# Command Line History

A proposal to enhance the Multics video windowing environment.

Author: Gary Dixon
Date: May 19, 2016

## Abstract

When connecting to Multics, a feature dearly missed is a mechanism for editing command lines as they are typed; and for editing prior command lines to fix mistakes, add control arguments, and so on. Such command line editing features are common-place on today's systems, and would be nice to have as part of the Multics user experience.

The Multics video windowing system provides functions for editing input lines as they are typed. It uses Emacs-like editor keystroke commands to position within the current input line, operate on characters or words surrounding the cursor, etc. This MTB proposes an enhancement to capture input lines (an input history), along with editing requests to select prior input lines for further edits and re-input.

The new history mechanism should permit:

- capturing each input line in a per-process file, for the user's future reference;
- selecting one of the captured input lines to use as the next input (instead of typing a new line);
- further editing of the selected input; and finally
- returning the line as a result from iox_$get_line.

This Multics Technical Bulletin suggests two alternatives for implementing a Multics command history feature.

*Table 1: Revision History*

| Date | Revision | Author | Comment |
|---|---|---|---|
| 2016-03-08 | 0.1 | Gary Dixon | Initial Revision. |
| 2016-03-22 | 0.2 | Gary Dixon | Incorporate comments from Eric Swenson (auditor). |
| 2016-04-13 | 0.3 | Gary Dixon | Include references to a new input_history command and I/O module. |
| 2016-04-15 | 0.4 | Gary Dixon | Incorporate comments from Eric Swenson (auditor). |
| 2016-04-17 | 0.5 | Gary Dixon | Incorporate comments from Olin Sibert, Chris Jones. |
| 2016-05-19 | 1.0 | Gary Dixon | Proposal for input_history_ command and video system changes. |

# Contents

Abstract ........ 1
Introduction ........ 3
Command History in Unix and Linux ........ 4
Multics Command Processing ........ 5
Terminal Capabilities ........ 8
Alternatives for a Multics Command History Feature ........ 10
Comparison of Alternatives ........ 14
History Segment Issues ........ 15
Implementing Alternative 1: input_history_ ........ 16
input_history.pl1 ........ 18
input_history_.pl1 ........ 21
I/O Module Entrypoints ........ 21
Line Editor Request Entrypoints ........ 23
input_history_data.incl.pl1 ........ 25
window_io_iox_.pl1 ........ 27
window_line_editor.incl.pl1 ........ 30
ihtest.pl1 ........ 31
window_editor_utils_.alm ........ 33
bound_video_.bind ........ 33
Documentation for input_history:  input_history.info ........ 34
Documentation for input_history_:  input_history_.info ........ 36
Documentation:  video_editing.gi.info ........ 40
Documentation for window_editor_utils_.alm:  window_editor_utils_.info ........ 43

Figure 1: Unix/Linux GNU Readline ........ 4
Figure 2: Multics Listener & Command Processor ........ 5
Figure 3: Multics Command Processors & exec_com ........ 6
Figure 4: Standard Multics Terminal Configuration with tty_ ........ 8
Figure 5: Video Manager I/O Configuration ........ 9
Figure 6: Input History Alternative 1 (Initial Design) ........ 11
Figure 7: Input History Alternative 1 (Corrected Design) ........ 12
Figure 8: Command History Alternative 2 ........ 13

## Introduction

People using Multics today expect ease-of-use features that are available in other operating systems. Among these is command history, a facility: for capturing every command line, as it is entered; and for selecting, editing, and re-inputting prior command lines.

Goals for implementing a command history facility:

A. At input time, allow commands to be edited as they are being entered.
    1. For ease-of-use, entering and editing command lines should use a WYSIWYG (what you see is what you get) interface, rather than following a qx-like editing paradigm.

B. At input time, capture each command line in a command history file.

C. Provide a way to select prior lines, allow them to be edited and re-input.

D. To simplify implementation, the new feature should be implemented with a minimum of changes to existing Multics software.
    - Reduce documentation changes needed for existing software.
    - Minimize testing required for any changed software.

Multics includes one facility that meets some of the above goals. The attach_audit command, and its audit_ I/O module capture all input lines as they are entered; and all output lines as they generated by software. They provide a qx-like interface for editing the current input line as it is being typed; and for re-inputting a prior input line selected from the audit file. However, its many features make it complex to use; and its cumbersome user interface is not very user-friendly. Overall, it is difficult to understand and use.

Let's examine Unix and Linux to get ideas for a user-friendly Multics implementation of command history.

## Command History in Unix and Linux

On Unix/Linux systems, a shell program reads and executes command lines by forking a new process to run each command found in the command line. In the forked process, the command has its own attachments for input and output.

Typically, the shell uses the GNU readline subsystem to obtain each command line. readline uses video terminal capabilities to implement several features:

- editing a command-line as it is typed, including moving cursor within the typed line to add/change characters entered earlier;
- providing functions for command name completion, and path starname expansion, as the command is typed;
- maintaining an optional history of each command line returned to its caller;
- providing functions to search for lines in this history, and to reenter a historical command line again (perhaps after additional editing).

Thus, readline treats all data it reads from stdin as a command line entered by the user, and optionally stores each line in a history file that can be: extended by adding a new command line; or searched to locate a prior command line. In either case, the command line can be edited by the user, prior to returning it to the shell for execution. (See Figure 1.)

readline obtains information about terminal capabilities (e.g., cursor manipulation sequences) from the Unix/Linux termcap system. Its editing features are tightly linked to termcap's terminal definitions, and to the capabilities they enable.

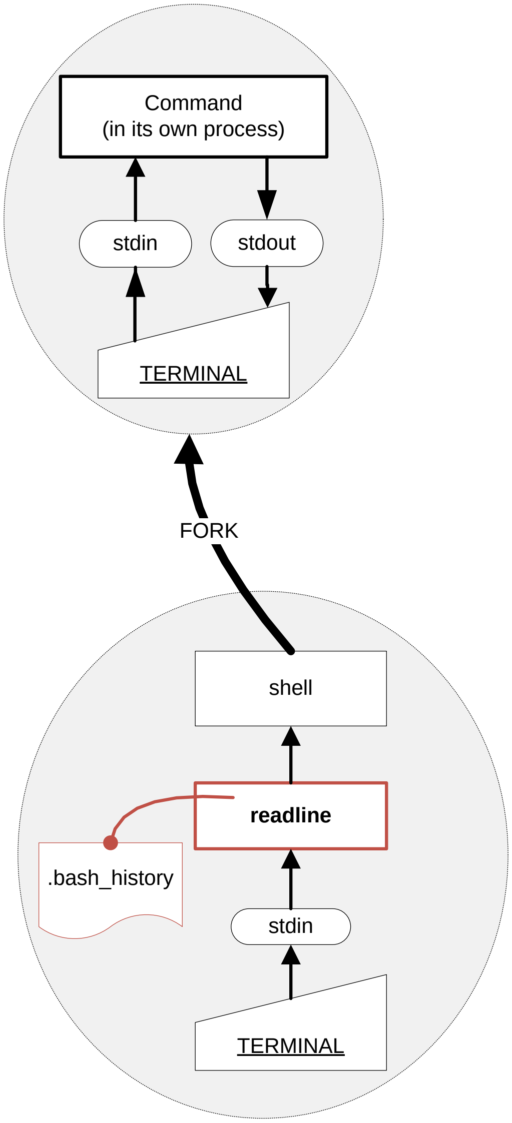

*Figure 1: Unix/Linux GNU Readline*

## Multics Command Processing

Multics divides Unix/Linux shell functions among several programs, and adds a few functions not present in the typical Unix/Linus shell.

The Multics command_processor_ evaluates and executes a command line by invoking each command in the same process used by the command processor. Multiple commands in a single command line are invoked sequentially.

Multics commands are read from the user terminal by the listen_ subroutine; and then passed to one (or more) command processor programs for evaluation and execution. (See Figure 2.)

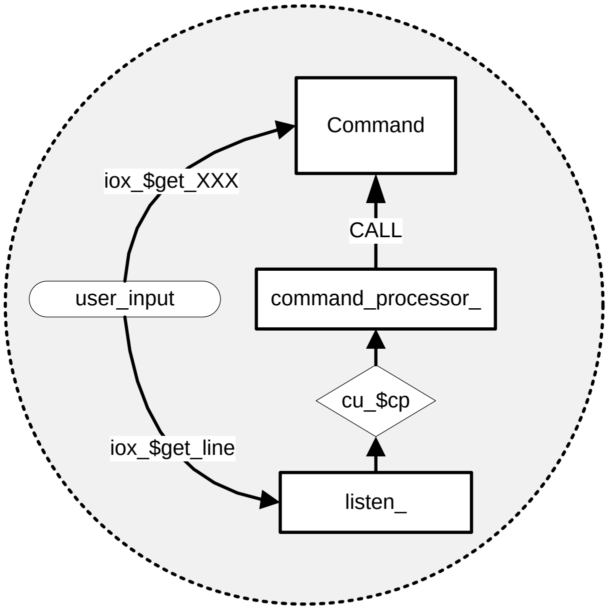

*Figure 2: Multics Listener & Command Processor*

The program known colloquially as The Listener (actually named listen_) reads a command line from the user_input I/O switch. It then calls cu_$cp, which is a transfer vector that calls the command processing program at the top of a cp-stack. When a Multics process starts, top of the cp-stack is the command_processor_, the program that implements the Multics command language.

However, many Multics users include the "abbrev -on" command in their start_up.ec. The abbrev facility expands abbreviations used within a command lines. "abbrev -on" pushes abbrev_ as a pre-processor onto the top of the cp-stack. In this configuration:

- listen_ passes each command line to topmost cp-stack program (abbrev_);
- abbrev_ expands any abbreviations in the command line; then passes the line to the next program in the cp-stack (which it remembers to be command_processor_);
- command_processor_ evaluates tokens in the command line from left-to-right, and invokes each command or active function as it is encountered (following command syntax rules).

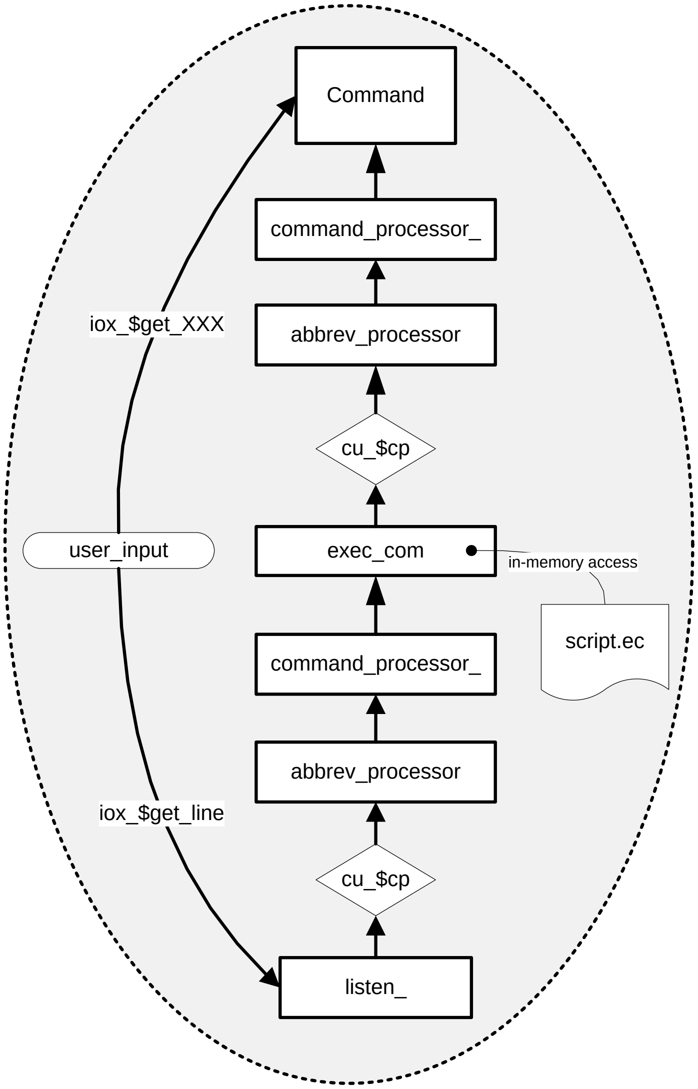

*Figure 3: Multics Command Processors & exec_com*

Scripting functions are provided by a separate exec_com command, which brings the script into memory, reads command lines within the script, and passes them via cu_$cp to the chain of command processors.

Note that, for any given command line, the Listener, Command Processor programs, and the invoked command are all running in the same process, and are reading terminal input from the same user_input switch while they are at the top of the execution stack. (See Figure 3.)

This sharing of user_input within the same process impacts possible strategies for implementing a command history functionality on Multics. In fact, broader sharing of user_input than is shown in Figure 3 may occur if some condition interrupts the running program. In such cases, the unclaimed_signal handler may push a new command level atop the stack, and run a second (or third, or fourth) listen_ instance within the same process. All of these listeners, and the subsequent commands they may invoke are sharing user_input. (See the stack trace in Table 2.)

*Table 2: Multiple Listeners on the Stack*

```
STACK FRAME  PROGRAM
               ON-UNIT(S) associated with frame above, if any
 ----------  ----------------------------------------------------------------
 19          command_processor_
               on command_abort_ ...
 18          abbrev_processor
               on cleanup ...
 17          listen_$release_stack
               on cleanup ...
 16          unclaimed_signal
               on cleanup call get_to_cl_|152
 15          default_error_handler_$wall
               on any_other call default_error_handler_|1004
 14          initialize_process_$any_other.2 (on-unit invoked)
  13.sig      signal_$signal_ (condition_info: quit)
  13.ops      pl1_signal_$pl1_signal_from_ops (PL/I: signal condition(quit);)
               on cleanup ...
 13          lisp_fault_handler_$ioc
               on cleanup ...
12           ..lisp.. (alm)
 11           Begin Block at lisp|34163
               on program_interrupt ...
               on any_other call lisp_default_handler_
               on cleanup ...
 10          lisp
  9          forward_command_ (alm in bound_multics_emacs_)
  8          emacs
               on record_quota_overflow, lisp_linkage_error, cleanup ...
  7          emacs
  6          command_processor_$read_list
               on command_abort_ ...
  5          command_processor_$complex_command_processor
               on cleanup ...
  4          command_processor_
  3          abbrev_$abbrev_processor
               on cleanup ...
  2          listen_
               on cleanup ...
  1          initialize_process_
               on any_other call initialize_process_|411
```

## Terminal Capabilities

The Multics terminal interface was designed many years ago, when hard-copy printout terminals were the main user interface device. With such terminals, it was considered impossible (or at least unsightly) for the cursor to back up to permit typed corrections atop data already printed on the terminal paper. So the standard Multics terminal I/O module, tty_, does not support any form of input line editing.

Upon login to an interactive Multics process, the user_input terminal I/O switch and associated I/O modules are typically configured as shown in Figure 4.

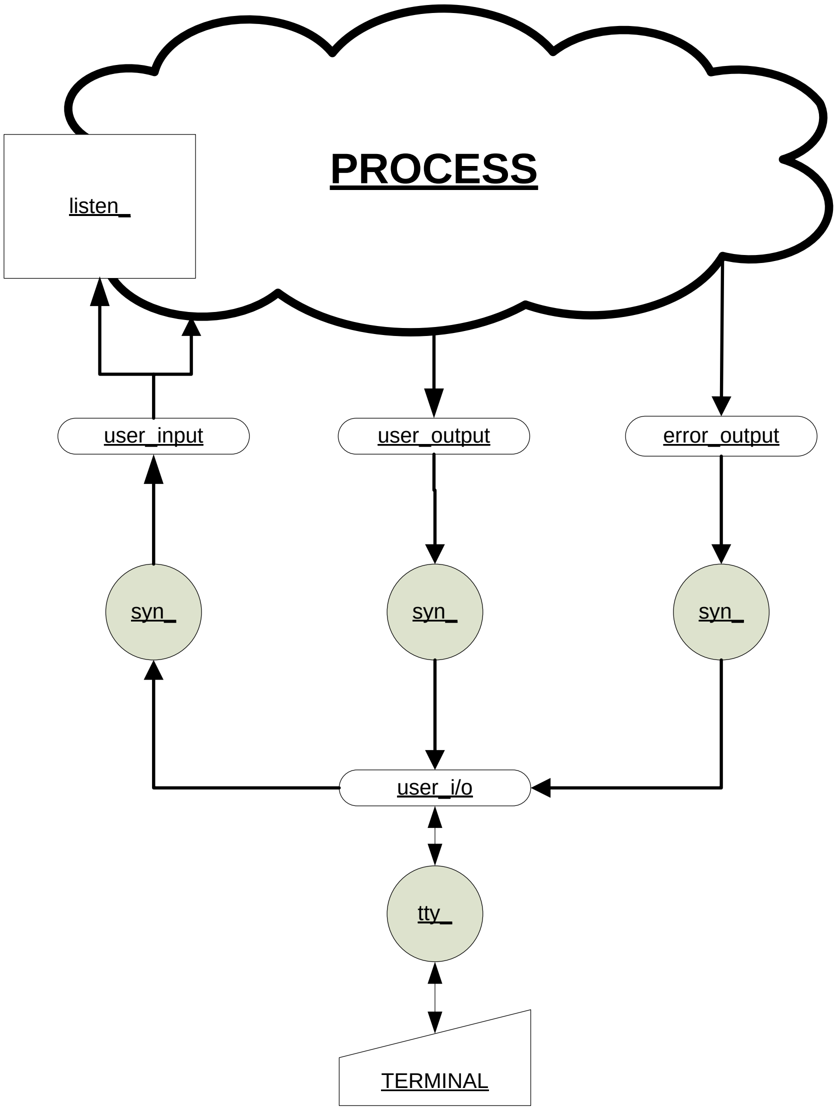

*Figure 4: Standard Multics Terminal Configuration with tty_*

However, in 1987, support for video terminal windowing was added to Multics. The window_io_ I/O module uses video terminal capabilities defined in the Terminal Type File to support full cursor movement within the video screen, as well as menu displays and forms entry, etc. Among its features are the window_io_ input line editor, providing emacs-like (WYSIWYG) editing requests that provide cursor movement within the line, moving or editing in character- or word-increments, etc.

Figure 5 below shows configuration of an interactive Multics process in which video windowing has been enabled, via: **`wdc invoke`**.

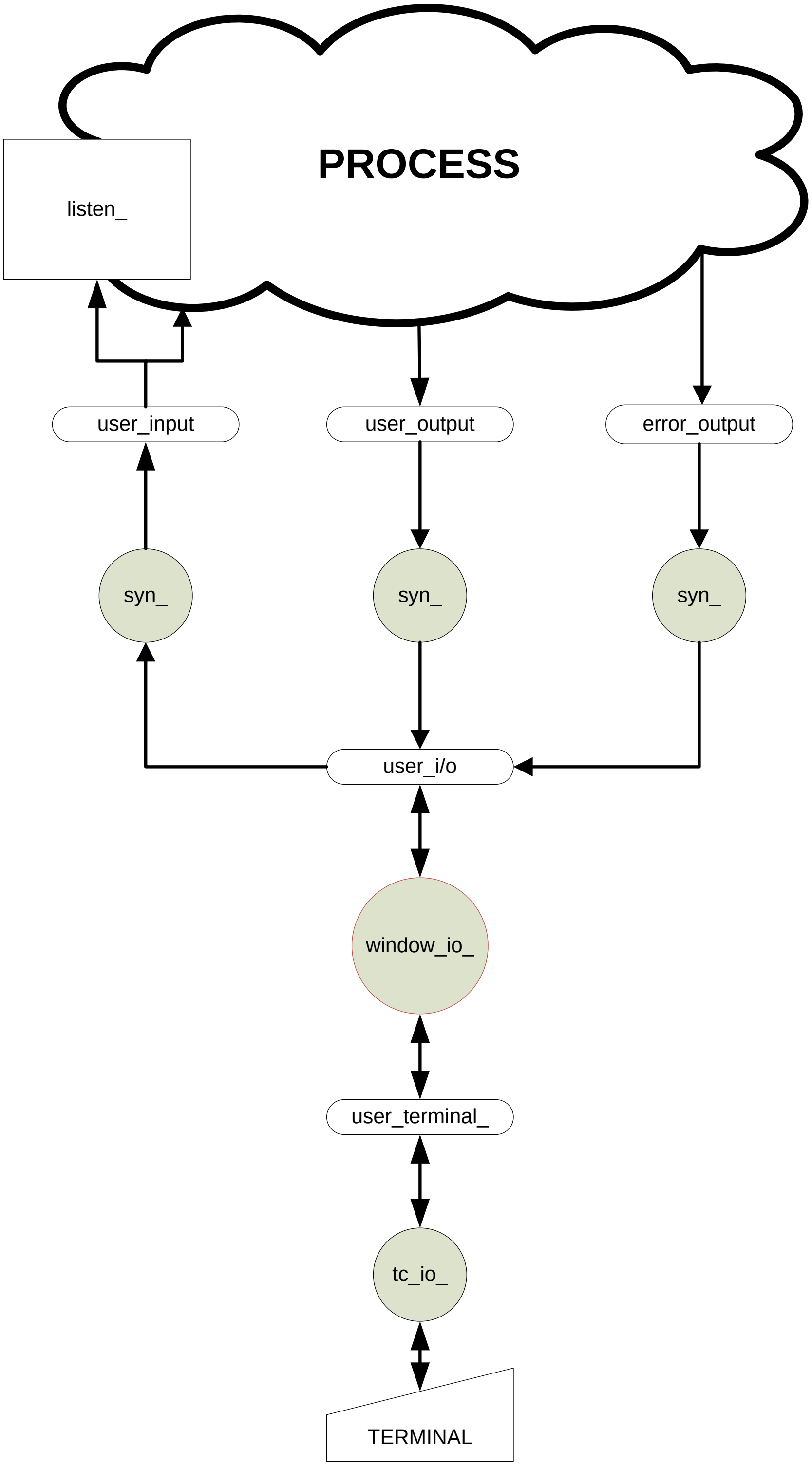

*Figure 5: Video Manager I/O Configuration*

Video system functions, including window_io_ input line editor capabilities, are described in the *Multics Menu Creation Facilities,* CP51-02. Section 2 introduces the video system; Section 4 describes the input line editing capabilities.

When the video system is first invoked (via: **`wdc invoke`**), the entire terminal screen is handled as a single window (often called "the main window" or "window 1"). It remains as a single window until menu creation commands are used to define sub-windows (areas of the screen) that are removed from the main window.

The window_io_ input line editor supports editing of input typed in each window defined on the screen. An application may call window_io_ control operations to tailor the editing functions by: adding, replacing or removing edit request key mappings; or providing application-specific editing functions, which are invoked by an application-specific key mapping.

This input editing functionality is similar to the command line editing provided by GNU readline.

## Alternatives for a Multics Command History Feature

Adding editing functions to the older tty_ view of terminal input is possible (see the attach_audit command). However, this implementation does not meet the ease-of-use goals for a command history implementation outlined above.

Given that window_io_ already performs some video-style input editing functions, a better solution would add new command history features to the window_io_ input line editor.

**ALTERNATIVE 1:** Implement a simple input history facility as a separate I/O module, which:

- captures all lines read from user_input in a per-user history segment; and
- adds several key bindings to the window_io_ line editor to select a past input line from the segment, for editing and re-input.

Figure 6 shows this implementation.

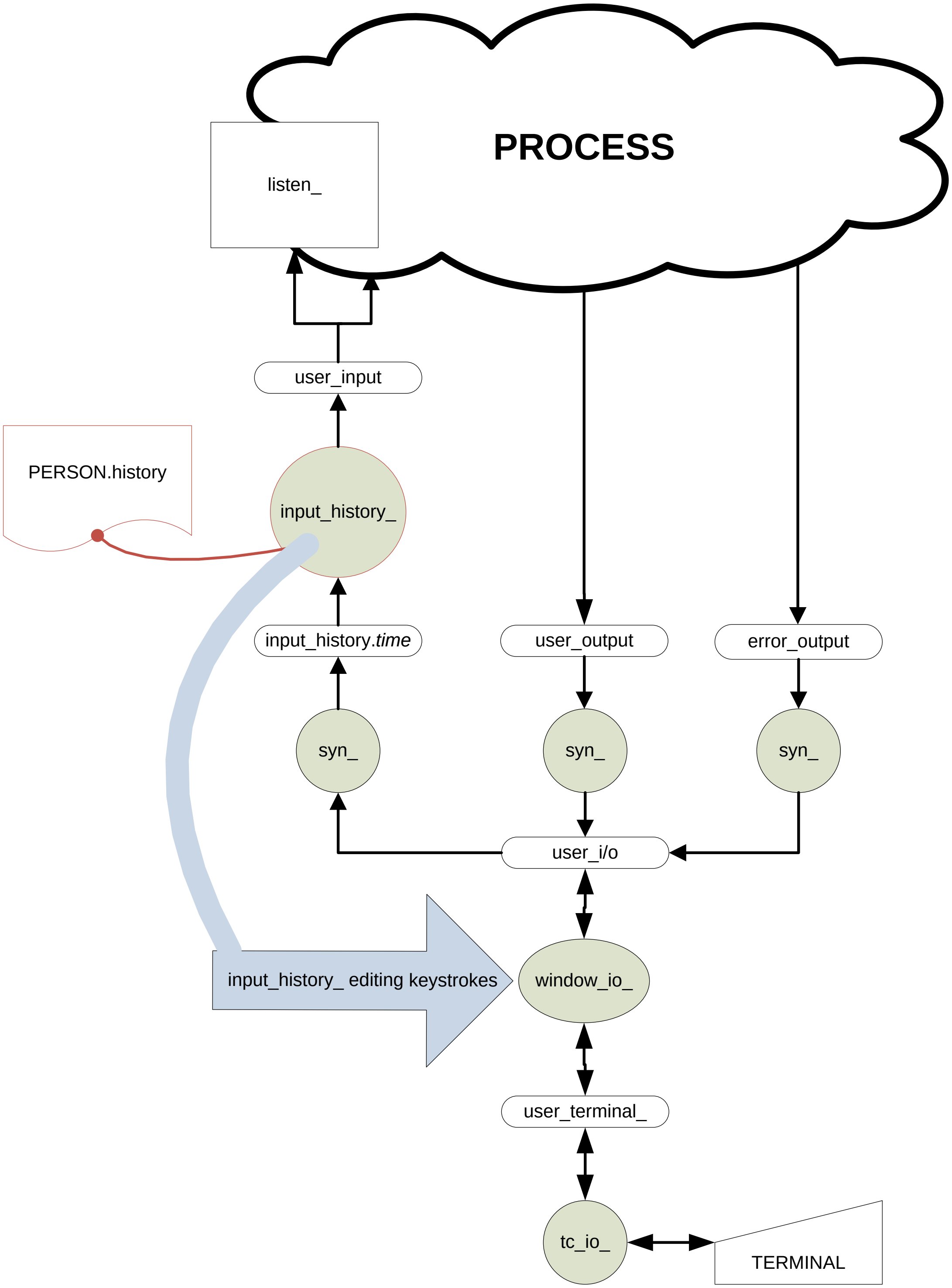

*Figure 6: Input History Alternative 1 (Initial Design)*

The configuration above does not work, however. When a new command level is pushed onto the stack, all three standard I/O switches (user_input, user_output, error_output) are saved and new syn_ attachments are made in the new command level. This is done to handle unexpected conditions occurring while input or output attachments have been redirected (e.g., using &attach in an exec_com file or file_output).

Such re-syn_ measures sideline input_history_ in the new command level.

To avoid this problem, the input_history_ I/O module must be attached between user_i/o and the window_io_ module. (See Figure 7)

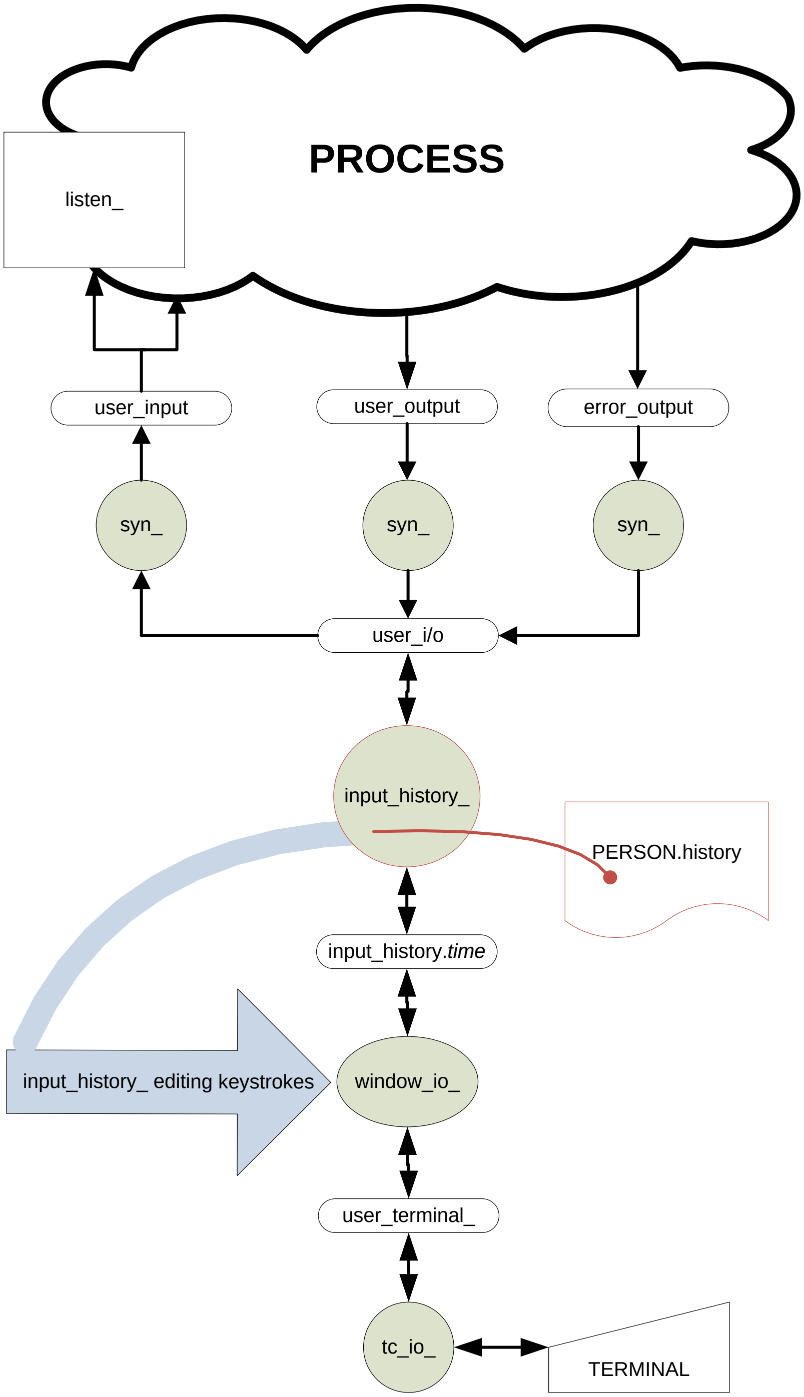

*Figure 7: Input History Alternative 1 (Corrected Design)*

**ALTERNATIVE 2:** Separating command lines (lines read by the Listener) from those read by other applications. This would require:

- changing the listener to read command lines through a separate I/O switch (command_input); and
- writing a command_history_ I/O module that would capture command input lines in the history segment; and
- having command_history_ add keystrokes to the editor to select a past command line from the segment only when a line is being read through command_input I/O switch.

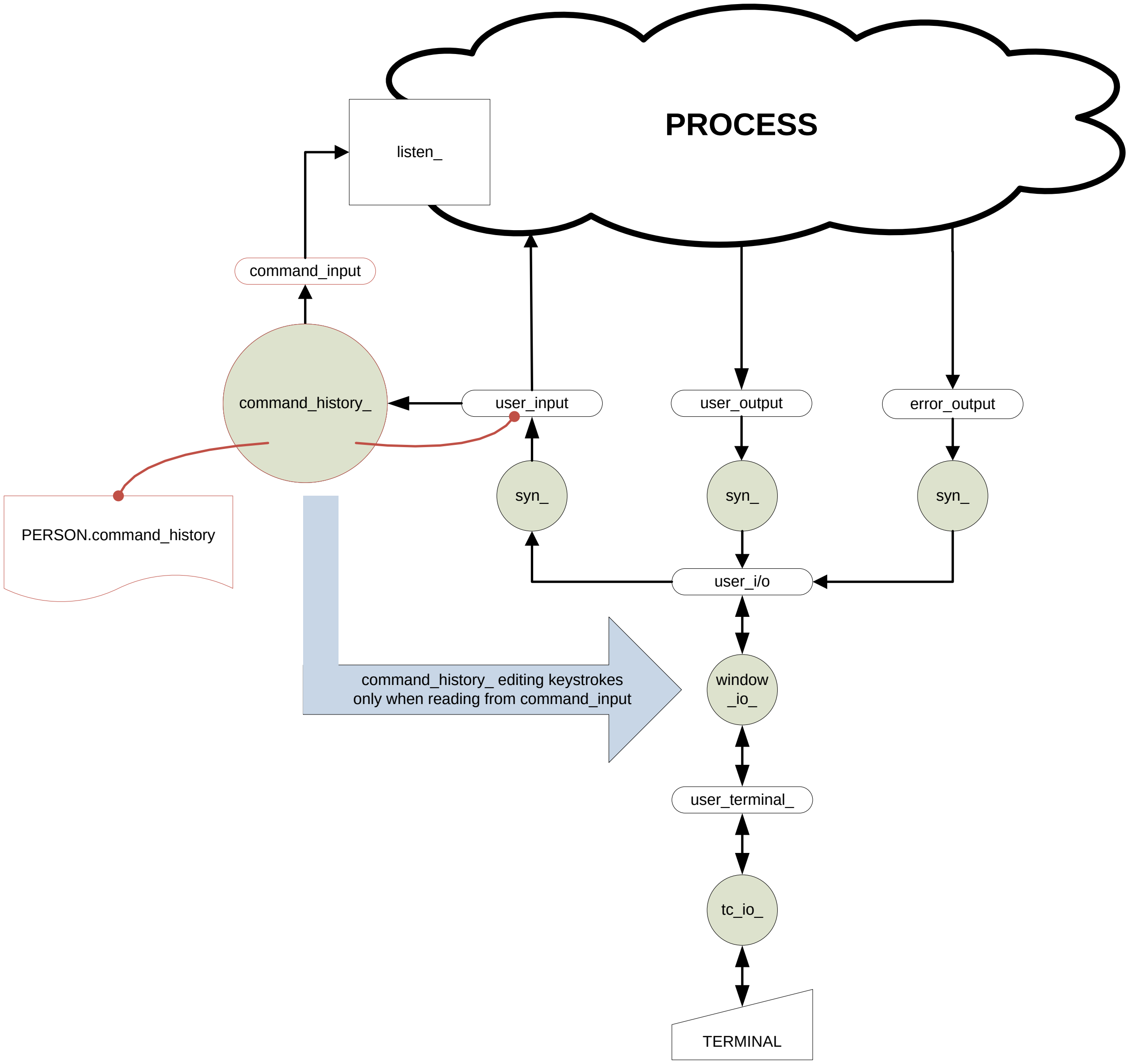

*Figure 8: Command History Alternative 2*

Each time listen_ calls iox_$get_line on the command_input I/O switch, the command_history_ module would:

- add video editing key bindings to search/select past command input lines from the command_history segment, for use by the window_io_ line editor.
- call iox_$get_line on its associated source switch, user_input to read a new input lines (possibly a resubmission from the command history).
- remove the video editing key bindings for command input lines from the window_io_ line editor.
- return the line to its caller (listen_).

Adding and removing key bindings is a simple operation provided by window_io_. Several new bindings may be added, or old bindings restored, in a single iox_$control call to window_io_.

**ALTERNATIVE 3:** Add input history features to the existing window_io_ I/O module implementation. This maintains the I/O configuration shown in Figure 5 above.

- The window_io_ attach description would be enhanced with control arguments to capture input history in a file.
- At attach time, window_io_ would add input line editor requests for selecting past input lines from the input history, only if a history was being captured.

This alternative would require changes across many areas of the wdc and window_io_ code. This code is already quite complex and convoluted. Adding more than 1000 lines of code would require a complete retesting of all features of this I/O module.

## Comparison of Alternatives

The choice between these three alternatives must balance several factors, as shown in Table 3 below.

Both alternatives 1 and 2 require small enhancements to the window manager for proper implementation of input history editing requests. The design of window_io_ was never fully completed; and a few essential features to support input edit requests external to window_io_ are missing.

- An interface to redisplay the current line being input/edited, as the external edit request operates on the line (or replaces its contents with another line). This deficiency was known to the window_io_ developers (as indicated by several comments in the code); but was never corrected.
- An interface for ungetting characters read by the external edit request, or for returning them for processing by the line editor.

Both are required to implement searching to select prior input lines. The search requests must read characters being searched for, and have some way to display intermediate search results. Also, searching must be ended by reading other characters that accept or reject the intermediate search results. These changes may be implemented in ~25 lines of code changes, localized within the window_io_ line editor.

*Table 3: Comparison of History Alternatives*

| | ALTERNATIVE 1 | ALTERNATIVE 2 | ALTERNATIVE 3 |
|---|---|---|---|
| Implementation | Simpler to explain, and code. | More complex to explain, and code. | Simplest to explain, but much more complex to code. |
| Scope of the changes | Minor changes to window_io_; no changes to listen_. | Listen_ must use new command_input I/O switch; minor window_io_ changes. | Major changes to window_io_; no changes to listen_. |
| Scope of testing | Full testing of input_history command, and new input_history_ I/O module. Small tests of minor changes to window_io_. | Full testing of command_history command, and new command_history_ I/O module. Small tests of minor changes to window_io_. | Complete retest of wdc and window_io_ I/O module, after significant additions to this code. |
| Operating costs | New input edit keystrokes added once, when input_history command turns on this facility. | New command-only edit keystrokes added/removed each time a command line (or residual command line fragment) is read. | New input editor keystrokes added once, when wdc is invoked, or attached to a subwindow. |
| Data in history file | All input lines entered from the window. | Only command lines. | All input lines entered from the window. |

Comments would be appreciated from readers on these three possible alternatives, other possible implementations, differences between alternatives not stated in Table 3, etc.

## History Segment Issues

Comments received so far have mostly supported Alternative 1, but pointed out several implementation issues for an input history segment.

A. Synchronizing read/write operations by several processes accessing the history file concurrently. Major problems in this area include:
   1. Delay appending new input while prior history is scrolled off top of history segment.
   2. Separating input lines added by each process sharing a history segment.

B. Providing history segment for a login at an elevated (from system_low) AIM access class.

C. Supporting user-named sets of input history for specific purposes.

The design proposed below resolves these issues, as follows.

- At input_history_ attach time, the user may specify location of the history file to use during that attachment, via a -pathname attach argument. An existing history file may be extended if no process is currently logging to that file. (Resolves issue C above.)
- Each input history segment will log input lines from a single active process. (Resolves issue A above.)

- A per-process history file, stored in the process directory, may be used if another process has locked the permanent history file named by the user; or if that permanent history file is not writable (due to mismatch of its AIM access class with current process authorization, or due to ACL restrictions on access). input_history_ automatically switches to a per-process file in such circumstances, unless -permanent is given in the attach description. (Resolves issues A and B above.)

## Implementing Alternative 1: input_history_

The input_history_ alternative is the best method for achieving the goals outlined in the Introduction.

- Compared with alternative 2, its implementation requires fewer changes to other Multics software, and more closely follows the terminal I/O strategy used in Multics processes (comparing Figures 5, 7, and 8).
- Compared with alternative 3, more code is required in an input_history_ I/O module to capture input lines as they are entered; and to add external line editor requests. But this code is focused on a few small tasks, and uses a well-defined, existing external line editor request interface already present in window_io_. Testing a new, focused implementation in a separate input_history_ I/O module will be much easier than testing equivalent code installed throughout the many different sections of window_io_, and its supporting programs (window_io_iox_ and wioctl_), in their highly complex operating environment.

Alternative 1 can be implemented as eight new or modified source files, plus several info files.

- **input_history.pl1**: a new command to oversee attachment of the input_history_ I/O module to the user_i/o switch, while moving its usual window_io_ attachment to a uniquely-named (or user-specified) switch to be monitored by input_history_. This command validates input_history_ attach arguments, then moves the user_i/o attachment to a uniquely-named switch, and attaches the input_history_ I/O module to user_i/o.

- **input_history_.pl1**: a new stream_input_output I/O module that captures input lines as they are read via iox_$get_chars calls; and adds 5 external edit requests to the window_io_ line editor for selecting prior input lines for re-input.

- **input_history_data.incl.pl1**: a new include file that defines the input_history_ attach data, and history segment lock file structures. It is used primarily by input_history_.pl1; and used superficially by the other two routines above.

- **window_line_editor.incl.pl1**: the existing include file that defines the line_editor_info structure passed by the window_io_ line editor to external editing requests. A new version 3 of this structure adds elements needed to enhance the window_io_ line editor interface.

- **window_io_iox_.pl1**: the existing code that implements the window_io_ line editor. Small enhancements to the line editor interface will be coded here. Also, the $get_top_kill_ring_string_ entrypoint is added to remedy the problem described below for window_editor_utils_.alm.

- **ihtest.pl1**: a new command to test the external line editor requests in input_history_, without actually attaching the input_history_ I/O module. It simulates calls from the window_io_ line editor to the external editing functions in input_history_. This permits probe-level debugging of the input_history_ code while not interfering with normal input through the user_i/o switch.

- **window_editor_utils_.alm**: the existing transfer vector that provides some window_io_ line editor support routines to external editor requests. While testing the above approaches, a major problem was found in the design of the window_editor_utils_$get_top_kill_ring_element. There is no way to tell the actual length of the returned string, or whether the char(*) argument supplied to hold the kill ring string was long enough to hold the entire string. A new $get_top_kill_ring_string will be added as a remedy.

- **bound_video_.bind**: the existing bind file changed to add the new input_history_-related objects to the bound_video_ object.

- **window_editor_utils_.info:** changes to describe the new $get_top_of_kill_ring_string entrypoint.

- **input_history_.info:** describing the new input_history_ I/O module.

- **input_history.info:** describing the user interface for the new input_history command.

- **video_editing.gi.info:** changes adding information about the input_history_ edit request key bindings, when input_history_ is in use.

The following subsections provide more details about these new or changed source files.

## input_history.pl1

The input_history (ih) command supervises a multi-step process that changes the standard video system I/O configuration (see Figure 5) into a configuration that includes the input_history_ I/O module (see Figure 7). These steps involve:

1. Get source_switch_name and monitored_switch_name (optional positional arguments), or their default values.
2. Form an input_history_ attach description from monitored_switch_name, and remaining control arguments. Let input_history_ I/O module diagnose any problem with the control arguments.
3. Look for source_iocbP (corresponding to source_switch_name). Make sure it is attached and open.
4. Find/create the monitored_iocbP (corresponding to monitored_switch_name). This will probably be created, if our default name is used. Make sure it is detached.
5. Validate attach description (created in step 2 above) if source_switch_name is user_i/o.
6. Move current attachment for source_iocbP onto monitored_iocbP. Establish a cleanup on-unit, so we can gracefully undo this move, if necessary.
7. Attach source_iocbP to input_history_ I/O module, using attach description (formed in step 2 above).
    - Use a silent attach operation if source_switch_name is user_i/o. Cannot print errors while user_i/o is detached.
    - Use a loud attach operation otherwise.
    - If errors occur, undo the move_attach, then print an error message.

Because the user_i/o switch attachment is moved to a unique I/O switch (step 6), it is effectively detached when the input_history_ I/O module is being attached to the user_i/o switch (step 7). Therefore, input_history_ attach operation cannot diagnose errors, give warning messages, and so on if problems or unexpected events are encountered during the attachment.

This problem is avoided by using a private, pre-attach interface in the input_history_ I/O module (step 5) to validate attach options, create/lock/initiate the history segment (and associated lock segment), and construct the attach data structure used when input_history_ is actually attached. Since step 5 occurs before the existing user_i/o attachment is moved (step 6), input_history_ can issue definitive messages to diagnose errors, and warn of events (like creation of a new history segment, or use of a per-process history when the permanent history segment is locked by another process).

This attach supervision parallels that provided by the window_call (wdc) command for its invoke operation, in which user_i/o is detached from tty_ and reattached to window_io_.

As in the window_call command, input_history actually provides several different operations. These are summarized below in the usage information displayed by the command: ih

```
Syntax as a command:
   ih attach {source_switch {monitored_switch}} {-control_args}
or:
   ih detach {source_switch}
or:
   ih version {source_switch}

Arguments:
source_switch
   is an existing switch attached to an I/O module open for stream_input or
   stream_input_output.  Subsequent get_line operations on this switch will be captured
   in the history file.   (DEFAULT: -- user_i/o)
monitored_switch
   is a new switch created by the input_history command to save the I/O module attachment
   currently on the input_switch, so subsequent I/O requests can pass-thru the input_history_
   module to the saved module.   (DEFAULT: -- input_history.time)

Control arguments: are input_history_ I/O module attach options.
-pathname PATH, -pn PATH
   use PATH as the location of the history file.  The default PATH is:
     [homedir]>[user name].history
-perprocess, -pp
   use a temporary history file created in the process directory.
-permanent, -perm
   use only a permanent the history file.

-lines N, -ln N
   recommends a size for the history file, in lines.  The default is 200 lines (about 2
   records of storage, if the average input line is 40 characters in length).
-truncate, -tc
   if the history file already exists, truncates this file as part of the attach
   operation.  The default is to extend the existing file.
```

The attach operation attaches the input_history_ I/O module. The detach operation undoes this attachment, and restores the I/O configuration to what it was prior to the attachment. The version operation displays version identifiers for the input_history command, and input_history_ I/O module.

The input_history command actually implements additional operations useful for debugging the input_history_ I/O module. These are summarized in the usage information displayed by the command: ih debug

```
- operation: debug (on)

Syntax as a command:
   ih attach {source_switch {monitored_switch}} {-control_args}
or:
   ih detach {source_switch}
or:
   ih version {source_switch}
or:
   ih debug
   ih db
or:
   ih print_attach_table
   ih pat
or
   ih data {source_switch}
```

```
Arguments:
source_switch
   is an existing switch attached to an I/O module open for stream_input or
   stream_input_output.  Subsequent get_line operations on this switch will be captured
   in the history file.   (DEFAULT: -- user_i/o)
monitored_switch
   is a new switch created by the input_history command to save the I/O module attachment
   currently on the input_switch, so subsequent I/O requests can pass-thru the
   input_history_ module to the saved module.   (DEFAULT: -- input_history.time)

Control arguments: are input_history_ I/O module attach options.
-pathname PATH, -pn PATH
   use PATH as the location of the history file.  The default PATH is:
     [homedir]>[user name].history
-perprocess, -pp
   use a temporary history file created in the process directory.
-permanent, -perm
   use only a permanent the history file.

-lines N, -ln N
   recommends a size for the history file, in lines.  The default is 200 lines (about
   2 records of storage, if the average input line is 40 characters in length).
-truncate, -tc
   if the history file already exists, truncates this file as part of the attach
   operation.  The default is to extend the existing file.

-debug, -db
   display debugging messages as input_history_ is attached or detached, and when
   input scrolls off top of history file.
-end
   separates source_switch or attach control arguments from next input_history command
   operation given within the same command.


Notes on multiple operations:
Several ih operations may be given with the same command.  For any operation that
requires a switch_name or control arguments, use the -end control argument to end that
operation, and begin the next.  For example:
   ih debug  pat  attach -truncate -debug -end  data -end  pat
```

The debug operation displays step-by-step information throughout the attach and detach operations. The print_attach_table operation invokes the command of the same name, to show the I/O configuration. The data operation displays input_history_ I/O module attach data, when this I/O module is attached.

## input_history_.pl1

This new I/O module is a monitor-style I/O module, modeled after the similar audit_ I/O module. Like other monitor-style I/O modules, it supports attach, close, and detach operations. It supports monitored switch opening modes of stream_input or stream_input_output.

Most of its I/O operations simply pass-through to the monitored I/O switch, without impacting the operation in any fashion.

### Entrypoint: $validate_attach_options (not retained in the bound_video_ segment)

This entrypoint is used by the input_history command to validate the I/O module's attach options before the user_i/o is moved (while user_i/o is attached). During this pre-attach validation, detailed messages may be displayed to diagnose error, warn of unexpected events, etc. Calling sequence for this internal-to-bound_video_ interface is shown below. attach_options contains the same array of options that would be produced by an iox_$attach_loud or iox_$attach_ptr call to the I/O module.

```
dcl  input_history_$validate_attach_options entry (char(*), char(*),
    (*) char(*) var, fixed bin(35));
```

```
call input_history_$validate_attach_options (caller_name, source_switch,
    attach_options, code);
```

## I/O Module Entrypoints

The following I/O module entrypoints implement standard I/O module interfaces, as described in the *MPM Subsystem Writer's Guide (AK92)*, section 4. Please see that manual for information about interface arguments, and operations.

### Entrypoint: $input_history_attach

The attach description names another I/O switch whose operations will be monitored by input_history_. The attach description also identifies a history segment, in which the get_line operation logs monitored input lines. For information about the attach options, see: "Documentation: input_history_.info" (below).

In addition, the attach description accepts an undocumented (-debug, -db) option, that enables extra debugging messages as operations are performed on the history segment.

As in all monitor-style I/O modules, the attach operation automatically opens the I/O switch with the opening mode of its monitored switch.

If the monitored switch is attached to window_io_, then the attach operation saves window_io_ line editor key bindings; and binds keys to its own line editor functions, as shown below.

*Table 4: input_history_ Line Editor Key Bindings*

| Keystroke | Bound to input_history_ Entrypoint |
|---|---|
| ^P | $ih_previous_line |
| ^N | $ih_next_line |
| ^R | $ih_reverse_search |
| ^S | $ih_forward_search |
| ^G | $ih_selection_abort |

### Entrypoint: $ih_get_line (get_line operation)

The get_line operation passes-through to the monitored I/O switch (usually attached to window_io_); but the resulting line (or line-fragment) is logged in the input history file before it is returned to the iox_$get_line caller. Each get_line operation also resets the line editor cursor in the history file to END_OF_HISTORY, so editing associated with the next get_line operation is positioned after the line just logged. A ^P edit request will position to that line.

### Entrypoint: $ih_get_chars (get_chars operation)

This operation is passed through to the monitored switch in a transparent fashion.

### Entrypoint: $ih_put_chars (put_chars operation)

This operation is passed through to the monitored switch in a transparent fashion.

### Entrypoint: $ih_modes (modes operation)

This operation is passed through to the monitored switch in a transparent fashion.

### Entrypoint: $ih_position (position operation)

This operation is passed through to the monitored switch in a transparent fashion.

### Entrypoint: $ih_control (control operation)

Checks whether the operation is one provided by input_history_. If not, the operation is passed through to the monitored switch in a transparent fashion.

input_history_ provides the following control operations. Only the first two are included in input_history_.info. The others are useful for debugging the I/O module.

- get_input_history_lines: gets current value of the -lines N attach option. info_ptr points to:
  ```
  dcl based_linesN fixed bin(21) based(info_ptr);
  ```

- set_input_history_lines: sets a new values for the -lines N attach option. info_ptr points to:
  ```
  dcl based_linesN fixed bin(21) based(info_ptr);
  ```

- get_input_history_version: gets a version identify describing the input_history_ I/O module. info_ptr points to:
  ```
  dcl based_version char(20) varying based(info_ptr);
  ```

- set_input_history_debug: sets value for the -debug attach option. info_ptr points to:
  ```
  dcl based_bool bit(1) aligned based(info_ptr);
  ```

- get_input_history_data: gets a pointer to attach data structure for the input_history_ I/O module. info_ptr points to the ihData structure documented in: input_history_data.incl.pl1.

- io_call, io_call_af: operations invoked by the io_call command or active function. All of the above operations are supported via io_call as a command. get_input_history_lines is supported via io_call as an active function.

### Entrypoint: $ih_close (close operation)

This operation closes the I/O switch. Because monitor-style I/O modules do not provide a separate open operation, only a detach operation is possible when the I/O switch is closed. In addition, if window_io_ line editor key bindings were configured during the attach operation, then they are removed during this close operation; original key bindings are restored in their place.

### Entrypoint: $ih_detach (detach_iocb operation)

This operation unlocks and terminates the history segment; and frees input_history's attach data.

## Line Editor Request Entrypoints

The I/O module line editor request entrypoints implement the interface described in *Multics Menu Creation Facilities (CP51-02)*, Section 4, "Writing Editor Extensions". Please see that manual for information about interface arguments; and see "window_line_editor.incl.pl1" below for proposed extensions to that interface.

### Entrypoint: $ih_previous_line - line editor request bound to ^P

Positions to a previous line in the history segment. The selected line replaces the original line being typed when ^P was entered in the window_io_ line editor. For each get_line operation, the cursor starts at the end of the history segment (looking back at the line logged by the prior get_line operation). Line editor requests within a single get_line operation maintain their cursor position, so subsequent requests in that editor invocation may move from the selected cursor location. Accepts a repetition count (via ESC-ddd or ^U) to position back a number of lines at a time. Or repeated ^P operations position back one line at a time.

### Entrypoint: $ih_next_line - line editor request bound to ^N

Positions forward to the next line in the history segment. This presumes a prior line editor request has positioned backward in the history segment to some location; ^N then positions forward from that location. Accepts a repetition count, as described above for ^P.

### Entrypoint:  $ih_reverse_search - line editor request bound to ^R

Searches backward in the history segment from the current cursor position, looking for a line matching a search string typed after ^R.  This is an incremental search.  As a new character is added to the search string, backward searching continues from the point of the last successful match.  Search continues until one of the following characters is entered:  ESC, ^G, RETURN, or another line editor request (e.g., ^A, ^E, ^B, ^F, etc).  See “Notes on video editing” in the Documentation for input_history_ (below).

### Entrypoint:  $ih_forward_search - line editor request bound to ^S

Searches forward in the input history segment from the current cursor position, looking for a line matching a search string typed after ^S.  This incremental search works as described for ^R above.  ^S may be entered during a reverse search to change search directions.  Similarly, ^R may be entered during a forward search, to change search directions.

### Entrypoint:  $ih_selection_abort - line editor request bound to ^G

Stops looking in the history segment, and returns to the window_io_ line editor with the line that was being typed when the first history-related editing request was called during this window_io_ invocation (get_line operation).

## input_history_data.incl.pl1

```
dcl   ihDataP ptr;
dcl  1 ihData aligned based(ihDataP),

     2 ioModule,                    /* I/O module data required by iox_.                     */
       3 history_iocbP ptr,         /* Pointer to input_history_ IOCB (often user_i/o)       */
                                    /*  Identifies IOCB that owns attach data structure.     */
       3 source_iocbP ptr,          /* Pointer to source (window_io_) IOCB                   */
                                    /*  Our routines use it to get/put terminal chars.       */
       3 source_is_window_io_ bit(1) aligned,
                                    /* Flag indicating source is attached to window_io_.     */
                                    /*  This permits adding input line editing requests to   */
                                    /*  search/edit/re-enter lines from the history file.    */
       3 ioModulePad fixed bin,
       3 attach_descrip char(128) var,
                                    /* input_history_ I/O attach description.                */
       3 open_descrip char(32) var, /* input_history_ open description.                      */

     2 hist,                        /* Captured input lines (The History)                    */

       3 attachOpt,                 /* History segment description from attach options.      */
         4 path_dir char(168) unal, /*  - directory containing this segment.                 */
         4 path_ename char(32) unal,/*   - entryname of this segment.                        */
         4 limit_linesN fixed bin(21),
                                    /*  - desired max lines kepts in this segment.           */
                                    /*     - Actual lines may be greater, as we limit data   */
                                    /*       movement within segment to 1 page at a time.    */
          4 attachOptPad fixed bin,

       3 segmentData,               /* Physical information about the history segment.       */
         4 segLockP ptr,            /*  - history lock segment (for permanent history)       */
         4 segP ptr,                /*  - baseptr of containing history segment.             */
         4 bc fixed bin(24),        /*  - length of this segment (in bits).                  */
         4 segL fixed bin(21),      /*  - length of this segment (in characters).            */
         4 linesN fixed bin(21),    /*  - length of this segment (in lines).                 */
         4 scrollableL fixed bin(21),
                                    /*  - length of first page (in chars).                   */
                                    /*     Note that if final scrollable line starts on first*/
                                    /*     page, and extends onto subsequent page(s),        */
                                    /*     scrollableL include all characters of this line.  */
         4 scrollable_linesN fixed bin(21),
                                    /*  - length of first page of this segment (in lines).   */
         4 extraLineL fixed bin(21),/*   - if >0: length of window_io_ editor line appended   */
                                    /*    temporarily to History Segment.                    */
                                    /*    Should be >0 only when in our XXX_search requests. */
         4 flags aligned,
           5 permanent bit(1) unaligned,
                                    /*  - TRUE = permanent history seg; FALSE = temporary    */
                                    /*  - TRUE = permanent history seg; FALSE = temporary    */
           5 flags_pad bit(70) unaligned,

     2 edit,                        /* Line editor request data saved between calls from the */
                                    /* window_io_ line editor.                               */
       3 currentLine aligned like historyLinePosition,
                                    /*  - If linesFromEnd ^= 0, then this is history line    */
                                    /*     when the prior editing request returned.           */
                                    /*     Otherwise, it does not contain meaningful data.   */
                                    /*      - Each time the window_io_ line editor returns an*/
                                    /*        input line (in iox_$get_line call), currentLine*/
                                    /*        is reset to END_OF_HISTORY.                    */
       3 whenEditing,               /*  Data valid only which running in an editor request.  */
         4 workingLine aligned like historyLinePosition,
                                    /*  - This is position data used while an editing        */
                                    /*     function runs.  It is constructed by editing fcn  */
                                    /*     support routines.  Just before editing function   */
                                    /*     returns, it is moved to currentLine.              */
           4 origEditorBuffer char(512) var,
                                     /*  - Contents of line_editor_info.buffer when a history */
                                     /*     editing request first used in this invocation of  */
                                     /*     the window line editor.  May need this to restore */
                                     /*     what user started typing before deciding to look  */
                                     /*     back at prior input lines.  (^N eventually        */
                                     /*     positions beyond end of History Segment, at which */
                                     /*     time this original buffer is given back to editor.*/
           4 origCursorI fixed bin(21),
                                     /*  - Cursor position in line_editor_info.buffer when    */
                                     /*     history edit request starts.                      */

       2 savedKeyBindingsP ptr;      /* window_io_ line editor key bindings to be restored at */
                                     /*  detach time.                                         */

  dcl 1 historyLinePosition,         /* Location identifier for lines in History Segment.     */
      2 linesFromEnd fixed bin(21),  /* = 0  positioned just after last line in history seg.  */
                                     /*      (at EOF, lineP may not be set                    */
                                     /* =-1  positioned at last (possibly incomplete) line.   */
                                     /* =-2  positioned at 2nd-to-last,  (complete) line      */
                                     /*      (Lines deemed "complete" when a line(-fragment)  */
                                     /*       ending with NL is read via iox_$get_line.)      */
      2 lineL fixed bin(21),         /* Length of this history line (not including any NL)    */
      2 lineP ptr,                   /* Pointer to start of this history line.                */
      2 cursorI fixed bin(21),       /* Base_1 index of cursor within the line:               */
                                     /*   assert: 0 < cursorI <= lineL+1   (AFTER end-of-line)*/
      2 matchL fixed bin(21);        /* Length of most recent matched string in this line.    */
                                     /*   = 0, when line reached via ^P or ^N.                */


  dcl 1 END_OF_HISTORY aligned int static options(constant),
                                     /* Special historyLinePosition that denotes start of a   */
                                     /*  window_io_ line editor invocation.  ih_get_line sets */
                                     /*  ihData.currentLine = END_OF_HISTORY each time a line */
                                     /*  (fragment) is read by the process.                   */
      2 linesFromEnd fixed bin(21) init(0),
      2 lineL fixed bin(21)        init(0),
      2 lineP ptr                  init(null()),
      2 cursorI fixed bin(21)      init(1),
      2 matchL fixed bin(21)       init(0);

  dcl 1 searchStackItem aligned,     /* Item in the line editor request searchStack.          */
      2 dir fixed bin(1),            /*  - search direction: -1 = REVERSE; +1 = FORWARD       */
      2 search char(40) var,         /*  - current searchString for the incremental search.   */
      2 pos like historyLinePosition;/*  - position after search succeeded.                   */

  dcl 1 hLockSeg aligned based (ihData.hist.segmentData.segLockP),
      2 lock_descriptor char(80),
      2 history_path char(168),
      2 lock_sentinel char(12),
      2 lock_word bit(36),
      2 lock_terminator char(28);

  dcl  hLockSegDESCRIPTOR char(80) aligned int static options(constant)
       init("Please do not modify or delete this segment.
It contains the lock word for:
    ");
  dcl  hLockSegSENTINEL_LOCKED char(12) aligned int static options(constant) init("

   Lock:  ");
  dcl  hLockSegSENTINEL_UNLOCKED char(12) aligned int static options(constant) init("

Unlocked  ");
  dcl  hLockSegTERMINATOR char(28) aligned int static options(constant) init("
          ________________
");
```

## window_io_iox_.pl1

This component of the window_io_ I/O module implements:

- the window_io_ input line editor, which:
    - processes each break character in the input lines, as it is entered by the user;
    - invokes line edit requests bound to a particular break character (including external requests such as those supplied by input_history_);
- the actual entrypoints called by the window_editor_utils_ transfer vector, entrypoints providing support functions to the external line edit requests.

A small set of changes are proposed to provide functions needed by external edit requests to properly implement selection of input history lines to return to the line editor. Objectives for these changes are as follows.

- input_history_'s incremental search editor requests read characters given by the user specifying the search string to look for in prior input lines. As each character is typed, an incremental search is performed. If a matching input line is found: the new search character is added to the total search string; and the total search string and match input line are displayed to the user. This display occurs on the window line being entered/edited by the window_io_ line editor.

  The current implementation of the line editor provides no way for a request to do such incremental processing, or to display intermediate results. Providing access to the window_io_ redisplay_input_line routine solves this problem.

  The developers of the current window_io_ code had envisioned providing this redisplay capability as one of the window_editor_utils_ support routines. However, due to dependencies of the internal redisplay_input_line procedure on its containing environment, there was no way to make this internal subroutine visible as a separately-callable external entrypoint (similar to the entrypoints called by the other window_editor_utils_ transfer vector entries).

  The current proposal resolves the dependency issue by passing the internal procedure as an entry variable in the line_editor_info structure. An entry variable consists of two pointers.
    - codeptr(entry_variable) points to code for the procedure to be called.
    - environmentptr(entry_variable) points to the stack frame of its containing procedure.

  The environment pointer links the redisplay_input_line code and to data it depends on in its containing stack frames.

- input history_'s incremental search editor requests also need a way to elegantly stop searching, if the user types a break character not known to the incremental search code. These include standard editor requests like ^A, ^B, ^D, ^E, ^F, ^T; and to two of the editor requests added by incremental search: ^N and ^P. When such character is entered, the search request needs some method to have the window_io_ line editor handle the break character. It has no way to "unget" the character (return it to terminal_io_, so it could be re-read by the window_io_ line editor). So some method is needed to return a break character as part of the result from an external line editor request.

- A design flaw was discovered in the window_editor_utils_$get_top_kill_ring_element interface. This routine accepts a text parameter, declared as char(*), in which the kill ring element is to be returned; but provides no information about actual length of the kill ring data, or whether its actual length exceeds the length of the text argument.

  ```
  dcl window_editor_utils_$get_top_kill_ring_element (ptr, char(*), fixed bin(35));
  call window_editor_utils_$get_top_kill_ring_element (line_editor_info_ptr, text, code);
  ```

  While the proposed implementation for input_history_ does not need this function, this serious design flaw needs to be remedied.

The following changes are proposed.

1. Change the edit_line/get_line entrypoints to initialize line_editor_info structure (declared as LEI in this code) including new version_3 elements. (See window_line_editor.incl.pl1, below). These include:
   a. LEI.version = line_editor_info_version_3;
   b. LEI.flags.break_given = “0”b; /* new_break_character has not been returned */
   c. LEI.new_break_character = “”;
   d. LEI.pad2 = “”;
   e. LEI.redisplay = redisplay_input_line.

2. Change the read_input_line subroutine (internal to edit_line/get_line) which calls line editor requests, passing a pointer to the LEI structure.
   a. For each break character it reads from the terminal, it calls the internal process_break subroutine; that routine actually invokes either the (internal or external) editor request bound to the break character.
   b. Upon return by process_break, read_input_line will now check if LEI.break_given is set. If so, it copies LEI.new_break_character into its local break_char variable; clears LEI.break_given and LEI.new_break_character; and loops back to call process_break with the additional break returned by the editor request.

3. Change the setup_util_call internal procedure, which validates arguments passed to internal line editor requests, to accept either line_editor_info_version_2 or line_editor_info_version_3 as the value in LEI.version. Since storage for the version 2 part of the structure was not changed in version 3, the existing line editor routines do not require modifications, other than to accept the new LEI.version value.

4. Repair a defect in the internal redisplay_input_line procedure, to cause it to reference the line_editor_info structure pointed to by its lei_ptr parameter, rather than the structure declared in its containing procedure. In most cases, these are the same structure. However, now that redisplay_input_line is being passed as an element in the line_editor_info structure, it is important that it access its parameter structure. An external editor request may pass a different copy of the structure to this routine.

5. Add a new window_io_iox_$get_top_kill_ring_string_ entrypoint, which works like the flawed $get_top_kill_ring_element_ entrypoint, but returns its data in a varying-length string, so actual length of the data is known to the caller. It also returns an error_table_$long_record if maxlength(text_var) is too small to hold the actual kill ring string.

```
dcl window_editor_utils_$get_top_kill_ring_string (ptr, char(*) varying, fixed bin(35));
call window_editor_utils_$get_top_kill_ring_string (line_editor_info_ptr, text_var, code);
```

Note that a search of the Multics libraries found no external editor requests extending the window_io_ line editor interface. Thus, no Multics code is affected by this change other than as described above.

If user-provided applications have their own editor requests, their code may break if they validate the LEI.version value pointed to by their input parameter. If they do not validate LEI.version but simply assume a version 2 structure, their code will continue to work correctly.

## window_line_editor.incl.pl1

Original file in black text; new or changed lines in blue.

The line_editor_info_ptr is passed as an argument to each window_io_ line editor request. Refer to the section on window_io_iox_.pl1 for information about how these new elements of this structure extend the window_io_ line editor interface.

```
dcl  line_editor_info_ptr     pointer;

dcl  1 line_editor_info       aligned based (line_editor_info_ptr),
       2 version              char(8),
       2 iocb_ptr             pointer,     /* to current window */
       2 repetition_count      fixed bin,  /* number of times to perform operation */
       2 flags,
         3 return_from_editor  bit(1) unaligned,  /* to end editing session */
         3 merge_next_kill     bit(1) unaligned,  /* don't touch */
         3 old_merge_next_kill bit(1) unaligned,  /* don't touch */
         3 last_kill_direction bit(1) unaligned,  /* don't touch */
         3 numarg_given        bit(1) unaligned,
         3 suppress_redisplay  bit(1) unaligned,  /* only meaningful if return_from_editor set */
         3 break_given         bit(1) unaligned,  /* version_3: new_break_character has been   */
                                                  /* set by editing function.                  */
         3 pad                 bit(29) unaligned,
       2 user_data_ptr  pointer,           /* for carrying user state information */
       2 cursor_index   fixed bin(21),     /* 0 < cursor_index <= line_length + 1 */
       2 line_length    fixed bin(21),     /* 0 <= line_length <= length (input_buffer) */
       2 input_buffer   character(512) unaligned,
       2 key_sequence          character(128),
                                     /* key sequence which caused user routine to be invoked  */
       2 redisplay      entry(ptr),  /* version_3: Redisplays line_editor_info.input_buffer.  */
                                     /* call line_editor_info.redisplay(line_editor_info_ptr);*/
       2 new_break_character   character(1) unal,
                                     /* version_3: if break_given, break char is returned     */
                                     /* to window_io_ line editor by external edit function.  */
                                     /* Line editor will evaluate/implement this break char.  */
       2 pad2                  character(3) unal;

dcl  line_editor_input_line   char(line_editor_info.line_length)
                      based (addr (line_editor_info.input_buffer));

dcl  line_editor_info_version_1
                      char(8) static options (constant) init ("lei00001");

dcl  line_editor_info_version_2
                      char(8) static options (constant) init ("lei00002");

dcl  line_editor_info_version_3
                      char(8) static options (constant) init ("lei00003");

/* User supplied editor routines may want to preserve state information of
   their own across calls.  user_data_ptr points to a chain of data structures
   that these routines may use.  The structures should all have the same header
   (declared here), and the id field can be used to identify which structures
   belong to which routines. */

dcl  1 line_editor_user_data_header
                      aligned based,
       2 id           fixed bin,
       2 next_user_data_ptr   pointer;
```

## ihtest.pl1

When debugging input_history_, its normal function of providing input lines to the process collides with the need to provide input controlling debugging operations. Therefore, input_history_ is debugged mostly by adding ioa_ calls at strategic points in the code to trace its activities on user_output without requiring input from the user.

However, when testing the requests which input_history_ adds to the window_io_ line editor, even this ioa_ tracing strategy fails. The trace output data interferes with presentation by the line editor of the input data being edited.

ihtest avoids this problem by simulating the window_io_ line editor calls to particular input_history_ edit requests, so the requests may be tested without input_history_ actually being attached to window_io_. This simulated environment is created as follows.

1. ihtest tells input_history_ to fabricate its attach data structure (ihData), using a special entrypoint in the I/O module: input_history_$validate_attach_options. Pointer to the returned ihData structure is then stored in input_history_'s iocb_dict cache (associated with the iox_$user_io IOCB).
2. The window system in invoked, with window_io_ attached to the user_i/o switch. input_history_ attach operation is NOT performed.
3. Each simulated invocation of an input_history_ line editor request (e.g., ^P or ^N) invokes one of the editing requests provided by input_history_. Each editor request invocation uses and updates the simulated attach_data in the iocb_dict cache.

This scheme works because the window_io_ editor normally calls external edit requests by passing them a line_editor_info structure, which contains an iocb_ptr identifying the window_io_ instance making the call (usually the user_i/o switch attached to the window_io_).

In the simulated test environment, ihtest creates its own line_editor_info structure, with a pointer to the user_i/o IOCB. Each input_history_ edit request uses this iocb_ptr to find the simulated attach data in the input_history_ cache.

### Test Operations:

Several operations may be given in the same command, to simulate calls to same/different edit routines during a single invocation of the window_io_ line editor.

-display shows all entries in input_history_’s iocb_dict cache.

-clear clears the user_i/o entry in the input_history_ iocb_dict cache. This simulates a new invocation of the window_io_ line editor, by resetting the input_history_ attach data.

-next, +nnn invokes input_history_$ih_next_line for testing. When the positive number form is used, nnn is the repetition count given in the line editor input when invoking the edit request (e.g., 3 or +3 invokes ih_next_line with a repetition count of 3).

-previous, -nnn
invokes input_history_$ih_previous_line for testing. When the negative number form is used, nnn is the repetition count given in the line editor input when invoking the edit request (e.g., -5 invokes ih_previous_line with a repetition count of 5).

-reverse_search, -reverse, -r
invokes input_history_$ih_reverse_search for testing. The user is then prompted for incremental search characters.

-search, -srch, -s
invokes input_history_$ih_forward_search for testing. The user is then prompted for incremental search characters.

## window_editor_utils_.alm

This routine was changed to add a new transfer vector entry:

window_editor_utils_$get_top_kill_ring_string => window_io_iox_$get_top_kill_ring_string_

## bound_video_.bind

Changes to this bind file add components related to the input_history_ I/O module.

- To bound_video_, add the names: input_history, ih, and input_history_
- To bound_video_.archive, add the components: input_history, input_history_, and ihtest.
- For the input_history component:
    - use ih as a synonym.
    - retain only the entrypoints: input_history, ih
- For the input_history_ component:
    - retain all entrypoints (relating to I/O modules, and to the line editor requests), except the following:
        - input_history_$input_history (not required for I/O modules)
        - test_iocb_dict_clear, test_iocb_dict_display, test_iocb_dict_get, test_iocb_dict_set (all test entrypoints used only by ihtest command)
        - validate_attach_options (special entrypoint supporting attach operations, used by the input_history command, and ihtest test program).
- For the ihtest component, retain only the entrypoint: ihtest. (It would be invoked as ih$ihtest, since ihtest is not a name to be added to bound_video_.)

## Documentation for input_history: input_history.info

```
05/16/16  input_history, ih

Syntax as a command:
   ih attach {source_switch {monitored_switch}} {-control_args}
or:
   ih detach {source_switch}
or:
   ih version {source_switch}


Function: the attach operation inserts the input_history_ I/O module
between the IOCB named by source_switch and its currently attached I/O
module.  This current attachment is saved for future use on a new IOCB
named by the monitored_switch argument.

Each subsequent get_line operation on source_switch passes-thru
input_history_ to the monitored_switch.  input_history_ appends each
line received from the monitor_switch to a history file, then returns
the line to its caller.  Other operations pass-thru to the
monitored_switch, with results returned to the caller without action
taken by input_history_.


If source_switch is attached to window_io_ before the input_history
attach operation, then keystrokes are added to the window_io_
real-time line editor (which runs for each get_line operation) to
select past input lines (saved in the history file) for editing and
re-input.  See video_editing.gi.info for further information.

The detach operation stops capture of input lines in the history file,
and reconnects the source_switch to its original I/O module.  This
reverses the changes made by the input_history attach operation.

The version operation displays version numbers for this command,
and for the input_history_ I/O module.


Arguments:
source_switch
   is an existing switch attached to an I/O module open for
   stream_input or stream_input_output.  Subsequent get_line
   operations on this switch will be captured in the history file.
   (DEFAULT: -- user_i/o)
monitored_switch
   is a new switch created by the input_history command to save the
   I/O module attachment currently on the input_switch, so subsequent
   I/O requests can pass-thru the input_history_ module to the saved
   module.   (DEFAULT: -- input_history.time)


Control arguments: are input_history_ I/O module attach options.
```

```
-pathname PATH, -pn PATH
   use PATH as the location of the history file.  The default PATH is:
     [homedir]>[user name].history
-perprocess, -pp
   use a temporary history file created in the process directory.
-permanent, -perm
   use only a permanent the history file.  See "Notes on the history
   file" for further information on these three control arguments.


-lines N, -ln N
   recommends a size for the history file, in lines.  The default is
   200 lines (about 2 records of storage, if the average input line is
   40 characters in length).  (See "Notes on the history file" below.)
-truncate, -tc
   if the history file already exists, truncates this file as part of
   the attach operation.  The default is to extend the existing file.


Notes on input_history_:
For more information about the input_history_ I/O module, use the help
command to display the following files:

  input_history_.info         (or: ih_.info)
  video_editing.gi.info
```

## Documentation for input_history_: input_history_.info

```
05/16/16  input_history_

Function: The input_history_ I/O module passes-thru I/O operations
to a monitored switch.  For a get_line operation, it captures the
line returned by the monitored switch in a history file, before
returning it to the caller.

If the monitored switch is attached to the window_io_ module, then
the window_io_ real-time input line editor is augmented with editing
requests to search the history file for past input lines to
re-enter.  See "Notes on video editing" below.

Use the input_history command to facilitate attachment of the
input_history_ I/O module.  See: input_history.info (or: ih.info)


Syntax:
   input_history_ monitored_switch_name {-control_args}


Notes on attach description:
monitored_switch_name
   is the name of an I/O switch whose input lines are captured.  It
   must be attached to an I/O module open with an opening mode of
   stream_input or stream_input_output.

-pathname PATH, -pn PATH
   use PATH as the location of the history file.  The default PATH is:
     [homedir]>[user name].history
-perprocess, -pp
   use a temporary history file created in the process directory.
-permanent, -perm
   use only a permanent the history file.  See "Notes on the history
   file" for further information on these three control arguments.


-lines N, -ln N
   recommends a size for the history file, in lines.  The default is
   200 lines (about 2 records of storage, if the average input line is
   40 characters in length).  (See "Notes on the history file" below.)
-truncate, -tc
   if the history file already exists, truncates this file as part of
   the attach operation.  The default is to extend the existing file.


Notes on the history file:
Past input lines are logged in the history file as a stream of
characters.  The file bit count is adjusted after each line is
appended.  This permits searching or examining the file: using the
print command or a file editor; or using video editing functions added
to the window_io_ line editor by the input_history_ module.  See
"Notes on video editing" below.
```

Permanent history file

Holds a log of past input lines, and remains in the file system after the process ends. A subsequent process may log additional input lines in this file, and may select and re-input lines from the earlier process.

Temporary history file

Holds input lines only for the current process. The user may select and re-input lines while this process runs. However, the file is deleted when the process ends.

One process per history file

Several processes may not share the same history file concurrently. For each permanent history file (XXX.history), a corresponding process identity file (XXX.hisLock) records the lock_id of the process using that history file. While that process is running, input_history_ prevents another process from attaching with that file.

Read/write access required

The process must have read/write access to attach using an existing history file. An access error might occur if using a file created by another user or group; or created by a process with an access class that differs from the current process authorization.

Permanent history file attach error

If a permanent history file cannot be reused (due to locking or access error), either: a temporary history file is created; or if -permanent was given, then an error is reported and the attach operation fails. If both -pathname and -perprocess are given, then a temporary file is created only when the file given with -pathname is locked or inaccessible. If the permanent history file was locked, the temporary file is initialized by copying the permanent file.

Requiring a temporary history file

If -perprocess is given without a -pathname, then a temporary file is always created.

Automatic purging of oldest input

When a history file reaches its recommended size of N lines (see the -lines attach argument), the oldest lines are removed from the top of the file to permit new lines to be appended to the end of the file. These scroll operations are grouped into page-size chunks, to avoid excessive overhead during each input operation. Thus, the file may actually contain N-lines, plus some additional lines.

Notes on video editing:
When the input_history_ I/O module is attached, several editing requests are added to the window_io_ input line editor. The previous-line (^P) and next-line (^N) requests accept numeric repetition counts (e.g., ESC 7 ^P to move back 7 lines; or ^U ^N to move forward 4 lines).

^P
: Select the previous line in the history file for editing and re-input.

^N
: Select the next line in the history file for editing and re-input.

^G
: Abort selection of a prior input line. Original line is again on display in the window_io_ line editor.

Incremental searches of the input history are started using one of the following requests.

^R
: Perform an incremental search backward in the history file, looking for a line that matches characters typed following ^R.

^S
: Perform an incremental search forward in the history file, looking for a line that matches characters typed following ^S.

As each character is added to the incremental search string, the history line matching the search string is displayed.

- Backspace, DEL, or # (your erase character)
  - Remove a character from the incremental search string to undo part of the search operation. Different characters may then be added to the search string.

Incremental searching ends with one of the following characters.

- ESC
  - Matching line returned to window_io_ line editor for editing and re-input.
- ^G
  - Incremental search is aborted as described in ^G request above.

- (other window_io_ line editor control key)
  - The selected input line is returned to window_io_ line editor; the window_io_ edit function bound to the given control key is then applied to the selected line. Two of the many window_io_ requests that can be applied are listed below.

- RETURN (^M), NL (^J)
  - The selected history line is re-input, as is.

- ^E
  - Move to end of the selected line, where further edit requests may be applied to the line.

Notes on the open operation:
Because input_history_ is a monitoring-style I/O module, most I/O operations pass-thru directly to the monitored switch. For this reason, the open operation for input_history_ is performed automatically at attach time. It uses the same opening mode as the monitored switch.

Notes on other operations:
Put Chars Operation
is a pass-thru to the monitored I/O module.

Get Chars Operation
is a pass-thru to the monitored I/O module.

Get Line Operation
This is primarily a pass-thru to the monitored I/O module. However, any lines (or partially-read line fragments) are logged in the history file before returning the input to the caller.

Modes Operation
is a pass-thru to the monitored I/O module. input_history_ has no modes of its own.

Position Operation
is a pass-thru to the monitored I/O module.

Control Operation
is primarily a pass-thru to the monitored I/O module. However, the following input_history_ control operations are also supported.

get_input_history_lines
returns the current recommended size for the history file, in lines.

set_input_history_lines N
gives a value for the recommended size for the history file, in lines.

## Documentation: video_editing.gi.info

The section titled “List of input_history_ input editor requests” is added. Unchanged information from the original info segment is shown in black text. New or modified lines are in blue.

```
04/15/16  Multics video system input editor requests

Function: A process obtains input from a video terminal using
the window_io_ I/O module (video system).  Each input line may
optionally be saved in a USER.history file using the input_history
command, with its input_history_ I/O module.

The window_io_ real-time line editor provides a subset of Emacs
requests for editing each input line as it is entered.


List of video system input editor requests:
   The following list first gives the ASCII character
   and then the operation associated with that character.

^F
   Position the cursor one character forward.
^B
   Position the cursor one character backward.
ESC F
   Position the cursor one word forward.


ESC B
   Position the cursor one word backward.
^E
   Position the cursor to end of the line.
^A
   Position the cursor to beginning of the line.
^D
   Delete one character forward.
DEL or #
   Delete one character backward.
ESC D
   Delete one word forward.
ESC DEL or ESC #
   Delete one word backward.


^K
   Delete from the cursor to end of line.
@
   Delete from the cursor to the beginning of the line.
^Y
   Retrieve the last deleted characters or line.
ESC Y
   Retrieve previously deleted characters or line.
^T
```

```
   Interchange the previous two characters with each other.
ESC T
   Interchange the current (or last) word with the previous word.
^Q
   Accept the next character without treating it as an editor request.


^L
   Clear the window and redisplay the input line.
ESC C
   Capitalize (only) the first character of the current (or last) word.
ESC L
   Change the current (or last) word to lowercase.
ESC U
   Change the current (or last) word to uppercase.
ESC ?
   List valid editor request characters.


List of input_history_ input editor requests:
When the input_history_ I/O module is configured, several more
requests are added to the input line editor.  For information, see
"Notes on video editing" in: input_history_.info


^P
   Select the previous line in the USER.history file for editing.
^N
   Select the next line in the USER.history file for editing.
^R
   Perform an incremental search backward in the USER.history file,
   looking for a line that matches characters typed following ^R.
   Search ends by typing ESC: line can then be edited further; or by
   hitting RETURN: line is re-input as is.
^S
   Perform an incremental search forward in the USER.history file,
   looking for a line that matches characters typed following ^S.
   Search ends by typing ESC: line can then be edited further; or by
   hitting RETURN: line is re-input as is.


List of numeric repetition pre-requests:
   Some requests may be preceded by a repetition count, which causes
   the request to be performed a given number of times.  A
   repetition count is entered by preceding the request with
   one or more of the following pre-requests.

ESC n
   Where n is one or more numeric digits: enter a count of n.
   For example, ESC 100 enters a count of 100.
^U
   Multiply the repetition count entered so far by 4.  If no count
   has been entered, set the repetition count to 4.
```

Summary of cursor positioning requests and deletion requests:

| | | One character | One Word | To Edge of Line |
|---|---|---|---|---|
| Move Cursor | Right | Control-F | ESC F | Control-E |
| | Left | Control-B | ESC B | Control-A |
| Delete | Right | Control-D | ESC D | Control-K |
| | Left | DEL<br>or # | ESC DEL<br>or ESC # | @ |

Notes:
The ASCII characters given in the above list are the characters associated with the corresponding functions by default. These associations can be displayed with the command

```
  io_call control WINDOW_SWITCH get_editor_key_bindings key_sequence
```

and may be changed with the command

```
  io_call control WINDOW_SWITCH set_editor_key_bindings
    key_sequence1 {user_routine1} {control_args1} ...
    key_sequenceN {user_routineN} {control_argsN}
```

A "word" is a string of one or more consecutive "token characters". The set of token characters may be displayed with the command

```
  io_call control WINDOW_SWITCH get_token_characters
```

and may be changed with the command

```
  io_call control WINDOW_SWITCH set_token_characters TOKEN_CHAR_STRING
```

Type "help window_io_" for details about these commands.

## Documentation for window_editor_utils_.alm: window_editor_utils_.info

Documentation for the new $get_top_of_kill_ring_string entrypoint is added. Unchanged information from the original info segment is shown in black text. New or modified lines are in blue.

```
05/10/16  window_editor_utils_


A library of editor utility routines is provided for the benefit of
user-written editor routines.  Some operations can be performed
simply by a user-written editor routine.  For example, to position
the cursor to the end of the line, set the cursor_index variable to
one greater than the value of the line_length variable.  Most
actions are more complex than this, however.  So it is recommended
that the following editor utility routines be used to perform most
changes.

The following is a description of these routines.  In all cases,
line_editor_info_ptr is the pointer to the editor data structure
that is supplied as an argument to user-written editor routines.


Entry points in window_editor_utils_:


:Entry:  insert_text:  07/31/92 window_editor_utils_$insert_text

Function: Inserts the supplied character string into the input buffer
at the current cursor location.  If the string is too large to fit in
the remaining buffer space, the code
error_table_$action_not_performed is returned.  This routine updates
the line_length field of the line_editor_info structure, and the
cursor_index if necessary.


Syntax:
dcl window_editor_utils_$insert_text entry (ptr, char(*), fixed bin (35));

call window_editor_utils_$insert_text (line_editor_info_ptr, "text",
     code);


Arguments:
line_editor_info_ptr
    pointer to editor data structure. (Input/Output)
"text"
    text string to be inserted.  (Input)
code
    status code.  (Output)


:Entry:  delete_text:  07/31/92  window_editor_utils_$delete_text
```

Function: Deletes a specified number of characters from the input buffer at the current cursor location. If there are not enough characters remaining between the cursor and the end of the line, error_table_$action_not_performed is returned and no characters are deleted. The line_length component of the line_editor_info_structure is updated, and the cursor_index if necessary.

Syntax:

```
dcl window_editor_utils_$delete_text entry (ptr, fixed bin, fixed bin (35));

call window_editor_utils_$delete_text (line_editor_info_ptr, count,
     code);
```

Arguments:

line_editor_info_ptr
    pointer to editor data structure. (Input/Output)

count
    number of characters to be deleted. (Input)

code
    status code. (Output)

:Entry: delete_text_save: 07/31/92 window_editor_utils_$delete_text_save

Function: This entrypoint is identical to delete_text, except that the deleted text is added to the kill ring. The kill_direction flag is used during kill merging to decide whether the killed text will be concatenated onto the beginning or end of the current kill ring element. "1"b is used to specify a forward kill (e.g. FORWARD_DELETE_WORD), "0" a backward kill.

Syntax:

```
dcl window_editor_utils_$delete_text_save entry
     (ptr, fixed bin, bit(1), fixed bin (35));
call window_editor_utils_$delete_text_save
    (line_editor_info_ptr, count, kill_direction, code);
```

Arguments:

line_editor_info_ptr
    pointer to editor data structure. (Input/Output)

count
    number of characters to be deleted. (Input)

kill_direction
    flag to determine which end of current kill ring element deleted text will be concatenated to. (Input)

code
    status code. (Output)

:Entry: move_forward: 07/31/92 window_editor_utils_$move_forward

Function: Advances the cursor forward a specified number of characters in the input line. If there are not enough characters between the cursor and the end of the line, error_table_$action_not_performed is returned.

Syntax:
```
dcl window_editor_utils_$move_forward entry (ptr, fixed bin, fixed bin (35));

call window_editor_utils_$move_forward (line_editor_info_ptr,
     count, code);
```

Arguments:
```
line_editor_info_ptr
    pointer to editor data structure.  (Input/Output)
count
    number of characters to move forward.  (Input)
code
    status code.  (Output)
```

:Entry: move_backward: 07/31/92  window_editor_utils_$move_backward

Function: Moves the cursor backward a specified number of characters in the input line. If there are not enough characters between the cursor and the end of the line, error_table_$action_not_performed is returned.

Syntax:
```
dcl window_editor_utils_$move_backward entry (ptr, fixed bin, fixed bin (35));

call window_editor_utils_$move_backward
    (line_editor_info_ptr, count, code);
```

Arguments:
```
line_editor_info_ptr
    pointer to editor data structure.  (Input/Output)
count
    number of characters to move backward.  (Input)
code
    status code.  (Output)
```

:Entry: move_forward_word: 07/31/92  window_editor_utils_$move_forward_word

Function: Updates the cursor_index to a position after the next word (or token) in the input line. A word is defined via the editor's set of token delimiters, set via the set_token_delimiters control order.

Syntax:
```
dcl window_editor_utils_$move_forward_word entry (ptr, fixed bin (35));
```

```
call window_editor_utils_$move_forward_word (line_editor_info_ptr,
     code);
```

Arguments:
line_editor_info_ptr
    pointer to editor data structure.  (Input/Output)
code
    status code.  (Output)

:Entry: move_backward_word: 07/31/92  window_editor_utils_$move_backward_word

Function: Updates the cursor_index to a position before the preceeding word (or token) in the input line.  A word is defined via the editor's set of token delimiters, set via the set_token_delimiters control order.

Syntax:

```
dcl window_editor_utils_$move_backward_word entry (ptr, fixed bin (35));

call window_editor_utils_$move_backward_word
    (line_editor_info_ptr, code);
```

Arguments:
line_editor_info_ptr
    pointer to editor data structure.  (Input/Output)
code
    status code.  (Output)

**:Entry: get_top_kill_ring_string: 05/10/16**
**window_editor_utils_$get_top_kill_ring_string**

**Function: Returns the top kill ring element.**

**Syntax:**

```
dcl window_editor_utils_$get_top_kill_ring_string entry
    (ptr, char(*) varying, fixed bin (35));
call window_editor_utils_$get_top_kill_ring_string
    (line_editor_info_ptr, text, code);
```

**Arguments:**
**line_editor_info_ptr**
    **pointer to editor data structure.  (Input/Output)**
**text**
    **string containing contents of top kill ring element.  (Output)**
**code**
    **status code.  (Output)  error_table_$long_record is returned if the top element of the kill ring is longer than maxlength(text).**

```
:Entry: rotate_kill_ring: 07/31/92  window_editor_utils_$rotate_kill_ring

Function: Rotates the kill ring.


Syntax:
dcl window_editor_utils_$rotate_kill_ring entry (ptr, fixed bin (35));

call window_editor_utils_$rotate_kill_ring
    (line_editor_info_ptr, code);


Arguments:
line_editor_info_ptr
    pointer to editor data structure.  (Input/Output)
code
    status code.  (Output)
```