECT and 12-12
 - T_VIEW and 14-19
 - windows scripts, setting in 2-58
- ECHO qualifier (.FIELD) 2-58
- ECLOSE_METHOD subroutine 7-59 to 7-61
 - U_FINISH and 16-46
- edit control frames, spacing for 16-232
- edit format method 8-127 to 8-129
- EDITFMT_METHOD qualifier (.FIELD) 2-47
- editing method, user field 7-67 to 7-69
- EDSPFLD_METHOD subroutine 7-62 to 7-66
 - USER qualifier (.FIELD) and 2-70
- EEDTDSP_METHOD subroutine 7-67 to 7-69
 - USER qualifier (.FIELD) and 2-70
- EENTRST_METHOD subroutine 7-70 to 7-71
- EFKEY_METHOD subroutine 7-72, 18-23
- EHELP_METHOD subroutine 7-73 to 7-79
 - help identifier for 2-52, 8-96
 - help window renditions, changing in 16-70
 - hyperlink and drill methods and D-7
 - I_INPFLD, called by 8-54

- I_INPUT called by 8-59
- M_PROCESS, called by 10-45
- S_SELECT, called by 12-11
- S_SELECTCB, called by 12-14
- T_EDIT, called by 14-5
- U_CHR, called by 16-17
- U_EDITKEYS, called by 16-40
- U_EDITREND, called by 16-43
- U_FLD, called by 16-50
- U_MODREND, called by 16-100
- %U_MSGBOX, called by 16-102
- elements (WPF), embedding 11-2 to 11-14
- embedding
 - ActiveX controls 4-3 to 4-61
 - .NET forms and controls 11-2 to 11-14
- Enabled published property 9-18
- ENABLED qualifier (.FIELD) 2-45
- enabled state for buttons, returning 5-23, 15-20
- enabling
 - ActiveX controls 9-18
 - automatic font switching 16-215
 - buttons 5-3, 5-17 to 5-18, 9-64 to 9-65, 15-12
 - child windows and lists 6-35
 - fields 2-45, 8-19 to 8-20
 - lists 9-75, 9-87
 - menu bar 10-4 to 10-5
 - menu columns, entries, lists,
 - subcolumns 10-15 to 10-16
 - pop-up menu entries 10-15
 - tabs 13-20
 - toolbars 15-37
- encapsulation 1-13
- .END script command 2-5, 2-26
- endian type, moving files between systems 1-23
- end-of-file characters, number allowed on input 8-5
- end-of-file key, customizing 18-22
- ENTER key
 - break processing and 2-39
 - characters, treating as 8-8
 - default button, simulating 7-27, 7-41
 - input fields and 2-58, 8-4, 18-7
 - leave methods and 8-132
 - list double-click and 9-8, 9-151
 - menu selection and 10-5
 - selection processing and 12-2
- .ENTRY script command 2-5, 2-27 to 2-30
- enumerated input fields
 - defining 2-48 to 2-49, 8-96
 - display length for 2-45
 - modifying at runtime 8-30
- ENUMERATED qualifier (.FIELD) 2-48 to 2-49
- environment methods
 - addresses for, getting 7-20
 - application close 7-59 to 7-61, 16-46
 - application move 7-46
 - application resize 7-47
 - application state processing 7-49
 - century processing 7-50 to 7-52
 - help processing 7-73 to 7-79
 - input override 7-70 to 7-71
 - key mapping 7-72
 - registering 7-35 to 7-37
 - script compilation error processing 7-80 to 7-81
 - user field display 7-62 to 7-66
 - user field editing 7-67 to 7-69
 - user field validation 7-53 to 7-58
 - utilities processing 7-82 to 7-83
 - See also methods*
- environment variables
 - button transparency 2-15
 - colors 18-25
 - font settings 7-7, 18-44, 18-46
 - key mapping 18-13
 - list types 9-5
 - renditions 18-33
 - repository files 2-92
 - See also specific environment variable name;*
Environment Variables & System Options
- environments 1-11 to 1-12, 7-2
 - entering new 7-3 to 7-4
 - exiting 7-5
 - font for, setting 7-6 to 7-7
 - information on, returning 7-8 to 7-34
 - levels of, maximum 7-19, 16-136, A-1
 - menus for, information on 10-18 to 10-22,
 - 10-34 to 10-35
 - state of, modifying 7-41 to 7-44
 - text for, placing 7-38 to 7-40
 - toolbars, logging with 15-32, 15-36
 - See also environment methods*
- EOF characters, number allowed on input 8-5
- .err file extension 2-120, 2-127
- \$ERR_NOCHAIN error 1-20
- \$ERR_TOOLKIT error 16-6

- %ERR_TRACEBACK, retrieving information for thrown errors with 16-6
- error trapping 16-5 to 16-7
- errors 1-18
 - abort processing and 16-5 to 16-7
 - ActiveX control exceptions 4-8
 - “Cannot access external routine” and debugger 17-3
 - channel/file opening 16-106
 - \$ERR_NOCHAIN 1-20
 - \$ERR_TOOLKIT 16-6
 - list class creation 9-69
 - list creation 9-78
 - list processing 9-114
 - menu column loading 10-37
 - “overflow of column build array” 10-62
 - pop-up window 16-109
 - rendition scheme 16-90, 16-132
 - script compilation
 - exit status and 2-119
 - information about, returning 2-122
 - log file for 2-120
 - method for 7-80 to 7-81
 - standard output stream, sending to 2-6
 - script-to-repository conversion 2-127
 - trapping 16-5 to 16-7
 - U_WAIT and 16-148
 - waiting for user response to 16-148
 - window library 8-62, 16-92, 16-127
 - window system, getting 16-69
- ESCRPTERR_METHOD subroutine 7-80 to 7-81
- European formatting 7-27, 7-41
- EUTILS_METHOD subroutine 7-82 to 7-83
 - called by U_ menu entries B-5
 - hot entry processing 8-41, 8-57
 - hyperlink processing D-7
 - I_INPFLD and 8-54
 - I_INPUT and 8-59
 - M_PROCESS and 10-45
 - S_SELECT and 12-11
 - S_SELECTCB and 12-14
 - T_EDIT, and 14-5
 - T_VIEW and 14-20
 - U_ABOUT and 16-8
 - U_CHAR and 16-17
 - U_FLD and 16-50
- event methods. *See* mouse events
- event-driven programming 1-13, 1-17 to 1-18
- EXACT qualifier (.FIELD) 2-56

- example programs and script files 1-2
- exceptions, ActiveX control 4-8
- Excluded by Toolkit option (Repository) 2-33
- Exif graphics 2-15
- EXITE, called by U_ABORT 16-6
- exiting to previous environment 7-5
 - See also* closing
- Ext_hWnd published property 9-18
- Ext_Visible published property 9-18
- extended properties 4-4

F

- FarPoint Spread control 4-6
 - creating Toolkit container for 4-36 to 4-38
- fatal errors
 - ActiveX control exceptions and 4-9
 - displaying on abort 16-5 to 16-7
 - See also* errors
- FATAL Synergy DBL subroutine 1-18
- field frames, displaying 8-43
- .FIELD script command 2-5, 2-31 to 2-72
 - organization of fields 8-3
 - overriding repository specifications 2-91
 - scridl and 2-127
 - setting fonts with 18-46
- field specifications 2-35 to 2-36
- FIELD_POSITION qualifier (.FIELD) 2-49
- fields
 - accepting from terminal 16-48 to 16-50
 - frames for, displaying 8-43
 - general window fields 2-114 to 2-115
 - input. *See* input fields
 - multi-line 2-32
 - See also* global variables
- file stacks E-2 to E-14
 - cache for E-7
 - information about in debugger 17-5, 17-7
 - linked lists and F-2, F-3, F-4
 - maximum size of A-1
 - subroutines for E-4 to E-14
- files
 - channels for. *See* channels
 - .chm. *See* HTML Help
 - determining if open 16-55 to 16-56
 - environments and 7-2
 - .hlp (WinHelp) 16-162
 - initialization. *See* initialization files
 - key mapping 1-23, 18-12

- library. *See* window library
- moving to other systems 1-23
- OCX (for ActiveX controls) 4-28, 4-30, 4-37
- opening 16-106 to 16-107
- prompting user for 16-61 to 16-66
- rendition 18-33
- repository 2-91 to 2-92, 2-125, 8-107
- script. *See* script file
- searching for 16-55 to 16-56
- Synergy Data Language 2-125 to 2-127
- See also specific filename*
- fill character, defining 8-27, 8-97
- .FILL script command 2-73 to 2-74
 - overriding 2-34
 - types of windows that use] 2-5
- fill string, defining 2-73 to 2-74
- find operations (in lists) 9-112, 9-116
 - information about 9-140
 - parameters for 9-88 to 9-91
 - S_FIND menu entry and 9-131
- find window (for lists)
 - creating 9-92 to 9-94
 - name size A-2
 - replacing default 9-133
 - S_FIND menu entry and 9-131
- fixed fonts 18-49
- flags
 - attribute, for renditions 14-13, 16-119
 - defined in tools.def file 1-16
 - state, environments and 7-2
 - submenu 10-2, 18-19
- FLAGS subroutine 7-42
 - See also Synergy DBL Language Reference Manual*
- fldinf.def file 8-22, 8-47
- FMT qualifier (.FIELD) 2-51
- focus
 - ActiveX controls, for 4-12
 - ActiveX Toolkit lists, for 9-15, 9-17, 9-19
 - buttons, for 5-2
 - check boxes, for 2-41
 - child lists, for 9-80 to 9-81
 - child windows, for 8-9 to 8-10, 16-24 to 16-25
 - combo boxes, for 2-65
 - composite windows, for 6-4 to 6-6
 - input methods on Windows and D-7
 - radio buttons, for 2-61
 - See also input context*
- Font dialog box 16-216 to 16-217, 18-41
- font handle, retrieving 16-220, 16-223, 16-228
- font palette 18-40
 - information about, returning 16-218 to 16-219
 - synergy.ini, synuser.ini and 18-40
 - See also fonts*
- font palette entries 18-40
 - assigning methods for 18-40
 - defining 16-228 to 16-229, 18-40 to 18-42
 - font height and width in pixels, returning for 16-222
 - font point size, returning for 16-223 to 16-224
 - handle, returning for 16-220
 - name, returning for 16-221
 - saving 16-227, 18-40
 - sizing character, returning for 16-223 to 16-224
 - typeface for, returning 16-223 to 16-224
 - window, returning for 16-226
 - See also fonts*
- FONT qualifier (.FIELD) 2-50
- .FONT script command 2-5, 2-75, 18-46
- font switching, automatic 16-215
- FONT_ALPHAFLD environment variable 18-46, 18-48
- FONT_ALTERNATE environment
 - variable 18-43 to 18-44, 18-48
- FONT_DEBUG environment variable 16-31, 18-45, 18-49
- FONT_FOOTER environment variable 7-7, 18-44, 18-48
- FONT_GLOBAL environment variable 18-43 to 18-44, 18-48
- FONT_HEADER environment variable 7-7, 18-44, 18-48
- FONT_HEIGHT environment variable 18-43, 18-45, 18-48, 18-49
- FONT_INFO environment variable 7-7, 18-44, 18-48
- FONT_LIST environment variable 18-46, 18-48
- FONT_NUMFLD environment variable 18-46, 18-48
- FONT_PROMPT environment variable 18-46, 18-48
- FONT_TEXT environment variable 18-46, 18-48
- FONT_WIDTH environment variable 18-43, 18-45, 18-48, 18-49
- FontName published property 9-18, 9-20, 9-39
- fonts
 - ActiveX Toolkit lists, for 9-20
 - alternate. *See* alternate font
 - application, setting for 18-42 to 18-45, 18-48
 - Composer, setting with 18-40, 18-41, 18-46
 - debugger, for 18-45, 18-49, 18-50
 - fields, setting for 2-50, 8-100

- fixed (on Windows) 18-49
- Font dialog box, displaying 16-216 to 16-217
- for header, footer, information line 7-6 to 7-7, 18-44, 18-48
- global 16-214, 18-42 to 18-45
- information about, retrieving 16-213 to 16-231
- objects (window elements), setting
 - for 18-45 to 18-47, 18-48
- precedence of settings 18-48 to 18-49
- prompts, setting for 2-60, 8-31, 8-100
- proportional, support for 18-49
- Repository, setting in 18-46, 18-48
- runtime, setting at 18-42, 18-44, 18-47
- scaling 16-225, 16-230
- script files, setting in 18-46
- windows, setting for 2-75, 16-231
- See also* font palette; font palette entries
- FontSize published property 9-18, 9-20, 9-39
- footer. *See* header and footer for application window
- FooterFontName published property 9-18, 9-44
- FooterFontSize published property 9-18, 9-44
- FOOTERS qualifier (.LISTCLASS) 2-82
- force buffer 8-41 to 8-42, 8-57
- foreground color
 - ActiveX list cell, for 9-23
 - fixed fonts and 18-49
 - for palette entries
 - displaying in Select Color Palette window 16-134
 - returning 16-152, 16-157
 - setting 16-116, 18-39
 - text, for 2-41
- See also* color
- FORMAT qualifier (.FIELD) 2-51
- format string 2-51, 8-26, 8-28, 8-96, A-1
- forms (.NET), embedding 11-2 to 11-14
- forward edit mode message 18-3
- FPOS qualifier (.FIELD) 2-49
- frames
 - edit control 16-232
 - field, displaying 8-43
 - window. *See* window frames
- FS_APPEND subroutine E-5
- FS_CLOSE subroutine E-6
- FS_INIT subroutine E-7
- FS_OPEN subroutine E-8
- FS_PARAM subroutine E-9
- FS_QUIT subroutine E-11
- FS_READ subroutine E-12

- FS_WRITE subroutine E-13
- function codes
 - files used for 7-72, 18-12
 - shortcut text for, returning 10-36
 - shortcuts for, assigning 18-11 to 18-23

G

- g_ActiveX_list global variable (tkctl.def) 9-5, 18-2
- g_beep global variable (tkctl.def) 18-2
- g_chk_char global variable (tkctl.def) 8-119, 12-4, 18-2
- g_date_order global variable (tkctl.def) 2-43, 18-3
- g_date_sep global variable (tkctl.def) 2-43, 18-3
- g_dtktbounds global variable (tkctl.def) 18-3
- See* bounds checking
- g_edt_fwrd_msg global variable (tkctl.def) 18-3
- g_edt_ins_msg global variable (tkctl.def) 18-3
- g_edt_ovr_msg global variable (tkctl.def) 18-3
- g_edt_rev_msg global variable (tkctl.def) 18-3
- g_entnam global variable (tools.def) 8-56, 10-8, D-10
- g_entnamctab global variable (tkctl.def) 18-3
- g_entnamctab global variable (tkctl.def) 18-3
- g_entnamstab global variable (tkctl.def) 6-5, 18-4
- g_entnamtab global variable (tkctl.def) 6-5, 18-4
- g_eof_max global variable (tkctl.def) 8-5, 18-4
- g_fldnam global variable (tools.def) 2-39, 8-56, D-3
- g_inp_trm_key global variable (tkctl.def) 18-4
- g_inp_trm_msg global variable (tkctl.def) 18-4
- g_periods global variable (tkctl.def) 2-43, 18-4
- g_plc_col_args global variable (tkctl.def) 18-4
- g_retaincontext global variable (tkctl.def) 18-5
- g_select global variable (tools.def) 8-56, 10-8, D-10
- g_tabfacecolor global variable 13-13, 18-5
- g_throwabort global variable 16-5 to 16-7, 18-5
- g_time_entry global variable (tkctl.def) 18-6
- g_time_sep global variable (tkctl.def) 2-67, 18-6
- g_txt_rtrn global variable (tools.def) 18-7
- g_txt_softreturn global variable (tools.def) 18-7
- g_updt_fld global variable (tools.def) 18-7
- g_updt_set global variable (tools.def) 18-7
- g_utlib global variable (tools.def) 8-60, 16-137, 16-149
- g_wait_time global variable (tkctl.def) 18-6
- g_wt_alrt_key global variable (tkctl.def) 16-149, 18-6
- g_wt_alrt_msg global variable (tkctl.def) 16-149, 18-6
- g_wt_err_key global variable (tkctl.def) 16-149, 18-6
- g_wt_err_msg global variable (tkctl.def) 16-149, 18-6
- general window fields 2-114 to 2-115

- general windows 2-116
 - display area for 2-24 to 2-25
 - loading with Proto 3-6
 - modifying font for 16-231
 - placement of, defining 2-89
 - retrieving font for 16-226
 - specifying font for 2-75
 - gennet40 11-3
 - GetCellColor published method 9-23
 - GetCellFont published method 9-25
 - GetColumnWidth published method 9-27
 - GetFooterCell published method 9-28
 - GetFooterCellAlignment published method 9-29
 - GetFooterCellFont published method 9-30
 - GetHeaderCell published method 9-32
 - GetHeaderCellAlignment published method 9-33
 - GetHeaderCellFont published method 9-34
 - gi_sts field (inpctl.def) 16-17, 16-50
 - GIF graphics 2-15
 - global channels 1-17
 - allocating 16-59, 16-106
 - information about, returning 16-15
 - promoting a local channel 16-57
 - global data section, names reserved for 1-16
 - global environment 7-2
 - ActiveX container windows and 4-29, 4-37
 - channels and. *See* global channels
 - composite container windows and 6-32
 - information on, returning 7-8 to 7-34
 - lists and 9-95
 - menu columns and 10-3, 10-10, 10-18, 10-62
 - .NET form container windows and 11-13
 - tabbed dialogs and 13-10
 - toolbars and 15-35, 15-36
 - window libraries and 2-123, 16-137
 - windows and 16-58, 16-94, 16-159
 - See also* environments
 - global variables 18-2 to 18-7
 - limitations for .NET 1-22
 - See also* g_ entries; gi_sts field
 - graphics for buttons 2-15
 - information on, retrieving 5-33
 - on lists 9-56
 - on tab sets 5-6, 13-15
 - on toolbars 15-7
 - on windows 2-15, 5-6, 8-88
 - transparent 2-15
 - %GRID_CREATE function 4-6, 4-36 to 4-38
 - grid.def file 4-37
 - GridRepaint published method 9-36
 - group fields, accessing 2-34, 2-35, 2-92, 8-108
 - group of toolbar buttons 15-9
 - gs_dfltptr pointer 8-46
 - gs_edtcol field 8-23
 - gs_edtlcol field 8-23
 - gs_edtlrow field 8-23
 - gs_edtrow field 8-23
 - gs_fldnam pointer 8-46
 - gs_fmtptr pointer 8-46
 - gs_hlpptr pointer 8-46
 - gs_info pointer 8-46
 - gs_inpfld fields 8-23
 - gs_pts array 8-47
 - gs_setcnt field 8-81
 - gs_setfld array 8-81
 - gs_setnam field 8-81
 - gs_usrptr pointer 8-46
 - gs_usrtyp pointer 8-46
- ## H
- handles
 - ActiveX Toolkit list 9-18
 - font 16-220, 16-223, 16-226, 16-228
 - HTML Help window 16-72, 16-83
 - runtime instance 16-168
 - toolbar and toolbar button 15-24
 - header and footer for application window 1-7, 1-8
 - environments and 7-2, 7-5
 - font for 7-6 to 7-7, 18-44, 18-48
 - maximum number of lines A-1
 - proportional fonts and 18-49
 - renditions for 16-118
 - rows available in, retrieving 7-24
 - size of, setting 16-136
 - text 7-25, 7-38 to 7-40
 - U_START and header creation 16-136 to 16-138
 - See also* list footer; list header; menu column: header
 - HeaderFontName published property 9-19, 9-48
 - HeaderFontSize published property 9-19, 9-48
 - HEADERS qualifier (.LISTCLASS) 2-83
 - height
 - ActiveX container window 4-31, 4-60
 - application window 7-47, 16-130, 16-137
 - box drawn in window 2-11
 - caption bar for window, retrieving for 16-199
 - dialog box (U_DIALOG) 16-32

- display area for an input window 2-24, 8-15, 8-92
- font palette entry 16-222
- general window 2-116
- horizontal scroll bar, retrieving for 16-179
- list 9-141
- message (U_MESSAGE) 16-96
- monitor display area 16-186
- rows in a window, retrieving for 16-182, 16-184
- screen 16-122, 16-136
- scroll bar window 16-22
- selection window 2-64
- sizing character for a window, retrieving for 16-173
- tab in tab set 16-187
- toolbar 15-32, 15-38
- toolbar button 15-7
- help
 - for Proto 3-3
 - for the Color dialog box (%U_WINCOLOR) 16-153
 - for the Font dialog box (%U_WNDFONT) 16-216
 - for the window debugger 17-2
 - HTML help. *See* HTML Help
 - Toolkit Help
 - for ActiveX controls 4-12
 - for input fields 2-52, 8-26, 8-46, 8-96
 - for lists 9-66, 9-129
 - for menus 10-43
 - for selection windows 12-10, 12-13
 - for text windows 14-3, 14-19, 16-35
 - general window, using for 2-116
 - menu entry for B-4
 - method for 7-73 to 7-79
 - renditions for 16-70 to 16-71, 16-118, 18-31
 - WinHelp (winhelp32.exe) 16-162 to 16-165
- HELP qualifier (.FIELD) 2-52
- Help Topics dialog box 16-163
- HFT terminal type 18-14
- HIGHLIGHT qualifier
 - .BORDER 2-9
 - .BOX 2-11
 - .FIELD 2-52
 - .FILL 2-73
 - .LINE 2-79
 - .TITLE 2-108
 - .WFIELD 2-115
- highlighting. *See* renditions
- .hlp file extension 16-162
- HorizontalScrollBar published property 9-19
- .hpf files 16-163

- HTML Help 16-72 to 16-86
 - for Color dialog (%U_WINCOLOR) 16-153
 - for Font dialog (%U_WNDFONT) 16-217
- hyperlink prompts
 - methods for 8-130 to 8-131, D-7
 - processing 8-130 to 8-131
 - returning color for 16-156
- HYPERLINK_METHOD qualifier (.FIELD) 2-52

I

- I_ACTIVATE subroutine 8-8
- I_CTRCONTEXT subroutine 8-9 to 8-10
- I_DISABLE subroutine 8-11 to 8-12
 - leave processing D-3
- I_DISPLAY subroutine 8-13 to 8-14
 - displaying previous find 9-134
- I_DSPAREA subroutine 8-15 to 8-16
- I_DSPFLD subroutine 8-17
 - drilldown processing 8-125, D-7
- I_ENABLE subroutine 8-19 to 8-20
 - leave processing D-3
- %I_FLDDIM function 8-21
- I_FLDINF subroutine 8-22 to 8-24, 12-15
- I_FLDMOD subroutine 8-25 to 8-35, 18-47, 18-48
 - hyperlink processing 8-130
 - leave processing 8-132, D-4
- %I_FLDPREC function 8-36
- I_FLDSEL subroutine 8-37 to 8-38, 12-6
- %I_FLDSIZE function 8-39
- %I_FLDSTRPOS function 8-40
- I_FORCE subroutine 8-4, 8-41 to 8-42
 - drilldown processing 8-125, D-7
- I_FRAMES subroutine 8-43
- I_GETFLD subroutine 8-44
- %I_GETSTRING function 8-46 to 8-47
- %I_INFO function 8-48
- I_INIT subroutine 8-49 to 8-50, D-4
- I_INPFLD subroutine 8-51 to 8-54
- I_INPUT subroutine 8-55 to 8-61
 - arrive method and 2-37
 - completion key code 18-4
 - hyperlink processing 8-130
 - menu entries recognized 8-59
 - termination processing message 18-4
 - window fields and 2-115
- I_LDINP subroutine 8-62 to 8-64

- I_NEXT subroutine 8-20, 8-65 to 8-68
 - leave processing and D-4
 - list selection processing and 9-134
- I_PROMPT subroutine 8-69
- I_PUTFLD subroutine 8-70
- I_READONLY subroutine 8-72 to 8-76, D-3
- I_SETDEL subroutine 8-77 to 8-79, D-4
- I_SETINF subroutine 8-80 to 8-81
- I_SNAPSHOT subroutine 8-77, 8-82 to 8-83
 - INPUT_SNAPSHOT and 2-83
 - leave processing and D-4
- I_TXTPOS subroutine 8-84 to 8-86
- I_USER subroutine 2-70, 8-87
- IARRIVE_METHOD subroutine 8-114 to 8-117
- IB_BUTTON subroutine 8-88 to 8-89
- IB_BUTTONSET subroutine 8-90 to 8-91
- IB_DSP_AREA subroutine 8-92
- IB_END subroutine 8-7, 8-93
- IB_FIELD subroutine 8-95 to 8-102, 18-47, 18-48
 - .FIELD vs. 8-7
 - hyperlink processing 8-130
 - leave processing 8-132
- IB_INPUT subroutine 8-7, 8-103 to 8-104
- IB_LOCAL subroutine 8-7, 8-105, 8-112
- IB_PAINT subroutine 8-7, 8-106
- IB_RPS_STRUCTURE subroutine 8-107 to 8-108
 - default 8-112
 - IB_FIELD and 8-7, 8-98
- IB_SET subroutine 8-109
- IB_STRUCTURE subroutine 8-7, 8-111 to 8-112
 - default 8-112
- IBM high-function terminal 18-14
- %ICHANGE_METHOD function 8-118 to 8-121
 - ECHKFLD_METHOD vs. 7-53
 - Exxx_METHODs and 7-54
 - I_NEXT and D-7
- icons, setting 16-87 to 16-88
 - for lists 9-96 to 9-97
 - for message boxes 16-104
- identifiers
 - ActiveX containers, getting for 4-20
 - ActiveX controls, getting for 4-21
 - ActiveX Toolkit lists, getting for 9-139
 - help 2-52, 8-26
 - method set, setting for 16-212
 - Toolkit container windows, getting for 4-28, 4-31, 4-36
- %IDISPLAY_METHOD function 8-122 to 8-124

- IDRILL_METHOD subroutine 8-125 to 8-126
- %IEDITFMT_METHOD function 8-127 to 8-129
- IHYPER_METHOD subroutine 8-130 to 8-131
- ILEAVE_METHOD subroutine 8-132 to 8-134
- implied-decimal fields 2-32
 - creating 2-32, 8-29
 - getting precision of 8-36
- include file, creating for shortcuts 10-36
- INCREMENT qualifier (.FIELD) 2-37, 2-53
- incrementing value in a field 2-53
- indexed ActiveX control properties 4-23, 4-24
- indexes
 - button 5-7, 5-11, 5-22, 5-24
 - child window 6-4, 6-21, 6-28, B-1
 - color 16-154
 - field specification 2-35 to 2-36
 - filter, for Open and Save As dialogs 16-61
 - HTML Help 16-73, 16-76
 - icon 9-96, 16-87
 - menu 10-26
 - tab 13-12, 13-14, 13-33
 - WinHelp 16-162 to 16-165
- information icon 16-104
- information line
 - creating 16-136 to 16-138
 - displaying text on 2-53, 8-26
 - font 7-6 to 7-7, 18-44, 18-48
 - ownership of 1-11, 7-2
 - placing 16-10
 - placing text in 7-38 to 7-40
 - removing 16-10
 - rows available in 7-24
 - text, returning for 7-25
- INFORMATION qualifier (.FIELD) 2-53
- information string for fields 2-53
- initialization files
 - application window settings, saving to 16-130
 - debugger font settings in 18-45
 - initial font palette 18-40, 18-43
 - initialization settings, saving to 16-128
- initialization settings, saving 16-128
- inp_ pointers 8-46
- inpctl.def file 7-72, 10-36, 18-23
- inpinf.def file 8-46, 8-114, D-4
- input
 - ActiveX control 4-12 to 4-13
 - allowable entries 2-36
 - canceled 8-58

- displaying and formatting 7-62 to 7-66
- echoing 2-58
- editing 8-52, 8-53
- hiding 2-58
- in non-input window 16-48 to 16-50
- insert mode 16-50
- list 9-66 to 9-67, 9-98 to 9-100, 9-101 to 9-104
- negative 2-57
- override processing method 7-70 to 7-71
- overstrike mode 16-50
- processing 1-9, 1-17, 8-3 to 8-134
- routine to call before 9-129, 9-133
- terminating 2-58, 8-4, 8-8
- time-out 2-72, 8-4, 10-43 to 10-46
- waiting for 16-148 to 16-149
- input context 8-57
 - break processing and 8-12, 8-20
 - changing D-4
 - disabled fields on Windows and 8-11
 - for child windows 6-4 to 6-6, 8-9 to 8-10
 - menu entries and 18-5, D-10
 - saving and restoring 8-82
 - setting 7-41 to 7-44, 8-9 to 8-10, 8-65 to 8-68
 - See also* focus
- input fields 1-9, 8-3 to 8-6
 - adding/defining 2-4, 2-31 to 2-72, 8-95 to 8-102
 - alpha. *See* alpha fields
 - blank 2-38, 2-63, 8-26, 8-96
 - blanks in 2-51
 - break processing for. *See* break processing
 - color for 2-34, 2-41, 8-26, 8-96
 - control record for 8-23
 - data and. *See* data: input field
 - data areas for. *See* data areas
 - data types, setting for 2-32, 8-29, 8-99
 - dates. *See* dates: in input fields
 - decimal fields. *See* decimal fields
 - deleted, restoring 8-82
 - dimension for. *See* dimension (alpha fields)
 - disabling 2-45, 8-11 to 8-12, 8-26, 8-96
 - display length for 2-45, 8-6
 - displaying data to 8-13 to 8-14, 8-17
 - editing 8-4, 8-59
 - empty
 - character to fill 2-59, 2-88
 - moving to 8-65, B-3
 - processing 2-37, 8-4, 8-81
 - enabling 2-45, 8-19 to 8-20
 - ENTER, specifying a new line with 18-7
 - enumerated. *See* enumerated input fields
 - font for 2-50, 8-100, 18-46
 - formatting 2-51, 7-62 to 7-66
 - frames for, displaying 8-43
 - group fields, accessing 2-34, 2-35
 - help and 2-52, 8-26, 8-46, 8-96
 - implied-decimal fields 2-32, 8-29, 8-36
 - information about, retrieving 8-21, 8-22 to 8-24, 8-39, 8-40
 - information string size A-1
 - initializing 8-49 to 8-50
 - integer fields 2-33, 8-29
 - justification of text in 2-51, 2-55, 8-27, 8-97
 - maximum/minimum value for, defining 2-62
 - name size, maximum for 2-100, A-1
 - names, reserved 1-16, 8-13, 8-43, 8-49, 8-65
 - number of, returning 8-48
 - organization of 8-3
 - processing 8-51 to 8-54, 8-65 to 8-68
 - prompts for. *See* prompt
 - read-only state, setting for 8-27, 8-72 to 8-76, D-3
 - renditions for. *See* renditions: input fields
 - scrolling 2-45, 8-6
 - selection windows, assigning to 2-66, 2-95, 8-37 to 8-38
 - set of. *See* input set
 - size of A-2
 - .FIELD, specifying with 2-32, 2-33
 - I_FLDMOD, specifying with 8-25, 8-29
 - retrieving with %I_FLDSIZE 8-39
 - structure
 - defining in 2-99 to 2-100, 8-112
 - retrieving position in 8-40
 - text 8-5
 - time 2-67 to 2-68
 - user data set 2-22, 2-112, 8-28
 - user fields. *See* user fields
 - user-defined strings for 8-87, A-1
 - validation of. *See* validation: input field
 - wait time for 8-33, 8-102
 - width of. *See* display length; input length; view length
 - window, maximum number in 2-32
 - zero, prohibiting entry of 2-63
 - See also* input
- input length 2-54, 8-5 to 8-6, 8-27, 8-97

- input methods D-3 to D-9
 - arrive 8-114 to 8-117
 - change 8-118 to 8-121
 - display 8-122 to 8-124
 - drilldown 8-125 to 8-126, 9-103
 - edit format 8-127 to 8-129
 - hyperlink 8-130 to 8-131, D-7
 - leave 8-132 to 8-134, D-5
 - modifying at runtime 8-25 to 8-35
 - specifying as parameter. *See* .FIELD script command; IB_FIELD subroutine
 - See also* methods
- input override method 7-70 to 7-71
- .INPUT script command 2-5, 2-76
 - IB_INPUT vs. 8-103
 - scribl and 2-125
- input set
 - adding to input window 8-109
 - break 2-39
 - changing processing order 8-65 to 8-68
 - context for. *See* input context
 - default 8-110
 - defining 2-97, 8-7
 - deleting fields from 8-77 to 8-79
 - disabling input field 8-11 to 8-12
 - information on, retrieving 8-80 to 8-81
 - list and 2-83
 - maximum number of fields in A-2
 - name size A-2
 - names, reserved 1-16, 8-13, 8-43, 8-49, 8-65
 - number of, returning 8-48
 - processing 8-55 to 8-61
 - saving or restoring definition 8-77
- input structure. *See* structure
- input window 1-9, 8-3
 - building at runtime 8-6, 8-103 to 8-104
 - buttons in 2-13 to 2-16, 8-88 to 8-89
 - graphics for 2-15, 5-6, 8-88
 - placing 8-90 to 8-91
 - composite window method for 6-51 to 6-53, 8-10
 - configuring as text window 14-11 to 14-14
 - context. *See* input context
 - data pointer, retrieving string with 8-46 to 8-47
 - defining
 - at runtime 8-93, 8-103 to 8-104
 - in a script 2-4, 2-76
 - display area 2-24 to 2-25, 8-15 to 8-16, 8-92
 - fields in 2-87
 - input fields
 - adding 8-95 to 8-102
 - number of, returning 8-48
 - input sets
 - adding 8-109
 - number of, returning 8-48
 - loading from library 8-62 to 8-64
 - local structure in 2-99, 8-105, 8-111 to 8-112
 - name size 2-76, A-2
 - paint character 8-106
 - placing 2-89
 - repository structure for 8-107 to 8-108
 - saving/restoring controls 8-82 to 8-83
 - terminating processing 1-13
- INPUT_LENGTH qualifier (.FIELD) 2-54
- INPUT_SNAPSHOT qualifier (.LISTCLASS) 2-83
- insert edit mode message 18-3
- integer fields 2-33, 8-29
- integers, ActiveX control properties as 4-24 to 4-25
- I/O 16-4
 - retrieving mode of opened file 16-15
 - wait time 8-33, 10-43 to 10-46
- .ism file extension 2-120
- ITALIC qualifier
 - .BORDER 2-9
 - .BOX 2-11
 - .FIELD 2-54
 - .FILL 2-73
 - .LINE 2-79
 - .TITLE 2-108
 - .WFIELD 2-114
- .ITEM script command 2-4, 2-5, 2-77 to 2-78
- items in lists. *See* list items
- ITEMS qualifier (.LISTCLASS) 2-83

J

- JPEG graphics 2-15
- justification
 - button set 5-25, 8-90 to 8-91
 - caption 7-39
 - input field text 2-55, 8-27, 8-97
 - line 2-104
 - menu column 10-2
 - menu column text 10-70
 - text 2-104, 16-122
- JUSTIFY qualifier (.FIELD) 2-55

K**key**

- assigning new for reserved function 18-23
- backspace 8-4, 18-22
- privatized (for lists) 9-123 to 9-124, C-1
- registering for ActiveX controls 4-33 to 4-35
- reserved 18-22 to 18-23
- sequence, adding alternate 18-16
- special, defining on menu bar 1-8
- unmapped 14-19, 14-20

key codes

- for ActiveX 4-33
- for I_INPUT 18-4
- for U_WAIT 16-149, 18-6
- virtual 9-123, 15-18, 16-169

key map file 18-12**key mapping 3-4, 18-11 to 18-23**

- enabling users to control 16-40 to 16-41
- files, moving to other systems 1-23
- reserved shortcuts, changing 7-72
- text for shortcuts, returning 10-36
- unmapped keys, returning 14-19, 14-20

key mapping method 7-72**key mapping script file 18-12, 18-19 to 18-20****KEY qualifier (.ENTRY) 2-27, 2-29****keyboard independence 1-13****keymap.ctl 18-12, 18-19 to 18-20****keymap.msw file 18-13****keymap.vt file 18-13****keys.dbl file 18-13, 18-21****keywords, searching (HTML Help) 16-73 to 16-74, 16-85 to 16-86****L****L_BORDER subroutine 9-54****L_BUTTON subroutine 9-55 to 9-61****L_BUTTONSET subroutine 9-62 to 9-63****%L_BUTTONSTATE function 9-64 to 9-65****L_CHR subroutine 9-66 to 9-67, 16-16****L_CLASS subroutine 9-68 to 9-71****%L_CLASSINFO function 9-72 to 9-74****L_CREATE subroutine 9-75 to 9-79****L_PROCESS and 9-112****L_STATUS and 9-142****option string for 2-84****overriding .LISTCLASS placement 2-85****L_CTRCONTEXT subroutine 9-80 to 9-81****L_DATA subroutine 9-82 to 9-84****L_DELETE subroutine 9-85****L_DISABLE subroutine 9-86****L_ENABLE subroutine 9-87****L_FINDSPEC subroutine 9-88 to 9-91****%LFINDWND and 9-92 to 9-93****retrieving field information 8-21, 8-39, 8-40****S_FIND menu entry and 9-131****%L_FINDWND function 9-92 to 9-94, 9-131****changing window placement 9-133****L_GLOBAL subroutine 9-95****%L_ICON function 9-96 to 9-97****L_INPFLD subroutine 9-98 to 9-100****L_INPUT subroutine 9-101 to 9-104****drilldown button processing 8-125****hyperlink processing 8-130****L_METHOD subroutine 9-105 to 9-106****L_NEXT subroutine 9-107 to 9-108****L_PLACE subroutine 9-109****L_PROCESS subroutine 9-111 to 9-117****called by S_FIND menu entry 9-131****canceled pending requests 9-122****context 9-83****double-clicking and 9-152****load method and 9-69, 9-105, 9-114****L_QUEUE subroutine 9-118 to 9-119****L_REMOVE subroutine 9-120****L_RESIZE subroutine 9-121****L_RESTART subroutine 9-122****L_RETURNKEY subroutine 9-99, 9-103, 9-123 to 9-124****L_SECT subroutine 9-125 to 9-126****L_SECTDRAW subroutine 9-127****L_SELECT subroutine 9-128 to 9-135****constructing find window for 9-92 to 9-94****L_CHR and 9-67****L_INPFLD and 9-99****load method and 9-69, 9-105****%L_SELSTYLE function 9-136 to 9-137****L_STATE subroutine 9-138****L_STATUS subroutine 9-139 to 9-143****%L_USER function 9-144****L_VIEW subroutine 9-146****L_PROCESS and 9-115****LARRIVE_METHOD subroutine 9-148****layout of screen display 1-7****LDBLCLICK_METHOD subroutine 9-151 to 9-153**

leading

- arrayed text fields, for 8-5
- for text windows 14-9, 14-14
- list or window, changing for 16-232
- retrieving in pixels 16-183

leave methods. *See* input methods; list methods

LEAVE_METHOD qualifier

.FIELD 2-55

.LISTCLASS 2-83

left arrow key. *See* arrow keys

left click events 4-41, 16-237

left double-click events 4-43 to 4-44, 16-237

LENGTH qualifier (.WFIELD) 2-114

length. *See* display length; input length; view length

library files. *See* window library

license string

for ActiveX control 4-28

for FarPoint Spread 4-36, 4-37

LightItemColor published property 9-19, 9-20

line

adding to menu column 10-68

blank 2-7, 2-111, 14-11, 18-20

color 2-79

drawing in window 2-79 to 2-80

list header/footer 9-127

name size A-2

selection window 12-6

.LINE script command 2-5, 2-79 to 2-80

line wrapping 18-7

linked lists F-2 to F-8

linking with tklib_qcheck.elb 1-14

linking with tklib.elb 1-14

list classes 9-5 to 9-6

creating 2-82 to 2-86, 9-68 to 9-71

default characteristics 9-77

default placement 2-89

loading with U_LDWND 16-92

maximum size of names for A-2

returning information about 9-72 to 9-74

list footer

ActiveX Toolkit lists, for

alignment 9-29, 9-43

font 9-30, 9-44

text 9-28, 9-42

adding a line to 9-127

placing text in 9-125 to 9-126

renditions for 16-118, 18-30

specifying 2-82

list header

ActiveX Toolkit lists, for

alignment 9-33, 9-47

font 9-34, 9-48

text 9-32, 9-46

adding a line to 9-127

placing text in 9-125 to 9-126

renditions for 16-118, 18-28

specifying 2-83

list icon, setting 9-96 to 9-97

list items

adding new 9-111

appearance 9-6

changing 9-70

color for (ActiveX Toolkit lists) 2-84, 2-86

deleting 9-112

highlighting C-1

input into (ActiveX Toolkit lists) 9-16

loading 9-11, 9-78, 9-122, 9-154

maintaining information for 2-83, 9-68

moving 9-112

non-window data for 9-75, 9-98, 9-101

number to be displayed 2-83

renditions for 9-113

list methods 9-8 to 9-9

ActiveX Toolkit list, published 9-22 to 9-49

arrive and leave methods 9-8, 9-148

double-click method 9-8, 9-151 to 9-153

information about, returning 9-72 to 9-74

load method 9-9, 9-154

registering 2-82 to 2-86, 9-68 to 9-71, 9-105 to 9-106

specifying as parameter. *See* L_METHOD subroutine;

.LISTCLASS script command

See also methods

.LIST script command 2-5, 2-81

list title 9-6

placing text in 9-125 to 9-126

renditions for 16-118, 18-28

.LISTCLASS script command 2-4, 2-5, 2-82 to 2-86

lists 1-10, 9-4 to 9-21

ActiveX. *See* ActiveX Toolkit lists

adding to tab set 13-14

associating with input windows 9-6

borders for, adding or removing 9-54

buttons on 9-55 to 9-61, 9-62 to 9-63

characteristics of (list class), specifying 2-82 to 2-86,
9-68 to 9-71

child. *See* child windows and lists

- composite window method for 6-54 to 6-56
- creating 9-75 to 9-79
- data (loaded list items) in, manipulating 9-82 to 9-84
- deleting 9-85, 9-128
- disabling 9-86
- double-clicking and 9-8
- drag bar for, adding or removing 9-54
- enabling 9-87
- environment levels for 7-17, 7-18
- extending 9-112
- find operation for 9-88 to 9-94
- footers. *See* list footer
- headers. *See* list header
- horizontal display area for, shifting 9-146
- input processing for 9-66 to 9-67, 9-98 to 9-100, 9-101 to 9-104
- items. *See* list items
- leading (vertical spacing) 16-232
- linked F-2 to F-8
- maximum number of A-2
- methods. *See* list methods
- mouse wheel and 2-85
- null (no loaded items) 2-84, 9-68
- placing 9-109
- privatized keys for, using 9-123 to 9-124
- processing 9-10 to 9-14, 9-111 to 9-117
- promoting to global 9-95
- queue commands for 9-118 to 9-119
- removing 9-120
- resizing 9-121
- restarting 9-122
- selected item, specifying 9-107 to 9-108
- selection processing 9-128 to 9-135
- state of, modifying 9-138
- status of, retrieving 9-139 to 9-143
- style of selected item in 9-136 to 9-137
- titles. *See* list title
- user data associated with 9-144, A-2
- LL_CLOSE subroutine F-3
- LL_OPEN subroutine F-4
- %LL_PROCESS function F-5 to F-7
- LL_SYNCH subroutine F-8
- LLEAVE_METHOD subroutine 9-148
- LLOAD_METHOD subroutine 9-154
- lnklst.def file F-2
- load methods. *See* list methods
- LOAD_METHOD qualifier (.LISTCLASS) 2-84

- loading
 - general window 16-92 to 16-93
 - input window 8-62 to 8-64
 - list class into memory 16-92 to 16-93
 - lists 9-9, 9-16, 9-154
 - menu column 10-3, 10-37 to 10-38, 10-62
 - rendition schemes 16-90 to 16-91, 16-125 to 16-126, 18-32
 - selection window 16-92 to 16-93
 - text window 16-92 to 16-93
 - toolbar button 15-6 to 15-8
 - user-defined string for input field 8-87
- local
 - channel, promoting to global 16-57
 - menu column
 - creating 10-61 to 10-64
 - placing, removing, and deleting 10-10
 - properties 4-4
 - structure, defining 8-111 to 8-112
 - window
 - logging 16-94
 - promoting to global 16-58
 - setting 8-105
- .LOCAL script command 2-5, 2-87, 2-127
- ..LOG script command 2-6
- logging
 - input window environment 8-103
 - script output 2-6
 - toolbar environment 15-32, 15-36
- logicals. *See* environment variables
- LOWER_CASE_ALLOWED qualifier (.FIELD) 2-69

M

- M_COLUMN subroutine 10-10 to 10-11
- M_DEFCOL subroutine 10-4, 10-8, 10-12
- M_DISABLE subroutine 10-4, 10-13 to 10-14
- M_ENABLE subroutine 10-4, 10-15 to 10-16
- %M_INFO function 10-17 to 10-35
- M_KEYTXT subroutine 10-3, 10-36
- M_LDCOL subroutine 10-8, 10-37 to 10-38
- M_PROCESS subroutine 10-8, 10-43 to 10-46
 - disabled entry 10-14
- M_RESET subroutine 10-47
- M_RPLENT subroutine 10-48
- M_SCINFO subroutine 10-50
- %M_SIGNAL function 10-7, 10-52 to 10-55, D-7
- %M_TEXT function 10-56
- M_USER subroutine 10-58

- mapping
 - coordinates between windows 16-180
 - key 1-23, 18-11
- MARGIN qualifier (.TEXT) 2-101
- margin, text in window 2-101
- MATCH qualifier (.FIELD) 2-36, 2-56
- maximize events
 - call extension for 4-45
 - creating a method for 16-237
- maximum values 2-62, A-1
- MB_BLANK subroutine 10-59
- MB_COLUMN subroutine 10-9, 10-61 to 10-64
- MB_END subroutine 10-65
- MB_ENTRY subroutine 10-9, 10-66
- MB_LINE subroutine 10-68
- MB_LIST subroutine 10-69
- MB_TEXT subroutine 10-70
- memory access errors, detecting 1-19
- menu 1-8, 10-2 to 10-9
 - information about, retrieving 10-17 to 10-35
 - navigating, valid characters 10-44
 - preventing flicker (example) 16-143
 - See also* menu column; menu entry; menu entry lists
- menu bar
 - disabling 10-47
 - enabling 3-2, 10-4 to 10-5
 - environments and 7-2
 - ownership of 1-12, 7-2
 - placing 16-10
 - processing 1-13
 - removing 10-5, 16-10
 - rendition 10-4
 - resetting 2-29
- menu column 1-8, 2-3
 - blank row, adding 2-8, 10-59
 - building at runtime 10-66
 - control argument size 10-62
 - creating
 - at runtime 10-61 to 10-64
 - in a script 2-20 to 2-21
 - multiple 10-38
 - default, setting 10-12
 - deleting 10-3, 10-10 to 10-11
 - disabling 10-4, 10-13 to 10-14, 10-61 to 10-64
 - enabling 10-4, 10-15 to 10-16
 - ending specification session 10-65
 - environments and 7-2
 - global vs. local 7-2, 10-3
 - header, information about 10-28
 - horizontal line, adding 10-68
 - information about, retrieving 10-17 to 10-35
 - justification of text in 10-70
 - list entry, creating 10-69
 - loading 10-3, 10-37 to 10-38, 10-62
 - maximum memory available for A-3
 - maximum number placed A-3
 - menu entries, number in 10-23, A-3
 - name size A-2
 - number on screen 16-136
 - ownership of 1-11, 7-2
 - placing 10-10 to 10-11, 18-4
 - limitations 2-20
 - Proto 3-7
 - T_EDIT and 14-5
 - U_EDIT and 16-37
 - pop-up. *See* pop-up menus
 - processing 2-28, 10-8
 - black-box 1-13, 1-18
 - quick-select character for 2-20, 10-61
 - rebuilding 10-47
 - removing 10-10 to 10-11
 - T_EDIT and 14-5
 - U_EDIT and 16-37
 - renditions 10-4
 - resetting 10-47
 - runtime generation 10-9, 10-61 to 10-64
 - selecting with arrow key 10-5
 - state of, returning 10-21
 - storing in window library 2-26
 - testing and viewing 3-5 to 3-7
 - text, adding 10-70
- menu entry 1-8
 - canceling 10-52 to 10-55
 - context 18-5
 - creating 2-3, 2-27 to 2-30, 10-66
 - CTRL+SHIFT+TAB 18-3
 - CTRL+TAB 18-3
 - customizing 3-7
 - default 10-8, 10-12
 - disabling 10-4, 10-13 to 10-14
 - multiple items 2-81, 10-3
 - enabling 10-4, 10-15 to 10-16
 - multiple items 2-81, 10-3
 - information on, retrieving 10-17 to 10-35, 10-50, 10-56
 - justification of 2-28

- list of 2-81, 10-30 to 10-33
- modifying and retrieving text 10-56
- name 2-28, A-3
- number in column 10-23, A-3
- pop-up 10-13, 10-15, 10-39 to 10-42
- replacing names with shortcut 10-48
- reserved. *See* reserved menu entry
- SHIFT+TAB 18-4
- shortcut. *See* shortcut
- signaling 10-7, 10-52 to 10-55
- simulating 10-7
- state of, returning 10-21
- TAB key 18-4
- translating 9-131
- unknown 1-13
- user text 2-28, 10-58, A-3
- utility (U_) 7-82 to 7-83, B-5
- menu entry lists
 - creating 2-81
 - name size A-3
 - retrieving information about 10-30 to 10-33
- menu processing 1-13, 10-8, 10-43 to 10-46, 16-16
- message
 - alert style vs. error style 16-148
 - customizing Proto's 3-7
 - displaying 16-95 to 16-97
 - flashing 16-47
 - Windows-style 16-102 to 16-105
 - error state for U_WAIT 18-6
 - forward edit mode 18-3
 - I_INPUT termination 18-4
 - insert edit mode 18-3
 - overstrike edit mode 18-3
 - reverse edit mode 18-3
 - splitting long 16-95
 - window for 2-116
- message box, Windows-style 16-102 to 16-105
- message file, moving to other systems 1-23
- message window
 - changing rendition 16-116 to 16-121
 - displaying 16-95 to 16-97
- MessageBeep sound (Microsoft Windows) 18-2
- METHOD qualifier (.BUTTON) 2-13
- methods D-2 to D-11
 - ActiveX controls, for 4-17 to 4-19, 4-39 to 4-61
 - application start-up method 16-236
 - asynchronous D-2
 - button 2-14, 5-26
 - composite windows, for 6-36 to 6-37, 6-38 to 6-39, 6-46 to 6-66
 - environment. *See* environment methods
 - help processing 7-73 to 7-79
 - input. *See* input methods
 - list. *See* list methods
 - %M_SIGNAL and D-10 to D-11
 - mouse event. *See* mouse events
 - name size maximum A-3
 - signaling 10-52 to 10-55
 - synchronous D-2
 - toolbar button 15-7, D-2
- middle click events
 - call extension for 4-46 to 4-47
 - creating a method for 16-237
- middle double-click events
 - call extension for 4-48 to 4-49
 - creating a method for 16-237
- minimize events, call extension for 4-50
- MINIMIZE_LEADING environment variable 8-5
- minimum value, defining for input field 2-62
- minus sign (-) 2-57
- mnemonic, function code 10-36
- monitor display area 16-185, 16-186
- months
 - abbreviation for, obtaining 7-22
 - alpha characters, representing as 16-101
 - number for, obtaining 7-23
 - See also* dates
- mouse cursor, position of 16-167
- mouse events
 - address of method set for, returning 16-211
 - and ActiveX controls, methods for 4-40 to 4-61
 - assigning methods to 16-203 to 16-209
 - child windows and lists, methods for 6-9
 - creating methods for 16-205 to 16-208, 16-237 to 16-240
 - handlers for .NET forms 11-4
 - method set ID for, returning 16-212
 - methods for toolbar buttons, returning 15-18
 - programmatically triggering methods for 16-210
 - simulating for toolbar button 15-10
- mouse wheel scrolling (lists) 2-85
- movable views. *See* display area
- move events
 - call extension for 4-51 to 4-52
 - creating a method for 16-237
- move method (application) 7-46

moving

- applications to other systems 1-23
- customized files 1-23
- library files 1-23
- message files 1-23
- method for (application) 7-46
- windows 16-205

multi-line text fields 2-32

- setting/getting input position 8-84 to 8-86

N

names reserved by Toolkit

- DTK_ and GLST_ 1-16
- names with asterisks 8-13, 8-43, 8-49, 8-65

naming overloadable routines 18-8

NEGATIVE qualifier (.FIELD) 2-56

negative values 2-56

.NET Framework, using Toolkit code with 1-21

- embedding forms and controls 11-2 to 11-14

NOALLOW qualifier (.FIELD) 2-36

NOARRIVE_METHOD qualifier (.FIELD) 2-37

NOAUTOMATIC qualifier (.FIELD) 2-37

NOBLANK_IF_ZERO qualifier (.FIELD) 2-38

NOBLANK_IFZERO qualifier (.FIELD) 2-38

NOBLINK qualifier

- .BORDER 2-9
- .BOX 2-11
- .FIELD 2-38
- .FILL 2-73
- .LINE 2-79
- .TITLE 2-108
- .WFIELD 2-114

NOBOLD qualifier

- .BORDER 2-9
- .BOX 2-11
- .FIELD 2-38
- .FILL 2-73
- .LINE 2-79
- .TITLE 2-108
- .WFIELD 2-114

NOBREAK qualifier (.FIELD) 2-39

NOBZRO qualifier (.FIELD) 2-38

NOCASE qualifier (.FIELD) 2-56

NOCHANGE_METHOD qualifier (.FIELD) 2-40

NOCHECKBOX qualifier (.FIELD) 2-40 to 2-41

NOCOLOR qualifier (.FIELD) 2-41

NODATE qualifier (.FIELD) 2-42

NODECIMAL qualifier (.FIELD) 2-57

NODECIMAL_REQUIRED qualifier (.FIELD) 2-57

NODEFAULT qualifier (.FIELD) 2-44

NODISABLED qualifier (.FIELD) 2-45

NODISPLAY_METHOD qualifier (.FIELD) 2-46

NODRILL_METHOD qualifier (.FIELD) 2-46

NODSP_LENGTH qualifier (.FIELD) 2-45

NOECHO qualifier (.FIELD) 2-58

NOEDITFMT_METHOD qualifier (.FIELD) 2-47

NOENUMERATED qualifier (.FIELD) 2-48 to 2-49

NOEXACT qualifier (.FIELD) 2-56

NOFIELD_POSITION qualifier (.FIELD) 2-49

NOFMT qualifier (.FIELD) 2-51

NOFONT qualifier (.FIELD) 2-50, 18-47

NOFORMAT qualifier (.FIELD) 2-51

NOFPOS qualifier (.FIELD) 2-49

NOHELP qualifier (.FIELD) 2-52

NOHIGHLIGHT qualifier

.BORDER 2-9

.BOX 2-11

.FIELD 2-52

.FILL 2-73

.LINE 2-80

.TITLE 2-108

.WFIELD 2-115

NOHYPERLINK_METHOD qualifier (.FIELD) 2-52

NOINFORMATION qualifier (.FIELD) 2-53

NOINPUT_LENGTH qualifier (.FIELD) 2-54

NOITALIC qualifier

.BORDER 2-9

.BOX 2-11

.FIELD 2-54

.FILL 2-73

.LINE 2-79

.TITLE 2-108

.WFIELD 2-114

NOJUSTIFY qualifier (.FIELD) 2-55

NOLEAVE_METHOD qualifier (.FIELD) 2-55

NONEGATIVE qualifier (.FIELD) 2-56

NonGridBackColor published property 9-19

NONULL qualifier (.LISTCLASS) 2-84

non-window data, searching 9-89

NOPAD qualifier (.SELECT) 2-95

NOPAINT qualifier (.FIELD) 2-59

NOPOSITION qualifier (.FIELD) 2-59 to 2-60

NOPROMPT qualifier (.FIELD) 2-60

NOPROMPTFONT qualifier (.FIELD) 2-60, 18-47

NORADIO qualifier (.FIELD) 2-61

NORANGE qualifier (.FIELD) 2-62

NOREADONLY qualifier (.FIELD) 2-63
 NOREQUIRED qualifier (.FIELD) 2-63
 NORESET qualifier (.ENTRY) 2-27, 2-28
 NORETAIN_POSITION qualifier (.FIELD) 2-63
 NOREVERSE qualifier
 .BORDER 2-9
 .BOX 2-11
 .FIELD 2-64
 .FILL 2-73
 .LINE 2-80
 .TITLE 2-108
 .WFIELD 2-115
 NOSELECT qualifier (.FIELD) 2-64 to 2-65
 NOSELWND qualifier (.FIELD) 2-66
 NOTERM qualifier (.FIELD) 2-58
 NOTIME qualifier (.FIELD) 2-67 to 2-68
 NOUC qualifier (.FIELD) 2-69
 NOUNDERLINE qualifier
 .BORDER 2-9
 .BOX 2-12
 .FIELD 2-69
 .FILL 2-73
 .LINE 2-80
 .TITLE 2-108
 .WFIELD 2-115
 NOUPPER_CASE_ONLY qualifier (.FIELD) 2-69
 NOUSER qualifier (.FIELD) 2-70
 NOVIEWS_LENGTH qualifier (.FIELD) 2-71
 null list 2-84
 null pointer 8-47
 numeric bar, loading 15-2
 numeric fields. *See* decimal fields; implied-decimal fields; integer fields

O

object class, setting fonts 18-46, 18-48
 OCX files (for ActiveX controls) 4-28, 4-30, 4-37
 ONERROR statements 1-18, 4-8
 See also Synergy DBL Language Reference Manual
 online help. *See* help
 OnOriginChange published method 9-50
 OPEN statement (Synergy DBL) 16-106
 OPTIONS qualifier (.LISTCLASS) 2-84
 options, searching command line for 16-19 to 16-20
 OriginColumn published property 9-19
 overloadable routines 18-8 to 18-10
 overstrike mode, message for 18-3

P

padding, suppressing in selection window 2-95, 12-5
 paint character 2-59, 2-88, 8-106
 PAINT qualifier (.FIELD) 2-59
 .PAINT script command 2-5, 2-59, 2-88
 Palette Entry dialog box 18-37, 18-38
 palette selection window, displaying 16-134 to 16-135
 palettes. *See* color palette entries; font palette
 parent/child windows. *See* composite windows
 password, hiding display of 2-58
 PC terminal 18-14
 period-style date, maximum value 2-43
 .PLACEMENT script command 2-5, 2-89
 placing
 button set 8-90 to 8-91
 child windows and lists 6-3
 composite windows 6-3
 list 9-109
 menu column 10-10 to 10-11
 selection window 8-37 to 8-38
 toolbars 15-37
 windows 8-63, 16-159 to 16-161
 platform, moving application to different 1-23
 PNG graphics 2-15
 point size 18-40
 pointer, retrieving data string 8-46 to 8-47
 pop-up Help
 HTML Help 16-78
 WinHelp 16-163
 pop-up menus
 disabling entries in 10-13
 enabling entries in 10-15
 processing 10-39 to 10-42
 pop-up windows 16-109 to 16-110
 position
 child window or list 6-8
 column (window) 16-177
 composite window 6-8
 display area 8-92
 field 2-59 to 2-60, 2-99
 input 16-49
 left click 4-41
 left double-click 4-43
 middle click 4-46
 middle double-click 4-48
 mouse cursor 16-167
 move event 4-51
 right click 4-54

- right double-click 4-56
 - row (window) 16-184
 - text in window 2-101
 - toolbar 15-26, 15-32
 - toolbar button 15-6, 15-25, 15-29
 - window 2-89
 - POSITION qualifier
 - .BOX 2-12
 - .FIELD 2-50, 2-59 to 2-60
 - .LINE 2-79
 - .TEXT 2-101
 - .WFIELD 2-114
 - Print Setup dialog box 16-113
 - printer setup dialog, displaying
 - (Windows) 16-113 to 16-114
 - printer, retrieving information about
 - (Windows) 16-111 to 16-112
 - printing on Windows 16-113 to 16-114
 - privatized keys 9-123 to 9-124, C-1
 - processing
 - abort 16-5 to 16-7
 - alert 16-95, 16-109
 - arrive 8-114 to 8-117
 - black box 1-12 to 1-13
 - break 8-12, 8-20
 - composite window 6-42 to 6-45
 - drilldown button 8-125 to 8-126
 - edit 8-5
 - errors 1-18
 - forced data 8-41
 - hyperlink 8-130 to 8-131
 - input field 8-4, 8-51 to 8-54
 - input set 8-55 to 8-61, 8-78
 - I/O, waiting for 10-43 to 10-46
 - leave 8-132 to 8-134
 - list 9-111 to 9-117
 - menu 10-8, 10-43 to 10-46
 - rendition 16-116 to 16-121
 - termination 8-55 to 8-61, 8-67
 - text field 8-5
 - process-menu key 10-4 to 10-5, 18-22
 - programs. *See* applications
 - prompt
 - defining 2-60
 - find dialog boxes 9-93
 - font 18-47
 - hyperlink 8-130 to 8-131, 16-156
 - loading 9-114
 - variable 8-69
 - PROMPT qualifier (.FIELD) 2-50, 2-60
 - PROMPTFONT qualifier (.FIELD) 2-60
 - properties
 - for ActiveX containers 4-4
 - for .NET forms 11-4
 - retrieving for ActiveX controls 4-22, 4-24 to 4-25
 - setting for ActiveX controls 4-26
 - Properties button 16-114
 - proportional fonts 18-49
 - window 2-75, 16-231
 - See also* fonts
 - Proto 1-5, 3-1 to 3-7
 - changing renditions 18-32
 - compiling script files 1-4, 3-5
 - U_EDITKEYS and 16-40
 - published events (ActiveX Toolkit list) 9-50
 - published methods (ActiveX Toolkit list) 9-22 to 9-49
 - published properties (ActiveX Toolkit list) 9-17 to 9-21
 - purging channels
 - when exiting environments 7-5
 - when exiting Toolkit 16-46
 - when starting Toolkit 16-137
 - with U_CLOSE 16-18
- Q**
- qcheck compiler option 1-19
 - qstrict compiler option 1-19
 - qualifier
 - input field 1-9, 2-31 to 2-72, 8-3, 8-25 to 8-35
 - list class 2-82 to 2-85
 - queuing requests for lists 9-118 to 9-119
 - quick-select character 10-5 to 10-6
 - buttons 2-13, 5-31, 8-88 to 8-89
 - default 2-78, 10-57
 - menu columns 2-20, 10-61
 - menu entries 2-27, 10-67
 - modifying 10-56
 - renditions 10-6
 - retrieving for a menu entry 10-25
 - selection windows 2-77 to 2-78, 12-5, 12-17
 - terminating input with 8-8
 - See also* shortcut

R

radio buttons

- default selection 2-77, 8-96, 12-8
- defining 2-61, 8-97
- display methods and 2-46
- edit format methods and 2-47
- lists and 9-103
- validating 2-65, 7-43
- width of, returning 16-175

RADIO qualifier (.FIELD) 2-61

RANGE qualifier (.FIELD) 2-62

READONLY qualifier (.FIELD) 2-63

read-only state, setting for input fields 8-27,
8-72 to 8-76, D-3

records

- appending to current file E-5
- linked list and F-2
- reading from logical files E-12
- writing to logical files E-13

redisplay format 2-51

redrawing screen 16-115

registering

- ActiveX control keys 4-33 to 4-35
- ActiveX controls 4-3, 4-15, 4-30, 4-37
- composite window event method
 - extension 6-36 to 6-37
- environment methods 7-35 to 7-37
- event handlers for .NET forms 11-4
- event methods 2-82 to 2-86, 9-68 to 9-71,
9-105 to 9-106
- list methods 9-105 to 9-106
- routine for overloading a Toolkit routine 18-8
- routine for script compilation errors 7-80 to 7-81

registering ActiveX controls 4-9

regsvr32.exe utility 4-9

removing

- child windows and lists 6-3
- composite windows 6-3
- information line 16-10
- list 9-120
- menu bar 16-10
- menu column 3-7, 10-10 to 10-11
- rendition schemes 16-44, 18-32, 18-36
- scroll bar 16-159 to 16-161
- tab in tab set 13-22
- toolbars 15-37
- window 3-7, 16-159 to 16-161

rendition file 18-33

rendition schemes 18-31 to 18-39

- creating 18-32
- enabling users to create, modify, delete, save 18-32
 - using U_EDITREND 16-42 to 16-44
 - using U_MODREND 16-99 to 16-100
- loading 16-90 to 16-91, 16-125 to 16-126, 18-32
- modifying 16-116 to 16-121, 18-32
- names, maximum length for A-3
- removing 18-32
- saving 16-132 to 16-133, 16-139 to 16-140, 18-32

renditions 18-25 to 18-39

- boxes 2-11 to 2-12
- button area for button set 9-63
- defaults (set by U_START) 18-25 to 18-30
- editing

- with Proto 3-3
- with the Renditions window
(Windows) 18-36 to 18-37
- with the Select Screen Area screen (UNIX,
OpenVMS) 18-38 to 18-39
- with U_EDITREND 16-42 to 16-44, 18-36
- with U_MODREND 16-99 to 16-100
- with U_REND 16-116 to 16-121

general window fields 2-114 to 2-115

help window 16-70 to 16-71

input fields

- defining for 2-31 to 2-72, 8-95 to 8-102
- modifying at runtime for 8-25 to 8-35
- overriding for 8-122 to 8-124, 8-127 to 8-129

menu 2-103, 10-4, 10-6

resetting 16-89

selection windows 12-3

tab 13-13, 13-23 to 13-25

text in windows 2-101 to 2-107

text windows 14-12 to 14-14

user-assigned 18-33 to 18-35

window borders 2-9 to 2-10

window fill character 2-73 to 2-74

window lines 2-79 to 2-80

window titles 2-108 to 2-110

See also color; color palette entries; fonts; rendition
schemes

Renditions window (Windows) 18-36 to 18-37

repainting ActiveX Toolkit lists, forcing 9-36

- repository
 - converting a script to 2-125 to 2-127
 - fields from, including 2-33, 2-91 to 2-93, 8-107 to 8-108
 - fonts, setting with 18-46, 18-48
 - repository field, overriding font 18-47
 - .REPOSITORY_STRUCTURE script command 2-5, 2-91 to 2-93
 - .FIELD and 2-33
 - scribl and 2-127
 - REQUIRED qualifier (.FIELD) 2-63
 - reserved colors 18-24
 - reserved keys
 - assigning to other functions 18-23
 - customizing 18-22 to 18-23
 - reserved menu entry 1-13, 8-4, 16-17, B-1 to B-5
 - help 16-17
 - input processing 8-52, 8-53, 8-58
 - key mapping 16-40
 - scrolling 14-20
 - selection window processing 12-11
 - text processing 14-5, 16-37
 - reserved names
 - DTK_ and GLST_ 1-16
 - names with asterisks 8-13, 8-43, 8-49, 8-65
 - resizing
 - applications 1-20, 7-47, 16-230
 - fonts 16-225, 16-230
 - lists 9-121
 - screen 16-122
 - scroll bars 16-123
 - windows 16-205
 - restarting list 9-122
 - restore events
 - call extension for 4-53
 - creating a method for 16-237
 - restoring
 - environment 7-2
 - input window controls 8-82 to 8-83
 - renditions 16-125 to 16-126
 - state for toolbars 15-37
 - user-modifiable renditions schemes 18-32
 - window contents 9-112
 - RETAIN_CONTEXT_CHANGE_ON_SIGNAL
 - environment variable 18-5
 - RETAIN_POSITION qualifier (.FIELD) 2-63
 - RETURN key. *See* ENTER key
 - RETURN_KEY completion code 18-4, 18-6
 - returning to previous environment 7-5
 - reverse edit mode message 18-3
 - REVERSE qualifier
 - .BORDER 2-9
 - .BOX 2-11
 - .FIELD 2-64
 - .FILL 2-73
 - .LINE 2-80
 - .TITLE 2-108
 - .WFIELD 2-115
 - reverse video. *See* renditions
 - RGB values 9-23
 - right arrow key. *See* arrow keys
 - right click events
 - call extension for 4-54
 - creating a method for 16-237
 - right double-click events
 - call extension for 4-56 to 4-57
 - creating a method for 16-237
 - RIGHT qualifier 2-17
 - routines for ActiveX controls 4-11 to 4-35
 - RowMode published property 9-19
 - rows
 - in display 7-24, 16-184, 16-201, 16-202
 - number available in display section 7-24
 - number needed to span pixels 16-150
 - retrieving height of 16-182
 - space between 8-5, 16-183, 16-232
 - RPSDAT environment variable 2-92
 - rpsmain.ism file 2-92
 - RPSMFIL environment variable 2-92, 8-107
 - rpstext.ism file 2-92
 - RPSTFIL environment variable 2-92, 8-107
 - runtime
 - minimum version 16-98
 - returning handle for 16-168
 - runtime terminal binding 18-11
- S**
- S_SELBLD subroutine 12-5 to 12-7
 - enumerated field and 2-48
 - S_SELDFLT subroutine 12-8
 - overriding defaults 2-78, 12-6
 - S_SELECT subroutine 12-10 to 12-12
 - .ITEM and 2-77
 - overriding defaults 2-78, 12-6
 - .PLACEMENT and 2-89

- S_SELECTCB subroutine 12-13, 18-2
 - .ITEM and 2-77
- S_SELINF subroutine 12-15
- S_SELLD subroutine 12-17
- S_UPDATECB subroutine 12-18, 18-2
 - .ITEM and 2-77
- saving
 - application window settings to file 16-130
 - initialization settings to file 16-128
 - input window controls 8-82 to 8-83
 - rendition schemes 16-132 to 16-133, 16-139 to 16-140
 - state for toolbars 15-37
- scaling percentage 16-225, 16-230
- %SCR_CLOSELIBRARY function 2-121
- %SCR_ERRORCOUNT function 2-122
- %SCR_OPENLIBRARY function 2-123
- %SCR_PROCESS function 2-124
- screen 1-7
 - body 1-12, 7-2
 - clearing 16-46, 16-115
 - defining at start-up 1-7, 16-136 to 16-138
 - redraw key, customizing 18-22
 - redrawing 16-115
 - renditions for. *See* renditions
 - resizing 16-122
 - updating 16-143 to 16-144
 - See also* application window; dialog boxes and screens
- scrldl utility 2-125 to 2-127
- script commands 2-7 to 2-116
- script compilation error method 7-80 to 7-81
- script compiler, accessing from an
 - application 2-120 to 2-124
- script conversion utility 2-125 to 2-127
- script file 1-4 to 1-5, 2-3 to 2-6
 - advantages of 1-5
 - alternative to 8-6
 - blank lines in 2-7
 - comments in 2-7
 - compiling 2-118 to 2-120, 3-5
 - continuation lines 2-7
 - converting to repository 2-125 to 2-127
 - examples 2-117
 - output logging 2-6
 - processing 2-103
 - setting font in 18-46
- .SCRIPT script command 2-94
- Script utility 1-4, 1-5
 - accessing with external interface 2-120 to 2-124
 - compiling with 2-118 to 2-120
 - maximum line size A-4
 - running automatically 2-119
- scroll bar 16-4
 - borders 16-23
 - copying and loading characters 16-12 to 16-13
 - creating 16-22 to 16-23
 - fields (multi-line) and 8-6
 - height of horizontal, returning 16-179
 - list 2-85
 - name size A-3
 - progress/regress indicator 16-145
 - renditions for 16-119, 18-29
 - requirements for 9-106
 - resizing 16-123
 - size 16-22
 - suppressing 2-85
 - text window 14-4, 14-19
 - updating 16-145 to 16-146
 - width of vertical, returning 16-192
- scroll events
 - call extension for 4-58 to 4-59
 - creating a method for 16-237
- scrolling
 - lists with mouse wheel 2-85
 - text fields 2-45, 8-6
- Search tab for HTML Help, displaying 16-77
- searching
 - Alink names, using (HTML Help) 16-73 to 16-74
 - command line for options 16-19 to 16-20
 - for list items. *See* find operations; find window
 - for loaded menu columns 10-37 to 10-38
 - for loaded windows 8-63
 - HTML Help index entries 16-85 to 16-86
- SelBackColor published property 9-19, 9-20
- SELECT qualifier
 - .BUTTON 2-13
 - .COLUMN 2-20
 - .ENTRY 2-27, 2-30
 - .FIELD 2-64 to 2-65
 - .ITEM 2-77
- Select Scheme window (UNIX and OpenVMS) 18-39
- .SELECT script command 2-5, 2-95 to 2-96
 - multiple columns and 2-8, 2-102

- selecting
 - from a menu 10-4 to 10-8
 - from a selection window 12-3
- selection character 2-13, 8-88 to 8-89
- selection processing 9-10, 9-128 to 9-135, 12-2 to 12-4
- selection window 1-10, 12-2 to 12-4
 - assigning to input field 2-66, 2-95, 8-37 to 8-38
 - blank row, adding 2-8
 - border 12-3
 - check box style
 - creating 12-3 to 12-4
 - display character 2-77, 12-3 to 12-4, 12-13
 - processing 12-13
 - updating 12-18
 - composite window method for 6-61 to 6-62
 - creating 2-4, 2-95 to 2-96, 12-5 to 12-7
 - entry. *See* selection window entry
 - global status, promoting to 16-58
 - information about, retrieving 12-15
 - justification of text in 2-78
 - line in 12-6, 12-17
 - list vs. 12-2
 - loading 12-17, 16-92 to 16-93
 - logging 12-6, 16-94
 - margins 12-6
 - menu column vs. 12-2
 - multiple column 2-102, 12-6
 - name 2-95, A-3
 - navigating 8-4, 12-2, 12-6
 - non-selectable text in 12-17
 - placing 2-64 to 2-66, 2-89
 - processing 12-2 to 12-4, 12-10 to 12-12
 - renditions 12-3
 - replacing text in 12-17
 - runtime building 12-2
 - selecting from 12-3
 - subroutines for 12-5 to 12-18
 - text in 2-102
 - width of 2-95
- selection window entry
 - blank padding in, suppressing 2-95, 12-5
 - default 2-77, 8-96, 12-6, 12-8
 - defining 2-4, 2-77 to 2-78
 - justification of 2-77
 - loading into window 12-17
 - quick-select character 2-77 to 2-78, 12-17
 - replacing 12-17
 - returning selected 12-2
 - selecting 12-3
- SelfForeColor published property 9-19, 9-20
- SELWND qualifier (.FIELD) 2-66
 - overriding .PLACEMENT 2-89
- SELECT vs. 2-66
- set. *See* input set
- .SET script command 2-5, 2-97
 - referencing aliases 2-92
- SetCellColor published method 9-20, 9-37
- SetCellFont published method 9-20, 9-39
- SetColumnWidth published method 9-41
- SetFooterCell published method 9-42
- SetFooterCellAlignment published method 9-43
- SetFooterCellFont published method 9-44
- SetHeaderCell published method 9-46
- SetHeaderCellAlignment published method 9-47
- SetHeaderCellFont published method 9-48
- setinf.def file 8-81
- SFWINIPATH logical 16-128, 16-129, 16-130
- shared image 17-3
- SHIFT+TAB
 - ActiveX controls and 4-13
 - arrive methods and 8-115
 - buttons and 2-14
 - composite windows and 6-4 to 6-5
 - leave methods and 8-132
 - lists and 9-4, 9-102, 9-132, C-5
 - menu entry for 18-4
 - radio buttons and 2-61
 - selection windows and 12-12
 - text windows and 14-5, 16-38
- shortcut 10-3
 - changing at runtime 18-19 to 18-20
 - defining 2-27, 2-29, 10-66 to 10-67
 - information on, retrieving 10-36, 10-50
 - invalid key sequences 2-29, 10-7
 - processing 1-13, 16-16
 - replacing menu entry name with 10-48
 - retrieving for a menu entry 10-25
 - retrieving text 10-36
 - See also* quick-select character
- simulating menu entry 10-7
- sizing character 18-40
 - height, retrieving 16-173
 - too wide or too narrow 18-50
 - width, retrieving 16-174
- snapshot, input window controls 8-82 to 8-83

- space, adding to end of line in script file 2-105
- spacing between lines. *See* leading
- specifications, field 2-35 to 2-36
- start-up method 16-236
- start-up processing 16-136 to 16-138, 16-236
- state
 - current environment 7-41 to 7-44
 - flags 7-2
 - method 7-49
 - of menu column, returning 10-21
 - of menu entry, returning 10-21
 - toolbars, saving or restoring for 15-37
- static display area 2-83
- status
 - messages, standardizing 16-4
 - of lists, retrieving 9-139 to 9-143
- .STD_BUTTONS. *See* .BUTTON; .BUTTON_SET
- STOP chain restriction 1-20
- strings
 - data pointers, retrieving with 8-46 to 8-47
 - date 16-26 to 16-28, 16-51 to 16-52
 - fields, populating with 8-41 to 8-42
 - fill, for windows 2-73 to 2-74
 - format strings for fields. *See* format string
 - information strings for fields 2-53
 - license 4-28, 4-36, 4-37
 - menu entries, for 2-28, 2-30, 10-58
 - months, for 16-101
 - online help, for. *See* help
 - searching 9-88 to 9-91
 - time 16-29 to 16-30, 16-53 to 16-54
 - toolbar buttons, for 15-7, 15-18
 - user text. *See* user text for input fields
 - version number, parsing 16-108
- structfields 2-34
- structure
 - alias 2-91 to 2-92, 8-107 to 8-108
 - creating 2-99 to 2-100, 8-7
 - default 2-100
 - defining in input window 8-111 to 8-112
 - function of 2-99
 - maximum number in input window 2-99
 - maximum size 2-100, A-3
 - name size A-2
 - repository 2-91, 8-107 to 8-108
 - retrieving field position 8-40
- .STRUCTURE script command 2-5, 2-99 to 2-100
 - created automatically 2-91
 - example for find window 9-134
 - scriidl and 2-125
- SUB qualifier (.ENTRY) 2-27, 2-30
- submenu
 - creating 10-2
 - disabling 10-13 to 10-14
 - enabling 10-15 to 10-16
 - navigating 10-5
 - symbol for 10-2, 18-19
 - user string, accessing 10-58
- submenu flag 10-2, 18-19
- subroutine 1-6
 - button 5-1 to 5-18
 - customizing 18-8 to 18-10
 - environment 7-1 to 7-83
 - input 8-8 to 8-112
 - menu 10-2 to 10-70
 - overloadable 18-8 to 18-10
 - selection 12-2 to 12-18
 - tab set 13-2 to 13-37
 - text 14-2 to 14-23
 - toolbar 15-2 to 15-39
 - utility 16-4 to 16-240
- supplied methods for ActiveX controls 4-39 to 4-61
- SYN_3D_TOOLBAR environment variable 15-2
- SYN_RESIZE_SCALE environment variable 1-20, 1-21
- SYN_TRANSPARENCY_THRESHOLD environment variable 2-15
- SYN_TRANSPARENT_COLOR environment variable 2-15
- SYN_CENTURY environment variable 7-51
- synchronous callbacks 1-20
- synchronous methods D-2
- Synergy colors 18-24 to 18-25
 - default 18-24
 - RGB values for, returning 16-154 to 16-155
 - setting 16-158, 18-25
- Synergy Data Language 18-46
- Synergy Data Language file, created by scriidl
 - utility 2-125 to 2-127
- Synergy runtime
 - minimum version 16-98
 - returning handle for 16-168

Synergy UI Toolkit Control Panel
 changing renditions 18-32
 key mapping 18-11
 moving text files 1-23
 synergy.ini file. *See* initialization files; *Environment Variables & System Options*
 SYNTXT environment variable 16-137
 syntxt.ism file 3-7, 9-93, 9-131, 16-137
 synuser.ini file. *See* initialization files; *Environment Variables & System Options*
 system colors 18-24

T

T_EDIT subroutine 14-3 to 14-6
 T_GETLIN subroutine 14-7, 14-23
 T_PUTTXT subroutine 14-8 to 14-10
 T_SETUP subroutine 14-9, 14-11 to 14-14
 calling before T_EDIT 14-4
 calling before T_PUTTXT 14-9
 T_TXTRD subroutine 14-15, 14-17
 T_TXTWRT subroutine 14-17
 T_VIEW subroutine 14-19 to 14-21
 T_WNDRD subroutine 14-22
 T_WNDWRT subroutine 14-23
 TAB key
 ActiveX controls and 4-13
 arrive methods and 8-115
 buttons and 2-14
 composite windows and 6-4 to 6-5
 leave methods and 8-132
 lists and 9-4, 9-102, 9-132, C-3
 menu entry for 18-4
 radio buttons and 2-61
 selection windows and 12-12
 text windows and 14-5, 16-38
 tab set
 attributes
 defaults 13-28
 for non-selected tab 13-26 to 13-27
 for selected tab 13-23 to 13-25
 button sets in 13-17
 buttons in 5-4 to 5-13, 13-15
 composite window method for 6-63 to 6-64, 13-36
 creating 13-2, 13-7 to 13-11
 deleting 13-10
 disabling tab 13-21
 enabling tab 13-20
 ID, returning 13-35

index, returning 13-33
 list, adding 1-10, 13-14
 name size A-3
 object on tab, returning information on 13-34
 processing 1-10, 13-2 to 13-6
 renditions 13-13, 13-23 to 13-25, 13-26 to 13-27
 separators 13-9 to 13-11, 13-28
 styles 13-9 to 13-11, 13-28
 tabs
 height of, returning 16-187
 number of, returning 13-32
 removing 13-22
 width of, returning 16-188
 windows in
 adding 1-10, 13-12
 distance to 16-189, 16-190
 tabbed dialog. *See* tab set
 tabbing indices for child windows and lists 6-4 to 6-6, 6-28
 TABSET_STYLE environment variable 13-10, 13-30
 %TB_BUTTON function 15-3 to 15-14
 %TB_INFO function 15-15 to 15-31
 %TB_TKCREATE function 15-32
 TB_TKDEL routine 15-34
 TB_TKGLOBAL routine 15-35
 TB_TKLOG routine 15-36
 %TB_TOOLBAR function 15-37 to 15-39
 template, font 18-46
 TERM environment variable 18-14
 TERM qualifier (.FIELD) 2-58
 terminal
 bell. *See* bell (terminal)
 closing 16-46
 deleting definition 18-18
 independence 1-13
 input, processing 8-55 to 8-61
 key mapping types 18-14
 unknown 18-14
 Terminal font 18-43, 18-48
 termination processing 8-55 to 8-61, 8-67
 testing
 key mapping 18-21
 windows and menu columns 3-5 to 3-7
 text
 adding to window or menu column 2-101 to 2-107
 blank lines between paragraphs 14-11
 editing 8-4, 14-3 to 14-6, 16-34 to 16-39
 field. *See* text field

- file, moving to other systems 1-23
- filling line 2-104
- getting from text message library 16-67 to 16-68
- indenting 14-11 to 14-14
- justification of 2-55, 2-102, 2-104
 - after window resized 16-122
 - forcing with \n 2-105
- library. *See* text library
- line length in script file 2-102
- loading into window 2-102 to 2-107, 14-8 to 14-10
- margins 2-101, 14-11 to 14-14
- moving to new line 2-105
- object name size A-3
- placing in header, footer, information
 - line 7-38 to 7-40
- positioning and displaying 2-102 to 2-107
- processing, overview 1-11
- reading from window library 14-15
- renditions 2-103
- return for display section 7-25
- subroutines 14-2 to 14-23
- trailing blanks, stripping 14-9, 14-12, 14-17
- unselectable 2-102
- window. *See* text window
- wrapping 2-106, 14-9, 14-12
- text field 8-5
 - edit processing 8-5
 - justification of 2-55
 - multi-line 2-32
 - setting/getting input position 8-84 to 8-86
 - retaining position in 2-63
 - scrolling 2-45, 8-6
 - space used by 2-100
 - word wrap 8-5
- text library
 - getting text from 16-67 to 16-68
 - loading and modifying entries 16-141 to 16-142
 - reading from 14-17, 14-22
 - storing data in 14-23
- TEXT qualifier
 - .BUTTON 2-14
 - .WFIELD 2-114
- .TEXT script command 2-5, 2-101 to 2-107
- text subroutines 14-2 to 14-23
- text window 1-11
 - composite window processing method
 - for 6-65 to 6-66
 - creating 2-116
 - display area 2-24 to 2-25, 14-4, 14-12
 - editing text in 14-3 to 14-6, 16-34 to 16-39
 - formatting text in 14-9
 - getting text from 14-7
 - help for 14-3, 14-19, 16-35
 - line breaks in 14-8
 - loading 14-22, 16-92 to 16-93
 - margins 14-11 to 14-14
 - placing 2-89, 14-11
 - putting text into 14-8 to 14-10
 - reconfiguring another window as 14-12
 - renditions, setting 14-12 to 14-14
 - scrolling 14-4, 14-19, 14-21
 - setting controls for 14-11 to 14-14
 - size 14-12
 - space between lines in 14-9, 14-14
 - storing contents into library 14-23
 - terminating processing 1-13
 - view, moving 14-19 to 14-21
 - word wrapping in 14-8, 14-12
- TextColor published property 9-20
- threshold for transparent graphics 2-15
- TIFF graphics 2-15
- time
 - converting alpha to numeric format 16-29 to 16-30
 - formatting for 2-33, 16-53 to 16-54
 - processing 2-67 to 2-68
 - range 2-62
 - resetting to decimal field 2-68
 - separator character 2-67
 - storing 2-67
- TIME qualifier (.FIELD) 2-67 to 2-68
- time-out, input 2-72, 8-4, 18-6
 - ActiveX controls 4-12, 4-16
 - I_FLDMOD and 8-33
 - IB_FIELD and 8-102
 - L_CHR and 9-66
 - L_SELECT and 9-129
 - M_PROCESS and 10-43
 - S_SELECT and 12-10
 - U_CHR and 16-16
 - U_FLD and 16-49
- timestamp fields 2-68, 8-33, 8-101
- title
 - list 9-125 to 9-126
 - window 2-108 to 2-110
- .TITLE script command 2-5, 2-108 to 2-110
- tkctl.def file 1-22, 18-2 to 18-6

- tklib_qcheck.elb 1-14, 16-147
- TKLIB_SH.EXE shared image 1-14, 16-147, 17-3
- tklib.elb 1-14, 16-147
- toolbar button methods 15-7, D-2
- toolbar buttons
 - 3-D edges for 15-2
 - defining group 15-9
 - deleting 15-11
 - disabled toolbars and 15-38
 - disabling or enabling 15-14
 - exiting environments and 7-5
 - graphics for 15-7
 - information about, returning 15-15 to 15-31
 - loading 15-3 to 15-8
 - methods for 15-7, D-2
 - numeric bar, loading 15-2
 - selecting 15-2
 - simulating selection 15-10
 - up/down state 15-14
- toolbars 15-2
 - creating 15-32, 15-37
 - deleting 15-34, 15-37
 - disabling 15-37
 - enabling 15-37
 - environment level, returning for 7-28
 - information about, returning 15-15 to 15-31
 - logging with environment 15-32, 15-36
 - maximum number of A-3
 - number logged for environment 7-29
 - placing (showing) 15-37
 - promoting to global 15-35
 - removing (hiding) 15-37
 - state of, saving or restoring 15-37
 - See also* toolbar buttons
- Toolkit. *See* UI Toolkit
- tools.def file 1-16, 18-2, 18-7
 - including 1-14
 - library creation macro 2-123
- ToolTips
 - list buttons, for 5-34, 9-56, 9-59
 - tab sets, for 13-12, 13-14, 13-15
 - toolbar buttons, for 15-7, 15-28
 - window buttons, for 5-7, 5-12, 5-34, 8-88
- topics, Help. *See* help
- transparent graphics 2-15
- trapping, error 16-5 to 16-7
- .TRIM script command 2-5, 2-111
- TRY-CATCH blocks 1-18

- %TS_PROCESS function 13-4 to 13-6
- %TS_TABSET function 13-7 to 13-7
- types. *See* data types
- TYPE_FACE initialization setting 18-43, 18-45, 18-48, 18-49
- typeface, specifying 18-40

U

- U_ABORT subroutine 16-5 to 16-7
 - number of characters in message line A-3
- U_ABOUT subroutine 16-8 to 16-9
 - number of characters in information line A-3
- U_BAR subroutine 10-5, 16-10
- U_BEEP subroutine 16-11
- U_CHARSB subroutine 16-12 to 16-13
- U_CHECKVERSION subroutine 16-14
- U_CHNINF subroutine 16-15
- U_CHR subroutine 16-16 to 16-17
- U_CLOSE subroutine 16-18
- %U_CMDLINOPT function 16-19 to 16-20
- U_CNVDAT subroutine 16-21
- U_CREATESB subroutine 16-22 to 16-23
- U_CTRCONTEXT subroutine 16-24 to 16-25
- U_DCDDAT subroutine 16-26 to 16-28
- U_DCDTIM subroutine 16-29 to 16-30
- U_DEBUG subroutine 16-31
- U_DIALOG subroutine 16-32
 - number of lines in dialog box A-3
- U_EDIT subroutine 16-34 to 16-39
- U_EDITKEYS subroutine 16-40 to 16-41
- U_EDITREND subroutine 16-42 to 16-44, 18-32, 18-36
 - on UNIX and OpenVMS 18-38
- %U_ENUMWNDS function 16-45
- U_FINISH subroutine 16-46
 - called by U_ABORT 16-6
- U_FLASH subroutine 16-47
- U_FLD subroutine 16-48 to 16-50
- U_FMTDAT subroutine 16-51 to 16-52
- U_FMTTIM subroutine 16-53 to 16-54
- U_FNDFIL subroutine 16-55 to 16-56
- U_GBLCHN subroutine 16-57
- U_GBLWND subroutine 16-58
- U_GETCHN subroutine 16-59 to 16-60
- %U_GETFILENAME function 16-61 to 16-66
- U_GETTXT subroutine 16-67 to 16-68
- %U_GETWNDERR function 16-69
- U_HELPREND subroutine 16-70 to 16-71
- %U_HTMLHELP function 16-72 to 16-86

- %U_ICON function 16-87 to 16-88
- U_INITREND subroutine 16-89
- U_LDREND subroutine 16-90 to 16-91, 18-32
- U_LDWND subroutine 16-92 to 16-93
 - called from U_POPUP 16-109
 - selection window and 2-89, 2-95
- U_LOGWND subroutine 12-6, 16-94
- U_MESSAGE subroutine 16-95 to 16-97
 - maximum number of characters A-4
 - maximum number of lines A-4
- U_MINVERSION subroutine 16-98
- U_MODREND subroutine 16-99 to 16-100, 18-2, 18-32
- U_MONTHS subroutine 16-101
- %U_MSGBOX function 16-102 to 16-105
- U_OPEN subroutine 16-55, 16-106 to 16-107
- U_PARSEVERSION subroutine 16-108
- U_POPUP subroutine 16-109 to 16-110
- %U_PRINTQUERY function 16-111 to 16-112
- %U_PRINTSETUP function 16-113 to 16-114
- U_REDRAW subroutine 16-115
- U_REND subroutine 16-116 to 16-121, 18-32
- U_RESIZE subroutine 16-122
- U_RESIZESB subroutine 16-123
- U_RLSCHN subroutine 16-124
- U_RSTREND subroutine 16-125 to 16-126, 18-32
- U_SAVE subroutine 9-69, 16-127
- U_SAVELOG subroutine 16-128
- U_SAVESETTINGS subroutine 16-130
- U_SAVREND subroutine 16-132 to 16-133, 18-32
- %U_SELPALETTE function 16-134 to 16-135
- U_START subroutine 16-136 to 16-138, 18-21
 - environment levels and 7-2
 - text message library and 16-67
 - tkctl.def fields initialized by 18-2 to 18-6
- U_STOREND subroutine 16-139 to 16-140, 18-32
- U_TEXT subroutine 16-141 to 16-142
- U_UPDATE subroutine 16-143 to 16-144
- U_UPDATESB subroutine 16-145 to 16-146
- U_VERSION subroutine 16-147
- U_WAIT subroutine 16-148 to 16-149, 18-6
- %U_WINCELLS function 16-150
- %U_WINCOLOR function 16-152 to 16-158
- U_WINDOW subroutine 16-159 to 16-161
- %U_WINHELP function 16-162 to 16-165
- %U_WININFO function 16-166 to 16-169
- %U_WINMETRICS function 16-170 to 16-202
- %U_WNDEVENTS function 16-203 to 16-209
- %U_WNDFONT function 16-213 to 16-231
 - defining font palette entries 18-40, 18-42
 - hierarchy of font assignment 18-48
 - setting application fonts 18-44
 - setting font palette entries 18-40
 - setting object fonts 18-47
 - setting window fonts 18-47
- %U_WNDSTYLE function 16-232
- %U_WNDTYPE function 16-233 to 16-234
- UC qualifier (.FIELD) 2-69
- UI Toolkit 1-2 to 1-15
 - control panel. *See* Synergy UI Toolkit Control Panel
 - determining if it is active 7-16
 - initializing 16-136 to 16-138
 - programs. *See* applications
 - terminating 16-46
 - version. *See* version
- UNBOUNDED qualifier (.PLACEMENT) 2-89
- UNDERLINE qualifier
 - .BORDER 2-9
 - .BOX 2-12
 - .FIELD 2-69
 - .FILL 2-73
 - .LINE 2-80
 - .TITLE 2-108
 - .WFIELD 2-115
- underscore. *See* renditions
- UNIX console 18-14
- up arrow key. *See* arrow keys
- updating screen 16-143 to 16-144
- UPPER_CASE_ONLY qualifier (.FIELD) 2-69
- user colors. *See* Synergy colors
- user data associated with a list, setting or
 - retrieving 9-144
- user data sets 2-22, 2-112, 17-6
- user fields
 - creating 2-70, 8-98
 - display method for 7-62 to 7-66
 - editing method for 7-67 to 7-69
 - validation method for 7-53 to 7-58
- USER qualifier
 - .ENTRY 2-28, 2-30
 - .FIELD 2-70, 7-53
- user text for input fields 2-70, 8-28
 - maximum size for A-1
 - retrieving or modifying 8-87

- user text for menu entries 2-28
 - g_entuser and 10-8
 - maximum size of A-3
 - returning 10-25, 10-58
- .USER_DEFINE script command 2-5, 2-112
- user-defined data types 2-70
- user-modifiable rendition schemes 18-32
- user-overloadable routines 18-8 to 18-10
- USTART_METHOD subroutine 16-236
- utility (U_) menu entries 7-82 to 7-83, B-5
- utility methods 7-82 to 7-83
- utility programs 1-5
- utility routines 16-4 to 16-240
- %UWNDEVENTS_METHOD
 - function 16-237 to 16-240

V

- validation
 - check box 2-41, 7-43
 - combo box 7-43
 - input field 2-40, 8-118, 8-120
 - drill method and D-6
 - hyperlink method and D-6
 - tab set and 13-5
 - methods 7-53 to 7-58
 - radio button 2-65, 7-43
 - user data type 2-70
- variable prompt 8-69
- version
 - checking for minimum 16-14, 16-98
 - parsing number 16-108
 - UI Toolkit 2-113, 16-147
- .VERSION script command 2-113
- VerticalScrollBar published property 9-20
- view
 - creating 2-24 to 2-25
 - input or general window 2-24 to 2-25
 - movable. *See* display area
 - text window 14-19 to 14-21
- view length 2-71, 8-5 to 8-6, 8-28, 8-98
- VIEW_LENGTH qualifier (.FIELD) 2-71
- viewing windows and menu columns 3-5 to 3-7
- virtual key codes 9-123, 15-18, 16-169
- vtkeys.wsc file 18-21
- VTxxx terminal 18-14

W

- W_AREA subroutine (Synergy DBL) 8-15 to 8-16
- WAIT qualifier (.FIELD) 2-72
- wait time. *See* time-out, input
- .WFIELD script command 2-5, 2-114 to 2-115
- WHELP_ commands 16-163 to 16-164
- width
 - ActiveX container window 4-31, 4-60
 - application window 7-47, 16-130, 16-137
 - box drawn in window 2-11
 - columns in a window, retrieving for 16-177
 - dialog box (U_DIALOG) 16-32
 - display area for an input window 2-24, 8-15, 8-92
 - font palette entry 16-222
 - general window 2-116
 - input field 2-49, 2-71, 8-28, 8-98
 - list column 9-15, 9-27, 9-41, 9-126
 - menu column 2-28, 2-29, 10-28
 - menu column heading 2-21
 - message (U_MESSAGE) 16-96
 - monitor display area 16-185
 - screen 16-122, 16-136
 - scroll bar window 16-22
 - selection window column 2-95, 12-5
 - sizing character for a window, retrieving for 16-174
 - tab in tab set 16-188
 - toolbar 15-32, 15-38
 - toolbar button 15-6
 - vertical scroll bar, retrieving for 16-192
- window borders
 - height of horizontal side 16-198
 - help window, for 16-70
 - renditions for 2-9, 16-42, 16-99
 - selection 12-3
 - width of vertical side 16-194
 - window titles and 2-109
- window coordinates
 - client area 16-176
 - entire window, for 16-193
 - mapping between windows 16-180
- window errors 1-18, 16-5
- window fields, general 2-114 to 2-115
- window frames
 - minimize and maximize buttons for 16-207
 - mouse events and 16-206
 - thickness of, retrieving 16-195, 16-200

- window library
 - creating 1-4, 2-118, 3-5
 - file, moving to other systems 1-23
 - global, promoted to 16-137
 - listing windows in a 16-45
 - loading window from 14-22, 16-92 to 16-93
 - opening 3-7, 16-136
 - reading text from 14-15, 14-17, 14-22
 - saving windows to 16-127
 - text entry in 14-17, A-4
 - Window manager error 16-69
 - .WINDOW script command 2-5, 2-116
 - ending 2-26
 - window scripts. *See* script file
 - windows (UI Toolkit) 2-3
 - application. *See* application window
 - blank lines, trimming from 2-111
 - borders for. *See* window borders
 - buttons in. *See* buttons
 - child. *See* child windows and lists
 - client area of. *See* client area of window
 - composite. *See* composite windows
 - container. *See* container windows
 - coordinates for 16-176, 16-180, 16-193
 - debugging 1-6, 17-2 to 17-7
 - defining
 - at runtime 8-6
 - in a script 2-26, 2-76, 2-95 to 2-96, 2-116
 - interactively 1-5
 - deleting 16-159 to 16-161
 - environment levels for, retrieving 7-31
 - event processing. *See* mouse events
 - filling with a string 2-73 to 2-74
 - font 2-75, 16-226, 18-46
 - modifying at runtime 16-231
 - frames for 16-195, 16-200, 16-206, 16-207
 - general. *See* general windows
 - global 7-2, 16-58, 16-94
 - handles for. *See* handles
 - icons, specifying for 16-87 to 16-88
 - information about, retrieving 16-170 to 16-202
 - input. *See* input window
 - libraries for. *See* window library
 - list class. *See* list classes
 - local 7-2, 16-58, 16-94
 - maximum number of A-2
 - menu column. *See* menu column
 - moving 7-46, 16-159 to 16-161
 - names for, size limit A-1
 - parent. *See* composite windows
 - placing 2-89, 16-159 to 16-161
 - pop up 16-109 to 16-110
 - reconfiguring as text windows 14-12
 - removing 3-7, 16-159 to 16-161
 - renditions for. *See* renditions
 - resizing 16-205
 - script files for. *See* script file
 - scroll bar 16-22 to 16-23
 - searching for 8-63
 - selection. *See* selection window
 - size of 2-116, 16-136
 - state processing for 7-49
 - style, changing 16-232
 - system errors for, retrieving 16-69
 - tab set, adding to 13-12
 - testing 3-5 to 3-7
 - text windows. *See* text window
 - text, adding to 2-101 to 2-107, 16-34 to 16-39
 - titles for, defining 2-108 to 2-110
 - type of, returning 7-32, 16-233 to 16-234
 - vertical spacing 16-232
 - viewing 3-5 to 3-7
 - See also* dialog boxes and screens
 - Windows Control Panel and Toolkit
 - renditions 18-25 to 18-30, 18-31
 - Windows operating systems
 - caption, used for header 16-137
 - color operations for 16-152 to 16-158
 - fonts on 16-213 to 16-231, 18-40 to 18-50
 - information about, returning 16-166 to 16-169
 - Print Manager, information from 16-111 to 16-112
 - Print Setup dialog box for 16-113 to 16-114
 - WinForms 11-2 to 11-14
 - WinHelp (winhelp32.exe) 16-162 to 16-165
 - WND environment variable 18-14
 - word wrap 2-104, 2-105, 2-106
 - WPF elements, embedding 11-2 to 11-14
 - wrapping, line 18-7
 - WYSIWYG tools 1-5
- X**
- xfNetLink .NET 1-21
- Y**
- years. *See* dates; environment methods
- Z**
- zero entry, not allowing in field 2-63

