call Command

A proposal for a command to invoke subroutines and functions from the command line.

Author: Gary Dixon
Date: December 18, 2016

Abstract

When developing new code, testing external subroutines and functions directly allows experimenting with their entrypoint inputs, examining outputs, and exploring side-effects. This bulletin proposes a command interface that: uses entrypoint calling sequence information to construct an argument list; invokes the entry point with that argument list; and upon return, displays selected arguments.

Table 1: Revision History

Date	Revision	Author	Comment
2016-07-14	0.1	Gary Dixon	Initial Revision.
2016-08-01	1.0	Gary Dixon	Completion of publishable MTB.
2016-08-09	1.1	Gary Dixon	Changes stemming from review by Eric Swenson.
2016-08-11	1.2	Gary Dixon	Adding -date_time, -date, and -time formats to call.
2016-12-18	1.3	Gary Dixon	Changes for call.01.04 revising functions in call_dtype_fcns.incl.pl1.

Table of Contents

INTRODUCTION

The Multics operating system is running again, thanks to the efforts of several hardware simulation experts. When exploring the Multics runtime environment, existing Multics commands provide only a top-level view; this might be called the *user interface viewpoint*. However, many Multics innovations lie below this top-level, at the *programmer's viewpoint*.

Multics provides a rich programming environment in which the wide variety of operating system functions and services are offered via subroutines and functions. To explore this programming level, it is often useful to: call these subroutines and functions directly from a command; and examine outputs and side-effects produced by particular input values.

This MTB proposes a **call** command that can invoke external Multics subroutine and function entrypoints from the command line.

Preparing to call a PL/I subroutine or function involves a sequence of tasks that are usually performed by the PL/I compiler. A **call** command will have to perform these tasks, which are summarized in the *Call/Return Tasks* section of this MTB.

Since most of Multics is written in the PL/I Language, a **call** command must follow Multics call/return standards, and should support most PL/I data types. The *Standards* section summarizes this interface information.

Multics provides many tools (include files, support subroutines, etc.) that can assist with the call/return tasks. The *Existing Tools* section identifies these tools.

Several of the remaining call/return tasks are sufficiently complex to justify programming them as separate source modules. Other tasks are useful beyond the purposes of a **call** command, and are therefore programmed external service interfaces to make them available to other programming applications. These are summarized in the *New Tools* section.

Many models exist to define and invoke an application program interface (API). The PL/I Language uses declare and call statements for this purpose. The proposed **call** command uses a more compact form, as described in the *User Interface* section.

Given the complexity of the **call** proposal, a variety of test-related interfaces are needed to exercise and verify **call** command functionality. These are summarized in *Testing the **call** Command*.

CALL/RETURN TASKS

When a PL/I source program is preparing a call to an external entrypoint, the PL/I compiler performs a variety of tasks that implement the call. These tasks are based upon a large amount of data provided within a PL/I source program, or defined within the user's execution environment. Such data includes:

- detailed declaration of the entrypoint to be called, including its parameters and optional function return value.
- declared or default attributes for each argument being passed to an entrypoint parameter.
- search rules for "snapping a link to the named entrypoint". This involves converting a declared entrypoint name into an entry variable (a segment number, and offset within the segment at when entrypoint code begins).

In addition, PL/I follows language-specific rules and operating system-defined protocols for actually making the call:

- rules for converting/promoting a declared argument value to match attributes of the corresponding entrypoint parameter.
    - standards for size and layout of storage for each argument, when conversion/promotion is required.
- PL/I operators for actually calling the entrypoint, and for handling return from the entrypoint.

A command designed to invoke an arbitrary entrypoint must have an equivalent source of information describing the entrypoint to be invoked, and the argument values passed to its parameters. And it must have equivalent mechanisms for performing the tasks outlined above. In particular, it must:

1. Obtain the name/path for the entrypoint to be called. Convert this virtual entry to a PL/I entry variable, following search rules in the user's "linker" search paths.
2. Get a count of the entrypoint's parameters, including any function return value. Get PL/I attributes of each parameter.
3. Get information from the user about each parameter: is it an input argument or an output argument (or both)? For an input argument, get an initial value (a character representation of the argument value to be passed to the parameter).
4. Allocate storage for each argument, and storage for an argument list describing all the arguments.
5. Assign an input argument's initial value to its allocated argument storage. Initialize storage for an output argument to a default value.
6. Fill in the argument list with a pointer to each argument's storage, and a pointer to its argument descriptor (containing actual size information for arguments passed to parameters with star extents).
7. Transfer to the PL/I call operator to actually invoke the entry variable.
8. When this call operation returns: display any output arguments (converting each to a character representation appropriate for the parameter data type); or return a selected output argument as an active function return string.

The PL/I compiler conveniently generates entrypoint calling sequence information in the object segment for each external entrypoint defined by the PL/I source program. Multics provides service subroutines to access this calling sequence data, and to assist with the other tasks above. These services are summarized in subsequent sections of this MTB.

In addition, Multics follows standards governing how an external entrypoint is called in the Multics environment; and how an entrypoint receives parameter data. These standards are summarized in the next section of this MTB.

STANDARDS

Standards applicable in calling external subroutine and function entrypoints fall into two broad categories: call/return standards and conventions; and subroutine parameter data type definitions.

Call/Return Standards

The mechanisms for making an inter-segment call to an external entrypoint are described in the Multics Programmer's Manual – Subsystem Writers' Guide (AK92-02D), Section 2. The manual describes how a call is made; but this MTB is concerned with steps prior to calling the entrypoint, and steps that follow a normal return from the entrypoint. Even so, the intervening steps should be understood by the reader.

- Pages 2-9ff describe the sequence for making a call, and returning from the call.
- Pages 2-13ff of this manual describe an argument list (arg_list structure in arg_list.incl.pl1), used to pass argument values for each parameter of the called entrypoint; and the argument descriptor that describes each argument (see arg_descriptor or fixed_arg_descriptor structures in arg_descriptor.incl.pl1).

An argument list has the format:

```
dcl 1 arg_list aligned based,
      2 header,
        3 arg_count fixed bin(17) unsigned unal,
        3 pad1 bit(1) unal,
        3 call_type fixed bin(18) unsigned unal,
        3 desc_count fixed bin(17) unsigned unal,
        3 pad2 bit(19) unal,
      2 arg_ptrs  (arg_list.arg_count) ptr,
      2 desc_ptrs (arg_list.arg_count) ptr;
```

An argument descriptor has the format:

```
dcl 1 arg_descriptor based aligned,
      2 flag bit(1) unal,
      2 type fixed bin(6) unsigned unal,
      2 packed bit(1) unal,
      2 number_dims fixed bin(4) unsigned unal,
      2 size fixed bin(24) unsigned unal;
```

In addition, the following include files contain data pertinent to the call/return mechanism.

- std_descriptor_types.incl.pl1: defines constants for each data type supported by Multics. These values are used in the arg_descriptor.type element.
- encoded_precision.incl.pl1: defines the encoded_precision structure that can be overlaid atop arg_descriptor.size to produce precision/scale information for a PL/I fixed-point data type. (See also the fixed_arg_descriptor structure in arg_descriptor.incl.pl1.)

Data Type Standards

Items passed in an argument list include a pointer to storage for each argument passed to the subroutine, along with a descriptor for the argument. Most subroutines and functions: declare a fixed number of parameters; with known attributes for each parameter (its data type, size, alignment, array dimensions). However, some subroutines are declared only as "options(variable)", meaning they accept: a variable number of parameters; or accept parameters with various attributes.

For both kinds of entrypoint, the argument list must point to the argument storage, and point to a descriptor for the argument. Subroutine parameters may be declared with unspecified size (star extents for strings and areas; star dimensions for arrays); the actual size/dimension of the corresponding argument value is given in its argument descriptor. For subroutines accepting variable number/type of argument, attributes of the actual arguments passed are defined by the descriptors.

The Multics PL/I Language Specification (AG94-02E) outlines the standards for data types supported by PL/I programs.

- Page 5-33ff gives standards for a complete/consistent parameter <descriptor set>, including:
  - a list of data types supported by PL/I.
  - alignment and sign-type attributes applicable to each data type.
  - aggregate forms supported (array dimensions and structures).
- Page 5-13ff gives defaults for attributes missing from a <descriptor set>.

The Multics PL/I Reference Manual (AM83-00A) describes storage size rules for a parameter with a given <descriptor set>. For given data type and alignment, it defines: the storage size (in words, bytes, or bits); and layout of data components within that storage.

The Multics Processor Manual (AL39-01C) gives further details about storage size and layout of data.

Finally, the system.incl.pl1 file defines constants that facilitate implementing storage size rules in the **call** program. std_descriptor_types.incl.pl1 provides named constants for each descriptor type. data_type_info_.cds provides arithmetic attributes of each data type; these are accessed via the structure defined in data_type_info_.incl.pl1.

EXISTING TOOLS

Multics provides a variety of tools to assist with the call/return tasks listed above, including:

- Calling sequence tools
- Calling sequence declaration parsing tools
- Data storage tools
- Data type conversion tools
- Call/return operator invocation tool

Existing commands are described in the MPM Commands manual (AG92-06B); existing subroutines are described in the MPM Subroutines manual (AG93-05A).

Calling Sequence Tools

The **display_entry_point_dcl** (**depd**) command displays a PL/I declaration for an external entrypoint of an object segment. Information to build this declaration comes from calling sequence information generated by the PL/I compiler just prior to the entrypoint location (in the text section of its object segment). Calling sequence information is stored as general attributes of the entrypoint, plus an optional list of parameter descriptors. **depd** converts this calling sequence information into a PL/I declare statement for the entrypoint. A user of the **call** command can first use **depd** to determine parameter count and parameter attributes for an entrypoint.

Some Multics subroutines do not provide calling sequence information prior to their entrypoint location. ALM assembler source files are the best example. This includes source files for Multics gates, that provide the interface to Multics service routines in inner rings. The **depd** command works around this limitation by searching for a PL/I declaration of the entrypoint in files named in the "declare" search paths. The default "declare" search path, `>sss>pl1.dcl`, names a file provided with Multics; it declares many of the ALM-defined service routines. In addition, each user may add his own `.dcl` file to the "declare" search paths, to add other declarations or even redefine existing declarations.

The **call** command accesses the same entrypoint information displayed by **depd,** using the get_entry_point_dcl_$emacs subroutine.

Declaration Parsing Tools

The get_entry_point_dcl_$emacs subroutine tells **call** whether the entrypoint declaration came from PL/I-defined calling sequence information, or from a declare statement in a `.dcl` file. If calling sequence information is available, it is directly converted into a list of parameter descriptors using the get_entry_arg_descs_$info subroutine. But when a PL/I declaration string is returned, the **call** command must convert this entrypoint declare statement into equivalent calling sequence information (subroutine/function flag; and list of parameter descriptors).

call will use a new subroutine for this conversion task. The new routine will split the PL/I declare statement returned by get_entry_point_dcl_$emacs into tokens, using the lex_string_$lex subroutine. A set of reduction statements (see the **reductions** command) will then parse the tokens, extracting attribute information to build a list of parameter descriptors. More on this in the New Tools section, below.

As the reductions parse the declare statement tokens, they record attributes for each parameter in a `type` structure (see the pl1_symbol_type.incl.pl1 file). The pl1_descriptor_type_fcn.incl.pl1 file defines a pl1_descriptor_type function that maps a completed, consistent set of data attributes in the type structure onto the corresponding arg_descriptor.type value.

Data Storage Tools

The lex_string_$lex subroutine uses translator_temp_ to quick-allocate space for tokens. This temporary space is stored in process directory segments managed by translator_temp_. Such allocate-once, never-free storage is also ideal for the argument storage tasks that **call** must perform.

Data Type Conversion and Data Copying Tools

The PL/I compiler uses the any_to_any_ operator to convert one data type to another, following PL/I conversion rules. The assign_ subroutine is the external interface to any_to_any_. **call** can use this subroutine to convert a character input value to the corresponding argument's data type, if PL/I defines a conversion from character to that data type. assign_ also copies the converted input value onto the argument storage location; or initializes storage for an output parameter to a default value appropriate for its data type.

For an argument data type having no PL/I-defined conversion rule from character data, the **call** command must use some alternative conversion mechanism. Noteworthy data types for which this is the case include: pointer, entry variable, and area parameters. Fortunately, Multics already provides subroutines to perform these conversions.

- cv_ptr_ converts a variety of character representations for pointer values to an aligned pointer.
- cv_entry_ converts a variety of character representations of an entry value into an entry variable. It uses "linker" search rules to map reference names onto a segment path/number.
- define_area_ initializes storage for a PL/I area argument.

Besides cv_ptr_, **call** needs a mechanism to make a pointer parameter point to storage for another data type (e.g., an argument which is a pointer to a character buffer). See the -addr option proposed for the **call** command, in the User Interface section of this MTB.

Call/Return Operator Invocation Tool

Multics provides the cu_$generate_call tool to call an entry variable, passing it a given argument list.

NEW TOOLS

Several new tools are needed to perform call-related tasks. These are summarized here, and described in more detail later in this MTB.

- call_entry_info_.rd: a subroutine to convert a character representation of an entrypoint into: an entry variable; and an array of descriptors for the parameters of that entry variable. call_entry_info_.incl.pl1 describes the structure returned by:
  - call_entry_info_$from_virtual_entry: obtains descriptors from calling sequence info. This is mostly an interface to the Multics get_entry_arg_descs_$info subroutine, which does the actual work of examining the calling sequence information for the target entrypoint, and obtaining pointers to its parameter descriptors.
  - call_entry_info_$from_declaration: creates descriptors from a PL/I declare statement for the target entrypoint (such as that returned by get_entry_point_dcl_). It uses a set of reduction statements (see the command: reductions) to *compile* a PL/I declare statement string into calling sequence info for the target entrypoint (subroutine or function; number of parameters; descriptor for each parameter).
- pl1_symbol_type_fcns.incl.pl1: an include file that defines two new functions:
  - type_for_descriptor_set: accepts a list of PL/I attributes (named in the `type` structure; see pl1_symbol_type.incl.pl1, as mentioned above), and outputs a complete set of attributes after applying PL/I default rules. The function returns a flag indicating whether the input set of attributes were mutually consistent. This function is used by call_entry_info_$from_declaration to convert parsed parameter declaration tokens into a complete <parameter attribute> set that can be handed to the pl1_descriptor_type_ function.
  - type_as_string: accepts a list of PL/I attributes (named in the type structure), plus additional information (structure level, variable name, array dimension, precision/scale or size, and picture attribute information). It returns a character string declaration based upon these attributes.
- call_dtype_fcns.incl.pl1: an include file that defines several new functions to group arg_descriptor.type values according to different data characteristics. Such groupings simplify coding within the **call** command. (The XXX_dtype values referenced below come from attributes of arithmetic descriptor types, provided by data_type_info_.cds).
  - bit_string_dtype: returns true if the given descriptor has the bit_string attribute in data_type_info_.cds.
  - char_string_dtype: returns true if the given descriptor has the char_string attribute in data_type_info_.cds.
  - star_extent_dtype: returns true if the given descriptor type may have an undetermined size (a size specified as an asterisk or star). This included descriptor types having the char_string or bit_string attribute in data_type_info_.cds; and the area_dtype descriptor.
  - string_dtype: returns true if the given descriptor has the char_string or bit_string attribute in data_type_info_.cds.
  - varying_string_dtype: returns true if the given descriptor has the varying attribute in data_type_info_.cds.

  - fixed_bin_dtype: returns true if the given descriptor type has the fixed & ^decimal attributes in data_type_info_.cds.
  - fixed_point_dtype: returns true if the given descriptor type is one of the many types in which a fixed-point number may be stored (a number with both a precision and scale factor); it has the fixed attribute in data_type_info_.cds.
  - numeric_dtype: returns true if the given descriptor type has the arithmetic attribute in data_type_info_.cds.
  - unsigned_dtype: returns true if given a descriptor type has the arithmetic and ^signed attributes in data_type_info_.cds.
  - supported_by_pl1_dtype: returns true if given descriptor type is one of those supported by the PL/I compiler.
  - supported_by_call_dtype: returns true if given descriptor type is one of those supported by the **call** command.
  - pl1_dtype_name: maps a given descriptor type onto its name, as specified in std_descriptor_types.incl.pl1.
  - storage_for_pl1_dtype: given a descriptor_type, a packed/unpacked switch, and a numeric precision or string/area size, this returns the amount of storage needed for an item with these attributes, in words, bytes, or bits; and a storage alignment indicator (even-word, word, byte, or bit alignment).
- call_et_.alm: a Multics status code table is code values more descriptive of the circumstances which the **call** command must diagnose.
- call_scalar_dcl_.rd: defines a call_scalar_dcl_ subroutine to convert a PL/I declare statement for a single, scalar parameter into a parameter descriptor. The proposed call command uses this function to *compile* the DECLARATION string of a -dcl DECLARATION or -addr DECLARATION option into a parameter descriptor.
- call_status_code_name_.pl1: defines a call_status_code_name_ subroutine that obtains the name from a Multics status code value. This routine originated as the get_status_name internal procedure of the probe_display_data_.pl1 source module.

USER INTERFACE

A command to call a user-specified entrypoint must gather four classes of information in order to invoke the entrypoint, and display or return its results to the user.

A. Information about the entrypoint to be called, including:
   - Entry variable locating code for the entrypoint within its containing object segment.
   - Entrypoint type: subroutine or function.
   - Count of parameters expected or supported.
   - Data type, alignment, sign type, aggregate type for each parameter.

   The call_entry_info_$from_virtual_entry subroutine can provide calling sequence information for an entrypoint expecting a specific set of parameters.

   However, for options(variable) entrypoints that accept variable parameter counts and/or variable parameter data types, no information is available from PL/I-defined calling sequence information. **depd** declares such entrypoints as:

```
dcl ioa_ entry() options(variable);
```

   The user can override this declaration by providing a definition for ioa_ in his own "declare" search path file. But that approach only works for the specific parameter configuration placed in that declaration file. For most options(variable) entrypoints, the call user interface must provide information about each argument to be passed to the entrypoint.

B. Direction information designating arguments passed to an entrypoint as input to the call (arguments for which the caller must supply an initial argument value); or output from the call. Such information must come from options supplied in the **call** command line.

C. Meta-information for an argument, not stemming from entrypoint calling sequence information. Examples are:
   - Giving an actual length of an argument passed to a parameter with star extents.
   - Designating a fixed bin(35) argument as a Multics status code.
   - Overriding entrypoint calling sequence information by giving a user-specified data type, size, and alignment for an argument.
   - Describing a pointer parameter as a reference to data storage of a particular data type, size, alignment, and aggregate structure. This referenced data should be: initialized for an input argument; and displayed for an output argument.

D. Information about how to display an output argument.
   - Name to be displayed with an output argument.
   - Preferred display length for an output string.
   - Choice of output argument to return when call is invoked as an active function.
   - Request octal dump of argument storage holding an argument.

It is difficult to provide a concise representation for all this information in a single command line. To reduce command interface complexity, call must impose some limitations on the types of parameters it supports.

- Call supports only a subset of the data types supported by PL/I.
  - Excluded data types are: complex numbers; picture strings; offset; file; format; label.
  - Excluded data types are those supported by other languages (i.e., Algol, Cobol, Fortran, Pascal data types not equivalent to a supported PL/I data type).
- Call supports only scalar parameters.
  - Aggregate parameters are not supported: arrays; structures. Giving initial values for them, and displaying aggregate output would be difficult to describe in a command-line interface.

COMMAND: call

The following interface is proposed for a **call** command.

Name: call, cl

SYNTAX AS A COMMAND

call {-global_options} virtual_entry {arg_value_specifier …}

SYNTAX AS AN ACTIVE FUNCTION

[call {-global_options} virtual_entry {arg_value_specifier …}]

FUNCTION

invokes external subroutine and function entrypoints directly from a command line, to facilitate testing entrypoint inputs, operation, and outputs.

Operating either as a command or an active function, call invokes an entrypoint, passing arguments constructed from a set of arg_value_specifiers. Upon return from the entrypoint, call displays selected arguments, or returns one argument as its active function value.

ARGUMENTS

virtual_entry

> character representation identifying an external entrypoint to be invoked. Format of this string is described in: virtual_entries.gi.info.

arg_value_specifier

> one or more strings and options defining an argument to be passed to the entrypoint. *See: List of arg_value_specifiers*.

LIST OF arg_value_specifiers

Information for each argument being passed as an entrypoint parameter is given in the call command, in parameter list order. Each argument definition begins with one of the following directional specifiers. The definition continues with an arg_value or arg_options. It ends when the directional specifier for the next parameter position is given. The arg_options are described separately. *See: List of arg_options*.

-input arg_value {arg_options},
-in arg_value {arg_options},
-i arg_value {arg_options},
arg_value {arg_options}

arg_value is a character representation of an initial value assigned to the argument prior to invoking the entrypoint. The argument is treated as an input which not displayed upon return from the entrypoint. This is the default when no direction specifier is given.

-inout arg_value {arg_options},
-io arg_value {arg_options}

arg_value is a character representation of an initial value assigned to the argument prior to invoking the entrypoint. The argument is treated as an input/output parameter; its value is displayed upon return from the entrypoint.

-output {arg_options},
-out {arg_options},
-o {arg_options}

an output argument for which no initial value is provided. call assigns a default value to argument storage before invoking the entrypoint. The argument is displayed upon return from the entrypoint.

-outignore {arg_options},
-ignore {arg_options},
-ig {arg_options}

an output argument for which no initial value is provided. call assigns a default value to argument storage before invoking the entrypoint. The argument is not displayed upon return from the entrypoint.

LIST OF arg_options

The following options apply to the argument definition started by the immediately preceding directional specifier.

-id ID

gives a name for the argument. ID must follow the rules of a PL/I identifier.

-return, -ret

selects the argument that should be returned when call is used as an active function. Only one argument may be selected as the return value. *See: Notes on call as an Active Function*.

-code, -cd

characterizes the argument as a Multics status code (fixed bin(35) aligned). For an input argument, the arg_value is a status code name (e.g., error_table_$bad_arg). For an output argument, the status code name and associated long message are displayed.

-date_time,
-date, -dt,
-time, -tm

characterizes the argument as a Multics clock value (fixed bin(71) aligned). For an input argument, the arg_value is a string representation of the date/time value, in a format acceptable to the convert_date_to_binary_ subroutine. For an output argument, the clock value is converted to the process default date_time, date, or time format. (See the print_time_defaults (ptd) and set_time_defaults (std) commands.)

-declare DECLARATION,
-dcl DECLARATION

argument attributes are given by the PL/I DECLARATION. These attributes override other attributes obtained for this argument. A descriptor with the declared attributes is passed for this argument when the entrypoint is invoked. -dcl is often used to override the default (char(*) unal) attributes used for parameters of an options(variable) entrypoint. Example: -dcl “fixed bin(35)”

-addr DECLARATION

for a pointer argument, sets the pointer to address storage described by the PL/I DECLARATION, which is a string giving data type and length. Example: -addr “char(20)”. For an input pointer argument, the arg_value gives an initial value assigned to the addressed storage. For an output argument, the addressed storage is displayed or returned.

A DECLARATION may include the ID of another input parameter. For example, -addr “char(buffL)” defines a character string whose length is the arg_value of the input argument having -id buffL. For a more complete example, *see Notes on Example – iox_$get_line*.

-max_length M, -ml M

for a string or area parameter with star extents (e.g., char(*)), M specifies how many characters, bits, or area words to allocate when argument is prepared for the argument list.

-length L, -ln L
-length L_ID, -ln L_ID

for a string argument, L gives the number of characters or bits to display when the entrypoint returns. L_ID specifies this display length as the value of another output argument with the arg_option: -id L_ID. For a complete example, *see Notes on Example – iox_$get_line*.

-octal, -oc

displays an octal representation of storage of an output argument, as well as its character interpretation. Note that the final octal block may include bytes (or bits) that are beyond the end of the argument's storage.

LIST OF global_options

Global options are given either before, or immediately after, the virtual_entry.

-all, -a

displays all arguments upon return from the entrypoint. The default is to display only arguments with an -inout or -out directional specifier.

-octal, -oc

displays an octal representation of storage for all arguments, as well as a character interpretation. When -octal is used as a global option, it implies -all.

-debug LEVEL, -db LEVEL

displays debug information as the argument list is prepared. LEVEL is an integer between 1 and 5. Level 1 displays basic information; higher levels include addition details. This option is most often used when debugging the call command itself; but it can provide information to further explain error messages from call, or to ensure the argument list is being constructed as expected.

NOTES ON call AS AN ACTIVE FUNCTION

When call is used as an active function, its return value is selected as follows:

- If any argument has the -code option and a non-zero status code was returned by the entrypoint, the active function returns the status code name and its associated long message.
- Otherwise, if any argument has the -return option, the active function returns the character interpretation of that argument.
- Otherwise, if the entrypoint is a function, the active function returns the character interpretation of that function return argument.
- Otherwise, the active function returns an empty string.

NOTES ON ENTRYPOINT PARAMETERS

When the PL/I compiler compiles a source file into an object segment, each external entrypoint is preceded by entrypoint calling sequence information that includes:

- entrypoint attributes:
  - subroutine or function.
  - fixed parameter list, or entrypoint declared options(variable).
- parameter attributes (if the entrypoint has a fixed parameter list):
  - attributes for each parameter (including any function return parameter) are specified by a parameter descriptor.
- options(variable) entrypoint:
  - has no parameter descriptors, because it accepts parameters of a variable count and/or data type.
  - must therefore be a subroutine (since there is no parameter descriptor for a return value).

The entrypoint parameter attributes include data type, storage size and alignment, and other attributes. The information is sufficient to generate a complete PL/I declare statement for the entrypoint.

External entrypoints coded in assembler (ALM compiler) do not include such entrypoint calling sequence information. This includes gate interfaces to inner ring entrypoints. Such assembler entrypoints are supported by call only if one of the files in the "declare" search paths provides a PL/I declare statement for the alm entrypoint.

Use the display_entry_point_dcl (depd) command to display entrypoint information (obtained either from its calling sequence information, or from a declaration found via the "declare" search paths). The call command uses this same information to build an argument list for the entrypoint. See "Notes on user-provided data files" in depd.info for information about declaring new entrypoints, or customizing existing entrypoints.

NOTES ON BUILDING THE ARGUMENT LIST

To invoke an external entrypoint, call creates an argument corresponding to each entrypoint parameter.

- Allocate storage to hold the argument.
- Convert the arg_value for an input argument to the parameter's data type. PL/I type conversion rules are followed; arg_value for a pointer or entry variable is in a form accepted by cv_ptr_ or cv_entry_ subroutines, respectively.
- Copy the converted value to the allocated storage location.
- For a parameter with star extents, modify the parameter descriptor to specify an actual extent size for the argument. This may have been determined by a -max_length option, or by the length of the arg_value initializer.
- Store pointer to the allocated storage, and the (perhaps modified) parameter descriptor in an argument list.
- Call the entrypoint, passing that argument list.

Upon return from the entrypoint, call does the following for each -inout or -output argument:

- Convert the argument value from its parameter data type to a character representation.
- Display the argument value.
- For a function, convert and display the function value unless -ignore was specified for the return argument.

*NOTES ON EXAMPLE: initiate_file_*

The initiate_file_ subroutine has the following PL/I calling sequence:

```
call initiate_file_( dir, entry, access_mode, ptr,  bit_count, code);
```

The attributes for each parameter may be displayed by using the depd command:

```
depd initiate_file_
  dcl initiate_file_ entry( char(*), char(*), bit(*), ptr, fixed bin(24), fixed bin(35));
```

The following command initiates a file for reading; the file in the current working directory is named 'my_file'. Upon return from initiate_file_, call displays the output arguments: a pointer to the initiated file, its bit count, and a Multics status code:

```
call initiate_file_  [wd]  my_file  100  -out  -out  -out -code
```

*NOTES ON EXAMPLE: cv_ptr_*

Calling sequence for a function includes a descriptor for its returns attribute. This defines data the function returns to the caller. For example:

```
ptr_value = cv_ptr_ (virtual_pointer, code);

depd cv_ptr_
  dcl cv_ptr_ entry (char(*), fixed bin(35)) returns(ptr);

call cv_ptr_  [hd]>[user name].profile  -out -code
```

The default action is equivalent to

```
call cv_ptr_  [hd]>[user name].profile  -out -code  -out
```

which displays both the status code and function return value (ptr_value). To display only the status code for the given virtual_pointer (not displaying the function return value), use:

```
call cv_ptr_  [hd]>[user name].profile  -out -code  -ignore
```

NOTES ON EXAMPLE: iox_$get_line

When calling a subroutine with an output buffer pointer, an input max length for this buffer, and an output count of valid characters in that buffer, use the -addr and -length options to display only the valid buffer characters. iox_$get_line is described as follows:

```
call iox_$get_line (iocb_ptr, buff_ptr, buff_max_len, n_read,  code);

depd iox_$get_line
  dcl iox_$get_line entry (ptr, ptr, fixed bin(21), fixed bin(21), fixed bin(35));
```

Use call to invoke iox_$get_line, as follows. [Input shown on several lines should actually be given in a single command line.]

```
call    iox_$get_line    -i [io find_iocb user_i/o]
                         -o -id buff -addr "char(buffL)" -length readN
                         -i 200 -id buffL
                         -o -id readN
                         -o -code
```

If the user types the following user input...

```
Results from iox_$get_line are this line.
```

call displays...

```
-- Return from: iox_$get_line -------
 buff          337|2056 -> Results from iox_$get_line are this line.
 readN          42
 code05        OK
```

Argument 1 is an IOCB pointer to a named I/O switch. call converts this initial value to a pointer argument.

Argument 2, named buff, is an output buffer pointer. Call creates storage for "char(buffL)" characters, where buffL is the input arg_value for the third argument. Upon return, -length readN tells call how many characters in this buffer to display, where readN references the output value of the fourth argument.

Argument 3, named buffL, is an input argument giving max length (200 characters) for the buffer in the second argument.

Argument 4, named readN, is an output argument giving actual length to display from the buffer in the second argument.

Argument 5 is a status code.

*NOTES ON EXAMPLE: ioa_*

When calling a subroutine that accepts a variable number and type of arguments, arguments have default attributes of: char(*) unaligned. Use the -dcl option to specify the attributes of a different type.

ioa_ is described as:

```
call ioa_ (control_string, arg1, ..., argN);

depd ioa_
 dcl ioa_ entry() options(variable);
```

Use call to pass a character control_string that displays a pointer and fixed bin(35) number. [Input shown on several lines should actually be given in a single command line.]

```
call    ioa_    "data at: ^p (^d bits)"
                247|400 -dcl ptr
                39786   -dcl "fixed bin(35)"

data at:  247|400 (39786 bits)
```

NOTES ON SUPPORTED DATA ATTRIBUTES:

Every parameter of a PL/I entrypoint has a data type, an alignment type, and an aggregate type. The call command follows PL/I rules when converting an arg_value string to the corresponding parameter attributes.

- Alignment of storage supports both aligned and unaligned data.
- Aggregate (array and structure) data is not supported.

The following scalar data types are supported.

real fixed bin(P,S)
fixed-point binary data type, with or without a precision (P) and scale factor (S).

real float bin(P)
floating-point binary data, with or without a precision (P).

real fixed dec(P,S)
fixed-point decimal data type, with or without a precision (P) and scale factor (S).

real float dec(P)
floating-point decimal data type, with or without a precision (P).

char(N), char(N) varying, char(*), char(*) varying
character data, with a given length (N) or unspecified length (*).

bit(N), bit(N) varying, bit(*), bit(*) varying
bit string data, with a given length (N) or unspecified length (*).

pointer, ptr
pointer data.

entry
entry variable data.

area
PL/I allocation area of given size.

LIST OF UNSUPPORTED DATA TYPES:

complex fixed bin(P,S)
complex float bin(P)
complex fixed dec(P,S)
complex float dec(P)
picture
offset
file
format
label

NOTES ON arg_value FORMATS:

Each input arg_value is a character representation of an initial value for a subroutine argument. In general, numbers and bit strings are initialized by character data, in any format accepted by PL/I conversion rules for character-to-numeric or character-to-bit.

real numbers (binary or decimal, fixed or float)

```
 23
-23.45
 2e3          (equivalent to 2000)
 0.2345e5
 2.45e-6
```

Fixed binary arguments may also be entered as octal or hexadecimal bit strings. Bits are assigned to argument storage from right-to-left, following PL/I rules for converting a bit string to a fixed bin(17) parameter:

```
 55b3        (uses octal suffix b3:        equivalent to  45)
 2aDb4       (uses hexadecimal suffix b4:  equivalent to 685)
 777777b3    (uses octal suffix b3:        equivalent to  -1)
```

bit string data

Entered as a sequence of 1 and 0 characters (e.g., 10101101) which are converted to a bit string following PL/I character-to-bit conversion rules, and assigned left-to-right to the parameter. Values may also be entered in octal or hexadecimal format, as shown above for real numbers.

pointer, ptr

Use any format given in: virtual_pointers.gi.info.

entry variable

Use any format given in: virtual_entries.gi.info.

area

Since there is no way to initialize an area, or display its output value, an area is usually specified using the -ignore arg_value specifier. If an area parameter is declared:

```
dcl  area_parm area(*);
```

you can specify an actual size for the corresponding argument using the -max_length M option:

```
-ignore -max_length 2000
```

Often, the parameter is declared as a pointer to an area:

```
dcl  area_ptr ptr;
```

In such cases, you can specify a pointer to an area of given word size:

```
-ignore -addr "area(2000)"
```

SUBROUTINE: call_entry_info_

The following interface is proposed for a call_entry_info_ subroutine.

Name: call_entry_info_

This subroutine returns information about the calling sequence of a procedure entry point.

Entry: call_entry_info_$from_virtual_entry

The from_virtual_entry entry point, given a virtual entry string, returns information about the entry point, including: an entry variable fabricated from the virtual entry string; flags indicating whether the entry point is a function, or accepts a variable number of parameters; and for an entry point with fixed number/type of parameters, an array of pointers to each parameter descriptor.

The returned information is allocated in a translator_temp_ area. A pointer to this area is included with the return information. When the caller no longer needs the entry information, use the call_entry_info_$cleanup entry to release these temporary segments.

USAGE

```
declare call_entry_info_$from_virtual_entry entry (char(*), char(*), fixed bin(3) unsigned, ptr,
        fixed bin(35));

call call_entry_info_$from_virtual_entry (caller, virtual_entry, debug_sw, entry_info_ptr, code);
```

ARGUMENTS

caller

> Name of the calling command or subsystem. This is used in naming temporary segments created by call_entry_info_. (Input)

virtual_entry

> String representation of an entry point. Any representation accepted by the cv_entry_ subroutine may be given. (Input)

debug_sw

> Values greater than zero display debug information, as cv_entry_ operates. For the from_virtual_entry subroutine:
>
> 3 – displays attributes of each parameter;
>
> 4 – displays details of calling sequence information found in the PL/I object referenced by the entry variable. (Input)

entry_info_ptr

> points to the entry_info structure described under “Notes” below. (Output)

code

is a standard status code. Most errors are diagnosed by call_entry_info_ invoking com_err_ to report a specific error. The following code can be returned to the caller without diagnosing an error:

error_table_$noalloc

the entry_info structure and parameter descriptors could not be allocated. (Output)

Entry: call_entry_info_$from_declaration

The from_virtual_entry entry point, given a PL/I declare statement for an entry point, returns information about the entry point, including: flags indicating whether the entry point is a function, or accepts a variable number of parameters; and for an entry point with fixed number/type of parameters, an array of pointers to each parameter descriptor.

The returned information is allocated in a translator_temp_ area. A pointer to this area is included with the return information. When the caller no longer needs the entry information, use the call_entry_info_$cleanup entry to release these temporary segments.

USAGE

declare call_entry_info_$from_declaration entry (char(*), char(*), char(*) var, fixed bin(3) unsigned, ptr, fixed bin(35));

call call_entry_info_$from_declaration (caller, virtual_entry, declare_statement, debug_sw, entry_info_ptr, code);

ARGUMENTS

caller

Name of the calling command or subsystem. This is used in naming temporary segments created by call_entry_info_. (Input)

virtual_entry

String representation of an entry point. Any representation accepted by the cv_entry_ subroutine may be given. If an invalid or empty string is given, entry_info.entryVar is set to the equivalent of a null entry variable (codeptr(entryVar) = null()). (Input)

declare_statement

String giving a PL/I declare statement for the entry point. (Input)

debug_sw

Values greater than zero display debug information, as call_entry_info_ operates. For the from_declaration subroutine:
1 – displays errors found while parsing declare_statement;
3 – displays attributes parsed for each parameter, and parameter attributes after PL/I defaults are applied;
4 – displays tokens scanned in the declare_statement; and reductions matching input when parsing the tokens. (Input)

entry_info_ptr

points to the entry_info structure described under “Notes” below. (Output)

code

is a standard status code. Most errors are diagnosed by call_entry_info_ invoking com_err_ to report a specific error. It can be:

error_table_$noalloc

the entry_info structure and parameter descriptors could not be allocated. (Output)

NOTES

The entry_info_ptr parameter points to the structure shown below, upon return from call_entry_info_. This structure, and all of the call_entry_info_ entry points, are declared in call_entry_info_.incl.pl1.

```
dcl  1 entry_info aligned based(entry_info_ptr),
       2 header,
         3 version char(4),
         3 areaP ptr,
         3 entrypoint,
           4 nameString char(256) var,
           4 entryVar entry variable options(variable),
         3 callingSequence,
           4 function bit(1) unaligned,
           4 options_variable bit(1) unaligned,
           4 pad1 bit(34) unaligned,
           4 parm_count fixed bin,
       2 cleanup_data_ptr ptr,
       2 descriptor_ptrs (entry_info_parm_count refer(entry_info.parm_count)) ptr;

dcl  entry_info_v1 char(4) aligned int static options(constant) init (“ei01”),
     entry_info_ptr ptr,
     entry_info_parm_count,
     eiParmCountNotDetermined fixed bin int static options(constant) init(-1);
```

STRUCTURE ELEMENTS

version

is the version number for this structure. Since the structure is output data, the caller may check whether the returned structure version matches the version declared in the calling program. This is set to entry_info_v1.

areaP

is an allocation area provided by the translator_temp_ subroutine. This is a never-free-allocations area. The caller may allocate additional storage in this extensible area, using either the translator_temp_$allocate subroutine, or the allocate subroutine defined in translator_temp_alloc.incl.pl1.

nameString

is the entry point name, as extracted from virtual_entry with any pathname components removed. It is suitable for use in error messages referring to the entry point or its calling sequence.

entryVar

is the entry variable returned by cv_entry_ describing the virtual_entry string. In most cases, it is ready to invoke (by passing it to cu_$generate_call).

function

is 1 if the entry point returns a function value; and 0 if it is a subroutine. For a function, descriptor_ptrs 1 through parm_count-1 describe attributes of the function parameters; descriptor_ptrs(parm_count) describes attributes of the function return value.

options_variable

is 1 if the entry point accepts a variable number and/or type of arguments. parm_count is 0 for such entry points; and function is 0.

parm_count

for a subroutine, is the actual number of parameters expected by a non-options(variable) entry point. For a function, it is one greater than the number of parameters expected; the descriptor_ptrs(parm_count) points to a descriptor for the function return value.

cleanup_data_ptr

is a pointer used by call_entry_info_$cleanup. The caller need not examine or change this element in any way.

descriptor_ptrs

is a parm_count array of pointers to descriptors for the entry point parameters; and any function return value.

Entry: call_entry_info_$cleanup

This entry point releases temporary segments used by the other call_entry_info_ entry points. It's final step is to null the entry_info_ptr, to prevent any subsequent call (perhaps from a cleanup on-unit) from attempting a second cleanup operation.

USAGE

declare call_entry_info_$cleanup entry (ptr);

call call_entry_info_$cleanup (entry_info_ptr);

ARGUMENTS

entry_info_ptr

points to the entry_info structure described under "Notes" above. If called with a null value, no cleanup operation is attempted. (Input)

NOTES

The correct usage for call_entry_info_ shown below provides full-time cleanup condition handling for the entry_info structure.

```
entry_info_ptr = null();

on cleanup call call_entry_info_$cleanup (entry_info_ptr);

call call_entry_info_$from_...(..., entry_info_ptr, code);

        ... [caller's use of the entry_info data]

call call_entry_info_$cleanup (entry_info_ptr);
```

SUBROUTINE: call_scalar_dcl_

The following interface is proposed for a call_scalar_dcl_ subroutine.

Name: call_scalar_dcl_

Converts a PL/I declare statement for a parameter variable into an argument descriptor.

USAGE

```
declare call_scalar_dcl_ entry (ptr, char(*), fixed bin(3) unsigned, char(256) var, bit(36) aligned,
   char(*) var, fixed bin(35));

call call_scalar_dcl (areaP, declare_statement, debug_sw, variable_name, descriptor_bits, code);
```

ARGUMENTS

areaP

is an allocation area provided by the translator_temp_ subroutine. (Input)

declare_statement

String holding a PL/I declare statement for the parameter. (Input)

debug_sw

Values greater than zero display debug information, as call_scalar_dcl_ operates.

1 – shows errors in parsing the declare_statement.

3 – shows parameter attributes found by the parse, and after applying PL/I defaults.

4 – shows declare_statement scanned into tokens; and reduction statements that match those tokens. (Input)

variable_name

is any parameter name found in the declare_statement. (Output)

descriptor_bits

is the parameter descriptor created from the declare_statement. (Output)

code

is a standard status code. It can be:

call_et_$bad_declaration

the declare_statement is not one defined by the PL/I language for a parameter.

call_et_$too_many_descriptors

the declare statement describes more than one scalar parameter.

call_et_$too_many_bounds

the declare statement describes more than 128 array bounds in its dimension clause. (Output)

TESTING THE call COMMAND

To test the various components described in the *New Tools* section, several testing tools were written. These fall into three categories.

- Tools that directly invoke supporting subroutines, accepting user-provided inputs and displaying outputs from the target subroutine. These include: call_ei_test.pl1, call_sd_test.pl1.
- A tool that provides a variety of subroutine and function interfaces which perform a known mapping from input data to outputs. This is: call_test_.pl1.
- A tool for invoking the call command with a variety of arguments that test specific parts of the call interface. This is: call_test.ec.

These testing tools are described in the next subsections. The tools will be captured in bound_trace_stack_.s.archive for possible use in the future; but will not be compiled or included in the object archive, or in the bound_trace_stack_ object segment.

Testing Supporting Subroutines

The call_entry_info_.rd module defines two main entrypoints.

- call_entry_info_$from_virtual_entry performs a clear-cut task, using the get_entry_arg_descs_$info routine to do most of the work. The from_virtual_entry interface does not require special test tools, because of its straightforward task and dependence only on existing Multics subroutine.
- call_entry_info_$from_declaration emulates the part of the PL/I compiler that parses a string (a PL/I declare statement for an entrypoint) into calling sequence information. It parses using a new set of reductions that need thorough testing to ensure they adequately emulate the PL/I compiler.

A new testing command, call_ei_test.pl1, was written to invoke call_entry_info_$from_declaration with a variety of PL/I declare statements. call_entry_info_.rd includes some internal debugging code (not used during its normal operation) that:

- Constructs a small PL/I program containing components of the PL/I declaration string, organized as a small PL/I source module. For example:

  ```
  dcl program$ep entry (char(*), fixed bin(35)) returns(ptr);
  ```

  into:

  ```
  program: proc;
  ep:      entry( parm1, parm2) returns(ptr);

     dcl  parm1 char(*);
     dcl  parm2 fixed bin(35);

  end program;
  ```

- Invokes the PL/I compiler to generate an object from this small program.

- Compares the calling sequence information in this PL/I-generated object with that created by the call_entry_info_$from_declaration entry, reporting any differences.

call_ei_test.pl1 was used to invoke call_entry_info_$from_declaration in debug mode, using more than 500 PL/I declare statements in >sss>pl1.dcl. This testing identified several bugs in the reductions used in call_entry_info_, allowing them to be fixed and retested successfully.

The call_scalar_dcl_.rd subroutine uses a similar set of reductions to compile a PL/I declare string for a single, scalar parameter into a corresponding parameter descriptor. The reductions are a subset of those used in call_entry_info_.rd.

Testing is performed by a similar tool, call_sd_test.pl1, that invokes the call_scalar_dcl_.rd subroutine in debug mode. This tool:

- accepts a DECLARATION string (the attributes from a PL/I declare statement) as it might be given in a call -dcl or -addr argument;
- converts it into a full PL/I declare statement string;
- passes this string to call_scalar_dcl_; and
- decodes the resulting descriptor returned by call_scalar_dcl_, and displays the descriptor interpretation.

Interface Fodder to Exercise call

The call command supports a full spectrum of PL/I scalar parameter types. Testing all of these types encounters several issues.

- Some parameter types are seldom used in standard Multics subroutines. For example: decimal arithmetic variables were not permitted in Multics standard interfaces.
- Many Multics subroutines cause side-effects (changes to files or other resources), in addition to converting input parameters to output values. Depend upon the nature of such effects, calling such routines repeatedly can be cumbersome or even destructive.

To avoid such pitfalls, it was useful to create a call_test_.pl1 subroutine, which:

- defines entrypoints with a wide variety of parameter types (including those seldom found in Multics system subroutines);
- displays input parameters (to ensure call has properly initialized each parameter);
- sets output parameters (and/or function return value) to known values; and
- has no side-effects.

The call_test.ec script captures **call** commands to invoke the call_test_ entrypoints, with a variety of argument values. Some of these test call's ability to accept a broad range of inputs of a given type, and to properly diagnose input values that exceed numeric ranges, or otherwise specify invalid data. Other

commands within the script exercise **call** debugging functions, and its ability to support a broad spectrum of parameter data types.

call_test.ec is organized as a sequence of calls to a particular call_test_.pl1 entrypoint. Calls to a particular entrypoint can be selected individually; or calls to the full sequence of entrypoints can be run as a regression test.