HONEYWELL

LEVEL 68 MULTICS SYSTEM PROGRAMMING TOOLS

SOFTWARE

MULTICS SYSTEM PROGRAMMING TOOLS

SUBJECT

Description of Multics System Programming Tools of Interest to Experienced Multics Users

SPECIAL INSTRUCTIONS

This revision supersedes Revision 1 of the manual dated February 1980.

Changes made with this revision are marked by change bars in the margin; deletions are marked by an asterisk in the margin. Commands and subroutines that are entirely new do not contain change bars (see Preface for a list of the new commands and subroutines).

SOFTWARE SUPPORTED

Multics Software Release 9.0

ORDER NUMBER

AZ03-02

January 1982

Honeywell

PREFACE

This manual contains information not of general interest to most users of the Multics system; most users will find all the information they need in the Multics Programmers' Manual (MPM). The information documented here is used infrequently and is intended for experienced programmers who are already familiar with the Multics system. The commands, subroutines, and data bases described in this manual receive a level of support that corresponds to their usage. Listed below there are commands and subroutines that are new to the system and there are those that are new to this manual having been transferred from the Tools PLM (Order No. AN51).

Changes to Multics System Programming Tools, Revision 2, Addendum A

New Commands

add_pnotice
copy_dump
display_label
display_pnotice
display_pvte
list_pnotice_names
ol_dump
peruse_crossref

File No.: 1L13, 1U13

7/82

AZ03-02A

CONTENTS

ILLUSTRATIONS

TABLES

SECTION 1

REFERENCE TO COMMANDS AND SUBROUTINES BY FUNCTION

COMMAND REPERTOIRE

The Multics commands described in this manual are organized by function into the following categories:

Administrative Utilities
Debugging Facility
Directory Checking
General Purpose
Object Segment Manipulation
MST Maintenance

Detailed descriptions of these commands, arranged alphabetically rather than functionally, are given in Section 2 of this document.

Administrative Utilities

check_mdcs	performs maintenance and verification of MDC data
compare_configuration_deck	compares the configuration deck for the running system to a second copy
compare_dump_tape	verifies tape copies of an original Multics storage-system-hierarchy dump tape
copy_dump_tape	generates tape copies of an original Multics storage-system-hierarchy dump tape
compare_dump_tape_status	prints true or false
date_deleter	directory-housekeeping tool: deletes segments not used in a given length of time
delete_old_pdds	deletes old copies of >process_dir_dir created during previous bootload
display_ioi_data	displays the ring 0 data base ioi_data
fix_quota_used	repairs inconsistencies in storage system
mcs_version	prints version of the core image most recently loaded into a specified FNP
monitor_log	monitors activity in standard format log segments
prelink	creates a prelinked subsystem

print_configuration_deck	prints system configuration information from configuration deck
privileged_prelink	creates a prelinked subsystem in lower rings
reset_tpd	resets the transparent paging device attribute of a segment
save_history_regs	saves processor history registers upon each occurrence of a signalable fault in the signaler's stack frame
set_timax	adjusts scheduling parameters of the calling process
set_tpd	sets the transparent paging device attribute of a segment

Debugging

change_kst_attributes	modifies specialized per-initiation segment attributes
copy_dump	copies an fdump image taken by BOS out of the dump partition into the Multics hierarchy
deactivate_seg	causes the deactivation of a segment
display_branch	displays internal system information in a directory branch without attempting VTOC access
display_kst_entry	displays internal system information in the KST entry
ol_dump	looks at selected parts of an online dump created by the BOS FDUMP command and copied into the Multics hierarchy by the copy_dump command
print_apt_entry	displays the contents of specified APT entries in either octal or interpreted form.
print_error_message	prints the interpretation of standard Multics status codes
print_sample_refs	interprets the three data segments produced by the sample_refs command and produces a suitable output segment
process_id	prints (or returns) the process id of a specified user
sample_refs	samples the machine registers in order to determine which segments a process is referencing
vfile_find_bad_nodes	examines a vfile_ keyed file to determine whether the vfile_ MSF components which contain certain keys are in a consistent state

vtoc_pathname	prints the pathname of a segment given the location of its VTOC entry

Directory Checking

comp_dir_info	compares and reports the differences between two directory information segments
get_library_segment	obtains system source programs from system libraries
list_dir_info	prints information about the state of a directory saved by the save_dir_info command
list_sub_tree	lists the segments in a specified subtree of the hierarchy
rebuild_dir	reconstructs a directory from information saved by the save_dir_info command
save_dir_info	saves information about the state of a directory and the segments and links in it

General Purpose

add_pnotice *	protects source code programs by adding, at the beginning of a program, a software protection notice in a box delimited as a comment
clear_partition	overwrites the contents of a disk partition with zeroes or an optional user-supplied word
cross_reference *	creates a cross-reference listing of object programs and include files
display_label	prints information recorded in the physical volume label for a storage system disk volume
display_psp	displays information about distributed software installed in online system libraries
display_pnotice	displays information on software protection notices contained in source programs
display_pvte	prints information recorded in the Physical Volume Table Entry (PVTE) for a storage system disk volume
do_subtree	executes one or two command lines after substituting the pathname of a given directory in the command line
dump_partition	displays data from a moved data partition
expand	generates copies of source programs with include file texts included

generate_pnotice	protects Multics source, object archives, and executable software via copyright
get_ips_mask	prints the current state of the IPS mask for the calling process
hp_delete_vtoce	deletes a specified VTOC entry
hunt	searches portions of the hierarchy for segments with names matching a given star name
list_partitions	lists the locations and sizes of all partitions on a specified physical volume
list_pnotice_names	displays the primary names of all protection notice templates
mexp	takes mexp source segments, expands any macros found therein, and generates as output an expanded text segment suitable as input to the ALM assembler
monitor_quota	notifies user of record quota overflow condition
nothing	does nothing: useful for metering and command language programming
pause	waits a specified real-time interval
peruse_crossref	displays information extracted from the output file generated by the cross_reference command
record_to_sector	converts an octal sector number to a disk sector address
record_to_vtocx	finds the VTOC entries, if any, corresponding to a specified record number on a storage system volume
reduction_compiler	compiles a segment containing reductions and action routines into a PL/I source segment
repeat_line	executes a given command line repeatedly
reset_ips_mask	sets the IPS mask for the current process to unmask some or all IPS signals
sector_to_record	converts an octal sector address to a Multics record number
send_ips	sends an IPS signal
send_wakeup	sends an IPC wakeup
set_ips_mask	sets the IPS mask for the current process to mask some or all IPS signals
teco	is a powerful, general-purpose, character-oriented, programmable text editor

teco_error	prints the long form of a teco error message given the short term
teco_ssd	allows the user to specify a directory for teco to search when trying to find a teco macro to execute
test_archive	checks an archive segment for archive format errors and other inconsistencies
vtocx_to_record	converts an octal VTOCE index to a Multics record number and sector offset

Object Segment

compare_object	details discrepancies between object segment
hunt_dec	searches a subtree for PL/I object segments
perprocess_static_sw_off	turns off an object segment's per-process static switch
perprocess_static_sw_on	turns on an object segment's per-process static switch

MST Maintenance

check_mst	produces report of contents of an MST and some cross-reference information
compare_mst	reports discrepancies between two MSTs
copy_mst	copies an MST onto a new reel of tape
excerpt_mst	extracts selected segments from an MST
generate_mst	creates an MST from storage-system segments and an MST header
list_mst	produces brief report of the contents of an MST
merge_mst	creates a new MST from an old one and storage-system segments, performing substitutions
write_mst	allows writing of short MST without the use of a header file

SUBROUTINE REPERTOIRE

The Multics subroutines described in this manual are organized by function into the following categories:

General Purpose
Interface to System Functions
Object-Segment Manipulation
Reduction-Compiler Utilities

Detailed descriptions of these subroutines, arranged alphabetically rather than functionally, are given in Section 3 of this document.

General Purpose

ask_	flexible terminal-input facility for numbers and strings
copyright_notice_	add (and optionally deletes) copyright notices to source-program segments
create_ips_mask_	returns a bit string that can be used to disable specified IPS interrupts
display_file_value_	outputs information about a file on a user-supplied switch
find_include_file_	locates an include file via system include-file search rules
find_partition_	ascertains information about a disk partition located on some mounted storage-system disk
hash_	is used to maintain a hash table; contains entry points that initialize a hash table and insert, delete, and search for entries in the table

hphcs_$read_partition	reads words of data from a specified disk partition on some mounted physical storage disk
hphcs_$write_partition	writes words of data into a specified disk partition on some mounted physical storage-system disk
mdc_$pvname_info	gets various information about a specified storage-system physical volume
parse_file_	parses ASCII text into symbols and break characters
phcs_$read_disk_label	reads the label of a storage-system disk volume
rehash_	reformats a hash table of the form maintained by the hash_ subroutine into a different size
sort_items_	provides a general sorting facility
sort_items_indirect_	provides a facility for sorting a group of data items
sweep_disk_	walks a given subroutine over a subtree of the directory hierarchy

Interface to System Functions

abbrev_	subroutine interface to the abbrev command
datebin_	decodes calendar clock values into binary integers
get_initial_ring_	obtains a process' initial ring number
hcs_$get_page_trace	retrieves trace of process' page faults from the supervisor
hphcs_$ips_wakeup	sends a specified IPS signal to a specified process
link_unsnap_	unsnaps all links pointing to a given segment
list_dir_info_	internal interface to the list_dir_info command
parse_channel_name_	parses a character string that is intended to be an IOM channel number
ring0_get_	supplies name, segment-number, and entry-point information about ring 0 segments
teco_error	prints the long form of a teco error message given the short term
teco_get_macro_	called by teco to search for an external macro

Object Segment

decode_definition_	returns information about a definition in the object segment
get_bound_seg_info_	supplies structural information about a bound segment
stu_	retrieves information from the runtime-symbol-table section of an object segment.
translator_info_	supplies source-segment information for use by translators building object segments

Reduction Compiler Utilities

lex_error_	generates compiler-style error messages
lex_string_	parses ASCII character strings
translator_temp_	provides a temporary storage-management facility for translators

SECTION 2

COMMAND DESCRIPTIONS

COMMAND DESCRIPTION FORMAT

This section contains descriptions of Multics commands, presented in alphabetical order. Each description contains the name of the command (including the abbreviated form, if any), discusses the purpose of the command, and shows the correct usage. Notes and examples are included when deemed necessary for clarity. When a command may also be used as an active function, its usage and function are described near the end of the command description. The discussion below briefly describes the content of the various divisions of the command descriptions.

Name

The "Name" heading lists the full command name and its abbreviated form. The name is usually followed by a discussion of the purpose and function of the command and the expected results from the invocation.

Usage

This part of the command description first shows a single line that demonstrates the proper format to use when invoking the command and then explains each element in the line. The following conventions apply in the usage line.

1. Optional arguments are enclosed in braces (e.g., {path}, {User_ids}). All other arguments are required.

2. Control arguments are identified in the usage line with a leading hyphen (e.g., {-control_args}) simply as a reminder that all control arguments must be preceded by a hyphen in the actual invocation of the command.

3. To indicate that a command accepts more than one of a specific argument, an "s" is added to the argument name (e.g., paths, {paths}, {-control_args}).

NOTE: Keep in mind the difference between a plural argument name that is enclosed in braces (i.e., optional) and one that is not (i.e., required). If the plural argument is enclosed in braces, clearly no argument of that type need be given. However, if there are no braces, at least one argument of that type must be given. Thus "paths" in a usage line could also be written as:

path1 {path2 ... pathn}

The convention of using "paths" rather than the above is merely a method of saving space.

4. Different arguments that must be given in pairs are numbered (e.g., xxx1 yyy1 {... xxxn yyyn}).

5. To indicate that the same generic argument must be given in pairs, the arguments are given letters and numbers (e.g., pathA1 pathB1 {... pathAn pathBn}).

6. To indicate one of a group of the same arguments, an "i" is added to the argument name (e.g., pathi, User_idi).

To illustrate these conventions, consider the following usage line:

command {paths} {-control_args}

The lines below are just a few examples of valid invocations of this command:

```
command
command path path
command path -control_arg
command -control_arg -control_arg
command path path path -control_arg -control_arg -control_arg
```

In many cases, the control arguments take values. For simplicity, common values are indicated as follows:

STR
: any character string. Individual command descriptions indicate any restrictions (e.g., must be chosen from specified list, must not exceed a particular length, etc.).

N
: number; individual command descriptions indicate whether it is octal or decimal and any other restrictions (e.g., cannot be greater than a certain limit).

DT
: date-time character string in a form acceptable to the convert_date_to_binary_ subroutine described in the MPM Subroutines.

path
: pathname of an entry; unless otherwise indicated, it may be either a relative or an absolute pathname.

The lines below are samples of control arguments that take values:

```
-access_name STR, -an STR
-ring N, -rg N
-date DT, -dt DT
-home_dir path, -hd path
```

Notes

Comments or clarifications that relate to the command as a whole are given under the "Notes" heading. Also, where applicable, the required access modes, the default condition (invoking the command without any arguments), and any special case information are included.

Examples

The examples show different valid invocations of the command. An exclamation mark (!) is printed at the beginning of each user-typed line. This is done only to distinguish user-typed lines from system-typed lines. The results of each example command line are either shown or explained.

Other Headings

Additional headings are used in some descriptions, particularly the more lengthy ones, to introduce specific subject matter. These additional headings may appear in place of, or in addition to, the notes.

add_copyright

Name: add_copyright

This command has been replaced by the add_pnotice command.

Name: add_pnotice

The add_pnotice command protects source code programs by adding, at the beginning of a program, a software protection notice (copyright or trade secret) in a box delimited as a comment. Multiple protection notices are supported. Archives of source code programs can be protected using this command. The archive pathname convention is supported. If a particular language or suffix is not supported, an appropriate message is printed. By default, protection notice templates in >tools are used.

Usage

add_pnotice path {-control_arg}

where:

1. path
   is the name of a source code program or an archive of source programs that require protection notices. The language suffix or archive suffix must be included.

2. control_args
   can be chosen from the following:

   -name STR, -nm STR
   where STR specifies the name of a protection notice template to be added (see Notes below).

   -trade_secret
   specifies that the notice to be added to the segment is the default Honeywell's trade secret notice (which has an added name of default_trade_secret.pnotice).

Notes

If no control arguments are specified, the notice added is the Honeywell's copyright notice with the name default.pnotice.

A list of available copyright and trade secret protection notice template names can be obtained with the list_pnotice_names command. The -name can be used to specify any of these templates.

For further information on the software protection facility see the Multics Library Maintenance PLM Manual (Order No. AN80).

Name: change_kst_attributes

The change_kst_attributes command allows a user to change selected per-process attributes of a segment. *

Usage

change_kst_attributes {-control_arg} target attributes

where:

1. control_arg
   can be -name (or -nm) and is only used if the target is a relative pathname that looks like a segment number.

2. target
   specifies the segment whose KST (known segment table) attributes are * to be changed. Either a relative pathname or an octal segment number may be specified.

3. attributes
   specifies those attributes that are to be changed. One or more of the following must be given:

   tpd
   if set, pages of this object are not placed on the paging device on the account of the user.

   tms
   if set, date-time-modified is not updated on the account of the user.

   tus
   if set, date-time-used is not updated on the account of the user.

   allow_deactivate
   if set, permits explicit deactivation of the segment. *

   allow_write
   if set, the user is not prevented from writing into the segment or directory if he has permission to do so.

   audit
   if set, enables auditing.

change_kst_attributes

Notes

Because directories are activated when their segment numbers are assigned, it is not possible to meaningfully set the tpd, tms, tus, or allow_deactivate attributes for a directory.

If an attribute is preceded by the circumflex character (^), then the attribute is reset. Otherwise, the attribute is set. Attributes not mentioned are unaffected.

This command requires access to the hphcs_ gate if the tms or tus attributes are to be set; otherwise, access to the phcs_ gate is required.

Name: change_tuning_parameters, ctp

The description of the change_tuning_parameters command may now be found in Multics System Metering, Order No. AN52.

Name: check_mdcs

The check_mdcs command checks for valid format and invalid unique identifier (UID) pathnames in the master directory control segment (MDCS) for a given volume. These segments are found in >lv, and are sometimes damaged by system crashes. Any errors found are reported via the syserr log and, if possible, corrected.

Usage

```
check_mdcs volume
```

where volume is the name of a storage system volume.

Note

Access to the mdc_priv_ gate is required to use this command.

Name: check_mst, ckm

The check_mst command reads a Multics system tape (MST) and provides information about improper combinations of attributes and missing procedures. It also provides information about the segment numbers assigned for supervisor and initialization segments, their names, attributes, and whether or not they were referenced.

Usage

```
check_mst reel_id {-control_args}
```

where:

1. reel_id
   is the reel identification number of the tape to be written. The reel identification number, which is site dependent, may be up to 32 characters long. The reel_id may also include a density specification to indicate the density of the tape being written, as in "060341,den=1600".

2. control_args
   can be selected from the following:

   -collection, -coll
   succeeding numeric arguments define the collections after which a cross-reference check is desired. If this argument is not supplied, a cross-reference check is performed only after collections two and three.

   -debug, -db
   sets a switch that preserves the tables constructed during checking; should not normally be used.

   -file
   checks a file rather than a tape.

Notes

For normal use in checking out a new Multics system, the usage is:

```
check_mst NNN
```

where NNN is the reel identification number desired.

Provision is made for up to three collections to be loaded and checked for cross-references.

A BOS tape may not be checked with this command.

Diagnostics and Results

Output produced is directed to the segment NNN.ckrout where NNN is the reel identification number of the specified tape.

After each collection is read, the system prints the number of words to be loaded into segments directly from the tape. When the tape is dismounted, the system prints the total number of words read from the tape, including both segment contents and loading information.

If the grand total line does not immediately follow a collection total line, the ckrout segment should be examined immediately to discover the cause. Certain diagnostics relating to improper format or sequencing should occur within the first 20 to 50 lines.

The following messages can be written into the ckrout segment while processing a tape.

1. "Loading collection No.i": The system is starting to read the ith collection from the tapes, extracting names and linkage information.

2. "Collection-mark K": A collection mark with a value of K was read from the tape, completing the loading phase for the collection.

Following the "Collection-mark" message, the running count of segments processed in the various categories and the number of words used for each category are listed. The previous two messages are the only ones that normally occur; however, various other diagnostics can appear in various other cases listed below.

3. "Illegal control word xxx after seg j": The format of the tape was incorrect, such that the 12-octal-digit string xxx was not valid in the context in which it was encountered; an error in generating the tape or an unnoticed tape error may have been responsible. If this condition occurs, no more information is read from the tape.

4. "Possible tape format error or incorrect switch setting": The physical tape cannot be successfully read by the Multics tape reader package; or, the number of collections to be checked as specified in the argument list to the command does not exist on the tape.

5. "Name <z> also on seg No.j": Duplicate names appear on the tape. This occurs normally only in collection 3 for two segments that are different in ring 0 and in some outer ring.

6. "Seg i: message": Certain checks are made for unusual combinations of attributes in the description of the segment, and message describes the conflict.

7. "Bad text-link sequence after z": The next segment after a text segment was not the linkage segment it should have been; or, a linkage segment was found not preceded by its text segment. The same actions occur as for (3) above.

8. "Status conflict": Wired code is referencing nonwired code. This condition is not necessarily an error, since program logic may permit it.

Cross-Reference Output

For each group of collections for which a cross-reference listing was requested, the following output appears in the ckrout segment.

1. For each collection, in sequence:

   Collection No. i, collection mark k

   (values of i and k are as before).

2. For each segment for which one or more diagnostics appear: its segment number, and all its names.

3. Diagnostic messages

   a. First character is <
   the link could not be satisfied, for the reason given.

   b. "message name"
   advisory diagnostic pertaining to possible mismatching attributes between the referencing and referenced segments.

Segment Summary Listing

For each segment on the tape(s) read, the following information appears:

1. Segment No. — the system assigned segment number (in octal).

2. Segment name — the primary name of the segment as specified on the tape (secondary names appear indented on successive lines).

3. Acc — the access to the segment as specified on the tape.

4. Switches (miscellaneous attributes)

Segment	Status
W	Wired down.
G	The segment is paged.
P	The segment is a per-process segment.
S	The segment is to be deleted at shutdown.
B	The segment is an abs segment.
T	The segment is a temporary segment.
L	The segment has an associated linkage segment.
C	The segment's linkage section will be combined.
K	The segment's linkage section will be wired.
N	The segment is not referred to by linkage reference, and is not itself a linkage section.

A The segment is encacheable.

* The segment is never referenced.

5. Ring brackets

6. Length (a multiple of 16 words)

7. Pathname

Notes

Between collections, the new collection number (i), is printed out. Segments listed for collection 0 are those bootstrap1 manufactures before loading real segments.

If unexpected status is received during reading of the tape, a message is printed on the user's terminal and the debug command is called. If maintenance personnel are available, they should be contacted for further information; otherwise, type .q<NL> to exit from the debug command so that cleanup operations can be performed.

Name: clear_partition

The clear_partition command overwrites the contents of a disk partition with zeroes or an optional user-supplied pattern words. See also dump_partition and list_partitions commands in this manual.

Usage

clear_partition pvname partname {control_args}

where:

1. pvname
   is the name of the physical volume on which the partition to be cleared exists.

2. partname
   is the name of the partition to be cleared. It must be four characters or less in length.

3. control_arg
   may be chosen from the following:

   -brief, -bf
   produces brief format messages.

   -long, -lg
   produces longer format messages. (Default)

   -pattern word
   overwrites the partition with data consisting of the specified octal pattern word. The specified word is written into every location in the partition. If this control argument is not specified, a default of all zeroes is used.

Notes

Access to the phcs_ and hphcs_ gates is required.

The user is always queried as to whether the partition should be overwritten; by default (if -brief has not been specified), the contents of the first eight words in the partition are displayed (in octal and as ASCII characters) as part of this question, to aid in preventing accidental overwriting of the wrong partition.

Name: compare_configuration_deck

The compare_configuration_deck command compares the configuration deck for the running system to a saved copy.

Usage

compare_configuration_deck path {-control_arg}

where:

1. path
   is the pathname of a saved copy of the configuration deck.

2. control_arg
   may be one of the following:

   -brief, -bf
   suppresses the message if the two configuration decks are identical, and suppresses printing of the identifying headers.

   -long, -lg
   prints all output. (Default)

Output format (-long mode)

The long output format consists of up to four sections, each of which is printed, with an identifying header, if it is not empty. The four sections are added cards, deleted cards, changed cards and MEM cards. The section for MEM cards is only printed if the order or number of MEM cards in the two decks differs; otherwise, only changed MEM cards are printed. The changed cards are listed in pairs, such as:

```
Was:    mem  a  123.  on
        mem  a  123.  off
```

The first line (prefaced by Was:) is the card from the saved deck, and the second line is the current card. If the two decks are different in order or number, this is announced, and both decks are printed in their entireties.

Output format (-brief mode)

The brief output format omits the section headings and the message: "The two decks are identical." Cards are identified by preface--added cards are prefaced by "New:" and deleted cards by "Old:". Changed cards are listed in pairs, in the same format as in the long output mode. If the MEM cards section is printed, it is the last section. The MEM cards are listed in two groups, with the first card in each group prefaced by "Was:" (for the first group) or "Now:" for the second group, and all the other cards in the group are listed with no preface.

compare_configuration_deck

Examples

Long mode output

Note that because the MEM cards have been reordered, the changed card for MEM A is not listed in the changed cards section.

```
Cards added in current deck:
  parm  chwm  dirw  ttyb  7000.
  salv  pdlv  1

Cards deleted from old deck:
  intk  warm  3  star

Changed cards:
Was:  cpu   b  6  off
      cpu   b  6  on

MEM cards are reordered:
Was:  mem   a  258.  on
      mem   c  258.  on
      mem   b  258.  on
Now:  mem   c  258.  on
      mem   a  258.  off
      mem   b  123.  on
```

Brief mode output

This output is equivalent to the sample output for the long mode shown above.

```
Old:  parm  chwm  dirw  ttyb  7000.
Old:  salv  pdlv  1
New:  intk  warm  3  star
Was:  cpu   b  6  off
Now:  cpu   b  6  on
Was:  mem   a  258.  on
      mem   c  258.  on
      mem   b  258.  on
Now:  mem   c  258.  on
      mem   a  258.  off
      mem   b  123.  on
```

Usage as Active Function

[ccdk path]

When used as an active function, ccdk returns either "true" or "false" to indicate whether the current configuration deck and the copy are equivalent.

compare_configuration_deck

Notes

This command attempts to be as accurate as possible when identifying "changed" cards--it knows about the cards (such as MEM, CPU, etc.) that may appear several times and specify multiple items and identifies them by their operands as well as by name. It decides that the two decks are "completely" different if there appear to be more than 32 differences between them.

Name: comp_dir_info

The comp_dir_info command compares two directory information segments created by save_dir_info and reports on the differences.

Usage

comp_dir_info path1 path2 {-control_arg}

where:

1. path1
   is the pathname of the old directory information segment. If the dir_info suffix is not supplied, it is assumed.

2. path2
   is the pathname of the new directory information segment. If the dir_info suffix is not supplied, it is assumed.

3. control_arg
   can be one of the following:

   -brief, -bf
   compares and prints minimum information.

   -verbose, -vb
   compares and prints almost all information.

   -long, -lg
   compares and prints all information.

Notes

If no control argument is specified, the -verbose control argument is assumed.

Output from the comp_dir_info command is written on the user_output I/O switch.

Unless the -brief control argument is specified, a form feed character is transmitted and then a heading is printed that identifies the directories being compared and the times the information was saved.

Output is in three sections:

modified entries
deleted entries
added entries

and is identified by entry type (dir, seg, or link) and the entryname.

For deletions and additions, a heading of the form:

deleted: entry entryname

is printed, followed by a listing of the attributes of the deleted or added entry, in the format:

item_name: value

For segments that have been modified, a heading of the form:

modified: entry entryname

is printed, followed by a line of the following formats:

item_name changed from value1 to value2
item_name added: value

(The second format is used to report the addition or deletion of names, ACL entries, etc.)

The list below shows the output items according to the control argument and entry type. The control arguments are listed in order of their verbosity; i.e., the -brief (-bf) control argument prints out the least information, the -verbose (-vb) control argument prints out more information (including the "-bf" items), and the -long (-lg) control argument prints out all of the items listed.

segments:

-bf names
ring brackets
damaged switch
property list
deletion of ACL
truncation

-vb safety switch
copy switch
tpd switch
no_complete_dump_switch
no_incremental_dump switch
security OOS switch
audit flag
multiclass switch
access class
author
bit count author
ACL
date branch modified
records used
max length

-lg date modified
volume
bit count
entry point bound

directories:

-bf names
ring brackets
damaged switch
property list
deletion of ACL
sons volume
master dir
quota
MSF indicator

-vb safety switch
copy switch
tpd switch
no_complete_dump switch
no_incremental_dump switch
security OOS switch
audit flag
multiclass switch
access class
author
bit count author
ACL
initial seg ACL
initial dir ACL

-lg ACL
date branch modified
date modified

links:

-bf names
type
link target

-vb date link modified

-lg link dumped

When looking for a match between the old and new dir_info segments, the comp_dir_info command looks first for a match on the unique ID item. If no match is found, it looks for any entry with a name matching the primary name of the old entry.

If a match is found, the comp_dir_info command checks a set of items (depending on the specified control argument) to determine whether to report the entry as modified.

The names item is always checked. The date dumped and date used items are never compared. Other checking is dependent upon the control argument.

If comp_dir_info completes a pass without finding any modifications, deletions, or additions, it prints "Identical." Invoking the command with a more verbose control argument may detect some changes.

Name: compare_dump_tape

The compare_dump_tape command provides a method of verifying that copies of an original Multics storage system hierarchy dump tape are correctly made.

Usage

compare_dump_tape {-control_args}

where:

1. control_args
   are optional and may be one or more of the following:

   -master_volume STR1...STRn, -mvol STR1...STRn
   provides a list of master tape volume names forming the master volume set used for comparison. The names are separated from one another by a blank. Up to 10 volume names may be given. If not specified, the user is queried for master tape volume names as each tape in the volume set is read.

   -copy_volume STR1...STRn, -cvol STR1...STRn
   provides a list of tape volume names forming the copy volume set being compared with the master volume set. The names are separated from one another by a blank. Up to 10 volume names may be given. If not specified, the user is queried for volume names as each tape in the copy volume set is compared with the master volume set.

   -track N, -tk N
   specified that tapes in one of the volume sets are to be mounted on a tape drive capable of handling tapes containing N tracks. The default track size is 9. (See "Notes" below.)

   -density N, -den N
   specifies that tapes in one of the volume sets are to be mounted on a tape drive capable of handling tapes written at density N. However, the actual density at which the tapes were written determines the density that is used. If omitted, the default density is 1600 BPI. (See "Notes" below.)

   -abort
   aborts the verification when the first tape discrepancy is found. The default is to report all discrepancies found between the master and copy tape sets.

   -select
   specifies that the copy set being compared is a subset of the contents of the master set being compared. The purpose of this control argument is to allow the comparison software to skip those components on the master set that do not appear on the copy set.

Notes

If the -master_volume or -copy_volume control argument or both are not specified, the user is queried for the tape labels of the master or copy tape set or both.

The -track and -density arguments may only be given after the -master_volume or -copy_volume control arguments. Thus, if default track and density are not correct, -master_volume or -copy_volume must be given.

The -track and -density arguments apply only to the tapes composing the master or copy volume set associated with the preceding -master_volume or -copy_volume argument. Different -density and -track arguments can appear with each -master_volume or -copy_volume control argument if the default track size and density are not appropriate for that volume set.

If the user wishes to specify nondefault track size or density, then specific values must be provided in the list of tape reels. The specification is made as part of the reel label. Each specification is separated by a comma. For example:

```
compare_dump_tape -abort -mvol M1, 9track,den=800 M2,9track,den=800 -cvol C1 C2
```

Examples

```
compare_dump_tape -mvol M1 M2 M3 -cvol C1 C2 C3
```

compares master tape volumes M1, M2, and M3 with copy volumes C1, C2, and C3.

```
compare_dump_tape -abort -mvol M1 -cvol C1 -den 800 -tk 7
```

compares master tape voluem M1 with copy volume C1, which is mounted on a 7-track, 800 BPI tape drive.

compare_dump_tape_status

Name: compare_dump_tape_status

The compare_dump_tape_status command prints either "true" or "false" depending on the just-completed invocation of the compare_dump_tape command. If invoked as an active function, the values true or false are returned to the caller. The program does not accept input arguments in either case.

Usage

compare_dump_tape_status

As an active function:

compare_dump_tape_status.

Name: compare_mst

The compare_mst command is used to read two Multics system tapes (MSTs) and to list all differences between them. All differences in segment headers, and the starting address of any inequalities or differing lengths of segment contents are noted. Additions, deletions, and moves of segments are handled. One can optionally save the contents of differing segments in the user's working directory for further detailed comparisons. Any number of collections can be handled, but a warning message is printed if a tape does not end in a collection mark. If the active_all_rings_data segment is found on the first tape, a message containing the system identifiers of both tapes is printed.

Usage

compare_mst reel_id1 reel_id2 {-control_arg}

where:

1. reel_id1
   is the reel identification number of the tape to be written. The reel identification number, which is site dependent, may be up to 32 characters long. The reel_id may also include a density specification to indicate the density of the tape being written, as in "060341,den=1600".

2. reel_id2
   is the reel identification number of the new tape.

3. control_arg
   is -save to save the contents of corresponding segments with discrepancies in the user's working directory under the names tp1.<segment_name> and tp2.<segment_name>. An added segment is saved under the name tp2.<segment_name>.

Name: compare_object, cob

The compare_object command compares two object segments and, optionally, prints out the changes made to the segment specified by oldpath to yield the segment specified by newpath. The assumption is that the first segment is older than the second, and that they were both produced from the same source segment but, potentially, by different versions of a language processor.

Usage

compare_object oldpath newpath {-control_args}

where:

1. oldpath
   is the pathname of the first segment.

2. newpath
   is the pathname of the second segment.

3. control_args
   may be chosen from the following:

   -brief, -bf
   prints out by section a summary of discrepancies in the object segments, suppressing detailed listing of the discrepancies.

   -text
   compares the text sections of the two segments.

   -defs
   compares the definition sections of the two segments.

   -link, -lk
   compares the linkage sections of the two segments.

   -all, -a
   compares the text, definition, and linkage section of the two segments. If the segments have separate static sections, these are compared also. This is the default.

   -static
   compares the static section of two segments with separate static; otherwise, compares the linkage sections.

Name: copy_dump

The copy_dump command copies an fdump image taken by BOS out of the dump partition into the Multics hierarchy. It creates as many segments (up to ten) in >dumps as necessary to hold the fdump image.

Usage

```
copy_dump
```

The name of each segment has the form:

```
mmddyy.tttt.s.eee
```

where:

1. mmddyy
   is the date the dump was taken.

2. tttt
   is the time the dump was taken.

3. s
   is a sequence number (0, 1, 2, ... 9).

4. eee
   is the error report form (ERF) number used in reporting this dump.

Entry: copy_dump$set_fdump_num, copy_dump$sfdn

This entry sets the value of the next FDUMP to be taken by changing the value associated with the ERF number in the dump partition.

Usage

```
copy_dump$set_fdump_num erfno
```

where:

1. erfno
   is the ERF number for the next fdump to be taken.

This entry point will modify the dump partition only after the last dump taken has been copied. If an attempt is made to change the ERF number before a dump has been copied, an error message will be returned.

Notes

This command does not allow a particular dump to be copied twice; therefore, it will return an error code if an attempt is made to recopy a dump.

This command interfaces to hphcs_$copy_fdump and to hphcs_$set_fdump_num; it requires hphcs_ access.

Notes

If no control arguments are specified, the text, definition, and linkage sections of the two segments are compared.

The equal convention may be used.

In comparing the lengths of the symbol sections of the two segments, the compare_object command uses a heuristic to determine whether a discrepancy is serious or trivial (e.g., caused by differences in pathnames of include files). This heuristic errs in the direction of caution and tends to be inaccurate for large object segments.

Name: copy_dump_tape

The copy_dump_tape command generates tape copies of an original Multics storage system hierarchy dump tape.

Usage

copy_dump_tape {-control_args}

where:

1. control_args
   can be one or more of the following:

   -density N, -den N
   specifies that output tape volume sets are to be written at a density of N. If specified for input tape volume sets, the tapes are mounted on a tape drive capable of handling tapes written at density N. However, the actual density at which the input tapes were written determines the density that is used. If omitted, the default density is 1600 BPI. (See "Notes" below.)

   -input_volume STR1...STRn, -ivol STR1...STRn
   provides a list of input tape volume names (STR) forming the input volume set being copied. The names are separated from one another by a blank. Up to 10 volume names can be given. If not specified, the user is queried for input tape volume names as each tape in the volume set is copied.

   -map {NAME}
   controls the generation and naming of a dump map. If -select is used, and -map is used without a NAME, the map will have a name of "SELECTION_FILE_NAME.map". If a NAME is given, the map will have a name of "NAME.map". If -select is not used and -map is used without a NAME, the map will have a unique name with the .map suffix added; otherwise, the map will have a name of "NAME.map".

   -output_volume STR1...STRn, -ovol STR1...STRn
   provides a list of tape volume names forming one of the output volume sets produced by the copy operation. Volume names are separated from one another by a blank. Up to 10 volume names can be given in each output volume set. More than one copy of the input volume set can be produced by specifying several -output_volume control arguments. If not specified, only one copy of the input volume set is produced. The user is queried for output tape volume names as each tape is written.

   -select STR
   STR is the pathname of a file similar to a standard backup_dump dump control file. See Notes below for further details. For complete information regarding the format of a dump file, see the writeup on backup_dump in the Multics Operator's Handbook (Order No. AM81).

-track N, -tk N
specifies that tapes in one of the volume sets are to be mounted on a tape drive capable of handling tapes containing N tracks. The default track size is 9. (See "Notes" below.)

Notes

If only one copy is being made, the -output_volume control argument is optional. The user is queried for the volume name of each output volume as it is written.

When several copies are made at the same time, the -output_volume control argument must appear once for each copy being made, followed by a list of tape volume names comprising that copy. The -track and -density arguments can only be given after the -input_volume or -output_volume control arguments, and apply only to the tapes composing the input or output volume set associated with the preceding -input_volume or -output_volume argument. If more than one copy is being made, different -density and -track arguments can appear after each -output_volume control argument if the default track size and density are not appropriate for that volume set.

Pathnames in the optional selection file must contain only primary names. They must also be full pathnames; relative pathnames are not allowed.

The map generated does not denote the tape number on which an object was written. Only one map is generated even though more than one output volume set may be specified.

Examples

```
copy_dump_tape -ivol i1 i2 -ovol o1 o2
```

copies volumes i1 and i2 onto output volumes o1 and o2.

```
copy_dump_tape -ivol i1 i2 -ovol o1 o2 -ovol o3 o4
```

copies volumes i1 and i2 onto two output volume sets, o1 and o2, and o3 and o4.

*

Name: copy_mst, cpm

The copy_mst command is used to copy a Multics system tape (either a Multics or BOS bootload tape) onto another reel of tape.

Usage

```
copy_mst reel_id1 reel_id2
```

where:

1. reel_id1
   is the reel identification number of the tape to be written. The reel identification number, which is site dependent, may be up to 32 characters long. The reel_id may also include a density specification to indicate the density of the tape being written, as in "060341,den=1600".

2. reel_id2
   is the reel identifier number of the tape onto which the copy is to be made.

Note

The message:

"Tape tape_id1 does not end in a collection mark"

is normal for BOS tapes.

copyright_archive

Name: copyright_archive

This command has been replaced by the add_pnotice command.

This page intentionally left blank.

Name: cross_reference, cref

The cross_reference command creates a cross-reference listing of any number of object programs. The listing contains information about each object module encountered, including the location of each program, its entry points and definitions, its synonyms, if any, and which other modules encountered reference each entry point or definition. It also optionally supplies a cross-reference listing of include files used by the modules encountered.

Usage

cross_reference library descriptions {-control_args}

where:

1. library descriptions
   have three forms:

   path1 path2 ... pathN

   -library library_name path1 path2 ... pathN

   -library library_name -all path1 path2 ... pathN

   (The control argument "-library" can be abbreviated as "-lb".) Each pathi is the pathname of a segment to be examined and cross-referenced. The star convention is allowed. The library_name can be any user-chosen identifier. All modules represented by path1 ... pathN are treated by the cross-referencer as if they were in a common library of that name. If the library description contains the control argument "-all," all the module names encountered are considered external (see "Resolving References" below). This control argument is generally used only for cross-references of the Multics Hardcore libraries.

2. control_args
   are one or more consistent combinations of the following:

   -input_file, -if path
   specifies that a control file describing the modules to be cross-referenced is to be used instead of the library descriptions. If the crl suffix is not part of the supplied filename, it is assumed. If this control argument is used, no library descriptions are allowed.

   -output_file file, -of file
   specifies that the cross-reference list is to be created in a segment of the specified name. If the crossref suffix is not part of the supplied filename, it is assumed. If this control argument is not supplied, but its -input_file control argument is supplied, the output file takes its name from the input file, with the suffix ".crossref" replacing the suffix ".crl". Otherwise, the output file is named "crossref.crossref".

   -brief, -bf
   suppresses nonfatal error messages. This control argument does not affect the reporting of error messages to the output file.

-first
specifies, in conjunction with the -input_file control argument, that once any instance of particular module has been located, the cross-referencer need not search the remaining directories for other instances of modules with the same name. If this control argument is omitted, the cross-referencer searches all libraries in the search list for each module name supplied.

-include_files, -icf
specifies that include files used by all modules examined are also to be cross-referenced.

-short, -sh
specifies that referenced modules that are not included in the scope of any library_desci should not be included in the output. This control argument causes the output to reflect only the interrelationships among the modules in the libraries specified.

-line_length N, -ll N
causes lines in the output file to be formatted to the given line length. The default is 132.

Module Examination

Module examination is performed in two passes. The first pass defines all the segment names, synonyms, and definitions. The second pass examines external references, and attempts to resolve them with existing definitions.

Segments encountered fall into four classes: nonobject, bound segments, stand-alone modules, and archives.

When a nonobject segment is encountered, a warning message is printed to that effect, and the segment is included in the results of the cross_reference.

When a bound segment is encountered, a warning message is printed to that effect, and the segment is ignored. Bound segments are of no use to the cross-referencer, since information necessary to determine which components make use of which external reference links is no longer available due to the binding process. Instead, the object archive from which it was bound should be used.

When a stand-alone segment is encountered, it is analyzed for entry points, definitions, and external references. All additional names on the segment are entered as synonyms for the module. This information is then included in the results of the cross-reference.

When an archive is encountered, each component is analyzed for entry points, definitions, and external references. If a bindfile exists, synonyms for each component are derived from "synonym" statements in the bindfile, when they exist. This information is then included in the results of the cross-reference.

Modules are also identified by the segment in which they were found (either themselves, for a stand-alone segment, or the containing archive, for an archive) and by the library_name of the directory in which they are found. If the directory is specified without a library_name, the pathname of the directory is used as the library_name. This makes it possible to have multiple occurrences of segments with the same name, as long as they differ by at least one of these identification criteria.

Resolving References

When a module is examined by the cross-referencer, its name and synonyms are classified as "internal" or "external" by the following criteria:

1. If the module is stand alone, its name and synonyms are external.

2. If the module is archived, and the library description contained the "-all" control argument, its name and all its synonyms are considered external.

3. If the module is archived, and the library description did not contain the "-all" control argument, its name and each of its synonyms is external only if it appears in the "Addname:" statement of the bindfile. If no bindfile exists, the name and synonyms are considered internal.

The cross-referencer attempts to resolve external references on a best-match basis by using the following criteria:

1. If the reference can be satisfied by a definition in the same module, that definition is used.

2. If the referencing module is part of a bound segment, and it can be satisfied by a definition in the same bound segment, that definition is used.

3. If the reference can be satisfied by an external definition in the same library_name, that definition is used.

4. Otherwise, the first external definition encountered that satisfies the reference is used. If more than one such definition exists, a warning message is printed.

Format of a Driving File

If the -input_file control argument is specified, the cross-referencer takes its input from a special file.

The first lines of the file must contain the names of one or more directories to be searched. They are specified in the following manner:

```
-library:       (OR  -library -all:)
pathname_1           library_name_a
pathname_2           library_name_b
   ....
pathname_N           library_name_z;
```

Each pathname_i specifies a directory to be searched. When present, a library name (which may contain spaces) is used to describe the preceding directory name. (See "Module Examination" above.) The tokens "-wd" or semicolon ends the search list.

The next information in the file is a list of the segments to be examined. They must appear one to a line.

If the user wishes to explicitly define synonyms for any modules that would not otherwise be generated (e.g., a nonapparent reference name by which a segment is sometimes initiated), they can be included in this section with one or more lines of the form:

```
modulename syn1 syn2 ...  synN
```

These lines will not by themselves cause the cross-referencer to search for the module "modulename," since it may not be a freestanding segment. Any synonyms defined in this manner are considered external.

A file can consist of several repetitions of the format described above; that is, a search list, segment names, another search list, more segment names, etc. Whenever a new search list is encountered, it replaces the old search list. If a driving file is to be used, the greatest efficiency can be gained by having it consist of multiple occurrences of a one-directory search list followed by the segments contained in that directory.

For example, a control file constructed to cross-reference a student subsystem might look like the following:

```
-library:
>udd>Class>systemdir>object CLASS SUBSYSTEM;

class_login_responder.archive
class_tests.archive
student_grades_database
audit_procedure
class_utilities.archive
unallowed_compiler_stub fortran pl1
unallowed_compiler_stub
```

Special Cases

Segments with unique names and segments whose last component is a single digit are ignored, since these are conventions used by the system library tools to denote segments that are to be deleted shortly.

Archives whose names are identical with the exception of a different numeric next-to-last component are considered the same archive.

Definitions or entry points in archive components that masquerade as segment names by the expedient of an added name on the bound segment, without benefit of being defined as a synonym for their containing component, are not cross-referenced satisfactorily.

Include Files

The cross-reference listing of include files, when requested, is appended to the regular output of the cross-referencer. Each include file encountered is classified by its entryname and its date/time modified. This ensures that modules that use different versions of the same include file are apparent.

Example

The following command produces a cross_reference listing of the Standard Service System in the file "standard.crossref":

```
cref -library STANDARD >ldd>sss>o>** -of standard
```

To produce a cross_reference listing of the hardcore library, the following command line can be used:

```
cref -library HARD -all >ldd>h>x>* >ldd>h>o>*.archive -of hard
```

(Note the use of the "-all" control argument.)

Output Example

Entries are separated by dashed lines in the output listing. The following is a sample entry:

```
----------------------------*****bound_x_ in SSS *****----------------------------
sample_segname        SYNONYM:  one syn, another_syn
 one_entrypoint       program_a program_b
 second_entrypoint    program_a program_c
 unused_entrypoint
 undefined_ent (?)    program_d
```

The entry shown is for segment "sample_segname", which is a component of bound_x_ in the library specified as SSS. It possesses three entry points: "one_entrypoint", "second_entrypoint", and "unused_entrypoint". The information shows that "sample_segname$one_entrypoint" is called by module "program_a" and module "program_b". The question mark after entry point "undefined_ent" signifies that this entry point is an implicit definition; that is, that module "program_d" refers to "sample_segname$undefined_ent", but that entry point does not actually exist. (A diagnostic is printed when this situation is encountered.)

All error messages produced during the run, including warning messages that may not have been printed at the terminal due to use of the "-brief" control argument, are appended to the end of the output file for reference.

Name: date_deleter

The date_deleter command is used to perform a delete-by-date operation in a directory by removing all segments and multisegment files older than a specified number of days.

Usage

date_deleter dir_path n_days {star_names} {-control_args}

where:

1. dir_path
   is the pathname of the directory in which the deletions are to occur. The dir_path can be -working_directory or -wd to indicate the working directory.

2. n_days
   is the number of days that must have elapsed since a segment was last modified in order for it to qualify for deletion. The time elapsed is measured from date_time_contents_modified.

3. star_names
   are the optional names of files to be deleted. If none are specified, all files older than the specified number of days are deleted. Otherwise, only files matching one or more of the starnames, and older than the specified number of days, are deleted.

4. control_args
   may be chosen from the following:

   -date_time_contents_modified, -dtcm
   uses the dtcm of each entry. This is the default.

   -date_time_dumped, -dtd
   uses the dtd of each entry instead of the dtcm.

   -date_time_entry_modified, -dtem
   uses the dtem of each entry instead of the dtcm.

   -date_time_used, -dtu
   uses the dtu of each entry instead of the dtcm.

   -name STR, -nm STR
   specifies a starname STR that begins with a minus sign, to distinguish it from a control argument.

Examples

The command line:

```
date_deleter >ldd>old 7
```

deletes all files in >ldd>old last modified more than one week ago.

The command line:

```
date_deleter >udd>Proj>listing_pool 2 **.list
```

deletes all listing files in the >udd>Proj>listing_pool directory that are more than two days old.

Name: deactivate_seg

The deactivate_seg command allows a user to deactivate a segment or directory.

Usage

deactivate_seg segment {-control_arg}

where:

1. segment
   is the pathname of the segment or directory to be deactivated, or a segment number.

2. control_arg
   may be the following:

   -force, -fc
   causes the segment to be deactivated, if at all possible, by using the highly privileged demand_deactivate entry.

Notes

This command requires access to the phcs_ gate. If the -force control argument is used, it requires access to the hphcs_ gate.

If -force is not specified, the segment is only deactivated if all processes connected to the segment have the allow_deactivate attribute set for it. See the change_kst_attributes command for a description of the allow_deactivate attribute. If -force is specified, the segment is deactivated unless its entry-hold switch is set or it is a directory with active inferiors.

Name: delete_old_pdds

The delete_old_pdds command deletes old copies of >process_dir_dir created during bootload. This command is intended mainly for use in the system_start_up.ec.

Usage

```
delete_old_pdds {-control_args}
```

where:

1. control_arg
   may be chosen from the following:

   -exclude_first N
   specifies that the first N old copies of >process_dir_dir (that is, the N oldest ones) are not to be deleted.

   -exclude_last N
   specifies that the last N old copies of >process_dir_dir (that is, the N most recent ones) are not to be deleted.

Notes

The old copies of >process_dir_dir are named pdd.[unique], and branch directly off the root.

The control arguments for specifying that some old process dir copies are not to be deleted are useful when it is necessary to have the process directory contents of processes at the time of a crash, when debugging system problems.

This command requires access to the hphcs_ gate.

Name: display_branch

The display_branch command prints out information about directory entries that is not returned by the status command. It also lists the segment UID and the location of the branch. No attempt is made to access the VTOCE of the segment for any information.

Usage

display_branch {-control_arg} target

where:

1. control_arg
   can be -name (or -nm) and must be specified before any pathname that is a valid octal number or that can be construed as a valid octal pointer.

2. target
   indicates the segment whose branch is to be displayed. Any of the following forms can be used to specify the target segment: the pathname of the segment, the octal segment number of the segment, or the octal pointer representation of the address of the branch to be displayed (e.g., 260|1664)

Note

This command requires access to the phcs_ gate.

Name: display_ioi_data

The dipslay_ioi_data command can be used to display the ring_0 data base ioi_data. It can be used on a running system, an fdump, or a copy of ioi_data made with the copy_out command.

Usage

display_ioi_data {-control_args}

where:

1. control_args
   are selected from the lists below. The following control arguments are used to select where ioi_data is to be found. Only one control argument may be selected from this list. If none are specified, ioi_data is copied from ring_0 of the running system.

   -erf erfno
   specifies the number of the fdump to be analyzed.

   -segment path, -sm path
   specifies the pathname of the segment containing ioi_data. Normally, this segment would be obtained from the running system using the copy_out command or from an fdump using the extract command.

   The following control arguments specify which control blocks in ioi_data are to be displayed. Only one control argument may be selected from the following list. If none of these control arguments are specified, all control blocks are displayed.

   -group {device_name}, -gp {device_name}
   displays the group table entry (gte) for the device specified. If device_name is omitted, all gte's are displayed. The device_name may be either the full name of a nonmultiplexed device, such as prta, or the full name or first four characters in the name of a multiplexed device, such as tape or tape_02.

   -device {device_name}, -dv {device_name}
   displays the device table entry (dte) for the device specified. If device_name is omitted, all dte's are displayed.

   -channel {channel_name}, -chn {channel_name}
   displays the channel table entry (cte) for the channel specified. If channel name is omitted, all cte's are displayed. The channel name argument is in the form {tag}number, where tag is an IOM tag (a through h) and number is an octal channel number. If tag is omitted, IOM a is assumed.

   -gte {octal_offset}
   displays the gte at iom_data|octal_offset. If offset is omitted, all gte's are displayed.

   -cte {octal_offset}
   displays the cte at iom_data|octal_offset. If offset is omitted, all cte's are displayed.

display_ioi_data

-dte {octal_offset}
displays the dte at iom_data|octal_offset. If offset is omitted, all dte's are displayed.

-user {Person_id.Project_id}
displays the dte's of all devices assigned to the specified user. If Person_id.Project_id is omitted, the user's person_id is assumed. Either Person_id or Project_id may be omitted or an asterisk (*) can be used in their place. This argument is not allowed for -erf and will not work with -segment if the segment was created during a previous Multics bootload or if the users have since logged out.

Other control arguments:

-header, -he
causes the ioi_data header to be displayed. This is the default if no control blocks are selected.

-no_header, -nhe
suppresses the display of the ioi_header. This is the default when a control block is selected.

-all, -a
may be used in conjunction with -group or -gte. Causes all the cte's and dte's associated with the group(s) selected to be displayed also.

-force, -fc
forces the display of certain control block or fields or both that the command might not otherwise display. For example, '-gte' will display only allocated gte's; '-gte -force' would display all gte's, allocated or not.

Notes

To use this command on a running system, access to the gate phcs_ is required.

The default action of this commmand, when invoked with no arguments is:

display_ioi_data -group -all -header.

display_kst_entry

Name: display_kst_entry

The display_kst_entry command prints the contents of a KST (known segment table) entry. The KST entry to be dumped may be indicated by either a segment number or a relative pathname of the associated object.

Usage

```
display_kst_entry {-control_arg} target
```

where:

1. control_arg
   can be -name (or -nm) and must appear if target is a relative pathname that looks like a segment number.

2. target
   is either a segment number or a relative pathname.

Example

```
! display_kst_entry start_up.ec

segno:     256  at  155|470
usage:     0, 0, 0, 0, 2, 0, 0, 0
entryp:    243|5452
uid:       033100743603
dtbm:      416334652254
mode:      7 (4, 4, 4)
ex mode:   000000000000 (0, 0, 0)
infcount:  0
hdr:       4
flags:     write
```

Name: display_label

The display_label command prints information recorded in the physical volume label for a storage system disk volume. Optionally, it displays information recorded in the Physical Volume Table Entry (PVTE) for the associated disk unit.

Usage

```
display_label {dskX_NN} {-control_args}
display_label {PVNAME} {-control_args}
```

where:

1. dskX_NN
   specifies the disk subsystem and unit on which the volume is mounted (e.g., dska_07).

2. PVNAME
   is the physical volume name of the disk volume (e.g., rpv).

3. control_args
   can be selected from the following:

   -pvid PVID
   is used to specify the disk volume by the unique identifier assigned to the volume at the time it was registered (PVID). PVID is a 12-digit octal number. This control argument cannot be used if either dskX_NN or PVNAME is specified.

   -long, -lg
   causes information from the PVTE to be printed also.

Notes

The disk volume specified must be a mounted storage system volume.

This command requires access to phcs_.

Example

```
! display_label root3

Label for Multics Storage System Volume root3 on dska_18 d501

          PVID                237203135203
          Serial              root3
          Logical Volume      root
            LVID              257744057715

          Registered          12/03/80  1206.8
          Dismounted          03/29/82  2107.7
          Map Updated         03/29/82  2108.4
          Salvaged            03/29/82  2008.2
          Bootload            03/29/82  2108.0
          Reloaded
          Dumped
            Incremental
            Consolidated
            Complete


          Inconsistencies           0

          Minimum AIM         0:000000
          Maximum AIM         7:777777


Volume Map from Label

          First Rec   (Octal)      Size
                  0         0         8     Label   Region
                  8        10      2423     VTOC    Region
               2431      4577      1000     hc      Partition
               3431      6547       256     log     Partition
               3687      7147     58129     Paging  Region
              61816    170570      5000     dump    Partition
              66816    202400       384     bos     Partition
                                  67200     Total   Size
```

Name: display_pnotice

The display_pnotice command displays information on software protection notices contained in source programs.

Usage

```
display_pnotice name {control_arg}
```

where:

1. argument

   name
   is the full or relative pathname of the source language module. The language suffix or the archive suffix must be included if an entire archive is to be processed. The archive pathname convention is supported, but the star convention is not.

2. control_args
   can be chosen from the following:

   -long, -lg
   specifies that the full text of notices found will be displayed.

   -brief, -bf
   specifies that the primary name of notices, without the "pnotice" suffix, is printed instead of text of notices found. This is the default.

Notes

By default, the primary names of protection notices are printed instead of the entire notice text. If path includes the full archive name, then archives of source code programs can be audited for protection notices. If a source module does not contain any notices, or contains conflicting notices (copyright and trade secret), an error message is displayed. A warning message is also displayed if there is an embedded notice found in a source program (protection notices should be the first comment encountered).

Example

```
! display_pnotice add_pnotice.pl1
add_pnotice.pl1: HIS.1981

! display_pnotice add_pnotice.pl1 -lg
Notices in add_pnotice.pl1:

     Copyright, (C) Honeywell Information Systems Inc., 1981

! display_pnotice farf.pl1
Warning: farf.pl1 has no protection notice.
```

Name: display_psp

The display_psp command displays selected information about distributed software found installed in online systems libraries. The information includes marketing identifier, software technical identifier, copyright, and titles for the software requested.

Usage

```
display_psp {-control_args}
```

where:

1. control_args
   may be chosen from the following:

   -all, -a
   returns selected information of all products found installed in the systems libraries. This is the default.

   -brief, -bf
   prints only the software technical identifier. This is the default.

   -long, -lg
   prints the marketing identifier, software technical identifier, copyright, and titles selected.

   -copyright
   returns the copyright notice for selected products if found installed in the system library.

   -match STR
   where STR is the marketing identifier. This argument returns selected information for a specific product if it is installed in the systems library.

   -name, -nm
   returns selected information about a named product. The name of the product will be the long name by which the product is most commonly referred, i.e., compose, cobol, or ted.

Notes

The -brief and -long arguments are mutually exclusive, and only one argument can be given in a command.

The -match, -name, and -all arguments are mutually exclusive, and only one argument can be given in a command.

Name: display_pvte

The display_pvte command prints information recorded in the Physical Volume Table Entry (PVTE) for a storage system disk volume. Optionally, it displays information recorded in the physical volume label.

Usage

```
display_pvte {dskX_NN} {-control_args}
display_pvte {PVNAME} {-control_args}
```

where:

1. dskX_NN
   specifies the disk subsystem and unit on which the volume is mounted (e.g., dska_07).

2. PVNAME
   is the physical volume name of the disk volume (e.g., rpv).

3. control_args
   can be selected from the following:

   -pvid PVID
   is used to specify the disk volume by the unique identifier assigned to the volume at the time it was registered (PVID). PVID is a 12-digit octal number, which can be obtained by using the list_volume_registration (lvr) command. This control argument cannot be used if either dskX_NN or PVNAME is specified.

   -long, -lg
   causes information from the volume label to be printed also.

Notes

The disk volume specified must be a mounted storage system volume.

This command requires access to metering_gate_. If the -long control argument is specified, then access to phcs_ is required.

Example

```
! display_pvte root3

PVTE for Multics Storage System Volume root3 on dska_18 d501 at pvt|636

          PVID                  237203135203
          LVID                  257744057715

          VTOCEs
            Number                12115
            Left                   3933

          Records
            Number                58129
            Left                   9842
          Inconsistencies             0

          Volume Map
            volmap_seg ASTE    17|15140
            record stock        77|1400
            Page 0 - Base          7147
                     Free            47
            Page 1 - Base        103147
                     Free         23113
            Page 2 - Base        203147
                     Free            0
            vtoce stock         77|2134

          ON:         storage_system permanent hc_part_used

          OFF:        being_mounted being_demounted device_inoperative
                      vacating

Volume Map from PVTE

          First Rec   (Octal)      Size
                0         0          8      Label Region
                8        10       2423      VTOC Region
             2431      4606       1256      Partitions
             3687      7147      58129      Paging Region
            61816    170570       5383      Partitions
                                  67200      Total Size
```

Name: do_subtree

The do_subtree command operates a given directory (called the starting node) and all directories inferior to the starting node, by executing one or two given command lines after substituting the pathname of that directory in the command line. The substitution is performed by the do command (See MPM Commands), the directory pathname being taken as the first executed at each node before inferior nodes are operated on (the top down command line) and after inferior nodes are operated on (the bottom up command line).

The do_subtree command enables the user to execute the argument command lines in several processes. The walking of the hierarchy can be substantially speeded up by use of this facility. The process to which the initial command lines in starting mode was given is called the master process: the other cooperating processes are called the slave processes. The cooperating processes communicate via a segment called dos_mp_seg, which is found (or created if not found) in the working directory at the time the do_subtree command is issued. The master process must be logged in and begin executing first when multiple processes are used.

Usage

Master process or single process invocations:

do_subtree path -control_args

or, for slave process invocations:

do_subtree -slave

where:

1. path
   is the starting node. This must be the first argument. A path of -wd specifies the working directory (of the master process, if multiple processes are being used.)

2. control_args
   may be chosen from the following. They can appear in any order on the command line.

   -bottom_up STR, -bu STR
   specifies the bottom-up command line. It is taken as one argument, i.e., if it contains blanks, it must be enclosed in quotes. The name of the directory of execution is the first do command argument. It is recommended that this value be accessed with the string "&r1" rather than "&1" in case any directory names contain special characters.

-first N, -ft N
makes N the first level of the directory hierarchy at which the command lines will be executed. By definition, the starting node is at level 1. The default is -first 1. See the description of the walk_subtree command in the MPM Commands for examples of the use of -first.

-last, -lt N
makes N the last level in the storage-system hierarchy at which the command lines are executed. The default is 99999, i.e., all levels.

-long, -lg
causes printing of the names of directories at which the command lines are executed. Unlike with the walk_subtree command, this printing is off by default. In multiprocess executions with a bottom-up command line, an asterisk will precede all directory names for which the process executing the bottom-up command line is not the process that entered the directory first.

-multiprocess, -mp
specifies that the invoking process is to be the master process of a multiprocess execution. The dos_mp_seg segment is created in the current working directory and execution begins. As slave processes are started, work is distributed by the master and slave processes amongst themselves. Execution ends in all processes simultaneously. The top-down/bottom-up order of execution is guaranteed by all processes: no command line is executed at a given directory until the top-down command line (if any) is executed in all superior directories. The bottom-up command line (if any) is not executed at a given directory until all command lines have been executed in all inferior directories.

-no_msf
causes multisegment files not to be treated as directories. Unlike with the walk_subtree command, multisegment files are treated as directories by default. Most storage-system maintenance operations should not specify this control argument.

-slave
causes the command with this control argument to be executed in another process. This other process must be in a working directory where an active master process has begun executing a multiprocess invocation of do_subtree. All control arguments and command lines of the slave process will be the control arguments and command lines of that master process. The do_subtree command will finish execution in all processes at the same time. No more than 35 slave processes may be used.

-top_down STR, -td STR
specifies the top-down command line. It is taken as one argument, i.e., if it contains blanks, it must be enclosed in quotes. The name of the directory of execution is the first do command argument. It is recommended that this value be accessed with the string "&r1" rather than "&1" in case any directory names contain special characters. At least one of the control arguments -top_down or -bottom_up must appear. It is permissible to specify both.

AUXILIARY ENTRIES

Entry: do_subtree$recover

The do_subtree$recover entry point is used to pick up the work load of a process that has died in a multiprocess execution.

Usage

```
do_subtree$recover processnumber
```

where:

1. processnumber
   is the process number of the dead process. The process number of a do_subtree process in a multiprocess execution is typed out as it joins the execution.

The process that is to pick up the work load of the dead process must have as its working directory the directory in which the dos_mp_seg segment for the current multiprocess execution exists.

Entry: do_subtree$abort

The do_subtree$abort entry point brings an immediate halt to a multiprocess execution of the do_subtree. All processes return to command level at once. The process that executes this command must have as its working directory the directory in which the dos_mp_seg segment of the current multiprocess execution exists.

Usage

```
do_subtree$abort
```

Entry: do_subtree$status

The do_subtree$status entry point prints out a large amount of debugging and status information about all processes involved in a multiprocess execution of do_subtree, including the process identifiers and command lines. The process that executes this command must have as its working directory the directory in which the dos_mp_seg of the current multiprocess execution exists.

Name: dump_partition

The dump_partition command displays data from a named disk partition. By default this data appears in octal, four words per line, though other output formats can also be selected. Also see the clear_partition and list_partition commands in this manual.

Usage

dump_partition pvname partname offset {length} {-control_args}

where:

1. pvname
   is the name of the physical volume on which the partition to be dumped exists.

2. partname
   is the name of the partition to be dumped. It must be four characters or less in length.

3. offset
   is the octal offset at which to begin dumping.

4. length
   is the number of words to be dumped. If not supplied, one word is dumped.

5. control_args
   may be chosen from the following:

   -short, -sh
   produces data in short form, similar to dump_segment -short.

   -long, -lg
   produces data in long form, similar to dump_segment -long.

   -bcd
   produces data including the BCD character representation.

   -no_header, -nhe
   suppresses the header.

Notes

Access to phcs_and hphcs_ gates is required.

Usage as an Active Function

[dump_partition pvname partname offset {length}]

As an active function, dump_partition returns the contents of the specified words in octal, separated by spaces, rather than printing them.

excerpt_mst

Name: excerpt_mst

The excerpt_mst command is used to excerpt given segments from a Multics system tape (either a Multics or BOS bootload tape).

Usage

excerpt_mst reel_id {names}

where:

1. reel_id
   is the reel identification number of the tape to be written. The reel identification number, which is site dependent, may be up to 32 characters long. The reel_id may also include a density specification to indicate the density of the tape being written, as in "060341,den=1600".

2. names
   are the names of the specific segments to be extracted. The star convention is allowed. If no namei arguments are given, all of the segments on the tape are extracted. If a given segment has a separate linkage and definitions on the tape, and has been extracted, the separate linkage and definitions are extracted as well. Segments extracted are created in the current working directory. Bit counts are set from the SLT entry on the tape, as opposed to the actual length of the segment on the tape.

Note

A message is printed whenever a segment is extracted. A diagnostic is issued if names are provided that match no segments on the tape.

Name: expand

The expand command substitutes appropriate files for % include statements in ASCII files that are in either PL/I or assembler (ALM) syntax. PL/I syntax is assumed unless the name of the file to be expanded ends in the suffix alm.

Usage

expand path1 {path2...pathN}

where path1, path2...pathN are the relative pathnames of files to be expanded.

Notes

The expand command checks for some PL/I or ALM syntax errors, but only when necessary.

The expand command does not query the user under any circumstances.

If the name of the file to be expanded is of the form id.lang, then the name of the expanded file is id.ex.lang. An include statement such as: % include a; looks for a file called a.incl.lang, according to the user's translator search paths. If lang is alm, then assembler syntax is assumed; otherwise, PL/I syntax is assumed.

Since processing of include files is exactly the same as processing of the original source file (include files may contain % include statements), it is not enough to specify the line number on which an error occurred. The filename and recursion level must also be specified. If more than one consecutive error occurs in the same file at the same recursion level, then a line is typed specifying the filename and recursion level followed by at least one line for each error that occurred.

If there is infinite recursion of include files, the message "Recursion of include files starting with a.incl.pl1 is two levels deep." is returned. This means that a.incl.pl1 contains an include statement such as: % include b; where b.incl.pl1 contains the statement: % include a;.

Name: fix_quota_used

The fix_quota_used command repairs inconsistencies in storage-system quota used for a directory.

Usage

fix_quota_used pathname

where:

1. pathname
   is the pathname of the directory for which quota is to be made consistent.

Notes

The normal use of this command is from the fix_quota_used.ec exec_com, or by the "x repair" operator command. When a quota (segment quota or directory quota) is found inconsistent and corrected, a message is printed. If the correction causes a directory to have greater quota used than allocated, another message is printed.

Access to the hphcs_ gate is required.

Name: generate_mst, gm

The generate_mst command is used to generate a Multics system tape (MST), which can later be "bootloaded" by BOS as the first step in bringing up a Multics system, or to create the BOS tape itself. The procedures that generate the MST must first find the necessary segments to place on the MST and put them there in a manner that can later be read by BOS and the initializing programs themselves. The MST generating procedures find this information by scanning a header segment. This header segment contains names of programs and data bases to be placed on the tape along with other control information about the segments.

There is a set of search rules specifying which directories are to be searched and the order of search when looking for the specified segments. These rules may be contained in a segment, or default rules may be used. The name of the header segment and the optional name of a segment containing the search rules are given as arguments to the generate_mst command. If no search segment is used, only the directory >ldd>hardcore>execution is searched for the programs to be placed on the tape.

The standard MST header used to generate the Multics supervisor is located in the segment:

>ldd>hardcore>info>hardcore.header

The standard MST header used to generate a BOS system tape is located in the segment:

>ldd>bos>info>bos.header

The standard headers contain many examples of valid header syntax. When a header is modified, an example of the modification should first be located elsewhere in the header if possible, since the semantics of the header are quite complicated.

Usage

generate_mst path reel_id {-control_args}

where:

1. path
   is the pathname of the header segment without the header suffix.

2. reel_id
   is the reel identification number of the tape to be written. The reel identification number, which is site dependent, may be up to 32 characters long. The reel_id may also include a density specification to indicate the density of the tape being written, as in "060341,den=1600".

3. control_args
may be chosen from the following:

-directory, -dr
specifies that a search rule segment is provided in the working directory. The name of the search rule segment is path.search, where path is the entryname portion of the pathname given as the first argument to the generate_mst command.

-notape
specifies that no tape is generated. This control argument can be used to check the consistency of the header segment and produce an output listing without actually generating a tape.

-file, -fl
specifies that output is directed to a file in the storage system rather than to a tape. The file name (which may specify a multisegment file) has the same name as the reel_id argument.

-sys_id STR, -sysid STR
sets the system identifier to STR (which may be up to eight characters long). If this control argument is omitted, the first eight characters of the entryname portion of the pathname given as the first argument to the generate_mst command are used by default.

-hold
does not detach the tape when generation is completed. A checker run may then be performed on the same tape without remounting the reel

-vers_id STR, -versid STR
sets the version identifier to STR (which may be up to eight characters long). If this control argument is omitted, the first eight characters of the entryname portion of the pathname given as the first argument to the generate_mst command are used by default.

Notes

The generate_mst command assumes the name of the header segment is path.header, where path is the first argument to the command. The output listing is placed in a segment path.list in the working directory. If the -directory control argument is specified, the command assumes the segment path.search exists in the working directory.

The search file must contain a list of directories to be searched, one directory name per line. A blank line signifies the working directory.

Format of an MST Header

An MST header is an ASCII file (in free format) consisting of keywords followed by optional control arguments. Comments may be placed anywhere in the header except within a keyword name or control argument and are separated from the rest of the text by "/*" and "*/".

There are two levels of keywords, major and minor. The major keywords are listed below:

add_segnames
boot_program
collection
data
delete_name
end
fabricate
fini
first_name
linkage
name
object
text

The fabricate, first_name, name, object, and text keywords are initial keywords and indicate the start of a description of control arguments for a single segment to be placed on the MST. The linkage keyword is only valid if found in a Segment Description List (SDL). The end keyword indicates the end of an SDL. The collection keyword, which cannot occur in an SDL, instructs the generator to write a collection mark (see below) on the MST. The fini keyword, which cannot occur within an SDL, instructs the generator to close out the tape by writing an EOF and dismounting it.

The syntax of the header consists of some number of SDLs occasionally separated by collection keywords and ending with a fini keyword.

Keywords that do not have arguments are followed immediately by semicolons. Those that have arguments are followed immediately by a colon, which is followed by arguments, separated by commas; the arguments end with a semicolon.

The keywords add_segnames, end, fini, and linkage have no arguments; all others have arguments as described below:

Keyword	Meaning of Arguments
add_segnames	no arguments. It causes the segnames defined in an object segment to be added to the list of names for that segment, as if they had appeared in the list following an "object" or a "name" statement. All names that appear as segname definitions in the object segment will be added to the list of names for this segment. This keyword can only be used in the SDL for a bound object segment and must come immediately after the keyword that begins the SDL. It can be usually used to replace the list of names associated with a bound segment.

boot_program — begins the definition of a segment that will be placed in the bootload portion of the MST label. This is used only for BOS tapes. The bootload_program portion of the MST label will be executed when the Initialize/Bootload sequence is executed via the IOM switch or OC command sequence. Only the text section of the program is placed on the tape, and it must be less than 1500 (octal) words long; if shorter, it is padded to 1500 words with NOP instructions. This keyword must appear as the very first keyword in the header file. It is incompatible with the first_name keyword.

collection — a number indicating which collection mark is to be written. This keyword causes a collection mark to be written on the tape containing the collection number that follows the collection keyword. It must appear between segments, not in a segment definition.

data — begins a list of names associated with the segment. This keyword places the entirety of the named segment on the tape, preceded by a preface area containing all the information specified in the SDL. The data keyword is used only for segments that are not Multics standard object segments, such as ASCII files. The linkage keyword cannot be used with the data keyword.

delete_name — is followed by a list of one or more names that are to be deleted from the list of names for the current segment. This keyword is used to remove extra names that were added with the add_segnames statement but that should not appear on the segment; like add_segnames, it can be usually used to replace the list of names associated with a bound segment. It must appear after add_segnames in an SDL.

end — no arguments. This keyword specifies the end of a segment definition. An end keyword must conclude every use of an object, name, first_name, fabricate, or text keyword.

fabricate — a list of names associated with the segment. This keyword fabricates a segment of all zeros and places it on the tape. The attributes for the segment (size, etc.) are derived from the SDL. The linkage keyword cannot be used with the fabricate keyword.

fini — no arguments. This keyword specifies the end of an MST header. Any keywords appearing in the header after the first fini keyword are ignored.

first_name — a name associated with the segment. This keyword indicates that the segment associated with this SDL is the first segment on the tape and is specially processed, i.e., the first 32 decimal words of the segment are overwritten with tape header information when the tape is bootloaded.

linkage

no arguments. This keyword causes the linkage and definitions sections of an object segment to be placed on the tape, following the object segment itself (if the object keyword was used to define it) or the text section (if the name or text keywords were used). The linkage keyword must appear in an object definition between the object, text, or name keyword for the segment and the end keyword. Any minor keywords following a linkage keyword, such as wired and init_seg, are applied to the linkage section rather than to the text section; this can be used to direct the linkage section into a different supervisor-combined linkage segment than would be used by default. The linkage keyword must be specified in order to cause definitions to be included on the tape and copied into the supervisor definitions segment, even if the segment has no linkage section. This is often true for object segments created with create_data_segment. If such an object segment is used by the supervisor, its definitions sections must be placed on the tape by specifying the linkage keyword, even if the segment is started with the object statement, so that the definitions section is included along with the text section.

name — a list of names associated with the segment. This keyword places the named segment on tape preceded by a preface area for the segment containing all of the information specified in the SDL. If the linkage keyword is found in the SDL, the generator splits apart the object segment named and places only the text on the tape. Then, the linkage section by itself (preceded by a preface area for the linkage section) follows the text and definitions section (preceded by its preface) on the tape. Otherwise, the entire object segment is placed on the tape. The name keyword must be used for nonobject segments. For a Multics supervisor tape, the names specified in the header for a segment are the only names by which the segment may be referenced. Extra names on the segment itself are ignored. When adding a new program to an existing bound segment, it is necessary to update the MST header, as well as the bindfile, before adding the name of the new program to the list of names for the bound segment.

object — a list of names associated with the segment. This keyword behaves exactly as the name keyword except that the entire object segment is placed on tape rather than just the text section. It is also followed by the (redundant) linkage and definition sections if the linkage keyword is used.

text — a list of names associated with the segment. This keyword places the text section alone on tape. This keyword is used if only the text part of an object segment is wanted.

The following are "minor" keywords with the meaning of their arguments:

abs_seg — either yes or no. Indicates whether or not to suppress creation of a segment when current length/maximum length is not zero.

access — the SDW access mode for the segment in the supervisor's address space. The list may contain any combination of read, write, execute, and privileged.

acl — an ACL entry placed in the branch of the segment. Only segments placed in the hierarchy (via "path_name") can have ACL entries. The ACL entry consists of a Person_id.Project_id.tag followed by a list of read, execute, and write access rights. The Person_id.Project_id.tag must include all three components. The abbreviated form (i.e., that which omits the tag) accepted by the ACL commands is not acceptable.

bit_count — a number specifying a bit count to be associated with the segment.

cache — either yes or no. Indicates whether or not to override the default encacheability of the segment. If the cache keyword is not given, the following defaults are used: If the per_process keyword is specified as yes, then cache is yes. Otherwise, if the init_seg or temp_seg keywords are specified as yes, or write access is specified under the access keyword, then cache is no. Otherwise, cache is yes.

cur_length — for unpaged segments and segments loaded in collection1, a number specifying the number of words to be allocated to the segment. If this segment is a collection1 segment that is to be made paged, cur_length is its length while unpaged.

delete_at_shutdown — either yes or no. Indicates whether or not to return the pages of the segment to the appropriate free pool at shutdown time.

init_seg — either yes or no. Indicates whether or not to delete the segment at the end of initialization.

link_sect_wired — either yes or no. Indicates whether or not the linkage for the segment is to be combined in the supervisor's wired linkage section even though the segment itself might not be wired.

max_length — for paged segments, a number specifying the number of pages to be allocated to this segment. The greater of max_length (if given) and cur_length (converted to pages) determines the size of the page table and the segment bound.

paged — either yes or no. Indicates whether or not the segment is to be constructed as a paged segment.

path_name — specifies that the segment is to be placed in the hierarchy, the value of the argument is the pathname of the directory in which the segment is placed. This keyword is required for segments in collection 3. If the path_name keyword is specified, all names listed for the segment are added to the version in the hierarchy. If an object segment is to be placed in the hierarchy, it should be defined with the object keyword, so the whole segment will appear rather than just the text section.

per_process — either yes or no. Indicates whether or not to suppress copying of the SDW for this segment at process creation time.

ringbrack — is 1, 2, or 3 numbers (separated by commas) to be interpreted as the ring brackets to be placed in the branch for segments that are to go in the hierarchy. Default ring brackets are (0,0,0). Rules for assigning ring brackets are described in the set_ring_brackets command in the MPM Subsystem Writers' Guide.

sys_id — specifies an external name in this segment identifying a location that will be set to the eight-character system identifier (which can be specified by the -sys_id control argument). This normally appears only for Multics system tapes, and identifies the symbol active_all_rings_data$system_id.

temp_seg — either yes or no. Indicates whether or not to delete the segment at the end of the collection in which it was loaded.

vers_id — specifies an external name in this segment identifying a location that will be set to the eight-character version identifier (which can be specified by the -vers_id control argument). This normally appears only for Multics system tapes, and identifies the symbol active_all_rings_data$version_id.

wired — either yes or no. Indicates whether or not the pages of the segment are to be wired.

The generator works by reading the header segments and performing one of the following:

1. If the word found is an initial keyword, the information about the specified segment (i.e., all information up to the next end keyword) is gathered together and written on the MST followed by the data for the segment itself.

2. If the keyword is collection, a special mark is written on the tape indicating the end of the specified collection.

3. If the keyword is fini, the tape is closed out and dismounted.

For segments that are placed on tape (i.e., segments specified with an initial keyword), the first argument to the initial keyword is the name used when searching for the actual segment to be placed on tape. All subsequent arguments are treated as secondary names and although they are placed on the tape in the preface area for each segment they are not used by the generator.

Hardcore profiling

If hardcore programs are compiled with the -profile or -long_profile options, it is possible to profile the behavior of the supervisor. See the description of the -hardcore control argument to the profile command, in MPM Commands and Active Functions (Order No. AG92).

There are several common pitfalls encountered in hardcore profiling. The size of the supervisor linkage segments must be increased to contain the additional static data generated by the profiling code. The required sizes can be determined from the loading summary information following collection two in the output file from check_mst. The supervisor linkage segments are as_linkage ("Active Supervisor"), ai_linkage ("Active Initialization"), ws_linkage ("Wired Supervisor"), and wi_linkage ("Wired Initialization"). They are defined near the beginning of the standard header. Unless the "init_seg" and "temp_seg" keywords are removed from initialization programs and their linkage sections, it is not possible to profile supervisor initialization programs (because the profiling information would otherwise be discarded as the system finished initialization), but this is rarely a problem.

If wired code is to be profiled, and the -long_profile option is selected, the hcs_ gate and its linkage section must be wired, because they are referenced by the virtual CPU time and paging calculation operators. This is not necessary if only -profile is used.

If profiling a procedure that is specified as wired in the header, but whose linkage section is specified as unwired, it is necessary to change the linkage section to be wired.

Interrupt side code can be meaningfully profiled only with -profile, not with -long_profile, because interrupt code is not run in any particular process, and therefore the virtual CPU time calculation (which is per process) will return random results. This may lead to overflow faults while running on the PRDS. Because -profile does not require these calculations, it may be used with interrupt code.

Name: generate_pnotice

The generate_pnotice command allows Multics source and object archives and executable software to be legally protected via copyright or trade secret notices. It also provides software identification via Software Technical Identifiers (STIs).

Usage

generate_pnotice {control_args}

where:

1. control_args
   can be chosen from the following:

   -name STR, -nm STR
   where STR specifies the product's generic name found in psp_info_.

   -id STR
   where STR specifies the Marketing Identifier (MI) of the product as derived from psp_info_. This argument and the -name argument are mutually exclusive.

   -sti STR
   where STR is a valid 12-character STI. This argument can be used to override the STI found in psp_info_ via use of the -name or -id arguments.

   -special
   this argument is intended for use in cases where there may be no entry in psp_info_ for the software being protected. This is most likely to occur when the user is protecting software in an experimental or development library. The user is prompted for the information to be put into the PNOTICE segments. See "Notes" for further information.

Notes

This command allows protection of software that resides in a library other than the one specified in psp_info_, as well as software not specified at all in psp_info_, via use of the -special control argument.

The command generates ALM source and object segments with the names of "PNOTICE.<generic name>.alm" and "PNOTICE.<generic name>", where <generic name> comes from the psp_info_ data base, or, if -special is used, the user may provide a name.

These segments contain the text of one or more software protection notices and three 12-character STIs. The segments are appended to a product's primary source and object archives, as defined in the psp_info_ data base. If -special is used, the user must provide these archive names.

If PNOTICE segments with the same name exist in the archives, they are replaced. The archives should be ordered by the owner such that these segments are the first components. The binding of the object archive places the protection notices and STIs into the bound segment as well. The bindfile "Order" statement should state that the PNOTICE component is first. The PNOTICE segment name should not be retained in the bound segment.

The information contained in the PNOTICE segments for installed products is available via the display_psp command.

Unless the -special control argument is used, the source and object archives must be in the user's working directory; in which case, the user must have sma access to the directory as well as rw access to the archives. Then, the user can specify archive pathnames to the command. On the other hand, if -special is used, access is checked, and if it is not sufficient, it will be forced; otherwise, access is not forced.

If the user wishes to protect software in a library other than that specified in psp_info_, the -special control argument should be used.

The following set of questions is asked by the command when -special has been specified. The user should have the requested information readily available.

Generic name?
User supplies a short (<= 20 characters) name that is descriptive of the module(s) being protected. The name may be the same as contained in psp_info_ if the module is a newer version; otherwise, the user can create the name.

STI?
The Software Technical Identifier. This is a 12-character identifier used by Honeywell to provide information on released software products. It may be blank for nonproducts. Type help sti.gi" for more information.

Include the notices from psp_info?
The module(s) being protected have an entry in psp_info_. User is asked whether the notices there are to be included.

Source pnotice name?
User is asked to provide primary names of notices, without the ".pnotice" suffix, for protection of source. When done, type "q". Available names can be determined by typing "list_pnotice_names".

Object pnotice name?
User is asked to provide primary names of notices, without the ".pnotice" suffix, for protection of object and executable. When done, type "q". Available names can be determined by typing "list_pnotice_names".

Pathname of source archive?
User is asked to provide an archive pathname of the source archive. The ".archive" suffix is not required, but can be given.

Pathname of object archive?
User is asked to provide an archive pathname of the object archive. The ".archive" suffix is not required, but can be given. These two archives need not reside either in the same directory or in the working directory.

Further information on this command and other commands for the software protection facility can be found in the Multics Library Maintenance PLM (Order No. AN80).

Name: get_ips_mask

The get_ips_mask command prints the current state of the IPS mask for the calling process.

Usage

```
get_ips_mask {-control_args}
```

where:

1. control_args
   can be selected from the following:

   -brief, -bf
   prints nothing if no IPS signals are masked; otherwise, prints the names of masked signals.

   -long, -lg
   prints a more descriptive message about the status of IPS signals, masked or unmasked. (Default)

Notes

If all undefined IPS signals are either masked or unmasked, they are not mentioned. If, however, some are masked and others are not, an octal list will be printed. This can only happen when an invalid (probably reinitialized) value has been supplied in a call to set that mask.

get_library_segment

Name: get_library_segment, gls

The get_library_segment command can be used to find source or object segments in the Multics system libraries and to copy the segments found into the user's current working directory. The user can specify which system libraries are to be searched, and the order in which they are to be searched. There are also provisions for searching user libraries that may or may not be organized like the Multics system libraries. (See "Operation" below.)

Usage

get_library_segment seg_names {-control_args}

where:

1. seg_names
   are the names of the segments to be found, including any language suffix.

2. control_args
   can be chosen from the following list. With the exception of -rename, each control argument in the command line applies to all seg_names.

   -sys lname
   specifies that get_library_segment should use the control segment "lname.control".

   -long, -lg
   prints the pathname of the segment from which each segment is copied.

   -brief, -bf
   does not print pathnames. This is the default.

   -rename new_name, -rn new_name
   copies the immediately preceding seg_name into the user's process directory and then into a segment in the working directory. The new_name may be an equal name, in which case the equal convention is applied to the seg_name; otherwise, the segment created in the working directory will be named new_name. The new_name may not be a pathname.

   -control path, -ct path
   looks in the directory specified by path to find the control segments. The path argument can be -working_directory or -wd, in which case the get_library_segment command looks in the current working directory for its control segments (see "Operation" below). If this control argument is not specified, the get_library_segment command looks in the directory >ldd to find its control segments.

get_library_segment

Notes

If the -sys control argument is not given, then get_library_segment uses all the control segments specified in the root directory. The default root directory is >ldd. For a complete list of the control segments, type:

```
list -pn >ldd -all **.control

hard
standard
unbundled
auth_maint
network
languages
tools
```

Several -sys control arguments can be specified in the same command invocation. If so, all of the control segments referenced by the lnames in these arguments are searched. The order in which the control segments are processed and then searched is determined by the order in which the lnames appear in the command and the order in which the directores referenced by each lname appear in the lname control segment.

Control arguments and segment names can be interspersed throughout the command invocation.

Examples

The command line:

```
get_library_segment abc.pl1 -sys tools -sys sss random.alm
```

copies abc.pl1 and random.alm from the directories specified in >ldd>tools.control and >ldd>sss.control if they exist.

```
get_library_segment -sys lang xyz.pl1 -sys os -sys hard
```

searches for xyz.pl1 in the directories specified by the set of control segments in >ldd.

```
get_library_segment gorp.pl1 -rename glop.pl1
```

searches the default group of directories for segment gorp.pl1 and copies it into the user's working directory with the name glrp.pl1.

```
get_library_segment fortran_blast_ bound_parse_.bind -sys lang.o
```

searches for the segment fortran_blast_ and the bind segment, bound_parse_.bind, in the directories specified in >ldd>lang.o,control.

Operation

If no -control control argument is specified, the get_library_segment command searches for segments in one or more of the Multics system libraries. From each keyword given in a -sys control argument, get_library_segment constructs a pathname of the form >ldd>keyword.control. It uses this as the pathname of a control segment. This control segment tells the get_library_segment command which directories are to be searched and how to search them.

Each control segment contains one or more lines of the form:

```
directory_path:  search_procedure;
```

where:

1. directory_path
   is the absolute pathname of a directory to be searched.

2. search_procedure
   is the name of a procedure that searches the directory to find seg_name. This name can have the form:

   ```
   segment_name
   ```
   or:
   ```
   segment_name$entry_name
   ```

For each directory_path specified in the control segment, the get_library_segment command initiates the search_procedure, and calls it to search the directory. The calling sequence for search_procedure is:

```
declare search_procedure (char(*), char(*), char(*), fixed bin(35));

call search_procedure (directory_path, seg_name, containing_seg, code);
```

where:

1. directory_path (Input)
   is the absolute pathname of a directory to be searched.

2. seg_name (Input)
   is the name of the segment to be found, including any language suffix.

3. containing_seg (Output)
   is the name of the segment in directory_path in which seg_name was found. This name is either the same as seg_name or the name of an archive containing seg_name.

4. code (Output)
   is a standardstorage-system status code. 0 seg_name was found in directory_path>containing_seg 1, seg_name was not found.

Notes

If code is 0, and the final eight nonblank characters of containing_seg are the archive suffix, get_library_segment issues the command:

archive x directory_path>containing_seg seg_name

to extract the segment into the current working directory. If the -rename control argument was specified for seg_name, the segment is extracted and given the new name.

If code is 0 and the final eight nonblank characters of containing_seg are not the archive suffix, the get_library_segment command calls copy_seg_ to copy directory_path>seg_name into the current directory, unless a -rename control argument has been specified, in which case the segment is copied into directory_path>new_name.

If code is 1, the get_library_segment command continues the search with the next directory_path in the current control segment. If the current control segment contains no more directory_paths, the search continues with the first directory_path in the next control segment specified by the user. If the segment has not been found after all control segments have been exhausted, the get_library_segment command prints an error message and begins searching for the next seg_name.

If search_procedure returns a code that is neither 0 nor 1, the get_library_segment command prints the error message corresponding to the error code, and continues the search as if code were 1.

The get_primary_name_ procedure is used to find segments in the Multics system libraries.

If no -sys control argument is specified, the get_library_segment uses all the control segments in >ldd.

User Libraries

The get_library_segment command can be used with the -control control argument to extract segments from a user library. This control argument causes the get_library_segment command to use a control segment with the pathname path>keyword.control. The -control control argument thus allows the user to search his own library structure, using his own search_procedure or one of the Multics system library search procedures listed above.

For example, user Person_id.Project_id can use the get_library_segment command to extract a copy of the source program alpha.pl1 from a private library archive with the command:

```
gls -ct >udd>Project_id>Person_id -sys source alpha.pl1
```

if >udd>Project_id>Person_id>source.control contains the line:

```
>udd>Project_id>Person_id>library:  get_primary_name_;
```

and if alpha.pl1 is a component of some archive segment >udd>Project_id>Person_id>library, having alpha.pl1 as one of its names.

Name: hp_delete_vtoce

The hp_delete_vtoce command deletes a specified VTOC entry. This can be used when cleaning up after a sweep_pv to get rid of orphans, or whenever a forward connection failure is desired.

Usage

hp_delete_vtoce pvname vtoc_index {-control_args}

where:

1. pvname
   is the name of the physical volume on which the VTOCE to be expunged exists.

2. vtoc_index
   is the index (in octal) of the VTOCE to be expunged.

3. control args
   may be chosen from the following:

   -force, -fc
   suppresses the question about really deleting the VTOCE if it is an orphan. If it is not an orphan, the -no_check control argument must also be supplied to suppress all questions.

   -no_check, -nch
   suppresses the check made to see whether the VTOCE is an orphan or not. If it is not an orphan, deleting it will cause a forward connection failure in its parent directory.

   -brief, -bf
   suppresses the message announcing the deletion of the VTOCE, which is only printed if no questions were asked, anyway.

   -query, -qy
   always asks the question about whether to delete the VTOCE, even if it is an orphan.

   -clear
   uses the privileged entry that sets an entire VTOCE to zero, rather than deleting it normally. This should be used only when a VTOCE contains invalid information that might cause problems (reused addresses, crashes, etc.) if it was deleted by the normal means, since it will leave the volume on which the VTOCE existed in an inconsistent (though benign) state to use -clear. The volume in question should be salvaged with the volume salvager after all the seriously inconsistent VTOCEs have been deleted. The -clear control argument should not be used to delete an ordinary orphan, reverse connection failure VTOCE.

hp_delete_vtoce

Notes

This program cannot be used to delete the VTOCE of an active segment. The default action is to check whether the VTOCE is an orphan VTOCE, and delete it if it is, or ask whether to delete it if is not an orphan. The question is suppressed by the -force control argument, and can be forced by using the -query control argument.

Access to phcs_ and hphcs_ is required.

Name: hunt

The hunt command searches a specified subtree of the hierarchy for all occurrences of a named segment that is either free standing or included in an archive file. The segment(s) searched for can be specified by a star name. Any matching segments are reported.

Usage

```
hunt name {path} {-control_args}
```

where:

1. name
   is the name of a segment for which the hunt command is to search. The star convention is allowed.

2. path
   is the pathname of a directory to be interpreted as the root of the subtree in which to search for the specified segment(s). If no path argument is specified, the hunt command searches the subtree rooted at the current working directory.

3. control_args
   can be chosen from the following:

   -all, -a
   reports on finding links and directories as well as segments.

   -first
   stops searching as soon as the first occurrence of the specified segment is found.

   -archive, -ac
   looks inside archives for components whose names match the name argument. This is the default.

   -no_archive, -nac
   suppresses searching of archives for matching components when seeking for an executable segment.

Notes

The hunt command displays the type of entry found (segment, directory, or link) followed by the pathname itself. The total number of occurrences found is displayed at the end of the list.

Usage as an Active Function

```
[hunt name {path} {-control_args}]
```

Notes

All arguments accepted by the hunt command are accepted by the active function.

When invoked as an active function, hunt returns a string of pathnames separated by spaces. Archive components are returned as "archive_path:: component_name".

Name: hunt_dec

The hunt_dec command searches a specified subtree of the hierarchy for all PL/I object segments that are either freestanding or included in an archive file. Each PL/I object segment is classified according to its use of arithmetic decimal instructions and how these instructions access the data. The three classes are "no decimal," "aligned decimal," and "unaligned decimal."

If no control arguments are specified, two ASCII segments are created in the working directory. One segment, aligned_decimal.hd, is a list of the absolute pathnames of PL/I object segments and archive segments containing PL/I object segments classified as "aligned decimal." The absolute pathname of the archive segment is followed by a space then by the name of the component of the archive that was classified as "aligned decimal." This occurs for each component of the archive that is classified as such. Similarly, a segment, unaligned_decimal.hd, is created in the working directory for the class "unaligned decimal." No segment is created for the class "no decimal."

Usage

hunt_dec {path} {-control_args}

where:

1. path
   is the pathname of a directory to be interpreted as the root of the subtree in which to search and classify PL/I object segments. If this argument is not specified, the working directory is assumed.

2. -control_args
   are used to override the defaults given above in the command description and can be selected from the following:

   -aligned_decimal path, -ad path
   specifies that the ASCII segment listing the absolute pathnames of PL/I object segments and archive segments containing component classified as "aligned decimal" will be created with the pathname path suffixed with "hd".

   -unaligned_decimal path, -ud path
   specifies that the ASCII segment listing the absolute pathnames of PL/I object segments and archive segments containing components classified as "unaligned decimal" will be created with the pathname path suffixed with "hd".

Notes

The hunt_dec command is a tool to aid the user when PL/I programs compiled using "unaligned decimal" are to be recompiled using the newer PL/I compiler implementing packed decimal, which was part of Multics Release 8.0. This was an incompatible change because the layout of variables containing both the unaligned and decimal attributes was changed. Therefore, it is necessary for the user to find those PL/I programs that used "unaligned decimal" so that the appropriate program and data base changes can be made before recompiling the program using the new compiler.

The algorithm hunt_dec uses to classify PL/I object segments is simple. The text section is scanned for EIS decimal arithmetic instructions generated by the PL/I compiler. If none are found the object segment is classified as "no decimal." If decimal instructions are found, they and their descriptors are examined for address modification and nonzero digit offsets. If either is present, the object segment is classified as "decimal unaligned"; otherwise, it is classified as "decimal aligned."

The validity of the classification algorithm rests upon knowledge of how the PL/I compiler generates machine code. Below is a table listing the reliability of the algorithm for the different classifications.

CLASSIFICATION	RELIABILITY
aligned decimal	Always correct.

unaligned decimal	Fails when an unaligned decimal variable happens to fall on a word boundary. For example,
	
	`dcl 1 record aligned,`
	`2 item1 fixed bin(17),`
	`2 item2 fixed dec(3) unaligned;`
	
	The variable, item2, is unaligned decimal. But, since it is located one word from the beginning of the structure, the instruction accessing it appears to be accessing aligned decimal data.

no decimal	If fixed decimal variables are present in the source program but are never referenced or do not have the initial attribute, no EIS fixed decimal instructions are generated by the compiler.

The important point to be made is that the hunt_dec command will correctly identify PL/I object segments that use unaligned decimal data most of the time while letting a few segments slip by misclassified as aligned decimal or no decimal.

The hunt_dec command attempts to force access to all segments in its search path. If unable to access a segment for any reason, hunt_dec bypasses the segment without classifying it.

Name: library_descriptor, lds

A library descriptor is a data base that associates directories or archives in the Multics storage system with the roots of a logical library structure. Library descriptors are discussed in detail in Section 2 of Multics Library Maintenance PLM Preliminary Edition (Order No. AN80).

This command prints information about library descriptors on the user's terminal, and controls the use of library descriptors by the other library descriptor commands. It can print the pathname of the directory or archive associated with a library root; can print detailed information about one or more library roots; can set and print the name of the default library descriptor used by the other library descriptor commands; and can print the default library and search names associated with each library descriptor command. The relationship between library_descriptor and the other library descriptor commands is discussed further in the Multics Library Maintenance PLM.

Usage

library_descriptor key {arguments}

where the keys and their arguments are described below.

Key: name, nm

The name key returns the name of the default library descriptor that is currently being used. The library_descriptor command may be invoked as an active function when the name key is used.

Usage

library_descriptor name

Key: set

The set key sets the name of the default library descriptor.

Usage

```
library_descriptor set {desc_name}
```

1. desc_name
   is the pathname or reference name of the new default library descriptor. If a reference name is given, the descriptor is searched for according to the search rules, which are documented in Section 3, "Reference Names," of the MPM Reference Guide (Order No. AG91). If desc_name is omitted, then the default library descriptor is set as the descriptor for the Multics System Libraries.

Key: pathname, pn

The pathname key returns the pathname of the library root(s) that are identified by one or more library names. The library_descriptor command may be invoked as an active function when the pathname key is used.

Usage

```
library_descriptor pathname library_names {-control_args}
```

where:

1. library_names
   are the names of the libraries whose pathnames are to be returned. The Multics star convention may be used to identify a group of libraries. Up to 30 library names may be given.

2. control_args
   are selected from the following list of control arguments and can appear anywhere after the key in the command:

   -descriptor desc_name
   gives the pathname or reference name of the library descriptor defining the library roots whose pathnames are to be returned. If the -descriptor control argument is not specified, then the default library descriptor is used.

   -library library_name, -lb library_name
   identifies a library name that begins with a minus (-) to distinguish the library name from a control argument. There are no other differences between the library names described above and those given with the -library control argument. One or more -library control arguments may be given in the command.

Key: default, dft

The default key prints the default library name(s) and search name(s) associated with one or more of the library descriptor commands.

Usage

```
library_descriptor default {command_names} {-control_arg}
```

where:

1. command_names
   are the names of the library descriptor commands whose default library and search names are to be printed. If no command names are given, the defaults for all of the library descriptor commands are printed.

2. control_arg
   may be the -descriptor control argument as described above. It may appear anywhere after the key in the command.

Key: root, rt

The root key prints detailed information about library roots on the user's terminal. The information includes the names on each library root, its pathname, and its type.

Usage

```
library_descriptor root library_names {-control_args}
```

where:

1. library_names
   identify the library roots about which information is to be printed. The Multics star convention may be used to identify a group of libraries. Up to 30 library names may be given.

2. control_args
   are selected from the following list of control arguments and can appear anywhere after the key in the command:

   -name, -nm
   causes all of the names on each library root to be printed.

   -primary, -pri
   causes the primary name on each library root to be printed.

   -match
   causes all library root names that match any of the library names to be printed. This is the default.

   -descriptor desc_name
   is as above.

   -library library_name, -lb library_name
   is as above.

Name: library_fetch, lf

This command copies entries from a library into the user's working directory. Control arguments allow copying the entries into another directory or renaming them as they are copied; select which library entrynames are placed on the copy; allow copying the library entry that contains a matching entry instead of the matching entry itself (e.g., copying the archive that contains a matching archive component), or copying all of the components of the containing entry. A documentation facility is provided for recording in a file the status of each entry that is copied.

This command uses a library descriptor and library search procedures, as described in the Multics Library Maintenance PLM Preliminary Edition (Order No. AN80). The initial default descriptor describes the Multics System Libraries and allows this command to extract source programs, object segments and bind files, include and info segments, and compilation listings from the System Libraries. This command functionally replaces the get_library_segment command. Refer to the library_descriptor command description in this manual for information about the default library descriptor and the library names defined in the library descriptors.

Usage

library_fetch {search_names} {-control_args}

where:

1. search_names
   are entrynames that identify the library entries to be copied. The Multics star convention may be used to identify a group of entries with a single search name. Up to 100 search names may be given in the command. If none are given, then any default search names specified in the library descriptor are used.

2. control_args
   are selected from the following list of control arguments and can appear anywhere in the command:

   -library library_name,
   -lb library_name
   identifies a library that is to be searched for entries matching the search names. The Multics star convention may be used to identify a group of libraries to be searched. Up to 100 -library control arguments may be given in each command. If none are given, then any default library names specified in the library descriptor are used.

   -name, -nm
   indicates that all of the names on each matching library entry are to be placed on the copy. See the discussion of naming considerations under "Notes" below.

   -primary, -pri
   indicates that the first name of each matching library entry is to be placed on the copy. See the discussion of naming considerations under "Notes" below.

-match
indicates that, for each matching library entry, the entrynames that match any of the search names are to be placed on the copy. See the discussion of naming considerations under "Notes" below. This is the default.

-into path
identifies the directory into which library entries are copied and indicates how they are renamed. An absolute or relative pathname may be given. The directory portion of the pathname identifies the directory into which each library entry is copied. The final entryname of the pathname is used to rename each library entryname being placed on the copy, under control of the Multics equal convention. The -into control argument may appear only once in a command line. If -into is not given, matching entries are copied into the user's working directory and no renaming occurs.

-chase
indicates that the target of a matching library link is to be copied.

-no_chase
indicates that a warning message is to be printed when a matching link is found in the library, and that no copying is to occur. This is the default.

-long, -lg
causes the pathname of each matching entry to be printed on the user's terminal as the entry is copied.

-brief, -bf
suppresses printing of the pathname of matching entries. This is the default.

-container
causes the library entry that contains each matching entry to be copied, instead of the matching entry itself. See the discussion under "Notes" below.

-components
causes all of the component library entries of a matching library entry to be copied, rather than just the matching entry itself. It also causes all components of a library entry containing a matching component to be copied. See the discussion under "Notes" below.

-entry, -et
causes each matching library entry itself to be copied. This is the default.

-search_name search_name
identifies a search name that begins with a minus (-) to distinguish the search name from a control argument. There are no other differences between the search names described above and those given with the -search_name control argument. One or more -search_name control arguments may be given in the command.

-descriptor desc_name
gives a pathname or reference name that identifies the library descriptor describing the libraries to be searched. If no -descriptor control argument is given, then the default library descriptor is used.

-retain, -ret
indicates that library entries that are awaiting deletion from the library (as determined by the library search program) are to be copied.

-omit
indicates that library entries awaiting deletion from the library are to be omitted from the search, and are not to be copied. This is the default.

-output_file file, -of file
indicates that status information for each copied library entry is to be appended to a file. A relative or absolute pathname of the file may be given. If it does not have a suffix of fetch, then one is assumed.

-all, -a
indicates that all available status information for copied library entries is to be recorded in the output file.

-default, -dft
indicates that only default status information is to be recorded in the output file. This is the default.

Notes

Any combination of the control arguments governing naming (-name, -primary, and -match) may be given in the command. However, the following groups of control arguments are mutually exclusive, and only one argument from each group may be given in the command: -chase and -no_chase; -long and -brief; -container, -components, and -entry; -retain and -omit; and -all and -default.

An -all or -default control argument may only be specified when the -output_file control argument is also given. The particular status information recorded in the output file for the -default control argument is under the control of the library search program. It includes the information deemed most important for the type of entry contained in the library.

If the file given in the -output_file control argument does not exist, it is created by library_fetch. If it does exist, new status information is appended to the end of the file preserving any previously recorded status. This feature allows the user to build a history of the entries copied out of a library.

When using the -into control argument, care must be taken to ensure that the equal name included in the -into pathname can be applied to all names to be placed on each of the copied entries. Name duplications can easily result when more than one library entry matches the search names.

The -container and -components control arguments are provided to facilitate copying all of the library entries included in a given bound segment or related to a given subsystem. For example, by identifying a component of the source archive for a bound segment and using the -container control argument, the entire source archive is copied into the user's directory. Similarly, by identifying a directory in the library containing all of the component entries of a subsystem and using the -components control argument, each component is copied into the user's directory.

When the -container, -components, or -chase control arguments are used, it may happen that none of the entrynames on a copied library entry matches any of the search names. Because the user may have requested that only matching names be placed on the copies, the library search program causes the first entryname to be placed on the copy when one of these three control arguments is used, in addition to any names requested by the user.

The user is automatically given re access to object segments that are copied and rw access to all other segments.

Examples

```
library_fetch abbrev.pl1 -into >udd>Multics>user>new_=.=
```

copies the source segment abbrev.pl1 into the directory >udd>Multics>user, renaming the copy new_abbrev.pl1.

```
library_fetch bound_qedx_.** -library online
```

copies all of the segments in the online libraries whose names begin with bound_qedx_ into the user's working directory. This might include the source archive, bindable object archive, bound object segment, and bind listing.

```
lf bound_qedx_.** -library online.source -components
```

copies all of the source components from the source archive for bound_qedx_ into the user's working directory.

```
lf qedx.pl1 -components
```

copies all of the source components in the archive containing qedx.pl1 into the user's working directory.

```
library_fetch *.alm -lb network.source -into new_=.alm
```

copies all ALM source segments from the network source library into the user's working directory, and adds a new_ prefix to the names placed on each segment.

```
library_fetch pl1_status.info -nm -lb info
```

copies the pl1_status.info segment from the info segment libraries into the user's working directory, copying all entrynames from the library entry onto the copy.

```
library_fetch **.ec -library online.??????
```

copies all exec_com segments from the online source and object libraries into the user's working directory.

```
library_fetch -lb supervisor.bc bound_sss_wired_.*
```

copies the bind segment from the bindable object archive called bound_sss_wired_.archive. Note that although the object archive itself matches the search name that was given, only the matching archive component is copied because the -container control argument was not given.

```
library_fetch -lb include stack_frame.incl.*
```

copies the stack frame declaration include segments for all source languages from the include library into the user's working directory.

list_dir_info

Name: list_dir_info

The list_dir_info command lists the contents of a directory information segment created by the save_dir_info command.

Usage

list_dir_info path {-control_arg}

where:

1. path
   is the pathname of the directory information segment. If path does not end in the suffix dir_info, it is assumed.

2. control_arg
   can be chosen from the following:

   -long, -lg
   produces a long form of output. All items are listed.

   -brief, -bf
   produces a short form of output.

Notes

If neither the -long nor -brief control argument is selected, an intermediate verbosity level is used.

The output of this command is written on the user_output I/O switch.

For each entry, a series of lines of the form:

item_name: value

is written. Entries are separated by a blank line.

See the description of the list_dir_info_ subroutine for information on the items printed for each verbosity level.

Name: list_mst

The list_mst command is used to find out what segments are on a Multics system tape (either a Multics or BOS bootload tape).

Usage

list_mst reel_id {names}

where:

1. reel_id
   is the reel identification number of the tape to be written. The reel identification number, which is site dependent, may be up to 32 characters long. The reel_id may also include a density specification to indicate the density of the tape being written, as in "060341,den=1600".

2. names
   are names of segments to be listed. The star convention is allowed. If no namei arguments are given, all of the segments on the tape are listed.

Note

A summary line, giving the length of each segment and its primary name, is printed out for each segment listed. A special name is printed out for segments written with the first_name keyword of the generate_mst command because the proper name of such segments does not appear on the tape. A diagnostic is issued if names are provided that match no segments on the tape.

Name: list_partitions

The list_partitions command lists the locations and sizes of all the partitions on a specified physical volume. Also see the clear_partition and dump_partition commands in this manual.

Usage

```
list_partitions pvname
```

where:

1. pvname
   is the name of the physical volume whose partitions are to be listed.

Output Format

The output consists of a header, which lists the physical and logical volume names, the PVID and LVID, the size of the volume in pages, the size of the VTOC in both pages and VTOCEs, the size of the paging region, and the number of partitions. It is followed by a table listing the name, first record, and size of all partitions and other regions on the volume. All numbers in the table are given in both decimal and octal (in parentheses), and all other numbers in the output are decimal.

Example

```
Volume root2 (740651611731) of logical volume root (225072707470):
 38258. total records. 2000. VTOC records, for 1000. VTOCEs.

Volume map (including 4 partitions):
 Name            First record          Size
Volume header       0. (0)              5. (5)
VTOC area           5. (5)           2000. (3720)
  BOS            2005. (3725)         200. (310)
  DUMP           2205. (4235)        2000. (3720)
Paging region    4205. (10155)       34712. (103630)
  HC            36917. (110065)       1200. (2260)
  ALT           38117. (112345)        141. (215)
```

list_pnotice_names

Name: list_pnotice_names

The list_pnotice_names command displays the primary names of all protection notice templates.

Usage

list_pnotice_names {-control_arg}

where:

1. control_args
   can be chosen from the following:

   -check, -ck
   specifies that the entire pnotice search list is to be processed and that all templates, including duplicates, are to be listed. Checks are also made to the text of each template, and any errors encountered are reported.

   -all, -a
   specifies that the entire pnotice search list is to be processed and all templates, including duplicates, are to be listed.

Notes

Default copyright and trade secret notices are indicated. Names displayed by this command are shown as they should be input to the add_pnotice and generate_pnotice commands. If no control arguments are used, names are output in search list order, omitting duplicates.

Name: list_sub_tree, lst

The list_sub_tree command lists the segments in a specified subtree of the hierarchy. The complete subtree is listed unless the -depth control argument is specified.

Usage

list_sub_tree {pathnames} {-control_args}

where:

1. pathname
   is the relative pathname of the subtree to be searched. If more than one pathname is specified, only the last one is listed. If no pathname is given, then the working directory is assumed.

2. control_args
   can be selected from the following:

   -all, -a
   specifies that all the names of a segment will be printed. The default is to print only the primary names.

   -depth NNN, -dh NNN
   specifies the depth to which the hierarchy is to be scanned. The depth is relative to the base of the specified subtree. The depth must be specified by a decimal integer.

Notes

For each level in the hierarchy listed, the names are indented three more spaces to indicate which segments exist at which depth in the hierarchy.

Each segment printed includes a number indicating the number of records used by the segment.

Name: mcs_version

The mcs_version command prints on the user's terminal the version of the core image most recently loaded into the specified FNP. This command can also be used as an active function.

Usage

mcs_version {fnp_tag}

where fnp_tag is the identifier of the FNP whose version is to be printed. It may be a, b, c, d, e, f, g, or h. If it is omitted, a is assumed.

Name: merge_mst

The merge_mst command is used to copy a Multics system tape (MST) (either a Multics or BOS bootload tape) onto another reel of tape, replacing selected segments with segments from the storage system.

Usage

merge_mst reel_id1 reel_id2 {-control_arg} {names}

where:

1. reel_id1
   is the reel identification number of the tape to be written. The reel identification number, which is site dependent, may be up to 32 characters long. The reel_id may also include a density specification to indicate the density of the tape being written, as in "060341,den=1600".

2. reel_id2
   is the reel identifier number of the new MST to be made.

3. control_arg
   may be either -stop name (or -sp name) to call debug before the segment identified by name is written out whether or not it has been replaced. This allows the user to inspect or modify the SLT entry in any arbitrary way. Before such a call is made, a message is printed, giving the address of the segment and the SLT entry. The -stop control argument can be given more than once in the command line and applies only to the segment name argument immediately following it. The segment name argument to the -stop control argument is operated on as described under names in the following paragraph.

4. names
   are names of segments to be copied. The star convention is allowed. Any segment on the tape that matches any of the names is sought in the current working directory and replaced in the tape copy. If no namei arguments are given, replacements are sought for all of the segments on the tape in the current working directory. If a segment with a separate linkage and definitions is replaced, separate linkage and definitions are sought in the working directory to replace it. However, if there is none, the linkage and definitions are obtained from the object segment in the working directory. If a segment on the tape being replaced is not an object segment, but the matching segment in the working directory is, only the text of the segment is written to the new tape. A separate linkage or definitions section cannot be replaced without replacing the segment from whence it was separated.

Notes

A message is printed out each time a segment is replaced on the tape with one from the working directory. A diagnostic is issued if names are provided that match no segment on the tape or match segments on the tape but match no segments in the working directory.

Segments written with the generate_mst command first_name keyword cannot be replaced because the names do not appear on the tape.

Name: mexp

This command is obsolete and should not be used in new programs. It has been replaced by the ALM macro facility.

The mexp command is a fairly simple text-manipulative program to be used in conjunction with the ALM assembler. The program takes mexp source segments, expands any macros found therein, and generates as output an expanded text segment suitable as input to the ALM assembler.

The mexp command is purely text manipulative and does not have the capability for doing any expand-time decision making other than comparison of character strings. Conditional expansion of code is possible with the use of the pseudo-operations ine, ife, and ifarg. In addition, the ability to generate unique symbols within macros is provided. A limited form of interaction is also provided that allows for repetitive expansion of macro components.

Usage

```
mexp name args
```

where:

1. name
   is the input text segment name. The mexp command searches for name.mexp (unless name ends in the suffix mexp) and generates as output name.alm.

2. args
   can be any character strings that can be embedded in expanded macros with the use of the &An control expansion (see below).

Notes

The format of a mexp source program is quite similar to an ALM source program. The main difference is that macro-definition and macro-expansion statements are interspersed with the normal ALM statements. To define a macro, the pseudo-operation &macro is used. The format of this is as follows:

```
&macro     macro_name
-
- macro-body
-
&end
```

If the string &macro is found in the context of an ALM opcode or pseudo-operation, it is interpreted as the start of a macro definition.

The name of the macro is the next "word" on the line. The body of the macro is all of the text up to but not including the next &end found in the source text. The body of the macro can include any text that, when expanded by the rules specified below, yields valid ALM source code.

Macros are used by specifying the name as if it were an opcode or pseudo-operation and specifying the arguments, separated by commas, in the variable field. A comment field can follow the parameter list separated from it by a quote (") or white space.

The following control sequences direct the macro expander to act in a special way:

1. &0, &1, &2, ...
   the character & followed immediately by any decimal integer (< 100) is replaced, upon expansion, with the corresponding argument passed to the macro (see "Examples" below.)

2. &u
   is expanded to be a unique character string of the form ...00000, ...00001, etc. that is different from any other such strings expanded with &u control.

3. &p
   is expanded to be the same string as the previous &u expansion.

4. &n
   is expanded to be the same string as the next &u expansion.

5. &U
   is expanded to be a unique character string of the form .._00000, .._00001; however, multiple occurrences of &U within the same macro yields the same string.

6. &(n
   indicates the beginning of an iteration sequence. The text following the &(n and up to but not including the next &) is expanded at expand time only if there are additional parameters to the macro iteration argument that have not been used up (see below).

7. ife (ine)
   if ife or ine occur in the context of an opcode or pseudo-operation, it causes conditional expansion of the text up to the next ifend found in the text, depending on the equality (inequality) of the first two parameters to the pseudo-operation. The equality comparison is strictly a character-string compare.

8. dup
   causes the text up to the next dupend found in the text to be duplicated n times where n is the decimal value of the (first) parameter to the pseudo-operation.

9. &i
   is expanded to be the particular parameter in an iterated list for which the current iteration expansion is being done (see below).

10. &x
    is expanded into the decimal integer corresponding to the argument position of the iteration argument for which the current iteration is being done (see "Examples" below).

11. &An
    is expanded to be the n+1'st argument to the mexp command.

12. ifarg
    if ifarg occurs in the context of an opcode or pseudo-operation, it causes conditional expansion of the text up to the next ifend depending on whether or not the first parameter to the pseudo-operation is one of the arguments to the mexp command (other than the source name).

If a parameter is not specified for a particular parameter position, a zero-length string is used for expansion.

The argument &0 expands to be the first label on the statement involving a macro.

Any parentheses around a parameter are stripped off upon expansion. Parentheses used in this manner are treated as quoting characters.

Blanks cannot appear in a macro parameter list unless within a parenthesized parameter.

Iteration

The iteration feature is invoked by passing a parenthesized list of parameters in the parameter position for the specified iteration. The parameter number for an iteration sequence immediately follows the &(of its definition. (If no parameter number is specified, 1 is assumed.) Iterated arguments are scanned in the same manner as macro arguments, and hence quoting can be done with the use of parentheses.

If more than one &i occurs within a single iteration bound, the same parameter is substituted for the &1 throughout the expansion. That is, the parameter number specifying which parameter is to replace the &i is only changed when the &) to end the iteration is reached.

External Macros

The pseudo-operation &include can be used to define macros from an external segment. When this is done, the parameter to the pseudo-operation is treated as a mexp include file of macro definitions. The file name.incl.mexp (where name is the parameter to the pseudo-operation) is searched for, using the include file search rules. The macros contained in the specified segment are defined in the same way as though the macro definitions were in the text directly. (The same rules of requiring a macro to be defined before it is used apply.)

A macro can be redefined with no ill effect. The latest definition is the one used.

Recursion

Macros can be used recursively with the following restrictions:

1. A macro must be defined before it is expanded. It can be used previously in another macro definition as long as the other macro is not expanded (i.e., the name of the macro occurs in the pseudo-operation position of some line).

2. A maximum allowed recursion depth of 32 is arbitrarily imposed.

Continuation

If all of the parameters to be passed to a macro do not fit on one line, they can be continued on the next line. This is indicated by leaving a comma (,) as the last character in a parameter list. No opcode or pseudo-operation should be specified for subsequent continued lines. It is not possible to split a single parameter (which means a parameter that is a list) in this way.

Examples

The following macro definitions show typical expansions.

```
&macro     load
ld&1       &2
&end
```

might be used as follows:

```
load x0,temp ldx0 temp
```

or:

```
load a,(sp|3,*) lda sp|3,*
```

The use of parentheses in the second example causes the comma to be ignored as a parameter delimiter.

```
           &macro     test
&U:        lda        &1
           tnz        &U
           sta        &2
           &end
```

might be used as follows:

```
test       a,b           ..0000: lda     a
                                 tnz     ..0000
test       c,d           ..0001: lda     c
                                 sta     d
```

The following example shows how iteration is used. The macro definition:

```
            &macro      table
&(1         vfd         18/&i,18/&0
&)
            &end
```

might be used as follows:

```
e1:         table       (4,6,8,10)          vfd         18/4,18/e1
                                            vfd         18/6,18/e1
                                            vfd         18/8,18/e1
                                            vfd         18/10,18/e1
```

The following example shows how conditional expansion can be used. The macro definition:

```
&macro      meter
lda         &1
ife         &2,on
aos         meterword,al
ifend
&end
```

might be used as follows:

```
meter       foo,on                  lda     foo
                                    aos     meterword,al
```

The following macro shows how &x might be used. The macro definition:

```
            &macro      callm
&(3         eppbp       &i
            spribp      &2+&x*2
&)
            eaq         2*&x-2
            lls         36
            staq        &2
            call        &1(&2)
            &end
```

might be used as follows:

```
callm           sys,arg,(=1,(=20aError from device
^d),did)
```

yielding:

```
eppbp       =1
spribp      arg+1*2
eppbp       =20aError from device ^d
spribp      arg+2*2
eppbp       did
spribp      arg+3*2

eaq         2*4-2
ll          36
staq        arg
call        sys(arg)
```

The following example shows how conditional expansion might be used. The macro definition:

```
          &macro    tab9
&(        ife       &x,1
          vfd       o9/&iifend
          ine       &x,1
,o9/&iifned
&)
          &end
```

might be used as follows:

```
tab9 (61,62,63,64,65,66)
```

yielding:

```
vfd o9/61,o9/62,o9/63,o9/64,o9/65,o9/66
```

Notice the position of the ifend and &) sequences.

Name: monitor_log

The monitor_log command monitors activity in standard-format log segments. Logs segments are periodically examined, and any new messages are printed on the user's terminal or given as arguments to a specified command line.

Usage

monitor_log {log_name} {-control_args}

where:

1. log_name
   identifies a log segment. If the log is already being monitored, the entryname alone may be used if there is no other log or that entryname being monitored. A log_name may not be specified with -all or -number.

2. control_args
   can be selected from the following:

   -all, -a
   applies the other control arguments to all logs currently being monitored. This control argument may not be given with -number or a log_name.

   -number N, -nb N
   where N identifies a log by its number. This control argument may not be given if -all or a log_name is given. Log numbers are displayed with the -print control argument.

   -print, -pr
   prints the current state of the selected log(s). The log number, wakeup interval, current-log message number, and any called command are printed.

   -off
   removes monitoring from the specified log(s).

   -time N, -tm N
   where N specifies the interval in seconds between examinations of the active logs. Each time the logs are examined, any new entries are processed.

   -call STR
   where STR specifies that the new log entries in the specified log(s) are to be given as arguments to a command line instead of being printed on the terminal. If STR is a null string (""), messages are printed on the terminal instead.

   -match STR
   selects only messages from the specified log(s) containing the string STR for processing. This control argument may be given more than once. If multiple matches are supplied, all messages matching all of the STRs are printed.

-exclude STR, -ex STR
where STR specifies messages to be excluded from processing. This control argument may be given more than once. If multiple excludes are given, all messages matching any of the STRs are excluded.

-remove_match, -rm_match
removes all match specifications from the specified log(s).

-remove_exclude, -rm_ex
removes all exclude specifications from the specified log(s).

-severity N, -sv N
processes only messages with severity greater than or equal to N.

-no_severity, -no_sv
processes all messages regardless of their severity.

Note

Read access is required to the log(s) being monitored.

Name: monitor_quota

The monitor_quota command calculates storage of a directory and will send a warning message at the approach of a record quota overflow condition.

Usage

monitor_quota {-control_arguments}

where:

1. control_args
   can be chosen from the following:

   -pathname, -pn
   is the pathname of the directory to be monitored. Only one path can be given when invoked. The default is the user's working directory.

   -call STR {N}
   specifies that STR is to be passed to the command processor as a command when a directory's segment quota used is found to be greater then 90 percent of the quota assigned. If {N} is given, then the default of 90 percent will be overridden. (See "Notes" below.)

   -console {N}
   sends a warning of an approaching record quota overflow condition to the system console. Access to the phcs_ gate is required to issue warnings on the system console. If {N} is specified, then the default percent value at which the warning is to be issued (as given in the functional description) will be overridden.

   -warn Person_id.Project_id {Person_id.Project_id...} {N}
   sends the warning message to the individual specified in Person_id.Project_id. A limit of ten users can be listed with the use of this control argument. The invoking user will be sent a message by default if this control argument and -console is omitted. If {N} is specified, then the default percent value at which the warning is to be issued (as given in the functional description) will be overridden.

   -repeat DT, -rpt DT
   identifies the interval for setting the monitor time. This argument overrides the default time calculation. The DT is a relative time >= 1 minute and acceptable to convert_date_to_binary_ (e.g. 10min, 1hr).

   -off
   turns off completely all monitoring in the current process. This control argument must not be given with any other arguments.

Notes

This command can be used several times in a process to monitor several different directories. Use of the -off control argument stops monitoring of all directories.

The number of records given with the -call, -console, and -warn {N} control arguments must be less than the quota assigned to the directory.

The default interval when invoked without the -repeat control argument will automatically be set with a time interval dependent on how much available storage was found. That is, if the directory was 50 percent full, then an alarm would be set to trigger in 30 minutes to check again. If the quota was found to be at 80 percent, then a message would be sent, and an alarm time of two minutes would be set. At 90 percent, it would send a warning every minute, and if -call has been provided, then the specified string will be passed to the command processor.

Name: ol_dump

The ol_dump command looks at selected parts of an online dump created by the BOS FDUMP command and copied into the Multics hierarchy by the copy_dump command. The command is designed to aid system programmers in the task of crash analysis.

Usage

ol_dump {erfno} {-control_arg}

where:

1. erfno
   is an error report form number given in decimal, or "last" if the latest dump taken is to be selected. If erfno is not specified, ol_dump enters its request loop described below. If an erfno is given, ol_dump searches its currently referenced dump directory (see below) for a copy of the dump: if it finds the dump, it initializes itself to be able to process the given dump; if it doesn't find it, the user is told and the request loop is entered.

2. control_arg
   can be:

   -pathname path, -pn path
   specifies a directory pathname where online dumps are to be found. If it is not given, the default dump directory of >dumps will be used.

Request Loop

Once ol_dump has processed the erfno argument, it enters a loop-reading requests from user_input. The requests allow the user to look at selected regions of the dump currently under analysis or to choose another dump (erfno) for analysis. The following requests are implemented (letters in parentheses are abbreviations):

erf arg
selects another dump for immediate analysis; arg can be either an erf number or "last" if the latest dump taken is to be analyzed.

quit (q)
returns.

command (c) or (..)
passes the rest of the request line onto the current command processor.

list (l)
lists the dumps in the current dump directory by showing the name of the first component of the dump. The names of dumps tell when the dump was taken and what the erfno is.

help (?)
lists the requests of ol_dump.

dump (d) {arg1 arg2 arg3 arg4 arg5}
displays selected words located in the current dump under analysis. Arguments are as follows:

arg1 must be one of the following:

seg s displays selected words from segment "s" in the current process, where "s" can be a segment number or name. If no other arguments are specified, then the entire segment is dumped in octal.

mem displays selected words starting at absolute memory location indicated by arg2. A search is made of all running process's descriptor segments and AST/PT entries. If the requested address is found, the segment number and name, segment offset, and the process DBR value is output as well as the requested number of words. If the requested memory address is not found, it is assumed to be in free store.

arg2 segment offset (if arg1 is seg s) or the absolute memory address (if arg1 is mem).

arg3 if the first character of arg3 is a "+" or "-", then the rest of arg3 is either added or subtracted (as an octal number) from the base of arg2. If the first character of arg3 is not a "+" or "-", then arg3 is the number of elements to be dumped.

arg4 if the "+" or "-" option of arg3 is present, then arg4 is the number of elements to be dumped. If the "+" or "-" option was not used with arg3, then arg4 is any of the output modes used with the debug command ("o" for octal, "a" for ASCII, "p" for pointer, "i" for instruction format, etc.). If the instruction mode ("i") is used, and if the requested segment is not found in the dump (only segments with read and write access are found in the dump, which fact usually precludes executable object segments from being dumped), then a search of the library directories is made. If the segment is then found, the segment is dumped in instruction format.

arg5 if present, arg5 is used to specify the output mode, as above.

dbr arg {n}
switches to another process (in the same dump). Arguments are as follows:

cpu switches to process that is executing on CPU n. n can be either the cpu number (0 to 7) or cpu tag (a to h).

value switches to another process by specifying the dbr value for the new process.

name (n) segno {offset}
displays the SLT or SST name.

segno is a segment number.

offset displays the bound segment name as well as the component name and the relative offset in that component (if the specified segment is bound).

amsdw (ams) {prds}
displays the saved contents of the SDW associative memory. If the optional prds argument is present, the saved SDW associative memory in the prds is displayed. If the prds argument is not present, then the saved contents of the SDW associative memory at the time of the dump in the bootload CPU is displayed.

amptw (amp) {prds}
displays the saved contents of the PTW associative memory. If the optional prds argument is present, the saved PTW associative memory in the prds is displayed. If the prds argument is not present, then the saved contents of the PTW associative memory at the time of the dump in the bootload CPU is displayed.

syserdta (sdta)
displays the message entries in the wired message segment "syserr_data".

syserlog (slog) n
displays the specified number of message entries in the paged message segment "syserr_log" starting with the most recent entry.

proc (p) arg
displays some APT data for the process specified. Arguments are as follows:

all displays all of the APT entries.

cur displays only the APTE for the current process (as defined by the dbr value).

run displays only those APTEs that are currently executing on the configured CPUs.

rdy displays only those APTEs whose execution state is "ready."

wat displays only those APTEs whose execution state is "waiting."

blk displays only those APTEs whose execution state is "blocked."

stp displays only those APTEs whose execution state is "stopped."

emp displays only those APTEs whose execution state is "empty."

n displays APTE whose number is n.

stack (s) seg {os args lg fwd}
displays a stack trace of the stack segment specified by seg.

seg is either a segment name or number, or the key word "ring"; in which case, the next arg would be the ring number of the stack to be traced (i.e. stack ring 0).

os starts trace at the frame whose offset is os and continues to the end of the stack. If the offset argument is omitted, then the trace is started at the stack base. The segment name of the return pointer is displayed for all segments. If the name is a bound segment, the component name as well as the relative offset is displayed in the form "bound_seg$comp_name|offset". If the return pointer indicates "pl1_operators", then Pointer Register 0 is picked up and used instead. This is indicated by the flag "[pr0]" being displayed after the segment name.

args displays the stack frame arguments in interpreted format.

lg produces an octal dump of each stack frame, as well as interpreting the arguments as above.

fwd starts the trace at the beginning of the stack (as defined by the stack begin_pointer) and continues to the end of the stack (as defined by the stack_end pointer).

mcprds (mcpr) arg {lg}
displays the PRDS machine conditions for the specified argument. Only an interpreted version of the SCU data is displayed, unless the "lg" argument is used.

arg can be one of the following:

int displays machine conditions for prds|interrupt data.

systroub displays machine conditions for prds|system trouble data.

fim displays machine conditions for prds|fim data.

all displays all machine condition save areas in PRDS.

lg displays pointer registers and processor registers as well as SCU data.

mcpds (mcp) arg {lg}
displays the PDS machine conditions for the specified argument. Only an interpreted version of the SCU data is displayed unless the "lg" argument is used.

arg can be one of the following:

pgflt displays machine conditions for pds|page fault data.

fim displays machine conditions for pds|fim data.

sig displays machine conditions for pds|signal data.

all displays all machine condition save areas in PDS.

lg displays pointer registers and processor registers as well as SCU data.

mc arg1 arg2 {arg3 lg}
displays machine conditions from anywhere. Only the SCU data is displayed unless the "lg" argument is used. Arguments are as follows:

arg1 segment name/number or "cond", to display machine conditions from a condition frame specified in the following arguments.

arg2 segment offset, if arg1 is a segment name/number, or segment name/number, if arg1 is equal to "cond".

arg3 if arg1 is equal to "cond", then arg3 is the segment offset of the condition frame. If arg3 is not specified, and arg1 is equal to "cond", then the entire stack segment (from arg2) is searched for a condition frame. In this case the first condition frame found (starting from the stack_end_ptr and working toward the stack_begin_ptr) is displayed.

lg displays the pointer registers and processor registers as well as the SCU data.

dumpregs (dregs) {arg}
displays the processor registers that were saved at the time of the dump, from the bootload CPU. If no arguments are given, all of the registers are displayed. The optional arguments are as follows:

ptr displays the pointer registers only.

preg displays the processor registers only.

scu displays the saved SCU data only.

all displays all of the above.

lrn {segno1 segno2}
displays a breakout of the descriptor segment (dseg) by printing the SDW's segment numbers and names for specified segment numbers of dseg. If no optional arguments are given, the descriptor segment is displayed from segment number 0 to the last segment in dseg.

segno1 segment number at which display starts.

segno2 segment number at which display stops. If this argument is not given, but segno1 is, display continues to the end of dseg.

segno (segn) name
displays the segment number for a given entry name.

ssd arg {paths}
allows the user to specify up to three directories for finding offsets and bindmaps for hardcore segments. The default directory is >ldd>hardcore>execution.

arg can be chosen from the following:

pr displays the current directories searched.

def resets the directories searched to the default value.

paths pathnames of directories to search (maximum of three). If no arg argument is given, at least one path argument must be given. When more than one path argument is specified, the directories are searched in the order specified.

hisregs (hregs) arg1 {arg2 arg3}
displays a composite analysis of the processor history registers. Arguments are as follows:

arg1 can be chosen from the following:

pds displays the stored history registers from the PDS.

dmp displays the history registers stored at the time of the dump by the bootload processor.

help displays both a list of the abbreviations used in the history register analysis and their meaning.

seg can a segment name or number.

cond displays history registers from a condition frame, the location of which is described by arg2 and arg3.

arg2 if arg1 is "seg" then arg2 describes the segment offset to the beginning of the history register area. If arg1 is "cond" then arg2 defines the segment name or number for the desired history registers.

arg3 if arg1 is "cond", then arg3 describes the segment offset to the start of a condition frame. If arg3 is not present and arg1 is "cond", then the entire stack segment (specified by arg2) is searched for a condition frame.

pcd {arg}
displays the contents of the "config_deck" segment in an interrupted fashion. Arguments can be any one of the card types found in the configuration deck (cpu, mem, prph, etc.). The pcd command will process from one to 32 arguments. If no arguments are given, the entire config deck is displayed.

ast (pt) name
displays the AST entry and page table for the given segment. Name can be an segment name or number.

queue (tcq)
displays the scheduler's priority queue in order of priority.

dumpdir path
sets the dump directory to that specified by path. If the request line is none of the above, an error message is displayed, and the request loop is reentered.

absadr segment {offset}
provides an absolute address for the given segment. Segment can be a segment name or a number. Offset is an octal offset from the base of the segment.

erf?
displays the current dump number and date dumped for the dump being processed.

dump_events (de) {args}
displays "interesting" events from an online dump in reverse chronological order. Use of this request will not change any of the current parameters used by ol_dump. Arguments are as follows:

-erf <n>
selects an online dump. If this argument is not provided, the current dump is processed.

-dump_dir path, -dd path
specifies a directory where the online dump can be found. If this argument is not provided, the current directory is used.

-last <n>, -lt <n>
specifies the number of events to print. The default is to print all.

-time <sec>, -tm <sec>
specifies the time in seconds before the dump was taken when events were "interesting." The default is 10 seconds.

-brief, -bf
specifies the brief output format, which is one line per event.

-long, -lg
specifies long output format, which is multiple lines per event.

why
The why request attempts to tell why the system crashed. In some cases, the reason given may be the actual cause of the crash, most often not. It chases down the cause of the crash to:

1. A fatal (code 1) call to syserr. The crash message will be printed.

2. A manual (execute switches) return to BOS by the operator. The BOS machine conditions, including the execution point at which the switches were forcibly executed, are displayed.

3. A fault taken under invalid circumstances; e.g., a parity fault in interrupt code, an illegal fault, or an execute fault. The fault machine conditions are displayed.

4. A deliberate call to BOS, perhaps by a dump taken after successful shutdown, or a call to hphcs_$call_bos, perhaps by the Initializer "bos" command. A message indicating that this is the case is printed.

The why request searches through the dump, following BOS pointers and sys_trouble_data in as many processes as necessary, to determine which process initiated the return to BOS. This request leaves ol_dump in that process.

If the why request cannot determine the crash cause of a dump, it says so. This can happen if a dump is in sufficiently bad shape or if the reason of the system's crash is obscure. In this case, manual checking of prds$sys_trouble_data, pds$fim_data, and pds$signal_data (with the "mc" or "mcpr" requests) in all processes dumped is a good place to start.

After the why request has run, tracing of the stack at the crash point (via "stack" with no arguments) is a good place to go. Specifically, when crawlouts or process terminations of the Initializer were the reason for a crash, a fim-frame is often found in the crash process. Investigation of such a frame with the "mc" request often produces more insight into the reason for system failure.

pdstrace {N}
formats and displays the contents of the system trace table in the pds. The number of trace entries displayed can be specified by N (where N is a positive decimal integer). If N is not specified, all entries in the trace are displayed.

If the request line is none of the above, an error message is displayed, and the request loop is reentered.

Name: nothing, nt

The nothing command performs a return to its caller and does nothing, thereby allowing timing tests to be made at command level.

Usage

nothing {args}

where:

1. args
   are optional arguments that may have any value and are ignored.

Notes

This command makes use of a special feature in the Multics Linking Mechanism that allows it to be executed by any reference name. Thus, it can be used as a "do nothing" substitute for the routine normally known by that name. To do this, initiate it with the reference name of the program it is supposed to replace. It cannot be used in this fashion if the entry-point name is different from the reference name.

Name: pause

The pause command is an interface to the timer_manager_$sleep entry point that allows the caller to "sleep" for a given number of seconds. (The timer_manager_ subroutine is described in the MPM Subsystem Writers' Guide, Order No AK92.)

Usage

```
pause {time}
```

where time is the number of seconds (decimal integer) to sleep. If time is not specified, a time of 10 seconds is used.

perprocess_static_sw_off

Name: perprocess_static_sw_off

The perprocess_static_sw_off command turns off an object segment's per-process static switch, so that the segment's internal static storage will be reset within a run unit.

Usage

perprocess_static_sw_off path

where:

1. path
   is the pathname of a segment whose per-process static switch is to be turned off.

perprocess_static_sw_on

Name: perprocess_static_sw_on

The perprocess_static_sw_on command turns on an object segment's per-process static switch. This switch should be on in all segments whose internal static storage is not to be reset within a run unit.

Usage

perprocess_static_sw_on path

where:

1. path
   is the pathname of a segment whose per-process static switch is to be turned on.

Notes

This switch is a property of the segment itself, not the branch, so it must be set again after recompilation.

If the segment is a bound segment, the bindfile keyword Perprocess_Static should be used instead of the command.

Name: peruse_crossref, pcref

The peruse_crossref command displays information extracted from the output file generated by the cross_reference command.

Usage

pcref {cref_path} search_name{s} {-control_args}

where:

1. cref_path
   is the pathname of the crossref output file to search. It may be an MSF. It must contain a ">" or "<" character in order to distinguish it from a search_name. If no cref_path is supplied, the total system cross-reference (>ldd>crossref>total.crossref) is used. To specify a cross-reference in the working directory, use -pathname.

2. search_name{s}
   are one or more names to search for references to in the crossref. They can be either symbolic linker references or include file names. They can have any of the following forms:

   ```
   segname
   segname$entryname
   XXX.incl.YYY
   ```

   Any component of a search name can be a starname, with the exceptions that neither a segname nor an include file name can begin with a starname character, and the string ".incl" must appear in toto. If no entryname is specified with the segname, all references to any entry points in the segment are listed. XXX.incl is accepted as an abbreviation for XXX.incl.*. The characters ">" and "<" cannot appear in a search name.

3. control_args
   can be one of the following:

   -brief, -bf
   do not print any information for selected cross-reference items that have no entries (callers).

   -long, -lg
   print selected cross-reference items that have no entries (default).

   -pn crossref path
   specifies crossref path as the crossref to search.

Examples

```
pcref  phcs_
pcref  hphcs_$*acl
pcref  >ldd>crossref>total.crossref  stack_frame.incl
```

Output Example

References to objects matching search names are displayed like this:

References to phcs_$ring_0_peek: (STAND-ALONE in HARDCORE)

as_meter_, copy_salvager_output, display_branch, namef_,
ring_zero_peek_, sweep_pv, vpn_cv_uid_path_

If a matching object is not referenced by anything, it will be identified as such. If a search name does not match anything found in the crossref, a diagnostic is displayed. The listing is a maximum of 72 characters wide.

Notes

This command uses a binary search to locate the desired information and thus is quite inexpensive, even when searching the total system crossref. Average cost for a single search of the system crossref seems to be about 45-page faults and 0.5 CPU seconds, or roughly 30 times cheaper and far more convenient than using an editor.

No attempt is made to combine the results of the search names--if you ask for something twice, it will get printed twice.

This command does not perform any significant validation on the input file, and is likely to either take faults or signal the logic_error condition if asked to search something other than a crossref output file.

There is no support for synonyms; a search name must be the primary name of a segment, and not a synonym established in a bindfile or the hardcore header.

There is no way to select specific types of things, such as all the unresolvable references in the crossref.

Name: prelink

The prelink command generates a set of data segments that can be used during process and ring initialization to minimize the overhead. Many of the linkage faults associated with bringing up a new process can be avoided, and some of the storage used by linkage sections can be shared. Both of these features lead to smaller working sets for the entire system.

The prelinker must be run anew after each bootload to assure that links to segments that come in off of the bootload tape are consistent. Thus, it is advantageous to use prelinked environments only when several user processes make use of them, per bootload, since it is somewhat costly to prelink a subsystem.

The prelinker takes an ASCII driving file as input and places all output segments it creates in the same directory in which the driving file is located. This directory contains the necessary data bases to initialize a prelinked process and is the directory specified with the -subsystem control argument to login (see MPM Commands) or the subsystem keyword used in the PMF (see MAM Project).

Usage

prelink {path} {-control_args}

where:

1. path
   is the pathname of the directory containing the prelinker driver table that must have an entryname of pldt. If path is not specified, the current working directory is used.

2. control_args
   may be chosen from the following:

   -delete, -dl
   causes the prelinker to delete any segments created by a previous invocation of the prelinker. These segments are named as follows:

   template_kst
   template_dseg
   stack_?
   *.area.linker
   *.area.prelinker

   where the star convention is applied to the above names. Note that template_kst and template_dseg have ring brackets of 0, and hence cannot be deleted by normal means.

   -debug, -db
   causes the prelinker to retain its environment in the event of an unexpected fault or condition during prelinking. The default action (if this control argument is not specified) is to clean up the environment and report an appropriate error message.

Notes

The key to generating a prelinked subsystem is the generation of the prelinker driver table (PLDT). The PLDT contains all instructions on which segments to prelink, which search rules to use during prelinking, which rings to prelink, and how to lay out the various linkage sections that are used.

The basic concept of the prelinker is to allocate linkage sections (and static sections when necessary) in template linkage segments, point to these sections with pointers in a template LOT (and ISOT), allocate segment numbers for all prelinked segments, and snap as many of the links in the template linkage segments as possible. Clearly, since snapping a link requires knowledge of the segment number of the target of the link, only segments in the prelinked set can be prelinked to. This means that if the set of programs prelinked is not "transitively closed" there are links in the template linkage segments that cannot be snapped. These links are snapped if and when they are referenced as the prelinked process actually executes.

Internal Static

The goals of prelinking are to:

1. minimize linkage faults in a newly created process (and any associated paging), and

2. minimize the system working set by sharing data that would otherwise be per process (i. e., linkage sections).

Since the default location of internal static storage is the linkage section and since internal static storage cannot be shared, some mechanism must be used to remove internal static from the linkage section so that the linkage section can be shared. There are two ways this is done. First, all internal static variables that might be allocated in the linkage section are removed from the program. This can be done by declaring "constant" internal static variables with the "options (constant)" attribute and thereby get the "variable" allocated in the (shared) text, or by recoding the program to remove such variables from the program.

The second way to remove internal static from the linkage section is to use the "separate static" option of the PL/I and ALM translators. (In PL/I this is done with the -separate_static control argument; in ALM this is done with a join/static/ statement.) This forces a separate region of the object segment for internal static that is allocated independently of the linkage. Programs whose static is separate can have their linkage shared and their internal static per process.

Unsnapped Links

Since most prelinked systems probably have some links left unsnapped, it is important to understand what happens when a linkage fault occurs on one of these links. If the link is in a per-process linkage segment (e.g., those containing linkage sections with internal static), the link is snapped in the normal way. If, however, the link is in a shared linkage segment, an attempt to snap the link results in a copy-on-write fault, which causes a copy of the shared linkage segment to be placed in the process directory (with write permission to the user) so that the link can be snapped. This has the disadvantage of causing the working set of the system to be increased by those pages referenced by the given user in the once-shared linkage segment. For this reason, it is often advantageous to force segments with unsnappable links into per-process linkage segments, even though they might appear to be good candidates for shared linkage.

Unsnapping Links

If a user wants to make a segment unknown that is linked to by a program whose linkage is allocated in the shared linkage segment, a similar problem to that mentioned above occurs. Namely, the attempt to reset such links to their virgin state causes a copy of the shared linkage to be created and used. Again, if it is expected that links allocated in shared linkage will commonly be unsnapped, it might be advantageous to force such linkage sections into per-process linkage.

Internal Static Storage

The prelinker does not allocate separate internal static sections unless some prelinked program links to the static section (in which case the address is required to map the link). Instead, ISOT faulting packed pointers are placed in the ISOT for segments with separate static. If, during execution of a prelinked process, the static storage is referenced, the ISOT fault occurs and results in allocation of the static storage in a per-process segment. This has the advantage of not allocating static storage until it is needed--again resulting in smaller working sets.

Caveats for Prelinking

There are some reasons why prelinking might not be advantageous for a particular subsystem. One such reason is that the prelinker acts similar to a multisegment binder and resolves name searches at prelink time instead of at runtime. This is a tradeoff of function for better performance that may not be most appropriate for all applications.

Another problem that prelinking can cause is the potential for larger linkage segments than would otherwise be used in a nonprelinked system. This can occur if too many programs are included in the prelinked set, many of which are never referenced. Some thought must therefore be given to which programs to include in a prelinked subsystem. Programs not included can, of course, be dynamically linked to in the usual manner.

General Recommendations

In order to minimize the number of segments used by a process, it is usually best to place the first linkage region (general area, in fact) in the stack segment. The linker is ready to handle any overflow by creating further segments in the process directory as part of the same logical area.

Similarly, since the entry sequence of PL/I programs references the LOT and ISOT, it is best to limit these to 512 words each so that they can both be placed in the first page of the stack. Again, the linker is ready to handle an overflow if 512 is not large enough.

Other tuning should be done by examining processes created from prelinked templates to check for unnecessary programs or programs that are dynamically linked to and should therefore possibly be included in the original prelinked set.

Prelinker Driving Table (PLDT)

The PLDT directs the prelinker by specifying which segments to create, how big to make them, and which linkage sections to allocate where. The format of the PLDT is a header consisting of a Max_segno statement and a Search_rules statement. The syntax of these is:

```
Max_segno:  N;

Search_rules:  [referencing_dir,] path1, ..., pathM;
```

These may occur in either order. The Max_segno statement is used to specify the initial size of the LOT and ISOT. The Search_rules statement is used to specify the search rules to use while resolving name ambiguities during the snapping of links.

One or more "ring groups" are required after the two required statements. A ring group specifies in which rings the following segments are to be prelinked. The format of a ring group is:

```
ring:  low;
     ring body statements

end;

  or

ring:  low, high;
     ring body statements

end;
```

where rings low to high are prelinked using the information in the ring body statements.

The first three ring body statements must be linkage statements to specify the maximum size that the three kinds of linkage regions can attain. The three regions are referenced as "stack," "shared," and "combined." For example:

```
ring:  4;
     linkage:  stack, 4096;
     linkage:  shared, 16384;
     linkage:  combined, 65536;
             .
             .
             .

end;
```

The linkage statements may appear in any order, but all three must appear.

After the linkage statements are the directory statements, which have either of the two forms:

```
directory:  path;
  segment:  segname1;
  segment:  segname2;
          .
          .
          .
```

or

```
directory:  path, -all;
```

Any number of directory and linkage statements can appear in the body of a ring group. (Subsequent linkage statements merely change the size of the specified type of linkage region.)

The -all parameter of a directory statement directs the prelinker to prelink all segments (or segments linked to) in the specified directory. All names on the given segments are potential reference names to be used in the segment search. If -all is not specified, only the segments specified (with all names on the segments being used) in subsequent segment statements are prelinked.

For each ring of each ring group specified, a stack, and possibly a shared linkage segment, and a combined linkage segment are created. The ring brackets on these per-ring segments is r,r,r where r is the ring number.

At any time where a break or white space appears in a PLDT, a comment may be inserted. The syntax of comments is merely /* ... */ as in PL/I.

Example PLDT

```
Max_segno:      512;
Search_rules:   referencing_dir,
                >system_library_standard,
                >system_library_unbundled,
                >system_library_1;

ring:           1;
  linkage:      stack, 0;
  linkage;      shared, 16384;
  linkage;      combined, 16384;

  directory:    >system_library_1, -all;

  directory:    >system_library_standard;
    segment:    mseg_;
    segment:    bound_message_segments_;
    segment:    mail;
end;   /*ring 1 */
ring:           4,5;
  linkage:      stack, 8192;
  linkage:      shared, 16384;
  linkage:      combined, 16384;

  directory:    >system_library_1, -all;
  directory:    >system_library_unbundled;
    segment:    bound_fast_;
    segment:    bound_basic_;
    segment:    bound_basic_runtime_;

  directory:    >system_library_standard;
    segment:    print;
    segment:    pl1;

    end;    /*rings 4 and 5*/

end;
```

Name: print_apt_entry, pae

The print_apt_entry command prints one or more Active Process Table entries (APTEs). Each APTE can be printed in octal form, interpreted form, or both.

Usage

print_apt_entry {identifiers} {-control_args}

where:

1. identifiers
   can be User_ids, channel names, or process IDs. The three types of identifier are distinguished from one another by their format (see "Notes" below). They can be preceded by control arguments to eliminate any ambiguity (see control_args, below).

2. control_args
   can be chosen from the following:

Entry-selection arguments:

-user User_id
   selects this user.

-channel CHN, -chn CHN
   selects the user logged in over channel CHN.

-process_id PID, -pid PID
   selects the specified process.

-interactive, -ia
   selects interactive users.

-absentee, -as
   selects absentee users.

-daemon, -dmn
   selects daemon users.

-all, -a
   selects all three process types. This is the default.

Output-format arguments:

-dump
   dumps the selected APTE(s) in octal.

-no_dump
   eliminates octal dump of APTEs. This is the default.

-short, -sh
   causes octal dumps (when selected) to be four words per line instead of the default of eight.

-long, -lg
causes octal dumps (when selected) to be eight words per line. This is the default.

-display
prints a header and a four-line interpretation of some of the variables in the APTE. (See "Output Format" below.) This is the default.

-brief_display
prints the heading and only the first line of the interpretation produced by -display.

-no_display
prints the heading, but none of the interpretation.

-process_dir, -pd
prints or returns the process directory pathname. See "Notes."

-term_channel, -tchn
prints or returns the process-termination event channel. See "Notes."

Notes

If no process-selection arguments are given, the APTE of the current process is printed.

The type of an identifier not preceded by a control argument is determined as follows: if it contains only octal digits, it is a process ID; if it contains any uppercase letters, it is a User_id; otherwise, it is a channel name.

Channel names and User_ids can be starnames. User_ids are of the form Person.Project.tag. Any of the three components can be omitted, along with any trailing periods. Omitted components are treated as if they had been "*". The presence of a tag component restricts the search to the corresponding user table, for that user only.

A channel is a communications channel for an interactive process (e.g., a.h017), an absentee slot number for an absentee process (e.g., abs3), or a message-coordinator source name for a daemon process (e.g., bk, prta).

If a process id of six digits or less is given, it is assumed to be the left half of a process id, which is the octal offset of the APTE.

When mutually exclusive control arguments are given, the last one on the line from each set is used. This allows each user to define his own defaults by means of an abbreviation and to override them conveniently by using opposing control arguments on a command line. In particular, -ia, -as, and -dmn are not mutually exclusive with each other but are all mutually exclusive with -all; -dump and -no_dump are mutually exclusive; -short and -long are mutually exclusive; and -display, -brief_display, and -no_display are mutually exclusive.

This command can be invoked as an active function, to return individual items from the APTE. Currently, only two items are available for active-function return: process directory pathname and process-termination event channel. Using it as an active function without specifying one of those items is an error.

Read access to the three user tables (absentee_user_table, answer_table, and daemon_user_table) in >sc1 is required, as well as access to the gate metering_ring_zero_peek_.

Output Format

The print_apt_entry command prints, for each APTE selected, a heading line, an optional interpretation of one to four lines, and an optional octal dump. The contents of the heading and interpretation are described here. Fields enclosed in square brackets ([]) are omitted if they contain null values, such as zero.

The heading:

Pers.Proj.tag <channel> at <offset> in tc_data >pdd><pdir>

gives the User_id, communications channel, octal offset of the APTE, and process directory name. This line is always printed.

Line 1:

[F:<flags>][E:<event>]PID:<proc_id> TRM:<term channel>

gives the flag word (omitted if zero or if line four, containing flag names, will be printed), the event word (omitted if zero), the process id, and the event channel over which this process's termination will be signalled. All of these are in octal. This line is printed unless -no_display is given.

The remaining three lines are printed by default, but are suppressed by the -brief_display or -no_display arguments.

Line 2:

<state> for <interval> (since <time>[<date>]).
Usage: cpu <sec>; vcpu <sec>; pf <n>.

gives the process state (blocked, running, etc.) and the time interval since state change and the time of state change (the date is printed only if it is different from the current date). These are followed by the total real and virtual cpu time used, in seconds, and the number of page faults.

Line 3:

```
te/s/i/x:  <te> <ts> <ti> <timax>.[<ips name> pending.]
     [Flags:  <flag names>.]
```

gives the four scheduling parameters, te, ts, ti, and timax, in seconds of CPU time. These parameters are described in Multics System Metering (Order No. AN52); briefly, they are time eligible, time since scheduled, min (time since interaction, timax), and the upper limit on ti. Following these parameters, any ips signals pending in the process are printed, as well as the names of any flags that are on (except for the "firstsw" flag, which is only printed if it is off, an indication that the process has never run).

Line 4:

```
[Alarm in <interval> (at <time>[<date>][(<interval> after block)]).
     [CPU monitor in <interval> vcpu sec.]
```

is omitted unless the process has an alarm timer or a CPU monitor set. If an alarm timer is set, its time (and date, if different from the current date) are printed. If the process is blocked, the interval between the time of blocking and the alarm timer is printed. If a CPU monitor is set, the amount of virtual CPU time remaining until it goes off is printed.

Examples

This example selects a process by its message-coordinator source name:

```
pae bk

Backup.SysDaemon.z bk at 4700 in tc_data, >pdd>!BM1CKBGBBBBBBB
PID:004700234010 TRM:064472406353 302704222432
Ready for 0.0 (since 01:08:53). Usage: cpu 31:18; vcpu 9:07.6; pf 363582
te/s/i/x: 0.294 2.291 8.235 32.000. Flags: wakeup_waiting, loaded, eligible.
```

This example selects a process by its APTE offset:

```
pae 3000

Initializer.SysDaemon.z sysctl at 3000 in tc_data, >pdd>!zzzzzzzbBBBBBB
PID:003000777777 TRM:000000000000 000000000000
Ready for 0.034 (since 01:08:53). Usage: cpu 1:51:28; vcpu 37:14; pf 1258108.
te/s/i/x: 0.000 0.000 0.000 0.000. Flags: wakeup_waiting.
Alarm in 53.947 (at 01:09:47).
```

Name: print_configuration_deck, pcd

The print_configuration_deck command displays the contents of the Multics config_deck. The data is kept up-to-date by the reconfiguration commands and, hence, reflects the current configuration being used.

Usage

print_configuration_deck {card_names} {-control_args}

Syntax as an active function:

[pcd {card_names} {-control_args}]

where:

1. card_names
   are the names of the particular configuration cards to be displayed. Up to 32 card names can be specified. (See Section 6 of the Multics Operator's Handbook, Order No. AM81, for the names of the configuration cards.)

2. control_args
   can be selected from the following:

   -brief, -bf
   suppresses the error message when a requested card name is not found. (Default)

   -exclude FIELD_SPECIFIERS, -ex FIELD_SPECIFIERS
   where field_specifiers can be used to exclude particular cards or card types from being displayed. One to 14 field specifiers can be supplied with each -exclude control argument, and up to 16 -exclude arguments can be specified. To be eligible for exclusion, a card must contain fields that match all field specifiers supplied with any -exclude argument.

   -long, -lg
   prints an error message when a requested card name is not found.

   -match FIELD_SPECIFIERS
   where field_specifiers can be used to select particular cards or card types to be displayed. One to 14 field specifiers can be supplied with each -match control argument, and up to 16 -match arguments can be specified. To be eligible for selection, a card must contain fields that match all field specifiers supplied with any -match argument.

   -pathname PATH, -pn PATH
   prints card(s) from the copy of the configuration desk at PATH, rather than the one for the running system.

print_configuration_deck

Notes

Field specifiers can consist of a complete card field or a partial field and an asterisk (*). An asterisk matches any part of any field. For example, the field specifier "dsk*" would match any card containing a field beginning with the characters "dsk". Specifiers for numeric fields can be given in octal or decimal, but if decimal they must contain a decimal point. Asterisks cannot be specified in numeric field specifiers. All numeric field specifiers are converted to decimal and matched against numeric card fields, which are also converted to decimal. Hence, the field specifier "1024." would match a card containing the octal field 2000, and the field specifier "1000" would match a card containing the decimal field 512.

Selection is performed as follows. If no card names are specified, all cards are eligible for selection. On the other hand, if any card names are supplied, only the cards matching those names are eligible; and if more than one card exists with a specified name, all such cards are displayed. If a nonexistent card is requested, and the -long control argument is specified, an error message is displayed.

If any -match arguments are supplied, those eligible cards are matched against all field specifiers of each -match argument group; however, at least one -match group must have all its field specifiers match some field on the card to make that card eligible. A similar algorithm is used for any -exclude argument groups. So, if a card is eligible, and if -exclude arguments are supplied, then at least one -exclude group must have all its field specifiers match some field on the card to make that card ineligible. If no match for a given card name or -match group is found in the config_deck, nothing is displayed for that name or group, and no error is displayed. If no arguments are present, the complete config_deck is displayed.

Note that all card names must be specified before the first -match or -exclude argument. Field specifiers following a -match or -exclude argument include all arguments until the next -match or -exclude argument.

When called as an active function, the selected cards are returned in quotes, separated by a single space.

No action is taken for misspelled arguments or valid arguments for which there are no corresponding configuration cards.

print_configuration_deck

Examples

```
! print_configuration_deck cpu
  cpu  a  7  168  80.  on
  cpu  b  6  168  80.  on
  cpu  c  5  168  80.  off
```

(For the configuration deck displayed above.)

```
! pcd cpu -match on
  cpu  a  7  168  80.  on
  cpu  b  6  168  80.  on

! pcd -match 16 -ex off -ex b
  cpu  a  7  168  80.  on
```

Name: print_error_message, pem, pel, peo, peol

The print_error_message command prints out the standard Multics (error_table_) interpretation of a specified error code. The various entries specified below allow the user to specify the error code in either decimal or octal and have the output come out in either the short or long error_table_ form.

Usage

print_error_message code

where code is the decimal integer to be interpreted. The short form of the error message is printed.

Entry: pel

This entry is the same as print_error_message except that the long form of the error message is printed.

Usage

pel code

Entry: peo

This entry is the same as print_error_message except that the input code is assumed to be octal.

Usage

peo code

Entry: peol

This entry is the same as pel except that the input code is assumed to be octal.

Usage

peol code

Name: print_relocation_info, pri

* The print_reloction_info command is obsolete and has been deleted from this manual.

Name: print_sample_refs, psrf

The print_sample_refs command interprets the three data segments produced by the sample_refs command, and produces a printable output segment that contains the following information: a detailed trace of segment references, a segment number to pathname dictionary, and histograms of the Procedure Segment Register (PSR) and Temporary Segment Register (TSR) segment-reference distributions. (See the description of the sample_refs command.)

Usage

```
print_sample_refs name {-control_arg}
```

where:

1. name
   specifies the names of the data segments to be interpreted, as well as the name of the output segment to be produced. name may be either an absolute or relative pathname. If name does not end with the suffix srf, it is assumed.

   The appropriate directory is searched for three segments with entrynames as follows:

   ```
   (entry portion of) name.srf1
   (entry portion of) name.srf2
   (entry portion of) name.srf3
   ```

   The output segment is placed in the user's working directory with the entryname:

   ```
   (entry portion of) name.list
   ```

2. control_arg
   may be the following:

   -brief, -bf
   specifies that the detailed trace of segment references is not to be generated.

Notes

The print_sample_refs command is able to detect a reused segment number. The appearance of a parenthesized integer preceding a segment number indicates reusage.

```
     234|6542  >udd>user>bound_alpha_|6542
(1)  234|2104  >udd>user>max35|512
(2)  234|6160  >system_library_languages>assign_|6160
```

The occurrence of the above three lines in the detailed trace indicates the following:

1. a reference was made to location 6542 in bound_alpha_. The particular component of bound_alpha_ being referenced could not be determined. bound_alpha_ was assigned segment number 234.

2. a reference was made to location 512 in max35. max35 is a component of a bound segment whose name can be determined from the segment number to pathname dictionary. The segment bound_alpha_ has been terminated and, when the segment of which max35 is a component was initiated, it was assigned segment number 234.

3. a reference was made to location 6160 in assign_. The segment of which max35 is a component has been terminated and, when assign_ was initiated, it was assigned segment number 234.

The appearance of a segment number suffix (i.e., 1, 2, etc.) indicates a component of a bound segment.

```
310       >system_library_standard>bound_ti_term_
310.1     tssi_
310.2     translator_info_
```

The appearance of the above lines in the segment number to pathname dictionary indicate that tssi_ was the first component of bound_ti_term_ to be referenced, and that translator_info_ was the second component of bound_ti_term_ to be referenced.

Name: print_tuning_parameters, ptp

The description of the print_tuning_parameters command may now be found in Multics System Metering, Order No. AN52. *

Name: privileged_prelink

The privileged_prelink command can be used to prelink subsystems that include rings lower than the validation level of the caller. Special access to the prelinker_gate_ is needed to use this command.

Usage

privileged_prelink {path} {-control_args}

where:

1. path
   is the pathname of the directory in which the prelinker driver table (PLDT) is to be found. If path is not specified, the current working directory is used.

2. control_args
   may be chosen from the following:

   -delete, -dl
   causes the prelinker to delete any segments created by a previous invocation of the prelinker. The segments are named as follows:

   template_kst
   template_dseg
   stack_?
   *.area.linker
   *.area.prelinker

   where the star convention is applied to the above names. Note that template_kst and template_dseg have ring brackets of 0, and hence cannot be deleted by normal means.

   -debug, -db
   causes the prelinker to retain its environment in the event of an unexpected fault or condition. The default action (if this control argument is not specified) is to clean up the environment and report an appropriate error message.

Name: process_id

The process_id command or active function prints or returns a 12-digit octal number, the process id of a specified process.

Usage

[process_id {identifiers} {-control_args}]

where:

1. identifiers
   can be User_ids, channel names, or APTE offsets. The three types of identifier are distinguished from one another by their format (see Notes below). Two of the types can be preceded by a control argument to eliminate any ambiguity (see control_args).

2. control_args
   can be selected from the following:

   -user User_id
   selects this user.

   -channel CHN, -chn CHN
   selects the user logged in over channel CHN.

   -interactive, -ia
   selects interactive users.

   -absentee, -as
   selects absentee users.

   -daemon, -dmn
   selects daemon users.

   -all, -a
   selects all three process types. This is the default.

   -single
   requires that the arguments select exactly one process. This is the default, unless more than one identifier is given.

   -multiple
   allows more than one process to be selected. Their process ids are returned, separated by spaces. This is the default if more than one identifier is given.

Notes

Unless the -multiple control argument is given, or more than one identifier is given, it is an error if the arguments do not select exactly one process.

If no identifier is given, the process id of the current process is returned.

The type of an identifier not preceded by a control argument is determined as follows: if it contains only octal digits, it is an APTE offset; if it contains any upper case letters, it is a User_id; otherwise, it is a channel name.

Channel names and User_ids can be starnames. User_ids are of the form Person.Project.tag. Any of the three components can be omitted, along with any trailing periods. Omitted components are treated as if they had been "*". The presence of a tag component restricts the search to the corresponding user table for that user only.

A channel is a communications channel for an interactive process (e.g., a.h017), an absentee slot number for an absentee process (e.g., abs3), or a message coordinator source name for a daemon process (e.g., bk, prta).

The APTE offset is given as a 4 to 6 digit octal number. (See the description of the print_apt_entry command, in this manual, for more information on the APTE.)

The -as, -ia, and -dmn arguments may be given in any combination. The default, when none of these arguments is given (and a User_id with a tag is not given) is to search all three user tables.

Read access to the three user tables (absentee_user_table, answer_table, and daemon_user_table) in >sc1 is required, as well as access to the gate metering_ring_zero_peek_ (the latter only if an APTE offset is given as an identifier). None of the above access is required when no identifier is given and the id of the current process is returned.

Example

```
ioa_ [process_id *.SysAdmin -as]
```

prints the process_id of the single absentee process from the SysAdmin project. The example is in error if there is more than one absentee process from that project.

Name: rebuild_dir

The rebuild_dir command compares a saved directory information segment created by the save_dir_info command with the current version of the directory in the storage system. If any subdirectories are missing, rebuild_dir attempts to recreate them. If any links are missing, rebuild_dir attempts to relink them. If any segments are missing, rebuild_dir prints a comment.

Usage

rebuild_dir path {-control_arg}

where:

1. path
   is the pathname of a directory information segment. If path does not have the dir_info suffix, it is assumed.

2. control_arg
   may be one of the following:

   -brief, -bf
   suppresses the comments "creating directory X" and "appending link X."

   -long, -lg
   prints full information about any missing segments.

   -priv
   sets quotas and attempts to set the sons logical volume identifier. Access to the hphcs_ gate is needed to use this control argument.

Name: record_to_sector

The record_to_sector command converts an octal sector number to a disk sector address.

Usage

```
record_to_sector record_no {device_name}
```

where:

1. record_no
   is the octal Multics record number.

2. device_name
   is a valid device name (e.g., "m400", "m451").

record_to_vtocx

Name: record_to_vtocx

The record_to_vtocx command finds the VTOC entries, if any, corresponding to a specified record number on a storage-system volume.

Usage

record_to_vtocx pv_name arg1...argn

or

record_to_vtocx pv_name -sector sector_arg1...-sector sector_argn

where:

1. pv_name
   is the name of the physical device.

2. argi
   is the octal number record.

3. sector_argi
   is the octal sector number.

Notes

The record_to_vtocx command requires access to the phcs_ gate.

This command scans the VTOCEs in ascending order for each argument looking for the correct match and therefore uses great amounts of CPU time and requires a lot of disk I/O.

Name: reduction_compiler, rdc

The reduction_compiler generates language translators. It compiles a segment containing reductions and action routines into a PL/I source segment. The reductions specify the syntax and semantics of a new language. The action routines perform the basic operations required to translate the language (such as generating code or building a data structure). The reduction_compiler compiles these reductions and action routines into a PL/I source program that translates the new language.

Reductions are expressed in a highly compact form that emphasizes the syntax and semantics of the new language. The reduction_compiler command compiles these reductions into tables that drive an rdc-provided semantic analyzer for the language. This analyzer compares the tokens (basic units) of a program written in the new language with the valid token phrases defined in the reductions. When a valid phrase is found, the action routines defined by the reduction are invoked to translate the phrase.

Translators generated by the reduction_compiler command can be written more quickly than hand-programmed translators because rdc provides the semantic analyzer for the language. They are easier to understand and to maintain because the all-important language syntax and semantics is concentrated in the reductions, rather than being spread throughout the semantic analyzer.

The reduction compiler command can generate translators for the simplest type of language, a right-linear (finite state automaton) language. Often such languages are composed of keywords with operands, such as the control language of the bind command.

The organization of an rdc translator, the translation process, and the reduction language are described below.

Usage

reduction_compiler path {-control_args}

where:

1. path
   is the pathname of a translator source segment that is to be compiled by the reduction_compiler command. If path does not have a suffix of rd, then one is assumed. However, the rd suffix must be the last component of the name of the source segment.

2. control_args
   can be chosen from the following:

   -long, -lg
   prints all error messages with a detailed description of the error that has occurred.

-brief, -bf
prints all error messages with only a brief summary of the error that has occurred.

Note

By default, a detailed description is printed the first time an error occurs in a given compilation, and a brief description is printed in subsequent occurrences of that error.

Overview of the Translation Process

A translator for a given language must perform three steps to translate a source string written in the language:

1. Parse the source string into a list of tokens. These tokens are the basic units of the language being translated. They are character strings in the source that are separated from one another by language-defined delimiter characters.

2. Analyze the syntax of the tokens to identify groups of tokens (token phrases) that are valid in the language. Also identify invalid token phrases.

3. Assign some semantic meaning to each valid token phrase by performing a translation action. Print error messages diagnosing invalid token phrases.

Parsing the source string into tokens is a process that depends upon the types of units that make up the language, the delimiter characters defined by the language, and other language characteristics. For most languages, the lex_string_ subroutine provides facilities that are sufficient to parse the language. However, some languages have syntax characteristics that cannot be handled by the lex_string_ subroutine. For such languages, the programmer must code a special parsing routine. The nature of the parsing is further considered under "Parsing the Source into Tokens" below. See the description of the lex_string_ subroutine for more information about its parsing capabilities.

Once the source string is parsed into a list of tokens, these tokens are analyzed by a semantic analysis procedure. This procedure is generated by the reduction_compiler command from the reductions specified in the translator source segment. The procedure contains tables generated from the reductions, tables that define all of the valid token phrases accepted by the language.

For each valid token phrase, the semantic analyzer invokes the programmer-supplied action routines given in the reductions to translate the phrase. For invalid phrases, the reduction_compiler command provides a mechanism for diagnosing the errors in printed error messages.

Contents of a Translator

The source segment for a translator to be compiled by the reduction_compiler command contains the following items, which are organized as shown in Figure 2-1.

1. An optional copyright notice and other PL/I comments.
2. A set of reductions consisting of reduction attribute declarations and reduction statements. The delimiter /*++ opens the set of reductions, which is closed by the delimiter ++*/ .
3. A PL/I procedure statement for the main procedure of the translator.
4. PL/I declarations for the translator's variables.
5. An optional PL/I declaration for an error_control_table, which defines the text of error messages to be generated by the translator. This error_control_table is used by the error-reporting mechanism provided by reduction_compiler.
6. Code to parse source strings of the new language into tokens. This is shown in Figure 2-1 as a call to the lex_string_ subroutine, but programmer-supplied code could be used here instead.
7. A PL/I call statement invoking SEMANTIC_ANALYSIS, the semantic analyzer procedure generated by the reduction_compiler from the reductions.
8. A PL/I return statement to return after the translation process is complete.
9. One or more optional PL/I function subprograms that are used to define the syntax of valid token phrases. These programmer-supplied functions are called relative syntax functions.
10. One or more PL/I subroutines that are invoked to translate valid token phrases. These programmer-supplied subroutines are called action routines.

```
/* ********************                    copyright notice
   * c Copyright ...  *
   ******************** */

/*++
     MAX_DEPTH 5 \                         reduction statements and
     BEGIN                                 attribute declarations
       /   /   /        \
       /   /   / RETURN \
                        ++*/

translator: procedure;

     dcl .... ,                            translator's
         .... ,                            declarations
         .... ;

     dcl error_control_table...;

     call lex_string_$lex(..);             calls to parse translator
     Pthis_token = ...;                    input into tokens,
     call SEMANTIC_ANALYSIS();             translate these tokens,
     return;                               & return

     fcn: procedure returns                relative syntax
       (bit(1) aligned);                   functions
     end fcn;

     action: proc(...);                    action
     ...                                   subroutines
     end action;
```

Figure 2-1. Organization of a Translator

The definition of the reductions, the error-reporting mechanism, the parsing routine, relative syntax functions, and action routines are described further below.

Notice that no PL/I end statement is included for the main procedure of the translator in the translator source segment. The reduction_compiler command appends code for the SEMANTIC_ANALYSIS subroutine and for other utility programs to the contents of the translator source segment. It then appends the end statement for the translator's main procedure. Therefore, care must be taken when coding the translator source segment to ensure that all of the relative syntax functions and action routines are ended correctly, and that no end statement is included for the main procedure of the translator.

reduction_compiler

Compiling the Translator

A typical translator source segment, translator.rd, is compiled by a two-step process as shown in Figure 2-2. First, the reduction_compiler compiles translator.rd into a PL/I source segment, translator.pl1, which it creates in the user's working directory. Second, the pl1 command compiles translator.pl1 into an object segment called translator that it creates, again, in the user's working directory.

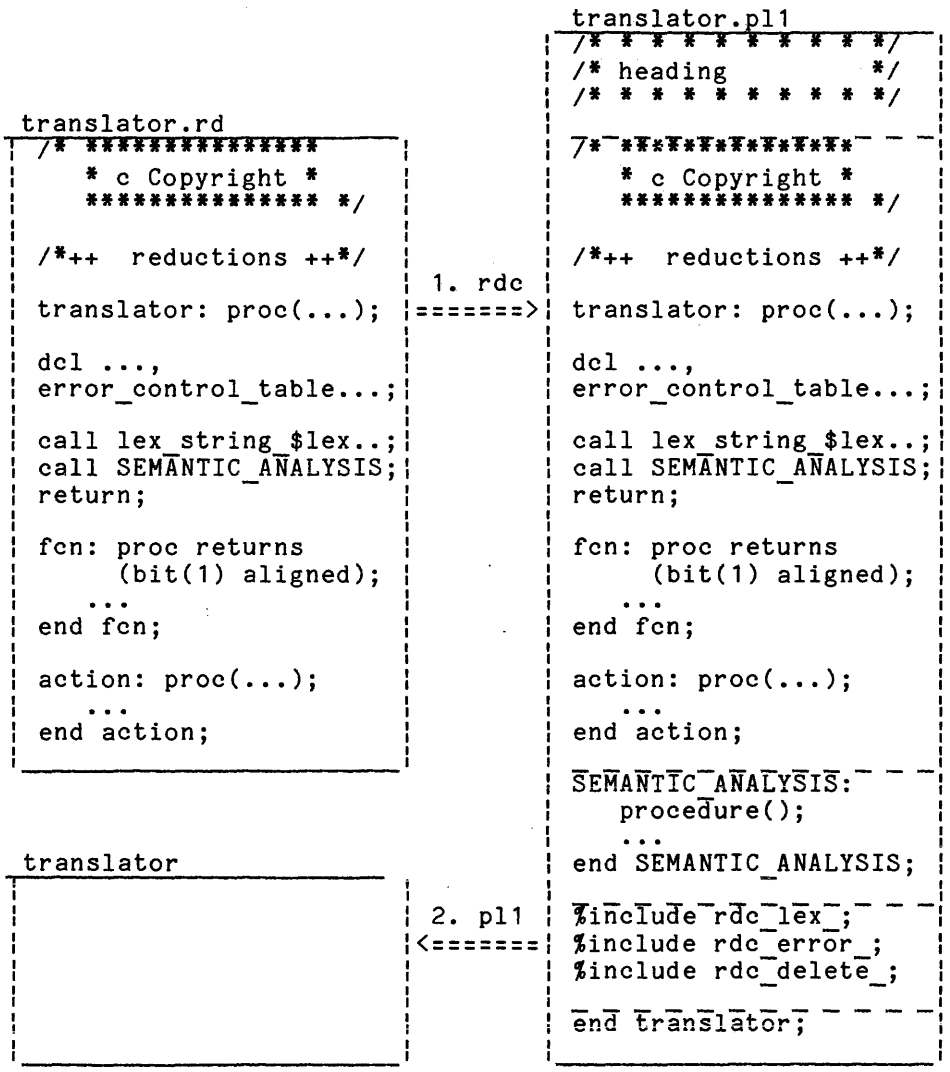


Figure 2-2. Two Steps of Compiling

The output of the reduction_compiler is a PL/I source segment that contains:

1. A heading that identifies the translator source segment, the version of the reduction_compiler command used to compile the translator source segment into the PL/I source segment, and the date and time of compilation.

2. The contents of the translator source segment.

3. The SEMANTIC_ANALYSIS procedure generated by the reduction_compiler command from the reductions in the translator source segment.

4. PL/I %include statements for utility procedures used in SEMANTIC_ANALYSIS and perhaps in the action routines to perform various functions.

5. A PL/I end statement for the main procedure of the translator. This is provided by the reduction_compiler command.

The Translation Process

The next few paragraphs describe the process of translating a language source string. It is important to understand how these steps are performed in rdc-generated translators. The reduction_compiler command or the lex_string_ subroutine provide code to perform many of these steps. For others, the programmer must supply a procedure to perform the steps.

PARSING THE SOURCE INTO TOKENS

As mentioned above, the first step of the translation process is for a translator to parse its input source string into a list of tokens. These tokens are the basic units of the language. For many languages, the lex_string_ subroutine provides sufficient facilities to parse the language. However, some languages may have a syntax that requires the special parsing facilities of a programmer-supplied parsing routine.

The lex_string_ subroutine or the supplied parsing routine must generate a chained list of token descriptors, as shown in Figure 2-3. Each descriptor describes one of the tokens in the source string. The token descriptors are chained together (forward and backward) in the order in which their respective tokens appear in the source string.

reduction_compiler

```
Volume: 70092;
Write;
File 4;

      might be
      parsed as
```

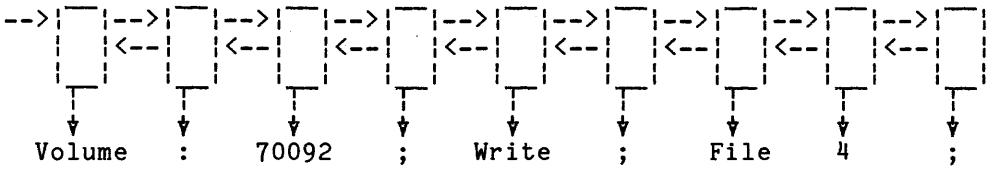


Figure 2-3. Input Tokens and Their Descriptors

Languages whose syntax includes statements separated by explicit statement delimiters can use a statement descriptor to identify the group of tokens forming each statement. The statement descriptor points to the descriptors for the first and last tokens in the statement. In turn, each token descriptor points to its respective statement descriptor. The statement descriptors are chained together (forward and backward) to create an ordered list of the statements appearing in the source string, as shown in Figure 2-4.

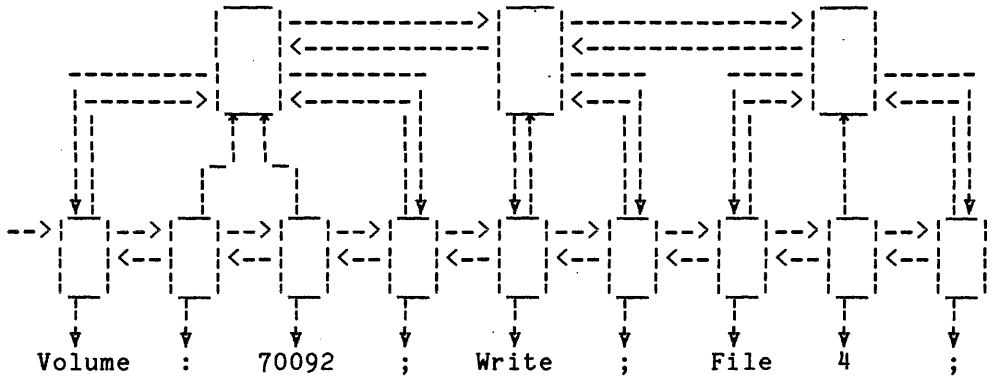


Figure 2-4. Tokens, Token Descriptors, and Statement Descriptors

There are no special requirements for a programmer-supplied parsing routine other than that it create a list of token descriptors and optional statement descriptors. The format of these descriptors is defined in the description for the lex_string_ subroutine. Refer to this description for more information about descriptors, as well as for information on the use of the lex_string_ subroutine.

Figure 2-5 shows the lex_string_ subroutine being invoked first to initialize the lex_delims and lex_control_chars break definition strings, and then to parse the translator's source string (described by Pinput and Linput) into tokens. In this example: a double quote (") character is used to open and close quoted strings; the characters /* open comments, which are closed by */; a semicolon (;) is the statement delimiter; and the colon (:), comma (,), space ( ), and all of the ASCII control characters including the PAD character operate as delimiters. The space character and all control characters except backspace are ignored delimiters that are not returned as tokens themselves, even though they separate tokens. Both token descriptors and statement descriptors are generated by the lex_string_ subroutine in this example. No descriptors are generated for the double quotes that enclose quoted strings, although descriptors are generated for the quoted strings themselves.

```
breaks = substr(collate,1,33) || ":," || substr(collate,128,1);
ignored_breaks = substr(collate,1,8) || substr(collate,10,24) ||
     substr(collate,128,1);
call lex_string_$init_lex_delims("""", """", "/*", "*/", ";", "10"b,
     breaks, ignored_breaks, lex_delims, lex_control_chars);
call lex_string_$lex(Pinput, Linput, Linput_ignore, Psegment, "100"b, """",
     """", "/*", "*/", ";", breaks, ignored_breaks, lex_delims,
     lex_control_chars, Pfirst_stmt_descriptor, Pfirst_token_descriptor,
     code);
Pthis_token = Pfirst_token_descriptor;
call SEMANTIC_ANALYSIS();
return;
```

Figure 2-5. Parsing Translator Input Into Tokens, Semantically Analyzing Those Tokens, and Returning

ANALYZING AND TRANSLATING THE TOKENS

Once the source string has been parsed into tokens, the translation continues by analyzing the syntax of the source tokens. The syntax specifications of the language are used to identify groups of tokens (token phrases). Valid token phrases are translated according to the language semantics (translation action specifications), and invalid token phrases are diagnosed to the user.

The language syntax and translation action specifications are coded in the set of reductions contained in the translator source segment. The reduction_compiler command uses these reductions to generate a SEMANTIC_ANALYSIS internal procedure that is appended to the translator when it is compiled.

When the SEMANTIC_ANALYSIS procedure is invoked as shown in Figure 2-5, it compares token phrases found in the list of source tokens with the syntax specifications defined in the reductions. If a token phrase matches the syntax specifications of a given reduction, the translation action routines associated with the reduction are invoked to translate the phrase. Then action routines are invoked to move on to the next token phrase, which is translated in a similar manner.

The translation is complete when each of the token phrases in the list of source tokens has been identified as a valid token phrase and translated, or has been diagnosed as an invalid token phrase.

Reduction Language

The reductions that define the syntax and semantics of a language to be translated are written in the reduction language. This translator generation language consists of two kinds of statements: reduction statements and attribute declarations.

Reduction statements specify the syntax of token phrases in the language being translated. They also name action routines that are invoked to translate valid phrases and to diagnose invalid token phrases.

Attribute declarations control the size of some fixed-length tables that the generated translator uses and cause translation action routines provided by the reduction_compiler command to be included in the translator. They are described below under "Attribute Declarations."

THE SYNTAX OF REDUCTION STATEMENTS

All reduction statements contain four parts: a reduction-label field, a syntax-specification field, an action-specification field, and a next-reduction field. A reduction statement has the form:

```
labels / syntax specifications / action routines / next-reduction label \
```

All of the fields must appear in each reduction, in the order shown above. The fields are separated from one another by a right slant (/) character, and the next-reduction field is terminated by a left-slant (\) statement delimiter. The fields of a reduction statement may span any number of lines in the translator source segment.

The syntax specifications, action routines, and other items that appear in a reduction statement are separated from one another by one or more of the delimiters shown in Table 2-1 below. When these delimiter characters are used, they are treated as part of the reduction. The meaning of left and right slant was described above. The double quote (") character is used as a quoting character to delimit quoted character strings in the PL/I convention. When any of the delimiter characters appears in a quoted string, it is treated as a regular character rather than as a delimiter. The use of the other delimiters is described in more detail as each field of the reduction statement is described below.

Table 2-1. Delimiting Characters Used by rdc

/	separates fields of a reduction statement.
\	ends each reduction statement or attribute declaration.
< >	delimits a syntax function in the syntax field of a reduction. For example, <no-token> .
[ ]	delimits a PL/I statement in the action field of a reduction. For example, [file_no = token.Nvalue] .
;	separates PL/I statements in the action field of a reduction when more than one statement is given between [ ] delimiters. For example, [a=b; c=d] .
( )	delimits the argument list of a PL/I subroutine call in the action field of a reduction. For example, perform_io (volume, file_no) .
,	separates arguments in the argument list of a PL/I subroutine call given in the action field of a reduction.
"	begins and ends quoted strings within a reduction statement. Inside a quoted string, a double quote (") character is expressed by two double quotes ("").
-	used to detect the special PL/I character sequence, -> , which may appear in an action specification.
=	used to detect the special PL/I character sequences, ^= <= >= , which may appear in an action specification.
^	used to detect the special PL/I character sequences, ^= ^< ^> , which may appear in an action specification.
backspace	used in the syntax field of a reduction to detect an underlined delimiter character. The special meaning of such a character is ignored, and the character is treated as a syntax specification character.
\"	begins a comment in a reduction statement. The comment ends with the next newline character.

There are also four delimiters that delimit items in a reduction but are ignored by the reduction_compiler command unless enclosed in a quoted string. These characters have no meaning in the reduction language but serve mainly to separate the specifications in a reduction statement. They are defined in Table 2-2.

Table 2-2. Ignored Delimiting Characters

space	horizontal tab
newline	newpage

SEMANTICS OF REDUCTION STATEMENTS

The most important part of any set of reductions are the syntax fields given in the reduction statements. These fields describe the syntax of the valid and invalid token phrases in the language to be translated. The syntax specifications can require a token in a particular phrase to have a specific character-string value, or to have a value that meets some general list of requirements defined in a syntax function (a PL/I function subprogram).

When a token phrase does not match the syntax requirements of a reduction it is compared with, it is compared with the syntax requirements of the reduction that follows. This process continues until the syntax requirements of some reduction are matched.

When a token phrase matches the syntax specifications of a particular reduction, the phrase is translated by invoking the action routines given in the action field of that reduction. Action routines can be simple PL/I statements or calls to PL/I subroutines with arguments. The routines can perform some constant translation operation, or an operation that depends on the values of one or more tokens in the matching token phrase. They can also skip over one or more of the tokens in the matching token phrase to permit the next token phrase to be examined.

After the action routines have been invoked, the next-reduction field of the matched reduction controls which reduction syntax field the next token phrase is compared with. The next reduction can be identified by label, using one of the reduction labels given in a label field. Also, the reduction following the matched reduction can be used next. In addition, special next-reduction operations are provided to return from the SEMANTIC_ANALYSIS procedure, and to return from a group of reduction statements used as a reduction subroutine.

Label Field of a Reduction Statement

One or more labels may be specified in the label field of a reduction statement to identify the reduction. A label is a character string that begins with an alphabetic character, and contains 32 or fewer alphanumeric or underline (_) characters.

The labels on a reduction statement can be referenced in the next-reduction field of reduction statements to direct the order in which tokens are compared with the reduction syntax specifications. To prevent any ambiguities in these references, each of the labels defined in a set of reductions must be unique.

In every set of reductions, any attribute declarations that are given must appear before all of the reduction statements. To distinguish between the attribute declarations and reduction statements, the first reduction statement must have a special first label called BEGIN.

The BEGIN label acts as a keyword that separates the attribute declarations from the reduction statements. It also identifies the first reduction with which token phrases are compared. Thus, the comparison of token phrases with reductions starts with this beginning reduction, the first reduction following the attribute declarations, the reduction with the BEGIN label.

With the exception of the BEGIN label on the first reduction statement, no labels are required on any reduction statement. Their use is optional, and is intended to facilitate the division of the set of reductions into groups of reduction statements or reduction subroutines. However, every reduction statement must have a label field even if it consists of an empty label field with a field delimiter (/). All four of the fields mentioned above must appear in every reduction statement.

Use of reduction labels is discussed further in the description of "The Next-Reduction Field" below.

Syntax Field of a Reduction Statement

The syntax field of a reduction statement defines the syntax of one token phrase in the language being translated. The tokens in the input list are compared with the syntax fields of one or more reductions. When the tokens match the syntax field of a reduction, then the action field of that reduction is invoked to perform a translation action. If the reduction specifies the syntax for a valid token phrase, the translation action can compile code to implement the semantic meaning of the phrase or it can immediately interpret the phrase or store a value in a table or perform any other translation action. If the reduction specifies the syntax for an invalid token phrase, then the translation action can diagnose the error in an error message.

CURRENT TOKEN PHRASE

Before learning how syntax specifications are defined, some terminology for dealing with the tokens in the token list must be developed.

In Figure 2-3 above, a list of tokens is described by token descriptors that are chained together. The reduction_compiler command declares a pointer in the main procedure of the translator that points to the particular tokens being compared with reductions at any given time. This pointer is called Pthis_token, and it points to the descriptor of the "current token." The current token and those tokens that follow it in the list of tokens are the tokens being compared with the reduction syntax specifications. This group of tokens is called the "current token phrase." These relationships are shown in Figure 2-6 below.

Notice that the current token phrase does not contain a fixed number of tokens. Instead, the number of tokens varies to accommodate the number of syntax specifications in the reduction being examined. Of course, if there are fewer tokens remaining to be translated than syntax specifications in a reduction, the current token phrase cannot match that reduction.

At any point in time, one of the tokens in the current token phrase is being compared with its corresponding syntax specification in a reduction. The descriptor for this token is pointed to by Ptoken, another pointer variable declared by the reduction_compiler command in the main procedure of the translator.

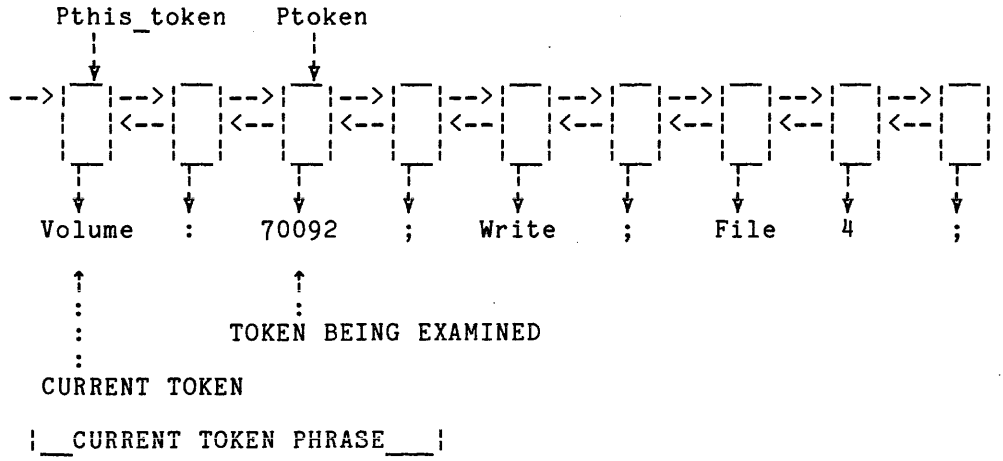


Figure 2-6. The Current Token Phrase Used by Reductions

SYNTAX SPECIFICATIONS

The tokens in the current token phrase are compared consecutively with the syntax specifications in a reduction syntax field to identify valid and invalid token phrases. The syntax specifications place requirements on the tokens in the current token phrase. If each token in the phrase meets the requirements of its corresponding syntax specification in the reduction, the entire phrase matches the reduction, and the reduction action field is invoked.

Three types of syntax specifications are allowed by the reduction language: absolute syntax specifications, relative syntax functions, and built-in syntax functions.

Absolute Syntax Specifications

Absolute syntax specifications require that their corresponding input token equal a particular character string. Absolute specifications are defined in the syntax field of a reduction statement by using their character-string value. For example, a reduction statement that would identify the first two tokens in Figure 2-6 might be:

```
vol_stmt  / Volume :                    /                    /       \
```

If reductions were written to translate all of the tokens in Figure 2-6 then "Volume," "Write," "File," ":", and ";" would probably be specified as absolute syntax specifications.

The delimiter characters used in the reduction language (see Table 2-1 and Table 2-2 above) may be used in an absolute specification by enclosing the entire specification in quotes. For example, "and/or", ">udd>Project_id>prog", """", "(", and ",". In addition, the delimiters that have a special meaning within the syntax field (/ < >) can be used as one-character absolute specifications by underlining the delimiter character. Thus, / ≤ and ≥ are treated as single-character absolute syntax specifications.

Relative Syntax Functions

Relative syntax functions are a second type of syntax specification. A relative syntax function requires that its corresponding input token meet some special requirements that are defined by a PL/I function subprogram. The requirements defined by such functions may be quite specific or very general in nature.

A relative syntax function is defined as a specification in the syntax field of a reduction by enclosing the name of the function in angle brackets (i.e., <function_name>). For example, if the volume_id function defines the requirements for a volume identifier like that used in Figure 2-6, the following reduction would match the first four tokens of Figure 2-6.

```
vol_stmt  / Volume : <volume_id> ; /                    /       \
```

Other examples of relative syntax functions might be: a <relative_pathname> function that requires that a token be a relative pathname, and that calls the absolute_pathname_ subroutine to associate an absolute pathname as the semantic value of this pathname token; a <positive_integer> function that requires that the token be a character-string representation of a positive integer; and <date_time> that requires a token that is acceptable as input to the convert_date_to_binary_ subroutine.

Relative syntax functions must be coded by the programmer and included in the main procedure of the translator source segment. Their calling sequence is shown below.

```
declare function_name entry returns (bit(1) aligned);

token_meets_requirements = function_name();
```

where the function returns a value of "1"b if the input token meets the requirements of the function, and "0"b otherwise. The function can have any valid PL/I function name that is 32 or fewer alphanumeric or underline characters in length, and that contains at least one lowercase alphabetic character. The lowercase letter is required to avoid naming conflicts with variables and procedures declared by rdc for use in the SEMANTIC_ANALYSIS procedure.

Relative syntax functions must be internal procedures of the main procedure of the translator so that they can reference the token to be examined. Ptoken points to the descriptor for this token as shown in Figure 2-6. The token descriptor itself is a structure variable named token that is based on Ptoken, as described in the lex_string_ subroutine description. The character-string value of the token can be referenced by way of the token_value variable. Ptoken, the token structure, and token_value are variables declared by the reduction_compiler in the main procedure of the translator.

A relative syntax function can associate a semantic value with the token being examined in one of three ways. It can set a variable that has been declared in the main procedure of the translator. It can set token.Nvalue to some integer semantic value, such as the numeric value of a token that matches the <positive_integer> function. Or it can allocate a semantic value structure in the temporary segment used for token descriptors, and chain this structure onto the token descriptor using the token.Psemant pointer. Refer to the description of the lex_string_ subroutine for a complete declaration of the token structure.

Built-in Syntax Functions

The third type of syntax specification that can be used in a reduction syntax field is the built-in syntax function. These are relative syntax functions that have been predefined by the reduction_compiler. Although several of these built-in syntax functions make requirements on the input token string that would be difficult to implement directly as relative syntax functions, most of the built-in syntax functions are defined merely to facilitate the implementation of the reduction_compiler itself.

Below is a list of the built-in syntax functions and the requirements they place on the input tokens.

1. <no-token>
   requires that no corresponding token exists in the current token phrase, that the list of input tokens is exhausted, and that no more tokens remain to be translated. It differs from other syntax functions that require the existence of a corresponding token in the token phrase. It is used to determine when the translation is complete.

2. <any-token>
   requires that a corresponding token exist in the current token phrase. It places no other requirements on the token. It is used when any token value is acceptable in the language being translated.

3. <name>
   requires that a corresponding token exist in the current token phrase, and that the token is a character string that begins with an alphabetic character and contains 32 or fewer alphanumeric, underline (_), or dollar sign ($) characters.

4. <decimal-integer>
   requires that a corresponding token exist in the current token phrase, and that the token is a valid, optionally signed decimal integer (as defined by the cv_dec_check_ subroutine). The numeric value of the token is stored as its semantic value in the token.Nvalue element of the token descriptor.

5. <quoted-string>
   requires that a corresponding token exist in the current token phrase, and that the token.S.quoted_string bit is turned on in the descriptor of the token. The lex_string_ subroutine turns on this bit if the token is enclosed within quoting delimiters when the input to the translator is parsed.

6. <BS>
   requires that a corresponding token exist in the current token phrase, and that the token is a single backspace character. It is used as a convenience for defining syntax specifications for one-character, underlined tokens.

COMPLETENESS OF THE SYNTAX SPECIFICATIONS

One of the most difficult aspects of writing a translator is identifying all possible invalid token phrases that could be received as input so that error messages can be issued. This problem must be addressed in each set of reductions, and in each group of reductions within a set as well, if the translator is to operate deterministically and to perform the expected translation.

A typical solution for the problem is to have a group of reductions that identify all possible valid token phrases, followed by one or more reductions that use the <any-other> built-in syntax function or an empty syntax field to identify all other invalid token phrases. For example, if the language for the tokens in Figure 2-6 requires that a colon, volume identifier, and semicolon always follow the Volume keyword, then the following group of reductions might be used to diagnose an error.

```
vol_stmt  / Volume : <volume_id> ; /                    /       \
          / Volume : <any-token> ; /                    /       \
          \" check for bad volume identifier.
          / Volume                 /                    /       \
          \" check for bad volume statement.
          /                        /                    /       \
          \" check for unknown or unexpected statement.
```

The Next-Reduction Field of a Reduction Statement

The next-reduction field governs the flow of control between reductions. When the translator calls the SEMANTIC_ANALYSIS procedure, control passes to the reduction whose label is BEGIN. The first of the current token phrases is compared with this beginning reduction and those that follow until it matches the syntax requirements of one of the reductions. The action field of that reduction is then invoked to translate to the current token phrase, and to make the next token phrase current.

The next-reduction field of the matched reduction controls which reduction the new current token phrase is compared with. The next-reduction field may be blank, or it may contain a reduction label. If it is blank, the reduction immediately following the matched reduction is used in the next comparison. If a reduction label is specified, then the reduction identified by that label is used in the next comparison. In either case, comparison of the new current token phrase with reductions continues until a matching reduction is found.

This process of analyzing token phrases continues until all of the input tokens have been translated. Each set of reductions must contain one or more reductions that use the <no-token> built-in syntax function to detect when all the input tokens have been translated. When such a <no-token> reduction is invoked, its next-reduction field usually contains the RETURN keyword, instead of a reduction label, to specify that the flow of control should return to the caller of the SEMANTIC_ANALYSIS procedure. On return from SEMANTIC_ANALYSIS, the translation is complete.

Often if several <no-token> reductions appear in a set of reductions, a reduction label is used in their next-reduction field (rather than a RETURN keyword) to branch to a final <no-token> reduction that performs epilogue actions and then returns via a RETURN keyword. Having only one of the <no-token> reductions perform the epilogue actions reduces the amount of translation code generated by rdc.

SAMPLE REDUCTIONS

Figure 2-7 shows the Backus-Naur Form (BNF) for the syntax of a language that identifies records to be read or written from a tape file on a particular volume, using a given record format. Several examples below employ this language to illustrate the use of reductions.

```
<spec>  ::=   Volume : <volume-id>[,{9track|7track}] ;
              {Read|Write} ;
              File <number> ;
              Records : <number>[, <number>]... ;
              Format : {F|FB|FBS|V|VB|VBS|U} ;
```

Figure 2-7. BNF Syntax for a Tape Language

Figure 8 shows how reduction statements can be used to define the syntax of the tape language. <positive_integer> and <volume_id> are the relative syntax functions described under "Relative Syntax Functions" above.

Note that reductions containing only an <any-token> or <no-token> syntax specification are included in each group of reductions to detect errors. The <any-token> reduction matches any token phrase except the empty token phrase (a phrase containing no tokens because all of the input tokens have been translated). The <no-token> reduction matches empty token phrases.

```
BEGIN
stmt    / Volume : <volume_id>          /                    / vol    \
        / Read ;                        /                    / stmt   \
        / Write ;                       /                    / stmt   \
        / File <positive_integer> ;     /                    / stmt   \
        / Records :                     /                    / numbers\
        / Format :                      /                    / format \
        / <any-token>                   /                    / stmt   \
        / <no-token>                    /                    / RETURN \

vol     / ;                             /                    / stmt   \
        / , 9track ;                    /                    / stmt   \
        / , 7track ;                    /                    / stmt   \
        / <any-token>                   /                    / stmt   \
        / <no-token>                    /                    / RETURN \

numbers / <positive_integer>            /                    / punct  \
        / <any-token>                   /                    / punct  \
        / <no-token>                    /                    / RETURN \

punct   / ,                             /                    / numbers\
        / ;                             /                    / stmt   \
        / <any-token>                   /                    / numbers\
        / <no-token>                    /                    / RETURN \

format  / F ;                           /                    / stmt   \
        / FB ;                          /                    / stmt   \
        / FBS ;                         /                    / stmt   \
        / V ;                           /                    / stmt   \
        / VB ;                          /                    / stmt   \
        / VBS ;                         /                    / stmt   \
        / U ;                           /                    / stmt   \
        / <any-token>                   /                    / stmt   \
        / <no-token>                    /                    / RETURN \
```

Figure 2-8. Reductions for the Tape Language

Action Field of a Reduction Statement

When a valid token phrase matches the syntax specifications of a reduction statement, the phrase must be translated according to the semantics of the source language. The translator does this by invoking the action routines specified in the action field of the matched reduction. These routines are invoked in the order of their appearance in the action field.

There are two types of action routines: those that perform some translation action on the current token phrase, and those that perform a lexing action to make another token the current token so that a new token phrase can be translated. Translation action routines are described below, and lexing routines are described under "Lexing Action Routines," which follows.

TRANSLATION ACTION ROUTINES

Translation action routines translate token phrases that match reductions according to the semantics of the source language. For example, they can construct tables; build compilation trees; generate object code, ALM statements, or PL/I statements; or perform any other type of translation function that can be expressed in the PL/I language.

There are two kinds of translation action routines: action statements and calls to action subroutines.

Action Statements

An action statement is a PL/I statement that appears in the action field of a reduction, enclosed in square brackets without its semicolon statement delimiter. For example, a tape language source string of:

```
Write;
```

might be translated by setting a mode variable as follows:

```
[mode = "w"]
```

Action statements can be used to perform the simplest translation operations, such as turning on a bit or assigning a particular value to a variable. Such simple operations occur frequently in translators, and are most clearly and easily expressed as a PL/I statement.

Action statements can use the token_value variable, just as relative syntax functions do, to reference the character-string value of the current token. For example, the tape language string:

```
Volume:  70092;
```

might be translated by making 70092 the current token, and then invoking an action statement like:

```
[volume = token_value]
```

Action statements can also use the token structure to reference the descriptor of the current token or a semantic value structure chained to the descriptor. For example, the tape language source string:

```
File 4;
```

might be translated by a reduction of the form:

```
/ File <positive_integer> ;        / LEX [file_no=token.Nvalue]
                                     LEX(2)                    /       \
```

where LEX and LEX(2) are a lexing action routines that make the next, or second next, token be the current token. Notice that, in the reduction above, the <positive_integer> relative syntax function sets token.Nvalue when it validates the syntax of the "4" token.

More than one PL/I statement can be used as an action statement if the PL/I statements are separated by a semicolon (;). This allows compound PL/I statements to be used as action statements. For example, the action statement:

```
[if token_value = "SCRATCH" then volume = "scratch";
                            else volume = token_value]
```

checks for the special tape volume name SCRATCH and uses scratch in its place if found; otherwise, the token value given in the source string is used as the volume name.

Action Subroutines

An action subroutine is a PL/I subroutine that performs some translation operation. It appears in the action field of a reduction as a PL/I call statement, without the call keyword or the semicolon statement delimiter. For example, the subroutine:

```
call perform_io ("tape_input", volume, file_no, mode, "1"b);
```

appears in the action field as:

```
perform_io ("tape_input", volume, file_no, mode, "1"b)
```

A subroutine with no arguments, such as:

```
call set_record_no();
```

appears as:

```
set_record_no
```

An example of a reduction containing action subroutines is

```
    / <no-token>                   / perform_io("tape_input",
                                     volume, file_no, mode, "1"b)/       \
```

The programmer must supply these action subroutines as part of the translator. Usually they are internal procedures defined in the main procedure of the translator. This facilitates references to the tokens being translated and to other data declared in the translator. However, an external procedure can be used as an action subroutine by calling it with arguments to pass any required information.

Naming Requirements for Action Routines

Several facts must be considered when defining action subroutines and other variables used in the action field of a reduction. First, action statements and subroutines are executed within the SEMANTIC_ANALYSIS procedure. Therefore, all variables used in action statements or as arguments to action subroutines must be declared in the main procedure of the translator. Similarly, internal action subroutines must be defined in this main translator procedure, and external action subroutines must be declared there. Figure 2-2 illustrates the relationship between the main translator procedure and the SEMANTIC_ANALYSIS procedure.

Second, care must be taken to avoid naming conflicts between the variables declared by SEMANTIC_ANALYSIS and the variables and subroutines used in the action field. With only a few exceptions, the variables used by SEMANTIC_ANALYSIS have uppercase names. Therefore, the programmer can avoid name conflicts by using names with one or more lowercase letters or digits.

There are three classes of exceptions to the uppercase naming rules used in SEMANTIC_ANALYSIS. First, SEMANTIC_ANALYSIS has declared the following PL/I built-in functions: addr, max, null, search, substr, and verify. Second, SEMANTIC_ANALYSIS has declared the cv_dec_check_ subroutine to implement the <decimal-integer> built-in syntax function. Third, the variables and structures required to reference tokens and their descriptors are declared by the reduction_compiler command in the main procedure of the translator. SEMANTIC_ANALYSIS assumes the existence of these declarations, which have lowercase names. (Refer to the description of the lex_string_ subroutine for a complete declaration of these variables.) All three classes of exceptions must be avoided when naming variables and action subroutines.

LEXING ACTION ROUTINES

Lexing action routines are useful in two ways. They can skip over a token phrase once it has been translated so that the next token phrase can be analyzed. Also, they can skip from the first token of a phrase to another of its tokens so that a translation action routine can reference that token. By default, the first token of the phrase that matches the reduction syntax field is the current token when the routines in the action field are invoked.

The following lexing action routines are provided by the reduction_compiler command.

1. LEX(N)
   makes the Nth token the new current token, where N is the token number relative to the existing current token. The current token has a relative token number of 0. Positive relative token numbers denote tokens following the current token, while negative numbers denote tokens preceding the current token. Thus, LEX(3) makes the third token following the current token the new current token.

2. LEX
   is equivalent to LEX(1).

3. NEXT_STMT
   makes the first token of the next statement (the statement following the statement that contains the current token) the new current token. This lexing routine can only be used when the tokens have been parsed with statement descriptors. NEXT_STMT is most useful to skip over the remaining tokens of a statement when an unrecoverable error has been detected in the statement.

4. DELETE(M,N)
   unthreads tokens from the token list so that they are not scanned by subsequent reductions. Tokens are unthreaded (deleted) from the Mth token relative to the current token through the Nth relative token. Thus, DELETE(2,3) deletes the second and third tokens following the current token. When the current token is one of those being deleted, the next token following those deleted becomes the current token. Thus, DELETE(-1,+1) deletes the token preceding the current token, the current token, and the token following the current token, and makes the second token following the current token the new current token.

5. DELETE(N)
   is equivalent to DELETE(N,N).

6. DELETE
   is equivalent to DELETE(0,0).

7. DELETE_STMT
   deletes all tokens of the current statement, making the first token of the next statement the new current token. The current statement is the statement containing the current tokens. DELETE_STMT can only be used when the tokens have been parsed with statement descriptors.

Using Lexing Routines in Translation Subroutines

Lexing action routines can be invoked from translation action subroutines if it is necessary for the subroutine to examine more than one token in the current token phrase. However, use of lexing routines from translation subroutines can obscure the translation process because the lexing is performed unexpectedly by a translation subroutine, rather than in the action field of a reduction where it is highly visible. If a translation routine examines only one token, it is best to place a LEX operation in the action field to make the desired token current before the translation routine is invoked.

A translation subroutine can call the internal procedures that rdc defines in a translator to perform the lexing actions. These internal procedures have the following calling sequences.

```
declare LEX entry (fixed bin);

call LEX(N);
```

where N is as described above for the LEX(N) lexing routine.

```
declare NEXT_STMT entry;

call NEXT_STMT();
```

```
declare DELETE entry (fixed bin, fixed bin);

call DELETE (M, N);
```

where M and N are as described above for the DELETE(M,N) lexing routine.

```
declare DELETE_STMT entry;

call DELETE_STMT();
```

Notice that only the two-argument version of DELETE and the one-argument version of LEX may be used from translation routines. If the particular routine to be called has not been used in any reduction, it must be explicitly included in the translator by using an INCLUDE attribute declaration statement, as described below under "Attribute Declarations."

SAMPLE REDUCTIONS

Figure 2-9 shows the reductions for our tape language, with the action fields filled in. Notice that only one of the <no-token> reductions performs epilogue functions, and that this reduction receives control from all other <no-token> reductions. The action field of reductions that identify invalid phrases have not, as yet, been specified.

```
BEGIN
stmt    / Volume : <volume_id>          / LEX(2) [volume=token_value]
                                          [track = 9] LEX            / vol    \
        / Read ;                        / LEX(2) [mode="r"]        / stmt   \
        / Write ;                       / LEX(2) [mode="w"]        / stmt   \
        / File <positive_integer> ;     / LEX [file_no=token.Nvalue]
                                          LEX(2)                     / stmt   \
        / Records :                     / LEX(2)                   / numbers\
        / Format :                      / LEX(2)                   / format \
        / <any-token>                   / NEXT_STMT                / stmt   \
        / <no-token>                    / perform_io("tape_input",
                                          volume, file_no, mode, "1"b)/ end   \

vol     / ;                             / LEX                      / stmt   \
        / , 9track ;                    / LEX(3)                   / stmt   \
        / , 7track ;                    / [track = 7] LEX(3)       / stmt   \
        / <any-token>                   / NEXT_STMT                / stmt   \
        / <no-token>                    /                          / end    \

numbers / <positive_integer>            / set_record_no LEX        / punct  \
        / <any-token>                   / LEX                      / punct  \
        / <no-token>                    /                          / end    \

punct   / ,                             / LEX                      / numbers\
        / ;                             / LEX                      / stmt   \
        / <any-token>                   / LEX                      / numbers\
        / <no-token>                    /                          / end    \

format  / F ;                           / LEX(2) format(1)         / stmt   \
        / FB ;                          / LEX(2) format(2)         / stmt   \
        / FBS ;                         / LEX(2) format(3)         / stmt   \
        / V ;                           / LEX(2) format(4)         / stmt   \
        / VB ;                          / LEX(2) format(5)         / stmt   \
        / VBS ;                         / LEX(2) format(6)         / stmt   \
        / U ;                           / LEX(2) format(7)         / stmt   \
        / <any-token>                   / NEXT_STMT                / stmt   \
        / <no-token>                    /                          /        \

end     / <any-token>                   / epilogue                 / RETURN \
        / <no-token>                    / epilogue                 / RETURN \
```

Figure 2-9. Reductions for the Tape Language
(Error-Diagnosing Actions Omitted)

ERROR-DIAGNOSING ACTION ROUTINES

Translators must identify and translate all valid token phrases in the source string, and must identify and diagnose all invalid token phrases to aid in their correction. Invalid token phrases can be detected in several ways.

1. Following a series of reductions that identify the valid token phrases for a given language construct, a reduction with an <any-token> syntax specification can be used to match all other invalid token phrases.

2. Specific reductions can identify predictable errors, such as tokens that do not match the specifications of the relative syntax function in a preceding reduction, or token phrases that have missing or invalid punctuation, misspelled or invalid keywords, and the like.

3. A reduction with a <no-token> syntax specification can be used to detect a premature end of the source string.

4. Action routines may detect an inconsistency in the semantic meaning of the source string and may diagnose the error.

When an error is detected, the translator must notify the user of the type and location of the error. The reduction_compiler command provides two facilities for printing error messages: the ERROR action subroutine and the lex_error_ external subroutine.

The ERROR Action Subroutine

The ERROR action subroutine is an internal procedure provided by the reduction_compiler command to print error messages. It may be called as follows.

```
declare ERROR entry (fixed bin(17));

call ERROR (error_number);
```

ERROR can be used in the action field of a reduction that identifies invalid token phrases. For example,

```
/ <any-token>          / ERROR(1) NEXT_STMT          / stmt   \
```

or it can be called from an action subroutine to diagnose a semantic inconsistency.

ERROR prints messages that have the following form.

```
prefix error_number, SEVERITY severity_no IN STATEMENT k of LINE l.
error_message_text
SOURCE:
statement_containing_current_token_phrase
```

For example,

```
ERROR 7, SEVERITY 2 in STATEMENT 2 OF LINE 2.
A bad track specification was given in a Volume statement.
9track has been assumed.
SOURCE:
Volume: 70082, 8track;
```

ERROR prints the error messages declared in an error_control_table structure array variable that the programmer declares in the main procedure of the translator. Each structure element in the array defines an error message, and the error_number is the array index of the desired error message. The structure contains a severity level associated with the error, a switch that controls the printing of the current statement as part of the error message, a long form of the error message text, and a brief form of the error message text. The error_control_table must be declared as a one-dimensional array of structures, with a lower bound of one, and an upper bound equal to the highest error_number that can be used. Figure 2-10 below shows a typical error_control_table declaration.

```
dcl 1 error_control_table (7) internal static options(constant),
      2 severity fixed bin(17) unaligned init (3, 2, 3, 2, 3, 2, 2),
      2 Soutput_stmt bit(1) unaligned
        init ("T"b, "1"b, "0"b, "1"b, "1"b, "1"b, "1"b),
      2 message char(70) varying init(
        "An unknown statement has been encountered.",
        "'^a' is an invalid record number.",
        "Translator input ends with an incomplete statement.",
        "'^a' is invalid punctuation in a list of record numbers.",
        "'^a' is an invalid record format.",
        "Input follows the end of the tape file specification.",
        "A bad track specification was given in a Volume statement.
        9track has been assumed."),
      2 brief_message char(28) varying init(
        "Unknown statement.",
        "Bad record number '^a'.",
        "Incomplete statement.",
        "Invalid punctuation '^a'.",
        "Invalid record format '^a'.",
        "Too much input.",
        "Bad track in Volume.");
```

Figure 2-10. error_control_table for the Tape Language

The severity_no associated with an error controls the prefix that is placed in the error message, as shown in Table 2-3 below.

Table 2-3. Relationship of error_control_table.severity_no to Error Message Prefix

SEVERITY	PREFIX	EXPLANATION
0	COMMENT	Comment. The error message is a comment, which does not indicate that an error has occurred, but merely provides information for the user.
1	WARNING	Warning only. The error message warns of a statement that may or may not be in error, but compilation continues without ill effect.
2	ERROR	Correctable error. The message diagnoses an error that the translator can correct, probably without ill effect. Compilation continues, but correct results cannot be guaranteed.
3	FATAL ERROR	An uncorrectable but recoverable error. The translator has detected an error that it cannot correct. Translation continues in an attempt to diagnose further errors, but no output is produced by the translation.
4	TRANSLATOR ERROR	An unrecoverable error. The translator cannot continue beyond this error. The translation is aborted after the error message is printed.

The statement and line numbers in the printed message are obtained from the descriptor of the current statement, if statement descriptors are available, or from the descriptor of the current token.

The phrase "IN STATEMENT k OF LINE l" appears if statement descriptors are available. Line l is the line number on which the statement containing the current token begins. Statement k identifies which statement in line l is in error, if more than one statement appears in line l. "STATEMENT k OF" is omitted from the message if only one statement appears in Line l.

If no statement descriptors are available, the phrase "STATEMENT k OF" is omitted from the message. Line l is the line number on which the current token appears.

If Pthis_token is null, the phrase "IN STATEMENT k OF LINE l" is omitted altogether, since there is no current statement and no current token.

When the output_stmt_sw of an error is on, the current statement is included in the printed error message. The stmt.output_in_err_msg switch is turned on in the statement descriptor to prevent the source statement from being reprinted in subsequent error messages. Since the current statement is obtained from its statement descriptor, the translator must parse its source string with statement descriptors. If statement descriptors are not present, error_control_table.output_stmt_sw has no effect. Refer to the description of the lex_string_ subroutine for information about the structure, contents, and generation of statement descriptors.

The printed error message contains either the error_message_text or the brief_message_text, depending upon the value of the SERROR_CONTROL variable. This variable is declared by the reduction_compiler command, in the main procedure of the translator, as follows:

```
dcl SERROR_CONTROL bit(2) initial ("00"b);
```

Table 2-4 below shows how the setting of these bits controls the r_message_text in the printed error message.

Table 2-4. SERROR_CONTROL Bits Control the error_message_text

SERROR_CONTROL	INTERPRETATION
"00"b	The printed error contains the error_message_text the first time the error occurs and the brief_message_text for subsequent occurrences of that error during a given translation.
"10"b	The printed error always contains the error_message_text.
"11"b	The printed error always contains the error_message_text.
"01"b	The printed error always contains the brief_message_text.

The reduction_compiler command declares the SERROR_PRINTED variable in the main procedure of the translator as follows.

```
dcl SERROR_PRINTED (dimension(error_control_table),1) bit(1) unaligned
     initial(dimension(error_control_table,1)(1)"0"b);
```

The ERROR routine turns on SERROR_PRINTED(error_number) whenever an error message is printed, and uses the current value of SERROR_PRINTED to control the printing of the error_message_text or brief_message_text when SERROR_CONTROL equals "00"b.

The translator can be implemented with control arguments to alter the use of error_message_text or brief_message_text in error messages. For example, the reduction_compiler command uses the normal value ("00"b) by default, but implements the -brief (-bf) control argument to set a brief value ("01"b) and the -long (-lg) control argument to set a long value ("10"b).

The error_message_text and brief_message_text of an error are defined as ioa_ control strings that may contain up to three occurrences of the ^a control code. Each occurrence of ^a is replaced by the token_value character-string value of the current token. In addition, any number of the following ioa_ control codes that do not require an input argument may be used in the error_message_text and brief_message_text strings: ^-, ^/, ^|, ^x, and ^^. The ioa_ subroutine imposes a maximum length of 256 characters on the error_message_text and on the brief_message_text after all ioa_ substitutions have been performed.

The ERROR routine maintains the severity of the highest-severity error encountered during a translation in the variable:

```
dcl MERROR_SEVERITY fixed bin(17) initial (0);
```

which the reduction_compiler command declares in the main procedure of the translator. The translator may reference the value of this variable to determine whether an uncorrectable error has occurred or to determine when to abort the translation due to an unrecoverable error.

The ERROR action routine and declarations for SERROR_CONTROL, SERROR_PRINTED, and MERROR_SEVERITY are automatically included in the main procedure of the translation when ERROR is used in the action field of one or more reductions. An INCLUDE attribute declaration can be used to include these error diagnostic facilities when the ERROR routine is used only by other action routines, and does not appear in any reductions. Refer to "Attribute Declarations" below for more information.

The lex_error_ Subroutine

The ERROR action routine is a very simple diagnostic tool, but this simplicity is possible only because ERROR does not generate highly specific error messages containing several different variable information fields. ERROR only allows the character-string value of the current token to be included in the message.

ERROR uses the lex_error_ subroutine to print its error messages. The translator can call the lex_error_ subroutine directly to produce more flexible error messages. In this way, error messages containing more than one token value, or containing variables defined by the translator, can be printed using a standard mechanism. Refer to the description of the lex_error_ subroutine for information about its calling sequence and operation.

SAMPLE REDUCTIONS - COMPLETE

Figure 2-11 shows the reductions for the tape language with errors being diagnosed by the ERROR action routine. The error_control_table used with these reductions was shown above.

```
BEGIN
stmt    / Volume : <volume_id>          / LEX(2) [volume=token_value]
                                          [track = 9] LEX              / vol     \
        / Read ;                        / LEX(2) [mode="r"]          / stmt    \
        / Write ;                       / LEX(2) [mode="w"]          / stmt    \
        / File <positive_integer> ;     / LEX [file_no=token.Nvalue]
                                          LEX(2)                       / stmt    \
        / Records :                     / LEX(2)                     / numbers\
        / Format :                      / LEX(2)                     / format  \
        / <any-token>                   / ERROR(1) NEXT_STMT         / stmt    \
        / <no-token>                    / perform_io("tape_input",
                                          volume, file_no, mode, "1"b)/ end     \

vol     / ;                             / LEX                        / stmt    \
        / , 9track ;                    / LEX(3)                     / stmt    \
        / , 7track ;                    / [track = 7] LEX(3)         / stmt    \
        / <any-token>                   / ERROR(7) NEXT_STMT         / stmt    \
        / <no-token>                    / ERROR(3)                   / end     \

numbers / <positive_integer>            / set_record_no LEX          / punct   \
        / <any-token>                   / ERROR(2) LEX               / punct   \
        / <no-token>                    / ERROR(3)                   / end     \

punct   / ,                             / LEX                        / numbers\
        / ;                             / LEX                        / stmt    \
        / <any-token>                   / ERROR(4) LEX               / numbers\
        / <no-token>                    / ERROR(3)                   / end     \

format  / F ;                           / LEX(2) format(1)           / stmt    \
        / FB ;                          / LEX(2) format(2)           / stmt    \
        / FBS ;                         / LEX(2) format(3)           / stmt    \
        / V ;                           / LEX(2) format(4)           / stmt    \
        / VB ;                          / LEX(2) format(5)           / stmt    \
        / VBS ;                         / LEX(2) format(6)           / stmt    \
        / U ;                           / LEX(2) format(7)           / stmt    \
        / <any-token>                   / ERROR(5) NEXT_STMT         / stmt    \
        / <no-token>                    / ERROR(3)                   /         \

end     / <any-token>                   / ERROR(6) epilogue          / RETURN  \
        / <no-token>                    / epilogue                   / RETURN  \
```

Figure 2-11. Complete Reductions for the Tape Language

Reduction Subroutines

Often a new language contains phrases that are similar in form but that differ in their use of keywords, types of keyword operand values expected, or in other minor ways. As an example, the value language specified in Figure 2-12 below includes three types of statements, each of which begins with a keyword followed by a punctuated list of keyword operand values.

```
<stmt> ::= Name :  <name>[,<name>]...  ;
           ¦ Attribute :  <attr>[,<attr>]...  ;
           ¦ Value :  <number>[,<number>]...  ;

<name> ::= is the name of a variable.

<attr> ::= fixed ¦ float ¦ decimal ¦ binary

<number> ::= is a numeric value to be assigned to a variable.
```

Figure 2-12. BNF Specification for the Value Language

Since all the punctuated lists used in each statement have the same form, a single group of reductions can be written to translate the punctuation for all three types of statements. This sharing of reductions reduces the total number of reductions needed to translate the value language. Reduction subroutines provide the facility for shared reductions.

A reduction subroutine is a group of reductions. As with a PL/I subroutine, a reduction subroutine has a primary entry point named by the label given in the label field of its first reduction. Alternate entry points are identified by the labels on other reductions in the subroutine. For example, the following reduction subroutine has a primary entry point of punct and an alternate entry of punct_no_comma.

```
punct      / ,             / LEX                          / STACK_POP  \
punct_no_comma
           / ;             / LEX                          / STACK_POP  \
           / <any-token>   / ERROR(7) NEXT_STMT           / stmt       \
           / <no-token>    / ERROR(4)                     / RETURN     \
```

A reduction calls a reduction subroutine by storing a return label in a label stack (a pushdown stack of label values), and then giving the subroutine entry-point name in the next-reduction field. The subroutine reduction labeled by that entry-point name is then the next reduction that is compared with the current token phrase. When the reduction subroutine has completed its translation of input tokens, it returns to the calling reduction (or group of reductions) at a label that the caller stores in a label stack prior to the call. For example, the punct subroutine shown above is called by each reduction in the group shown below.

```
attr       / fixed         / LEX attr(1) PUSH(attr)       / punct      \
           / float         / LEX attr(2) PUSH(attr)       / punct      \
           / <any-token>   / ERROR(5) LEX PUSH(attr)      / punct      \
```

The label stack performs the same function as the activation stack for PL/I subroutines. A caller stores the desired return point label on the top of the stack by giving that return point label in the PUSH label stacking action routine. The caller then transfers to the desired subroutine entry point by giving that entry-point label in its next-reduction field. The called subroutine returns by using the STACK_POP keyword in the next-reduction field of one or more of its reductions. STACK_POP causes a transfer to the label on top of the label stack as it removes that label from the stack.

The next few paragraphs describe more fully the facilities for writing and calling reduction subroutines. A set of reductions for translating the value language follows this description.

LABEL STACK ACTION ROUTINES

Two action routines manipulate the label stack used by reduction subroutines: PUSH and POP.

1. PUSH(label
   pushes the named label onto the top of the stack. Up to 10 labels may be stored in the stack by default.

2. POP
   pops the top label off the top of the label stack. The label below the popped label becomes the new top of the stack. If the popped label is the only label in the stack, the stack becomes empty. If no labels are on the stack before popping, the POP is ignored.

If a PUSH would cause the label stack to overflow, then PUSH calls the lex_error_ subroutine to report a severity 4 error and then calls the cu_$cl entry point to return to command level. A start command cannot be given, but translator maintenance personnel can perform debugging operations to determine why the stack has overflowed.

By default, only 10 labels can be stored in the stack. This number can be increased by use of the MAX_DEPTH attribute declaration. See "Attribute Declarations" below for more information.

POP is useful for reduction subroutines that are called by stacking two return labels, a normal return label, and an error return label, before transferring to the subroutine. The following example illustrates this usage.

```
attr        / fixed          / LEX attr(1) PUSH(attr)
                               PUSH(syntax_err)          / punct      \
            / float          / LEX attr(2) PUSH(attr)
                               PUSH(syntax_err)          / punct      \
            / <any-token>    / ERROR(5) LEX PUSH(attr)
                               PUSH(syntax_err)          / punct      \
syntax_err
            /                / ERROR(6) POP NEXT_STMT  / stmt       \

punct       / ,              / LEX POP                 / STACK_POP  \
            / ;              / LEX POP                 / STACK_POP  \
            / <any-token>    /                         / STACK_POP  \
```

The label stack is implemented as an array of fixed binary integers. The reduction_compiler command converts all labels appearing in a PUSH action routine to a reduction number that is passed as an argument to a PUSH internal procedure provided by the reduction_compiler command. The PUSH procedure increments a STACK_DEPTH variable that records the array index of the top stack element, and then stores its input reduction number in the new top-of-label-stack element. POP pops the top label from the stack by decrementing STACK_DEPTH. It is sometimes useful to clear the label stack when an error occurs. This can be done by an action statement that sets STACK_DEPTH to zero.

LABEL STACK NEXT-REDUCTION KEYWORDS

Two keywords may be given in the next-reduction field of a reduction to perform reduction subroutine return operations: STACK and STACK_POP.

1. STACK
   transfers to the label stored on top of the label stack. If the label stack is empty, then no STACK operation occurs, and a transfer occurs to the next reduction (the reduction following the one that used the STACK keyword) just as if an empty next-reduction field had been given.

2. STACK_POP
   performs a STACK operation followed by a POP operation. This implements the typical subroutine return operation.

SAMPLE REDUCTIONS USING REDUCTION SUBROUTINES

Figure 2-13 below shows the reductions required to translate the value language described in Figure 2-10. The punct reduction subroutine is called to process the list punctuation symbols by the names, attr, and values reduction groups. These groups in turn are reduction subroutines that are called to process statement operands by the stmt group of reductions.

The error messages used below may be summarized as follows: ERROR(1)--severity 2, unrecognized statement; ERROR(2)--severity 2, unexpected '^a' punctuation mark in a name list; ERROR(3)--severity 2, invalid name '^a' in a Name list; ERROR(4)--severity 3, incomplete statement; ERROR(5)--severity 2, invalid attribute '^a' in an Attribute list; ERROR(6)--severity 2, invalid number '^a' in a Value list; and ERROR(7)--severity 3, unexpected '^a' when a punctuation mark was expected in a name list.

```
MAX_DEPTH 2 \
BEGIN
stmt      / Name :          / LEX(2) PUSH(stmt)          / names      \
          / Attribute :     / LEX(2) PUSH(stmt)          / attr       \
          / Value :         / LEX(2) PUSH(stmt)          / values     \
          / <any-token>     / ERROR(1) NEXT_STMT         / stmt       \
          / <no-token>      /                            / RETURN     \

names     / <name>          / set_name LEX PUSH(names)   / punct      \
          / ;               / ERROR(2) LEX               / STACK_POP  \
          / ,               / ERROR(2) LEX               / names      \
          / <any-token>     / ERROR(3) LEX PUSH(names)   / punct      \
          / <no-token>      / ERROR(4)                   / RETURN     \

attr      / fixed           / attr(1) LEX PUSH(attr)     / punct      \
          / float           / attr(2) LEX PUSH(attr)     / punct      \
          / decimal         / attr(3) LEX PUSH(attr)     / punct      \
          / binary          / attr(4) LEX PUSH(attr)     / punct      \
          / ;               / ERROR(2) LEX               / STACK_POP  \
          / ,               / ERROR(2) LEX               / attr       \
          / <any-token>     / ERROR(5) LEX PUSH(attr)    / punct      \
          / <no-token>      / ERROR(4)                   / RETURN     \

values    / <decimal_number>
                            / set_num  LEX PUSH(values)  / punct      \
          / ;               / ERROR(2) LEX               / STACK_POP  \
          / ,               / ERROR(2) LEX               / values     \
          / <any-token>     / ERROR(6) LEX PUSH(values)  / punct      \
          / <no-token>      / ERROR(4)                   / RETURN     \

punct     / ;               / LEX POP                    / STACK_POP  \
          / ,               / LEX                        / STACK_POP  \
          / <any-token>     / ERROR(7) NEXT_STMT POP     / STACK_POP  \
          / <no-token>      / ERROR(4)                   / RETURN     \
```

Figure 2-13. Reductions for the Value Language

Attribute Declarations

Two attribute declarations control the maximum depth of the reduction subroutine label stack and inclusion of rdc-provided internal procedures for use in translator-provided action subroutines. These attribute declarations are described below.

1. MAX_DEPTH number \
   defines number, a decimal integer, as the maximum depth of the reduction subroutine label stack.

2. INCLUDE action_routine \
   causes the reduction_compiler command to include an internal procedure that implements the named lexing or error action routine. NEXT_STMT, ERROR, DELETE, and DELETE_STMT may be given as action_routine values. The action routine internal procedures can then be called by the translator's action routines.

Summary of the Reduction Language

Figure 2-5 below summarizes the elements of the reduction language.

Table 2-5. Elements of the Reduction Language

```
labels     syntax          actions         next-reduction
-------    ------------    -----------     --------------

MAX_DEPTH label_stack_depth_number \
INCLUDE NEXT_STMT \
INCLUDE ERROR \
INCLUDE DELETE \
INCLUDE DELETE_STMT \
BEGIN    / absolute_spec  / semant(...)  / label          \
         / <relative_fcn> / [var="1"b]   /                \
label
label2   /                /               /               \
         / <no-token>     / LEX           / RETURN        \
         / <any-token>    / LEX(n)        / STACK         \
         / <name>         / NEXT_STMT     / STACK_POP     \
         / <decimal-integer>
                          / DELETE        /               \
         / <quoted-string>/ DELETE(n)     /               \
         / <BS>           / DELETE(m,n)   /               \
         /                / DELETE_STMT   /               \
         /                / ERROR(n)      /               \
         /                / PUSH(label)   /               \
         /                / POP           /               \
```

Name: repeat_line, rpl

The repeat_line command allows certain limited testing of the performance of a user's interactive terminal by "echoing" an arbitrary message typed in by the user.

Usage

repeat_line {N} {string}

where:

1. N
   is the number of times the message is to be printed. If N is not specified, or is 0, its previous value is used; the default first-time value is 10.

2. string
   string is the message to be printed (quotes can be used to permit embedded blanks in the message). If string is an asterisk (*), the previous message is reused. The first time the repeat_line command is used in a process, a canned message, consisting of "The quick brown fox..." (alternate words in red- and black-shift), followed by three separate lines, each containing one horizontal tab character plus ASCII graphics in ascending numeric sequence, is used. If string is not specified, the user is requested to type in a new string (see "New Input" below). Once the message to be printed has been determined, it is printed N times. (Notice that in the case of the "quick brown fox" message, 4N lines are printed.)

New Input

When printing of the message is completed (or no initial message is specified), the line:

Type line (or q or <NL>):

is printed. Typing only the newline (<NL>) character causes the previous message to be printed another N times. The letter q (in lowercase), followed by <NL>, causes the repeat_line command to return to its caller. Any other line is interpreted as a new message to be printed N times.

Name: reset_ips_mask

The reset_ips_mask command sets the IPS mask for the current process to unmask some or all IPS signals.

Usage

reset_ips_mask {signal_names} {-control_args}

where:

1. signal_names
   are the names of one or more IPS signals to be unmasked. The signal names must be defined in sys_info$ips_mask_data. Presently, the defined signal names are quit, alrm, neti, cput, trm_, sus_, and wkp_. At least one signal_name or the -all control argument must be specified.

2. control_args
   can be selected from the following:

   -all, -a
   sets the IPS mask to unmask all IPS signals. This may not be specified if any signal names are also specified.

   -brief, -bf
   suppresses printing of the previous state of the IPS mask after setting it.

   -long, -lg
   prints the previous state of the IPS mask after setting it. (Default.)

Notes

If all undefined IPS signals are either masked or unmasked, and -long is specified, they are not mentioned. If, however, some are masked and others are not, an octal list will be printed. This can only happen when an invalid (probably uninitialized) value has been supplied in a call to set that mask.

Name: reset_tpd

The reset_tpd command resets the transparent paging device switch of a directory entry. Resetting this switch allows pages of the segment or directory to be placed on the paging device. If the system is not using a paging device, then this switch has no effect.

Usage

reset_tpd path

where:

1. path
   specifies the relative pathname of the object whose transparent paging device switch is to be reset.

Note

This command requires access to the gate hphcs_.

Name: ring_zero_dump, rzd

The ring_zero_dump command prints the locations of the specified ring 0 or user-ring segment in full-word octal format. This command does not require access to phcs_ for those segments accessible through the ring_zero_peek_ subroutine.

Usage

ring_zero_dump segname {offset} {length} {-control_args}

where:

1. segname
   is either an octal segment number, the name of a ring 0 segment, or a pathname. To specify a segment name that consists entirely of octal digits, the name must be preceded by the -name control argument.

2. control_args
   may be chosen from the following:

   -4bit
   prints out, or returns, a translation of the octal or hexadecimal dump based on the Multics unstructured four-bit byte. The translation ignores the first bit of each nine-bit byte and uses each of the two groups of four bits remaining to generate a digit or a sign.

   -address, -addr
   prints the address (relative to the base of the segment) with the data. This is the default.

   -bcd
   prints the BCD representation of the words in addition to the octal or hexadecimal dump. There are no nonprintable BCD characters, so periods can be taken literally. This control argument causes the active function to return BCD.

   -block N, -bk N
   dumps words in blocks of N words separated by a blank line. The offset, if being printed, is reset to initial value at the beginning of each block.

   -character, -ch, -ascii
   prints the ASCII representation of the words in addition to the octal or hexadecimal dump. Characters that cannot be printed are represented as periods. This control argument causes the active function to rerun ASCII.

   -ebcdic9
   prints the EBCDIC representation of each nine-bit byte in addition to the octal or hexadecimal dump. Characters that cannot be printed are represented by periods. This control argument causes the active function to return nine-bit EBCDIC.

-ebcdic8
prints the EBCDIC representation of each eight bits in addition to the octal or hexadecimal dump. Characters that cannot be printed are represented by periods. If an odd number of words is requested for dump, the last four bits of the last word do not appear in the translation. This control argument causes the active function to return eight-bit EBCDIC.

-header, -he
prints a header line containing the pathname (or segment number) of the segment being dumped as well as the date or time printed. The default is to print a header only if the entire segment is being dumped, i.e., if neither the offset nor the length argument is specified.

-hex8
prints the dumped words in hexadecimal with nine hexadecimal digits per word rather than octal with 12 octal digits per word.

-hex9
prints the dumped words in hexadecimal with eight hexadecimal digits per word rather than 12 octal digits per word. Each pair of hexadecimal digits corresponds to the low-order eight bits of each nine-bit byte.

-long, -lg
prints eight words on a line. Four is the default. This control argument cannot be used with -character, -bcd, -4bit, -ebcdic8, -ebcdic9, or -short. (Its use with these control arguments, other than -short, results in a line longer than 132 characters.

-name PATH, -nm PATH
indicates that PATH is the name of a ring 0 segment or a pathname, even though it may look like an octal segment number.

-no_address, -nad
does not print the address.

-no_header, -nhe
suppresses printing of the header line, even though the entire segment is being dumped.

-no_offset, -nofs
does not print the offset. This is the default.

-offset N, -ofs N
prints the offset (relative to N words before the start of data being dumped) along with the data. If N is not given, zero is assumed.

-short, -sh
compacts lines to fit on a terminal with a short-line length. Single spaces are placed between fields, and only the two low-order digits of the address are printed, except when the high-order digits change. This shortens output lines to less than 80 characters.

Notes

Only one of the control arguments -ebcdic8, -ebcdic9, -character, -bcd, or -4bit can be specified.

When invoked as an active function, ring_zero_dump returns only one word of information, which is located at offset with the segment. If the -4bit, -bcd, -character, -ebcdic9, -ebcdic8, -hex8, or -hex9 control arguments are invoked, the information is returned in the specified format only. All other arguments are ignored in active-function invocation.

Name: sample_refs, srf

The sample_refs command periodically samples the machine registers in order to determine which segments a process is referencing. Three output segments are produced that are interpretable by the print_sample_refs command. (See the description of the print_sample_refs command.)

Usage

```
sample_refs -control_args
```

where:

control_args can be chosen from one of the following two control groups:

1. arguments that initiate sampling are:

   -time n, -tm n
   specifies the rate in milliseconds at which the process is sampled. n must be a positive integer. The default is n = 1000; i.e., the process is sampled once every second.

   -segment name, -sm name
   specifies the names to be given the three output segments. The name argument can be either an absolute or relative pathname. If name does not end with the suffix srf, it is assumed. The output segments are named as follows:

   ```
   (entry portion of) name.srf1
   (entry portion of) name.srf2
   (entry portion of) name.srf3
   ```

   The default causes the output segments to be placed in the user's working directory, with entrynames as follows:

   ```
   mm/dd/yy__hhmm.m_zzz_www.srf1
   mm/dd/yy__hhmm.m_zzz_www.srf2
   mm/dd/yy__hhmm.m_zzz_www.srf3
   ```

2. the argument that terminates sampling is:

   -reset, -rs
   specifies that the process is no longer to be sampled.

Notes

Only one active invocation per process is permitted. Attempting a secondary invocation of sample_refs causes the first invocation to be terminated, whereupon the new invocation proceeds normally.

The machine registers can be sampled only when the process is running in a ring other than ring 0. Were a process to use, for example, a total of 100 seconds of processor time, and sample_refs, running at a sample rate of n = 1000, were to record only 23 samples, it would indicate that 77 seconds of processor time were spent in ring 0.

Under certain conditions, the contents of one of the machine registers sampled--the Temporary Segment Register (TSR)--can be invalid. This invalidity is noted, but does not necessarily indicate that the process is in error.

At the maximum sample rate, 1 millisecond, execution time can be increased by as much as 50%. Using a 1 second sample rate, the increase in execution time is negligible.

Accuracy of sample rates less than 1000 milliseconds (sample rates n < 1000) is not guaranteed due to load factors. The accuracy of such sample rates increases with load.

If the process being sampled should be terminated without an invocation of sample_refs with the -reset option, interpretable output segments are still produced; however, both the off-time and the last recorded sample can be invalid.

Name: save_dir_info

The save_dir_info command creates a segment containing all information available from the storage system about a directory and its contents. The command is not recursive; that is, the entire subtree inferior to the selected directory is not scanned, just the immediately inferior branches and links. The saved information segment can be processed by the comp_dir_info and list_dir_info commands.

Usage

```
save_dir_info dir_path {seg_path}
```

where:

1. dir_path
   is the pathname of the directory to be scanned.

2. seg_path
   is the pathname of the directory information segment to be created. If seg_path is omitted, the entryname of dir_path is assumed. If seg_path does not end with the dir_info suffix, it is assumed.

Name: save_history_registers

The save_history_registers command allows a user to save processor history registers upon each occurrence of a signalable fault in the signalers stack frame. By default, the history registers are not saved, and the history register block in the signalers stack frame is set to all zeros.

Usage

save_history_registers {state} {-control_args}

where:

1. state
   can be either "on" or "off." If state is not specified, it is off.

2. control_args
   can be chosen from the following:

   -print, -pr
   will display the current state of the history register save switch if it is present without the state argument; with this argument, the state of the switch will be displayed before the new state is applied.

   -priv
   specifies manipulation of the per-system state by directing the state and -print arguments to operate on the per-system history register save switch, wired_hardcore_data$global_hregs. When set, this switch will cause all processes to save their history registers upon each occurrence of a signalable fault in the signalers stack frame. If -priv is not specified, then the state and -print arguments operate on pds$save_history_regs, the per-process history register save switch of the user's process executing this command.

Note

When the -priv control argument is used, hphcs_ access is required.

Name: sector_to_record

The sector_to_record command converts an octal sector address to a Multics record number.

Usage

```
sector_to_record record_no {device_name}
```

where:

1. record_no
   is the octal Multics record number.

2. device_name
   is a valid device name (e.g., "m400", "m451").

Name: send_ips

The send_ips command sends an IPS signal. It is a command interface to the hphcs_$ips_wakeup subroutine entry point, described in Section 3 of this manual.

Usage

send_ips process_id signal_name

where:

1. process_id
   is a 12-digit octal number specifying the ID of the process that is to receive the signal.

2. signal_name
   is the four-character name of one of the system-defined ips signals. See the description of the set_ips_mask command in Section 2 of this manual for a list of valid IPS signal names.

Notes

No error message is given if an undefined ips signal or a nonexistent process is specified.

Leading zeros may be omitted from the process_id.

The process_id active function (described in Section 2 of this manual) is a convenient way of obtaining a process_id, given a user_id or channel name.

Access Requirements

Access to the highly privileged gate, hphcs_, is required.

Name: send_wakeup

The send_wakeup command sends an IPC wakeup. It is a command interface to the hcs_$wakeup subroutine entry point, described in the MPM Subsystem Writers' Guide (Order No. AK92).

Usage

send_wakeup process_id event_channel {event_message}

where:

1. process_id
   is a 12-digit octal number specifying the ID of the process that is to receive the wakeup.

2. event_channel
   is a 24-digit octal number specifying the event channel over which the wakeup is to be sent.

3. event_message
   is an optional 72-bit event message. It can be given as either a 24-digit octal number or an eight-character ASCII string. The default is all zero bits.

Notes

Leading zeros may be omitted from the process_id and event channel. Leading zeros or trailing blanks may be omitted from the event message. The event message is assumed to be in octal form if it contains only octal digits.

Nonexistent processes and event channels of invalid format are diagnosed; however, validly formed but nonexistent event channels are not diagnosed.

The process_id active function (described in Section 2 of this manual) is a convenient way of obtaining a process id, given a user id or channel name.

Name: set_ips_mask

The set_ips_mask command sets the IPS mask for the current process to mask some or all IPS signals.

Usage

set_ips_mask {signal_names} {-control_args}

where:

1. signal_names
   are the names of one or more IPS signals to be masked. The signal names must be defined in sys_info$ips_mask_data. Presently, the defined signal names are quit, alrm, neti, cput, trm_, sus_, and wkp_. At least one signal_name on the -all control argument must be specified.

2. control_args
   can be selected from the following:

   -all, -a
   sets the IPS mask to unmask all IPS signals. This may not be specified if any signal names are also specified.

   -brief, -bf
   suppresses printing of the previous state of the IPS mask after setting it.

   -long, -lg
   prints the previous state of the IPS mask after setting it. (Default)

Notes

If all undefined IPS signals are either masked or unmasked, and -long is specified, they are not mentioned. If, however, some are masked and others are not, an octal list will be printed. This can only happen when an invalid (probably uninitialized) value has been supplied in a call to set that mask.

Name: set_timax, stm

The set_timax command is used to set the value of timax for the user. The user must have access to both the privileged and the highly privileged gates phcs_ and hphcs_.

Usage

set_timax N

where N is the number of seconds to which timax is to be set. A value less than or equal to zero causes it to use the default timax from tc_data$timax.

Examples

The command line:

set_timax 3.5

sets timax to 3500000 microseconds for the current user's process and prints appropriate messages on both the user's terminal and operator console.

The command line:

set_timax 0

sets timax to the default timax and prints messages on the user's terminal and operator console.

Name: set_tpd

The set_tpd command sets the transparent paging device switch of a directory entry. Setting this switch prevents pages of the segment or directory from being placed on the paging device. If the system is not using a paging device, this switch has no effect.

This command requires access to the hphcs_ gate.

Usage

set_tpd path

where path specifies the relative pathname of the object whose transparent paging device switch is to be set.

Note

The paging device is not flushed of pages of the object when the transparent paging device switch is set.

Name: teco

The teco command is a character-oriented text editor that provides a basic set of requests for creating and editing ASCII text segments and an extensive macro facility for creating sophisticated text editing-request combinations.

Usage

teco {path1} {path2}

where path1 is the pathname of a text segment to be read into the teco text buffer and path2 is the pathname used when writing out the text. If neither pathname is specified, the buffer is initially empty and the user can read in or enter text segments from the terminal. If path2 is not specified, path1 is used when writing out the segment. (See "Start-Up Macro" below.)

OVERVIEW OF TECO

The teco editor implemented for Multics is modeled after the TECO in general use on the Digital Equipment Corporation PDP-10, which was originally written at MIT's Artificial Intelligence project. The teco editor allows simple editing requests on a line basis as well as a character basis. In addition, iterative and conditional facilities are provided for writing macro definitions. These permit the user to do simple manual editing of ASCII files or to write complex macros that do automatic editing. Although this implementation is modeled after the teco editor in general use, many new requests and features have been added that make the macro facility more powerful and easy to use. Some of the additions include adding if ... then ... else ... statements, allowing the contents of Q-registers to be used as quoted strings; allowing numeric and string arguments to be passed to macros; allowing searches using regular expressions, automatically executing a start_up macro whenever teco is invoked; and allowing macros that reside in files to be called directly from the editor.

The line-oriented features of teco are similar to those of the edm and qedx commands. The character-oriented requests use a pointer that can be positioned between any two characters in the buffer, permitting insertion, deletion, and so on of characters without the need to retype the line.

The teco editor reads request lines from the user's terminal line by line until a line ending with a dollar sign ($) is typed. Execution of the complete request string is started when this line is read. The teco editor will type "*" when it is waiting for a new request string. To exit from the editor, type the EQ request (followed by $ and a newline).

Macro Usage

```
teco$macro macro {macro_arguments}
```

where macro is the name of a teco macro to be executed when the editor is invoked and macro_arguments are optional arguments processed by the macro invoked. This entry point is provided for users who write teco "programs" that are intended to run without ever reaching teco request level. The command line:

```
teco path1 path2
```

is equivalent to:

```
teco$macro start_up {path1} {path2}
```

TECO STORAGE AREAS

The teco editor uses four storage areas:

1. The buffer
   an area where text to be edited is examined and modified. At all times it contains a (possibly null) character string. There is a pointer into the buffer, denoting the current position. This pointer does not point to a character; it points between two characters. The pointer can assume any value between 0 and Z, where Z is the number of characters currently in the buffer. 0 indicates that the pointer is to the left of the first character, and Z would represent the position to the right of the last character in the buffer. The value of the pointer is represented by ".".

2. Request String Area
   editor requests are read into the request string area as a continuous character stream for subsequent parsing into operational requests. Uppercase and lowercase letters can be used interchangeably in requests.

3. Q-registers
   locations for storing either numeric quantities or strings of text for later use. Each Q-register is designated by a single character name. There are 95 Q-registers, one for each printing ASCII character. Each Q-register can contain a positive or negative integer or a character string.

4. Q-register pushdown list
   a last-in-first-out (LIFO) list that can be used to temporarily store the contents of a Q-register. It is cleared (i. e., the contents are lost) every time the user returns to teco request level (i. e., a "E" is typed).

NUMERIC EXPRESSIONS

The teco editor uses numeric expressions for many of its operations. These consist of operators and operands. Operands can be decimal numbers, octal numbers, teco requests that return values, teco macros that return values, or teco special symbols. Operators are unary minus (-), arithmetic binary operators addition (+), subtraction (-), multiplication (*), division (/), and the boolean binary operations or (¦), and (&). All operators are of equal precedence and expressions are evaluated from left to right. Notice, however, that parentheses can be used in their normal manner. Spaces are ignored except to terminate numbers. If two numeric quantities are given with no operator between them, the default operator + is used. Notice that a string of digits followed immediately by a "." is interpreted as an octal rather than a decimal number. Division using the "/" operator is integer division, i.e., the remainder is ignored. The special symbols allowed in an expression at any point are:

B (Beginning) equivalent to 0.

Z equivalent to the number of characters in the buffer.

. equivalent to the current value of the pointer or the number of characters to the left of the pointers.

H (wHole) equivalent to 0,Z. It is the only symbol to have two values. It is useful for referring to the entire buffer.

Requests that return values can also be used in expressions, but they cannot appear immediately to the right of an operator if it requires arguments. This is because requests that take arguments assume that everything to its left is part of one of its arguments. If a request appears within parentheses, its arguments are entirely contained by the the closest left parenthesis that encloses the request. A request does not read parts of an expression outside the parentheses in which it is enclosed.

The plus and minus binary operators assume a right operand of 1 if none is given.

The examples below show the evaluation of numeric expressions in teco. Assume that the current value of the pointer is 500.

	expression	value
(1)	(7 12)/3	= 6
(2)	9+	= 10
(3)	b-	= -1
(4)	-	= -1
(5)	4+8/2	= 6
(6)	101.	= 65
(7)	3¦10	= 11
(8)	1++++ ++ +++ +	= 11
(9)	9*-2	= -18
(10)	9*--2	= 18
(11)	.10	= 510
(12)	10.	= 8

QUOTED STRINGS

Quoted strings are strings of text delimited by a quoting character. The quoting character can be any character not contained in the string except a letter or a digit. The contents of a Q-register can be used as a quoted string if the letter "q" followed immediately by the letter specifying the Q-register is typed instead of the first quoting character. The following examples show valid quoted strings.

(1) "hello"
(2) /This is a quoted string/
(3) ,This string is delimited by the comma character and contains 2 newline characters.
,
(4) q1

ERROR MESSAGES

Error messages are printed by teco in one of two modes: long or short. Short error messages are from one to eight characters long while long error messages are less than 50 characters long. The default mode is short. To change the error mode teco is using, give the following Multics commands:

```
teco$teco_error_mode long
```
or
```
teco$teco_error_mode short
```

If a short error message, such as "/: ?", cannot be understood, the following Multics command prints the long error message:

```
teco$teco_error "/:  ?"
```

The above holds for teco error messages only.

IMPLEMENTATION RESTRICTIONS

The maximum number of characters allowed in a Q-register, in a quoted string, or in a teco request line is 1044480 characters. Notice that these sizes are all one segment long. When the Multics segment size changes, these restrictions also change. The maximum number of items in the pushdown list is 20. The maximum depth of macro calls is 20. The maximum depth of parentheses is 20.

teco REQUESTS

The teco editor requests have the basic form:

m,nX/string/

where m and n are optional numeric arguments, X is the request to be executed, and /string/ is a quoted string. In most cases, the request is just one character, though in some cases, it is two characters. Not all of the requests take arguments. Those that do generally have default values for missing arguments. Only a few requests expect quoted strings. The string must not be omitted if the request expects one. Some requests also return values; this is discussed later in "Advanced teco Commands."

Some letters chosen for requests have mnemonic meanings, which are indicated in the description of each request. Unfortunately, teco has a fairly long history, having originally been developed for editing paper tapes, and so some of the mnemonic meanings are lost now. As many requests as one wishes can be typed at a time. Execution of the requests does not start until after a line is typed ending with a "$". Spaces can be inserted anywhere except in the middle of numbers, and newline characters can be inserted anywhere except between a request and its arguments. Uppercase and lowercase letters can be used interchangeably as requests.

Reading a File - EI (External Input)

EI/pathname/
: reads in the file specified by pathname, which is assumed to be a standard Multics pathname. The contents of the file are inserted in the buffer at the current pointer position, and then the pointer is moved to the right of the text inserted.

Writing a File: - EO (External Output)

EO/pathname/
: is equivalent to HEO/pathname/. It writes the contents of the entire buffer to the file specified by pathname. This request takes arguments similar to the T request; it writes out that part of the buffer that would be printed by T. However, if no arguments are given, EO assumes B, Z as the default rather than 1.

NOTE: The pointer is never moved by the EO request.

Typing the Buffer - T (Type)

T
: is equivalent to 1T

nT n>0
: prints the buffer beginning at the current pointer position and terminating after n newline characters have been encountered. T prints the rest of the current line, and 2T prints the rest of the current line and the next line. The last character printed by T is a newline. If n is greater than the number of new line characters to the right of the pointer, all text to the right of the pointer is printed.

n≤0
prints the characters between the (-n+1)th newline character and the pointer. The (-n+1)th newline character is not printed. If (-n+1)th is greater than the number of newline characters to the left of the pointer, all text to the left of the pointer is printed. 0T prints the beginning of the line up to the current pointer. -T prints the previous line and the beginning of the current line. If the pointer is at the beginning of a line, -T prints the previous line.

m,nT
prints the (m+1)th through the nth characters of the buffer.

NOTE: The pointer is never moved by the T request. Usually two T requests are given at once, such as 0TT, which prints the entire line that the pointer is in.

Moving the Pointer - J (Jump), C (Characters), R (Reverse), and L (Lines)

nJ
moves the pointer to the right of the nth character in the buffer, i.e., sets "." to the value of n. If n is not specified, 0 is assumed. That is, the pointer is moved to the left of the first character in the buffer. The value of n must be from 0 to z.

nC
moves the pointer n characters to the right of its current position (equivalent to .+nJ). If n is omitted, 1 is assumed. The new value of "." must be from 0 to z.

nR
like nC except it moves the pointer to the left (equivalent to -nC). If n is omitted, 1 is assumed. The new value of "." must be from 0 to z.

nL n>0
positions to the beginning of a line. Moves the pointer to the right, stopping after it has passed over n newline characters. If n is omitted, 1 is assumed. L moves the pointer to the beginning of the next line. There must be at least n newline characters to the right of the pointer.

n≤0
moves the pointer to the left, stopping after it has passed over (-n+1) newline characters and then moving it to the right of the last newline character passed over. 0L moves the pointer to the beginning of the current line. -L moves the pointer to the beginning of the previous line. There must be at least (-n+1) newline characters to the left of the pointer.

Deleting Text - D (Delete) and K (Kill)

nD
deletes n characters. If n is positive, the characters are deleted to the right of the pointer. If n is negative, the characters are deleted to the left of the pointer. If n is omitted, 1 is assumed. If n is zero, nothing is deleted.

K
takes arguments like the T request except it deletes the text T prints. The pointer is moved to where the deletion took place. If no arguments are specified, 1K is assumed.

n>0
deletes all the characters beginning at the current pointer position and terminating after n newline characters have been encountered. There must be at least n newline characters to the right of the pointer. K deletes the rest of the current line and the newline character at the end of the line, while 2K deletes the rest of the current line and the next line. 0LK deletes the current line as does 0KK.

n≤0
deletes all the characters between the (-n+1)th newline character and the pointer. There must be at least (-n+1) newline characters to the left of the pointer. 0K deletes the beginning of the current line without deleting the newline character at the end of the previous line. -K deletes the previous line and the beginning of the current line. To ensure that only the previous line is deleted, the request sequence 0L-K is used.

m,nK
deletes the (m+1)th through the nth characters of the buffer. The pointer is moved to m. Equivalent to mJ n-mD. HK deletes the entire buffer.

Inserting Text - I (Insert)

I/text/
inserts the text of the quoted string at the current pointer position and moves the pointer to the right of the inserted text. /text/ can also be specified as a Q-register, for example, Iq2.

nI
inserts the character whose ASCII code value is n. It moves the pointer to the right of the inserted character.

Search for Text - S (Search)

S/string/
is equivalent to 1S/string/

nS/string/
searches for the nth occurrence of the quoted string. If n is positive, the text is searched from the current pointer through the end of the buffer for the nth occurrence of the string. If found, the pointer is set to the right of the matching string. Otherwise, the pointer is not moved, and an error message is printed. If n is negative, the text is searched from the current pointer position to the beginning of the buffer for the (-n)th occurrence of the quoted string. The pointer is set to the left of the matched string. If the string is not found, the pointer is not moved, and an error message is printed.

m,nS/string/
searches m lines from the current pointer for the nth occurrence of the quoted string instead of searching the entire buffer. If m is positive, n must be positive, and the only part of the buffer that is searched is from the current pointer to just after the mth newline character after the current pointer. If m is 0 or negative, n must be negative, and the only part of the buffer that is searched is from the current pointer to just after the (m+1)th newline before the current pointer. 1,1S/text/ only searches the rest of the current line. 0,-1S/text/ only searches the beginning of the current line.

Search for Regular Expression - N

N/string/
is equivalent to 1N/string/; searches from the current pointer position through the end of the buffer for the first occurrence of the regular expression, string.

The term "regular expression" refers to the character string used to address a line of text that contains that string of characters. In its simplest form, a regular expression is a character or string of characters delimited by the right slant character (/). For example, in the following text, the regular expression /abc/ matches line 2:

```
a:procedure
  abc = def
   x = y
    end a
```

nN/string/
searches from the current pointer position to the end of the buffer for the nth occurrence of the regular expression, string. The value of n must be greater than 0.

m,nN/string/
searches the next m lines for the nth occurrence of the regular expression. The values of m and n must be greater than 0.

Printing Values - = (Equals)

n= or m,n=
prints the decimal value of its arguments, separated by spaces and followed by a newline.

n:= or m,n:=
prints the octal value of its arguments, separated by spaces and followed by a newline.

Leaving TECO - EQ (External Quit)

EQ
returns to the caller of teco (e.g., Multics command level). (The user must remember to do an EO request before the EQ if the editing is to be saved.)

Restarting teco After a Quit

If a quit signal is used to abort a request string, the Multics program_interrupt (pi) command can be used to restart the teco editor. Issuing a quit does not abort the entire command string; only those commands not yet executed. The current request is aborted when it is completed.

At times it is desirable to get around this feature. When doing an EO, for instance, teco does not allow the user to return to teco request level until it has completed writing the file. To get around this, the user types:

```
(quit)
teco$abort
```

When teco$abort is called, the most recent invocation of teco aborts its current operation without checking for consistency of states. This is useful if an E0 request fails because of insufficient access. Using the program_interrupt command would cause teco to reattempt the write. Notice that teco is in a consistent state whenever it actually accesses a file, and so there should be no problems encountered if this feature is used to get out of an E0 request. Under other circumstances, however, it is wise for the user to type:

```
-5t5t
```

to ensure that control is maintained. Except for the case of an unsuccessful E0 request, this feature should not be used.

STAND-ALONE EXAMPLES

Entering Teco

`teco source.pl1`
enters teco and reads in the file source.pl1 from the working directory.

`teco <x>y>z>a.ec`
enters teco and reads in the file specified.

`teco`
enters the buffer initially empty.

`teco >t>start_up.teco start_up.teco`
enters teco and reads in >t>start_up.teco. Q-register * is set to start_up.teco.

Reading a File

`EI/source.pl1/`
inserts the text contained in source.pl1 at the current point in the buffer.

Writing a File

EO/new_source.pl1/
: writes the whole buffer out into new_source.pl1.

.,zEO/bottom/
: writes out the buffer from the current pointer to the end into the file named bottom.

2EO/lines/
: writes out two lines starting at the current pointer position to the file named lines.

Printing Text

2T
: prints from the pointer to the end of the next line.

0T
: prints the current line from its beginning to the pointer.

0TT
: prints all of the current line.

25,100T
: prints the 25+1 (26th) through the 100th character of the buffer.

Moving the Pointer

J
: positions the pointer at the beginning of the buffer.

ZJ
: positions the pointer at the end of the buffer.

L
: positions the pointer at the beginning of the next line in the buffer.

0L
: positions the pointer at the beginning of the current line.

-L
: positions the pointer at the beginning of the previous line.

R
: backs up the pointer by one character position.

812-388C
: moves the pointer ahead 812-388 (424) character positions.

Deleting Text

19,22K
: deletes the 19+1 (20th) through the 22nd character of the file. Sets the pointer to 19.

19J 3D
: moves the pointer to the right of the 19th character and then deletes the next three characters (20-22).

HK
: deletes the whole buffer.

-D
: deletes the character just to the left of the pointer.

Inserting Text

I/abc
/
: inserts the line abc followed by a newline character at the current pointer position.

I.abc.
: inserts the string abc without a newline character.

65I
: inserts the character with ASCII code 65 (A) at the current pointer position.

Printing Values

Z =
: prints how many characters are in the buffer.

Z,.=
: prints how many characters are in the buffer followed by the current pointer position.

=
: prints a newline character.

Q6+53 =
: prints the value 53 plus the value contained in Q-register 6.

Searching for Text

J S/Hello/
: positions the pointer just to the right of the first occurrence of the string Hello in the buffer.

ZJ -S"Hello"
positions the pointer just to the left of the last occurrence of the string Hello in the buffer.

J 3S"*
"
positions the pointer just after the third occurrence of a line ending with an asterisk (*).

J 1,1S/Hello
/
positions the pointer just after the first line in the buffer if it ends in Hello. If the first line does not end in Hello, prints an error message.

EXAMPLES OF BASIC EDITING REQUESTS

In the following examples, underlined text is produced by teco.

teco abc.pl1
enters teco and reads in the segment abc.pl1.

B5LT$
moves to the 6th line and prints it out.
dcl a fixed bin;

BS/a/-DI/b/0LT$
changes the "a" to a "b" and prints the line.
dcl b fixed bin;

BS/dcl d/0LKT$
searches for "dcl d" and deletes the line that contains it. Then prints out the next line.
dcl f fixed bin;

BKI/dcl g char(2);
7$
deletes this line and then insert a declaration of g.

BEO/abc.pl1/EQ$
writes the edited text out to the file and then returns from teco.

ADVANCED teco COMMANDS

In "teco Requests" above, the general form of a teco request was given. Some items were left out, however. A more complete format is:

m,nXq/string1//string2/.../stringn/

The q indicates a Q-register on which the request is to act.

It should also be noted that more than one string can be given. Although no teco request currently accepts more than one quoted string, a macro can be called with multiple string arguments that can be retrieved inside the macro by the :X request.

"Numeric Expressions" specifies that expressions can be built from numbers, special valued requests, and symbols. Examples of valued requests are given in this section. Notice that requests with values that require arguments only appear on the left side of the first operator, or within parentheses. Otherwise, the part of the expression preceding the request is considered to be an argument to the request.

The effect of many requests can be changed by preceding the request with a colon (:). The colon has no fixed meaning--it is defined for each request individually. The following requests given earlier have the following changed effect:

:Iq/string/ or n:Iq
: is similar to the I request except that the specified string is inserted into Q-register q instead of the buffer. The former contents of Q-register q are lost.

n:L
: is equivalent to nLR. :L moves to the end of the line rather than the beginning.

:S/string/
: is similar to S except that it returns a value. The value is 0 if the search fails and -1 if it succeeds. Even if the search fails, teco continues execution.

:n/string/
: is similar to N except that it returns a value. The value is 0 if the search fails and -1 if it succeeds. Execution continues even if the search fails.

:T/string/
: prints the specified string on the user's terminal. This request takes no arguments.

:=
: is identical to = except it prints values in octal instead of decimal.

:EI
: is similar to EI except that it returns a value. The value returned is -1 if the read succeeds and 0 if the read fails. No error is printed if the read fails.

:J,n:J
: is similar to J except that errors cannot occur. If n is less than 0, the pointer is moved to the beginning of the file. If n is greater than Z, the pointer is moved to the end of the file.

:C,:R
: are similar to C or R except that errors cannot occur. If the pointer would be moved to before B, move it to B. If the pointer would be moved beyond Z, move it to Z.

Numeric Q-Registers

Q-registers can be used to hold numeric values. These values can be used in expressions.

SAVING A VALUE - U (Update)

Uq
: sets Q-register q to a very large positive number.

nUq
: sets Q-register q to n.

m,nUq
: sets Q-register q to n and returns m as its value.

READING Q-REGISTERS - Q (Q-register)

Qq
: returns the number stored in Q-register q as the value. Notice that Q is not a request--it is a special symbol. Thus, in the expression 5+Q3 the 5+ is not considered an argument to Q; the result is the sum of Q3 and 5. Notice if Q-register q contains text, the length of the text, in characters, is returned.

INCREMENTING Q-REGISTERS - %

%q
: adds 1 to Q-register q and returns the new number as the value. Q-register q cannot contain text. Notice that %, like Q, is a special symbol, not a request.

Text Q-Registers

Q-registers can also be used to hold character strings. They can be used to move text from one place in the buffer to another, to save request lines for execution as macros, or to provide quoted strings.

EXTRACTING TEXT TO A Q-REGISTER - X (eXtract)

Xq
: takes arguments like the T request, but copies the text that T would print into Q-register q. The former contents of Q-register q are deleted. The text is not deleted from the buffer and the current pointer is not moved.

nXq n>0
: copies all the text from the current pointer to just past the nth newline character to the right of the pointer into Q-register q. X1 copies the rest of the current line including the newline at the end of the line into Q-register 1. 2Xa copies the text on the rest of the current line and all of the next line into Q-register a.

n≤0
copies the characters between the (-n+1)th character and the pointer. The (-n+1)th newline character is not copied. 0X/ copies the beginning of the current line into Q-register /. In this case, no newline character is put into Q-register /. -Xa puts the previous line and the beginning of the current line into Q-register a.

m,nXq
copies the (m+1)th character through the nth character into Q-register q.

APPENDING TEXT TO A Q-REGISTER - P (aPpend)

Takes arguments like the X request, except it appends to the former contents of the q-register instead of deleting the former contents. The text is not deleted from the buffer and the current pointer is not moved.

INSERTING TEXT DIRECTLY INTO A Q-REGISTER - :I (Insert)

:Iq/string/
is similar to the normal I request except that the text is inserted into Q-register q rather than the buffer. The former contents of Q-register q are deleted. The text buffer is not affected.

n:Iq
is similar to :I except that it puts the character corresponding to n into the Q-register q.

GETTING TEXT FROM A Q-REGISTER - G (Get)

Gq
inserts the text contained in Q-register q into the buffer to the left of the current pointer. If the Q-register contains a number, the decimal representation of the number is inserted.

Obtaining Quoted Strings from Q-Registers

Whenever teco expects a quoted string, it is possible to indicate that the string is in a Q-register. Normally, letters and digits are considered invalid quoting characters. If, however, the letter Q is found where a quoted string is expected, the next character after the Q is considered a Q-register name. Whenever a quoted string is retrieved by any request, it is loaded into Q-register ". As an example, SQ", immediately after another search, searches again for the same string. This notation is invalid if the specified Q-register contains a number.

The Q-Register Pushdown Stack

There is one Q-register pushdown stack (not one per Q-register) in which the values of Q-registers can be saved. It is organized as a pushdown (Last-In-First-Out) list. It is emptied every time teco waits for a new request string, i.e., a "▣" is typed.

PUSHING A VALUE ONTO THE STACK - [(opposite of])

[q
pushes the current value of Q-register q onto the top of the stack. The Q-register is not affected.

POPPING A VALUE FROM THE STACK -] (opposite of [)

]q
pops the top value on the stack into Q-register q. The previous contents of the Q-register are lost. It is an error to do a] request if the stack is empty.

Loops

The teco editor has the ability to execute a request string repeatedly, just as FORTRAN or PL/I provides do-loops.

LOOPS - < and > (opposite of each other)

<
begins a loop. It is equivalent to n< except that n is set to a very large number that is for all practical purposes infinite.

n<
begins a loop to be executed n times. The value of n and the position of the < in the request string are saved. The value of n must not be negative.

:<,n:<
is similar to < except that errors that occur within the iteration group just terminate the iteration group and the > returns a value. The returned value is -1 if no errors occurred, and it is 0 if the group was terminated by an error. The error message that terminates the loop is not printed.

>
ends a loop. It returns to just after < if the string has not yet been executed n times.

n<...>
executes the string between the angle brackets n times.

TERMINATING A LOOP BEFORE n EXECUTIONS - ;

n;
if n is less than 0, then nothing is done. Otherwise, execution of the current loop is aborted and teco skips to just after the closing >. If n is not specified, the result of the most recent S or N request is used (terminate loop if search failed). The ; request cannot appear outside of a loop.

:;
is similar to ; except that the sense of the test is inverted. If n is less than 0, execution of the current loop is terminated and teco skips to just after the corresponding >.

SPECIAL LOOP FACILITIES - throwing and catching values

F<!label!
provides for a nonlocal transfer of control. F< and > define an iteration group like < and >. From the time that the F< iteration group is entered until the time it is exited, F< sets up a handler to "catch" values "thrown" by the F;/label/ request. If no F;/label/ request with matching-string argument is executed before the F< iteration group is exited, the iteration group returns -1 as a value. If, however, an F;/label/ request is executed (where the label string matches the one in the F<!label!), the execution of all macros and iteration groups encountered since the F< is abandoned, and the F< iteration group returns the numeric argument of the F; request as a value.

:F<!label!
is similar to F< except that if an error is encountered during the execution of the :F< iteration group, the latter returns zero as a value.

nF;/string/
"throws" the numeric value n to the most recent F< or :F< iteration group where the string argument matches the string argument of the F; request. It is an error to execute a F;/string/ request when there is no F< or :F< iteration group in execution with a matching-string argument.

NOTE: These requests provide a method of exiting several nested loops at once. Execution of a F; request terminates the F< loop as well as any contained loops.

Gotos

The teco editor provides the ability to transfer control to a different part of the request string.

GOTO - O (gOto)

O/string/
searches the current macro (or, if we are not in a macro, the request line) for the label !string!. If it is found, teco begins interpreting requests just after the label. If not found, but execution is currently in a macro, the search is repeated in the previous execution level, i.e., the caller of the macro. This is repeated until teco has checked all the way down to the request line typed by the user. Notice that although teco can exit a macro using an O request, it cannot use that request to exit a loop. Only a semicolon (;) can be used to terminate a loop.

Macros

The teco editor has the ability to execute strings of text (macros) other than those read from the user's terminal. The associated requests are listed below:

EXECUTING A MACRO IN A Q-REGISTER - M (Macro)

Mq
: executes the contents of Q-register q as a request string. Notice that if the M request is given any numeric arguments, they are passed to the first request inside the macro. String arguments can be fetched by the :X request.

:Mq
: is similar to the M request except that if issued within a macro, the return from Q-register q causes the invoking macro to return also.

EXECUTING A MACRO IN A FILE - EM (External Macro)

EM/string/
: is similar to the M request except that the request string is found in a file whose entryname is string.teco. This file is looked for in three directories: the working directory, the user's login directory, and the teco library.

OBTAINING A STRING ARGUMENT TO A MACRO

:Xq
: suspends execution of the current macro, returns to its caller to fetch a quoted string into Q-register q, and then restores the macro that was being executed. Notice that each :X request in a macro fetches another quoted string. The U request(s) should be the first request in a macro if one wishes to fetch numeric arguments in a macro.

NOTES

1. Loops cannot cross macro boundaries, i.e., a loop cannot start in one macro and end in another. This does not, however, prohibit the M or EM request from being used within a loop.

2. A macro can modify itself if it is in a Q-register. Notice, however, that the current invocation of the macro is not affected; only future accesses to the Q-register. If the macro is invoked by the EM request, the results of modifying the file are hard to predict as teco reads the request string directly from the file.

3. When a macro is invoked by the EM request, it should be noted that the name of the macro is found in the Q-register named ". Thus several macros can be put in one segment with the first request in the segment being 0Q". (The user must not forget to put all the appropriate names on the segment.)

4. If an M or EM request is given as the last request in one macro, the request is interpreted as a goto rather than a call. Thus, unlimited M's can be done in this manner although there is an implementation-defined limit to the depth of calls.

5. When the teco editor is entered, a macro named start_up is searched for. If it is found, the arguments to teco are put onto the pushdown stack, and the start_up macro is executed. There is a default start_up macro if the user does not provide his own. This macro is described below.

CODING CONVENTIONS FOR MACROS

Since there are only a small number of Q-registers (95), each with a one-character name, there are serious problems in writing a set of macros that are compatible. A set of macros become incompatible if one macro uses a Q-register for long-term storage that any other macro uses at all. There are two ways this effect can be combated. First, by establishing certain coding conventions, and second, by use of a documented macro library. Probably the most important coding convention is the specification of which Q-registers can be used inside a macro for temporary storage. Many library teco macros use the ten Q-registers 1,2,3,4,5,6,7,8,9, and 0 for temporary storage. If one macro calls another macro that destroys the contents of one of these registers, the calling macro can save the value of the Q-register in the pushdown list and then restore it after the other macro has been called.

Fortunately, calling a macro is a very inexpensive operation in teco if the macro is in a Q-register. The EM request is more expensive. This leads to the practice of creating a macro in a macro library that loads a Q-register with a useful macro. When the user realizes that he wants the macro, he gives the EM request that loads the macro he wants into a Q-register, where he can then call it whenever he wishes. It now becomes necessary to have coding conventions that specify which registers can be loaded permanently with macros. Since it should be easy to type the macro names, the lowercase alphabetic letters should be used for this purpose. Sometimes a macro uses a Q-register for long-term storage. If the user does not have to type the name of this Q-register, names that must be escaped are good; otherwise, other special characters can be used. This leaves the uppercase alphabetic letters entirely to the user to use to store intermediate results in editing. Also the special characters -, ,, ., /, space, tab, and newline should be reserved for the user since these are all lowercase on most terminals.

An extremely useful feature of teco is that the last quoted string is loaded into Q-register ". To allow this to continue to be useful, all macros should make sure that Q-register " either contains the last quoted string argument to the macro, if there are any, or contains what it contained before the macro was called. Q-register " can be saved on the pushdown list on entry to a macro and then restored just before leaving the macro. Use of the pushdown list is very inexpensive.

RELATIVE COSTS IN teco

The teco editor stores the buffer in two pieces. The first piece, all the characters from the beginning of the buffer to the current pointer, is stored at the beginning of one buffer segment. The second piece, all the characters from the current pointer to the end of the buffer, is stored at the end of another buffer segment. Inserting text merely adds text to the end of the first buffer segment and increases the number of valid characters in the first buffer segment. Deleting text merely changes the number of valid characters in one or both of the buffer segments. In order to move the pointer, a string copy from one buffer segment to the other is performed unless an unmodified copy of the string already exists in the other buffer. It does not matter to teco which direction the pointer is moved.

Reading a file into an empty buffer causes that file to be used as the buffer until the text is modified. Thus this request string causes an invalid segment fault:

```
HK EI/File/ EC/dl File/$
```

Positioning to the end of the buffer (ZJ) puts all the text into one temporary segment. Thereafter, pointer moves do not actually move text. As long as all the text remains in one temporary segment, pointer moves do not actually move text. An insertion or deletion anywhere but at the end of the text causes the text to be split up.

Each text Q-register is presently kept in its own segment. This means that if a start_up macro loads many Q-registers with macros, entering teco for the first time in a process is somewhat slow since all these segments must be created. The teco command has its own segment manager (get_temp_seg_) that allows it to reuse segments without calling hardcore to create and delete segments when the values of Q-registers are changed. Whenever a string is quoted, or a Q-register loaded with text, a new segment is retrieved from get_temp_seg_ and loaded with the value. If the string that is being loaded into the Q-register is in another Q-register, the new Q-register is just made to point to the same copy of the text in the first Q-register. :IAQB is therefore a very simple operation, as are [(Push) and] (Pop). The feature of keeping the last quoted string in Q-register " lets the user take advantage of this scheme.

If the user wants to write a macro that must do some editing on another file, it is much cheaper if he saves the value of "." and "Z-." , inserts the text to be edited, edits it, writes it out or copies it into a Q-register, and then deletes what he was just editing from the buffer. The net change to the buffer by all these operations is zero, but the text that the user was editing was never moved. This method is much cheaper than storing the entire buffer in one Q-register, the value of the pointer in another, and then using the buffer for the editing within the macro.

There are four ways to transfer control in teco, by the > request, the ; request, the " or :' request, and the O request. Of these, the > request is the fastest, since teco already knows exactly where to transfer it. The ;, ", and :' requests are next, since they merely search from where they are. Although the > request and the ; request cannot change macro levels, the ", and :' requests can. This adds a small expense. The ; and :; requests have to check so that a ; request completely skips over another nested loop and looks beyond it for a >. Similarly, the " transfer skips over nested if statements, as does the :' request. Usually, the matching ' or > is not far from the transfer, so this only causes a short search.

WARNING: The teco editor implements < and > searches very simply. It does not check the semantics of the request string. The request string is searched forward for the first < or >. If a < is encountered, a counter is incremented. If a > is encountered and the counter is zero, the search is complete; otherwise the counter is decremented. Any and all < and > appearing in the searched portion of a request string participate in this process.

O is the most general and most expensive transfer of control in teco. It must search the entire macro from the beginning, then the entire macro that called the present macro, etc., until it finds it or finishes searching the request line and gives an error. Although this is the most expensive transfer, its cost is proportional to the distance of the label from the goto request.

Conditionals

The teco editor has the ability to conditionally execute strings. The " request corresponds to the PL/I statement "if ... then do;". The ' request corresponds to the PL/I statement "end;". " and ' are matched and can be nested.

WARNING: The teco editor implements " and ' searches very simply. It does not check the semantics of the request string. The request string is searched forward for the first " or '. If a " is encountered, a counter is incremented. If a ' is encountered and the counter is zero, the search is complete; otherwise the counter is decremented. Any and all " and ' appearing the searched portion of a request string participate in this process.

The letter following the " determines what test is made.

NUMERIC COMPARISONS - "E (Equals), "N (Not equal), "G (Greater), "L Less than)

m,n"E
if m=n, then execution continues; otherwise, execution skips to just after the corresponding '.

n"E
is identical to n,0"E.

m,n"N
is similar to m,n"E except it tests for m^=n.

n"N
is identical to n,0"N.

m,n"G
is similar to m,n"E except it tests for m>n.

n"G
is identical to n,0"G.

m,n"L
is similar to m,n"E except it tests for m<n.

n"L
is identical to n,0"L.

TESTING FOR A SYMBOL CONSTITUENT - "C (Symbol Constituent)

n"C
if n is the ASCII code for a letter, a digit, or one of the characters ., _, or $; then execution continues. Otherwise, execution skips to the corresponding '.

STRING COMPARISON - "M (Match)

"m/string/
if the specified string appears immediately to the right of the pointer, then execution continues; otherwise, execution skips to just after the corresponding '.

:"m/string/
is similar to "m/string/ except that the sense of the test is inverted.

TERMINATING A CONDITIONAL DO - ' (Matches ")

'
is ignored when executed in normal execution. It is used to close a conditional statement.

:'
transfers to the next ', just as a 1"e does. Since this request looks like a ', it can serve to close a conditional statement. This is useful if an if ... then ... else ... statement is desired. The if expression is a " statement, then the expression is terminated by the :' request and the else expression is terminated by the ' request. (See the warning under "Conditionals" above.)

Reading Input from the User's Terminal - VW (V then Wait for input)

VW
does a V request (presently does nothing on Multics) and then reads one character from the user's terminal. The ASCII value of the character is returned as the value of the request.

:VWq
does a V request and then reads one line from the user's terminal. The line is put into Q-register q. The newline is the last character read in.

Passing a Command to the Command Processor - EC (External Command)

EC/string/
passes the specified string to the Multics request processor for execution.

Invoking an Active Function - EA

EAq/string/
passes the specified string to the command processor's active-function application entry. The result of the active-function application is returned in Q-register q. The specified string must not include square brackets.

Examining a Character in the Buffer - A (ASCII)

nA
The ASCII code for the (.+n)th character in the buffer is returned as the value of the request. n must be specified. (Notice that 1 indicates the character just to the right of the current pointer; 0 indicates the character just to the left.)

Tracing Command Execution - ?

?
turns tracing on. When tracing is on, each request executed by teco is printed on the user's terminal just before it is executed.

??
turns off tracing.

Translating Numbers to ASCII and Vice Versa - \

\
reads the decimal number found to the right of the current pointer and returns its value as the value of the request. The pointer is moved to the right of the number. The number can be signed and can be preceded by any number of blanks or tabs. It is an error if no number is found.

n\
inserts the decimal interpretation of n into the buffer to the left of the current pointer.

m,n\
inserts the decimal interpretation of m into the buffer to the left of the current pointer. The interpretation is padded on the left to be at least n characters wide.

:\
is similar to \ except that it converts to and from octal representations of numbers.

Null Command - W

W
does nothing. It is most useful for throwing away unneeded numeric arguments.

newline character
has the same effect as W.

$
has the same effect as W.

EXAMPLES OF MACROS

Write Macro

This macro writes out the entire buffer into a file whose name is in Q-register *. The pathname is changed by doing:

```
:i*/new_name/
```

EOQ*
assumes that the name of the file we are editing is in Q-register *. It writes out the entire buffer into this file.

A Restart Macro

This macro zeros out the buffer, changes Q-register * to be a new file name and reads the file into the buffer:

```
:x* hk eiq* j
```

:X*
takes one string argument and loads it into Q-register *.

HK
deletes all the text in the current buffer before editing is restarted.

EIQ*
reads the new file into the buffer.

J
puts the pointer at the beginning of the buffer.

Start-Up Macro

This macro only uses the first and second argument to teco. It treats it as a file name, loads it into Q-register * and reads the file into the buffer. It also loads the writing macro into Q-register w:

```
]1 :iw¦eoq*¦ifqwq1"n ]* eiq* j q1-1"q ]*'
```

`]1`
pops the top item off the pushdown list and puts it into Q-register 1. This is the number of arguments teco was called with.

`:iw¦eoq*¦`
loads Q-register w with the write macro given in the above example.

`:ifqw`
loads Q-register f with a copy of the contents of Q-register w.

`q1"n`
if the contents of Q-register 1 are not zero, then execute the following statements; otherwise, transfer to the matching '.

`]*`
pops the first argument to teco off the pushdown list and into Q-register *.

`eiq*`
reads the file whose name is the contents of Q-register *, into the buffer.

`j`
moves the pointer to the beginning of the buffer.

`q1-1"q`
if the value of the expression (q1-1) is greater than zero, execute the following; otherwise, skip to the matching '.

`]*`
pops the next (second) argument off the pushdown list and into Q-register *.

`'`
matches q1-1"q. End of request string for second argument.

`'`
matches q1"n. End of request string for processing arguments.

Substitute Macro

This macro takes two string arguments. The first string argument is searched for, then it is deleted, and the second string inserted.

```
:x1 :x2 sq1 -q1d g2
```

`:x1`
loads the first string argument into Q-register 1.

:x2
loads the second string argument into Q-register 2.

sq1
searches for the first string.

-q1d
deletes the first string when it is found. (Could also be -q"d.)

g2
replaces the string found with the second string argument.

When the macro returns Q-register, 1 and 2 contain the first and second strings, respectively. Q-register " contains the second quoted string.

A teco SUMMARY

NAME USE AND EXPLANATION

a
nA
The value of the request is the ASCII code for the (.+n)th character in the buffer.

b
B
The value of this symbol is always zero.

c
nC
moves the pointer n characters to the right. If n is omitted, 1 is assumed.

:C
n:C
is similar to c, only error messages are not printed.

d
D
deletes the one character to the right of the pointer.

+nD
deletes n characters to the right of the pointer.

-nD
deletes n characters to the left of the pointer.

ea
EAq/string/
passes the string to the Multics active-function command process; result is put in Q-register q.

ec
EC/request/
passes the string to the Multics command processor.

ei
EI/file/
reads the file into the buffer to the left of the current pointer.

:ei
:EI/file/
is similar to EI, only no errors are possible. Returns 0 if read fails; -1 if it succeeds.

em
EM/macro_name/
searches for the file macro_name.teco, first in the working directory, then the login directory, then the teco library. If found, it executes it as a macro.

eo
EO/file_name/
writes out the entire buffer into the file specified.

+nEO/file_name/
writes out the next n lines.

(0 or -n)EO/file_name/
writes out the last n lines.

m,nEO/file_name/
writes out the (m+1)th through the nth characters.

eq
EQ
returns to its caller.

NAME	USE AND EXPLANATION
g	GQ inserts the text contained in Q-register q into the buffer to the left of the pointer. If Q-register q contains a number, it is converted to a character string and inserted.
h	H is equivalent to 0,Z. It is the only symbol that has two values.
i	I/string/ inserts the quoted string to the left of the pointer.
	nI n is the ASCII code for a letter that is inserted.
:i	:Iq/string/ inserts the quoted string into Q-register q.
	n:Iq inserts the single character whose code is n into register q.
j	nJ moves the pointer to the right of the nth character in the buffer. If n is omitted, 0 is assumed.
:j	n:j is similar to j, only no errors.
k	K deletes the rest of the current line from the buffer.
	+nK deletes the next n lines from the buffer.
	(0 or -n)K deletes the last n lines from the buffer.
	m,nK deletes the (m+1)th through the nth characters from the buffer.
l	L moves the pointer to the beginning of the next line.
	+nL moves the pointer to the beginning of the next nth line.
	(0 or -n)L moves the pointer to the beginning of the last nth line.
:l	:L moves the pointer to the end of the current line.
	+n:L moves the pointer to the end of the next (n-1)th line.
	(0 or -n):L moves the pointer to the end of the 1st (n+1)th line.

NAME	USE AND EXPLANATION
m	m,nMq/string1//string2/.../stringn/ starts executing the text in Q-register q as a macro. m and n are numeric arguments to the first request in the macro. string1 through stringn are string arguments to the macro that can be retrieved with the :X request. EM also takes all these arguments.
:m	m,n:M/string1//string2/.../stringn/ is similar to m only when control returns from Q-register q, macro containing :m request returns as well.
n	N/string/ searches from the current pointer to the end of the buffer for the regular expression "string."
:n	:N/string/ is similar to N/string/ except that :N returns a value. It returns 0 if the string is not found; -1 if it is.
o	O/label/ transfers control to just after label in the current macro, its caller, etc., or the request string.
P	Pq appends texts to Q-register q.
q	Qq the value of this request is the value of Q-register q if it is a numeric Q-register or the number of characters in Q-register q if it contains text. This request can also replace any quoted string if Q-register q contains text. The contents of the Q-register are used as the quoted string.
r	R moves the pointer one character to the left. nR moves the pointer n characters to the left.
:r	R is similar to R only no errors are possible.
s	S/string/ searches from the current pointer to the end of the buffer for "string"; if found, it moves the pointer to the right of the string. +nS/string/ searches for n occurrences of the string. Moves the pointer to the right of the nth occurrence. -nS/string/ searches for n occurrences of "string" from the current pointer to the beginning of the file. If found, it moves the pointer to the left of the nth occurrence. +m,+nS/string/ only searches from the current pointer to the beginning of the next mth line.

NAME USE AND EXPLANATION

(0 or -m),-nS/string/
only searches from the current pointer to the beginning of the last mth line.

: :S/string/
takes arguments in all the ways S does, except that if S does not find the string, it prints out an error message and returns to teco request level. :S does not. Instead, :S has the value -1 if the search succeeds and 0 if the search fails.

t T
prints out the rest of the current line of the terminal.

+nT
prints out the buffer from the current pointer to the beginning of the next nth line.

(0 or -n)T
prints the buffer from the beginning of the last nth line to the current pointer.

m,nT
prints the (m+1)th through the nth characters of the buffer.

:t :T/string/
prints the quoted string on the terminal.

u Uq
sets Q-register q to a very large positive number.

nUq
sets Q-register q to n.

m,nUq
sets Q-register q to n and returns m as its value. This may be used inside a macro to get the numeric arguments to the macro.

vw VW
when this request is executed, one character is read from the terminal. The ASCII code for the character read is the value of the VW request.

:vw :VWq
reads in an entire line from the terminal and puts it into Q-register q. The newline character is the last character in the register.

w W
is used for throwing away unwanted numeric arguments.

x Xq
loads the rest of the current line into Q-register q.

+nXq
loads Q-register q with everything from the current pointer to the beginning of the next nth line.

(0 or -n)Xq
loads Q-register q with everything from the beginning of the last nth line to the current pointer.

NAME	USE AND EXPLANATION
	m,nXq loads Q-register q with everything from the (m+1) character to the nth character.
:x	:Xq loads Q-register q with the next string argument to the macro we are executing in.
z	Z is the total number of characters in the buffer. ZJ moves the pointer to the right of the last character in the buffer.
%	%q if Q-register q contains a numeric value, this request increments the register by 1. The value of the request is the new value of the Q-register.
$	$ throws away its arguments and does nothing.
newline	newline throws away its arguments and does nothing.
?	? turns tracing on.
??	?? turns tracing off.
\	\ is the decimal number immediately to the right of the pointer. It moves the pointer to just after the number.
	n\ inserts the decimal representation of n to the left of the pointer.
	m,n\ inserts the decimal representation of m to the left of the pointer. The representation is padded on the left to be at least n characters wide.
:\	:\ is similar to \ except the values are octal and not decimal.
:[	[q pushes the contents of Q-register q onto the pushdown list.
]	]q pops the top element off the pushdown list and into Q-register q.
<	< marks the place in the request string that is transferred to by the > request. This loop can only be exited by the ; request.
:<	:< is similar to < except inhibits errors within the loop and causes > to return a value.

NAME USE AND EXPLANATION

> >
transfers control to just after the last < request executed and decrements the loop count. If enough loops have occurred, this request does nothing. Nested loops are allowed.

; ;
if the last :s,n, *s, or :n request was unsuccessful, transfers to just after the next > and exits the present loop; otherwise, does nothing.

n;
if n is positive, transfers control to just after the next > request and exits the present loop; otherwise, does nothing.

:; :;
is similar to ; except that the sense of the test is inverted.

"C n"C
if n is the ASCII code for a letter, a digit, ., _, or $ does nothing. Otherwise, it transfers to just after the matching '.

"e m,n"E
if m=n, then "E does nothing; otherwise, transfers to just after the next '.

n"E
if n=0, then "E does nothing; otherwise, it transfers to just after the matching '.

"g m,n"G
if m>n, then "G does nothing; otherwise, it transfers to just after the matching '.

n"G
if n>0, then "G does nothing; otherwise, it transfers to just after the matching '.

"l m,n"L
if m<n, then "L does nothing; otherwise, it transfers to just after the matching '.

n"L
if n<0, then "L does nothing; otherwise, it transfers to just after the matching '.

"n m,n"N
if m^=n, then "N does nothing; otherwise, it transfers to just after the matching '.

n"N
if n^=0, then "N does nothing; otherwise, it transfers to just after the matching '.

"m "m/string/
if characters immediately to the right of the pointer are equal to string, "m does nothing; otherwise; it transfers to just after the matching '.

NAME	USE AND EXPLANATION
:"m	:"m/string/ if the characters immediately to the right of the pointer are not equal to string, "m does nothing; otherwise, it transfers to just after the matching '.
'	' marks the location a " request transfers to. If executed as a request, it does nothing.
:'	:' marks the location a " request transfers to. If executed as a request, it transfers to just after the next '.
!	!label! is a label; it is ignored if it is executed.
.	. its request is the value of the current pointer.
=	= prints a newline.
	n= prints n in decimal followed by a newline.
	m,n= prints the value of m followed by a space, followed by the value of n, followed by a newline. The values are printed in decimal.
:=	m,n:= is similar to = except that the values are printed in octal.
F F;	used to define throw-catch loops.

Name: teco_error

The teco_error command prints the long form of a teco error message given the short term.

Usage

```
declare teco_error entry (char(*));

call teco_error (name);
```

where name is the short form of a teco error message. (Input)

Name: teco_ssd

The teco_ssd command allows the user to specify a directory for teco to search when trying to find a teco macro to execute. The directory so specified is searched instead of the user's directory.

Usage

teco_ssd path

where path is the absolute pathname of a directory to be searched by teco_get_macro_ instead of the user's home directory.

Name: test_archive

The test_archive command is a library maintenance tool that checks an archive segment for archive format errors and other inconsistencies. It is run weekly to check all archive segments in the online libraries.

Usage

test_archive paths

where paths are the pathnames of the archive segments in question (without the suffix archive).

Name: vfile_find_bad_nodes

As a command, the vfile_find_bad_nodes command examines a vfile_ keyed file to determine whether the vfile_ MSF components that contain keys are in a consistent state. The keys in a keyed file are maintained in a tree structure in which each node of the tree is stored in a separate page of an MSF component. The consistency checks that are performed are summarized below. Nodes reported as bad by this heuristic are almost certainly damaged.

Usage

vfile_find_bad_nodes {pathname} {-control_args}

Node-branch checks:

1) Is this a freed node? If so, skip further checks.
2) Are there any branches (keys) in this node? If not, skip further checks.
3) Is branch_count > 313? If so, node is bad, because space in a page limits a node to having, at most, 313 one-character keys.
4) Is branch_count < 0? If so, node is bad.

Key-region checks:

5) Is start_of_key_region > 4096? If so, node is bad, because the character position of the first key must lie within the node page.
6) Does start_of_key_region overwrite the branch array? If so, node is bad, because keys have overwritten the array of branches in the node.
7) Is scattered_free_key_space > 4096-start_of_key_region? If so, node is bad, because the count of unused space within the key region is greater than the size of the key region itself.
8) Is scattered_free_key_space < 0? If so, node is bad.

Key-location checks:

9) Does any branch declare its key to begin prior to start_of_key_region? If so, node is bad.
10) Does any branch declare its key to extend beyond end of node page? If so, node is bad.

Key-overlap check:

11) Does the storage for any key overlap storage for another key? If so, node is bad. Note that this test is somewhat time-consuming.

Key-order check:

12) Are the keys within the node ordered in increasing ASCII collating sequence? If not, the node is bad. Note that this test is somewhat time-consuming.

where:

1. pathname
   is path of the indexed file whose nodes are to be checked.

2. control_args
   can be chosen from the following:

   -io_switch STR, -isw STR
   identifies an I/O switch that is already attached to the indexed file to be checked. The switch may be closed. If open, it must be opened for keyed_sequential_input.

   -request_loop, -rql
   enters the request loop when bad nodes are found.

   -no_request_loop, -nrql
   simply prints information about the bad nodes, and then continues checking.

   -check MODES, -ck MODES
   enables only the types of checking given in the MODES string. See "List of Modes" below. (Default: -check default.)

List of modes:

node_branch, ^node_branch
performs node-branch checking, as described above.

key_region, ^key_region
performs key-region checking, as described above.

key_loc, ^key_loc
performs key-location checking, as described above.

key_overlap, ^key_overlap
performs key-overlap checking, as described above.

key_order, ^key_order
performs key-order checking, as described above.

default
is a shorthand way of enabling checks that can be quickly performed. It is equivalent to node_branch,key_region,key_loc. The settings of other modes are not affected.

all
is a shorthand way of enabling all possible checking. It it equivalent to node_branch,key_region,key_loc,key_overlap,key_order.

Request loop operation:

When a bad node is found, its location is printed out, followed by the number of branches in the node, its low_key_pos, and its unused key space (scat_space). Then a request loop is entered that allows the user to continue checking other nodes, to quit further checking, or to enter a totaling loop that counts the number of damaged nodes in the current component without printing their statistics. The request:

```
..ds node_seg node_offset count -ch
```

is a useful thing to do. Type "c" in the request loop to continue checking the next node.

List of requests:

continue, c
: continues searching for damaged nodes.

quit, q
: stops further processing, reporting total of damage found so far.

total, tt
: for remainder of this MSF component, stops reporting information about each damaged node and just counts the damaged nodes in this component.

.
: gives name and version number of this program, plus pathname or I/O switch of file being examined.

.. command_line
: escapes a command_line to Multics command level.

?
: lists available requests.

Name: vfile_find_bad_nodes

As an active function, the vfile_find_bad_nodes command returns true if bad nodes are found, false otherwise. Normal diagnostic messages are still printed.

Usage

```
[vfile_find_bad_nodes {pathname} {-control_args}]
```

Notes

Either a pathname argument or -io_switch must be given to identify the file to be checked. When invoked as an active function, -no_request_loop is the default. When invoked as a command, -request_loop is the default.

Examples

```
     ! vfile_find_bad_nodes >sc1>perm_syserr_log -ck default,key_order
[1]
[2]    Begin checking free node list (node_ptr = 473|314000).
[3]    Found 59 undamaged free nodes.  Processing continues.
[4]
[5]    Begin checking component 0, node:
[6]     25 50 75 100 125 150 175 200 225 250
[7]    Begin checking component 6, node:
[8]     25 50 75 100 125 150 175 200 225 250
[9]    No damaged nodes.
```

Lines [2-3] of the output show that the key containing components of the file contain some unused node pages. These free node pages are catalogued, and no further checking occurs on them.

Line [5] shows the beginning of testing in component 0 of the file. Each component contains 255 pages, numbered from 1 to 255. The numbers printed on line [6] show the progress of checking through these pages (i.e., 25 is printed after the first 25 pages are checked, 50 is printed when 50 pages are checked, etc).

Line [9] is printed when no damage is found.

```
      ! vfile_find_bad_nodes user_reg -check all
[1]
[2]     Begin checking component 0, node:
[3]
[4]     ERROR 13 in Comp 0, node 5 (node_ptr = 464|10000)
[5]     Key(2) > Key(3)
[6]         branch_count = 203 keys
[7]         start_of_key_region = char position 2470
[8]         key_space = 1626 chars,
[9]         scattered_free_key_space = 0 chars
[10]    vfile_find_bad_nodes:  ! c
[11]
```

```
[12]    ERROR 1 in Comp 0, node 6 (node_ptr = 464|12000)
[13]    branch_count > 313
[14]        branch_count = 9420723823 keys
[15]        start_of_key_region = char position 15733420590
[16]        key_space = -15733416494 chars,
[17]        scattered_free_key_space = 11171849844 chars
[18]    vfile_find_bad_nodes:  ! tt
[19]     25 50 75 100 125 150 175 200 225 250
[20]    4 bad nodes in comp 0
[21]
[22]    4 key nodes were damaged.
```

Line [2] shows beginning of checking on component 0 of another file. The error message on lines [4-9] shows a key-order error in node 5 of component 0.

Line [10] shows the request loop. The c request was issued to continue checking.

Lines [12-17] show a second error, in node 6 of component 0. In this error, more than 313 keys were found in the node.

Line [18] shows issuing the tt request to simply get a count of the remaining errors in component 0. The count is shown in line [20], 4 bad nodes, which includes the two for which errors were shown plus two others.

Line [22] prints a summary of all checking, showing that a total of found-damaged nodes were found among all of the components of the file.

Name: vtoc_pathname

The vtoc_pathname command is used to determine the pathname of a segment from the location of its VTOC entry (VTOCE). The location of the VTOCE is specified by giving its volume name (or physical volume table index, if known) and an index into the VTOC of that volume.

The vtoc_pathname command requires access to the phcs_ gate, since it must copy directories.

Usage

```
vtoc_pathname volname vtocx {-control_arg}
or
vtoc_pathname pvtx vtocx {-control_arg}
```

where:

1. volname
   is the physical volume name of the volume on which the VTOCE resides. This volume must be mounted and must be part of a mounted logical volume.

2. pvtx
   is the physical volume table index of the volume on which the VTOCE resides, if known. It must be given in octal.

3. vtocx
   is the VTOC index of the VTOCE. It must be given in octal.

4. control_arg
   can be -brief or -bf to suppress the printing of an error message when the VTOCE is free.

Note

The user's process must have status access to each of the containing directories of the segment in question. The command supplies "-NO-ACCESS-" as the entryname at the level at which further access is necessary, if needed. If one of the containing directories specified in the VTOCE does not exist in its containing directory, "-NOT-LISTED-" is supplied as the entryname at that level. The command supplies "-????-" as the entryname at any level below that at which either of the problems mentioned occurs.

Name: vtocx_to_record

The vtocx_to_record command converts an octal VTOCE to index a Multics record number and sector offset.

Usage

vtocx_to_record vtoc_index {device_name}

where:

1. vtoc_index
   is the octal VTOCE index.

2. device_name
   is a valid device name (e.g., "m400", "m451").

Name: write_mst

The write_mst command is used to write Multics system tapes. This is usually useful only for Multics tapes for BOS, either complete BOS tapes, or tapes to be used with the BOS LOADDM command (see the Multics Operators' Handbook, Order No. AM81). Each segment is written with the entryname supplied as its name on the tape. Its bit count, current length, and actual length are derived from its bit count in the storage system. Other than the bit count and current length, its SLT entry is given as zero. The various control arguments allow the user to specify collection marks on the tape and text-link definitions decoding of the segments.

Usage

write_mst reel_id {-control_args} names

where:

1. reel_id
   is the reel identifier number of the tape to be written.

2. namei
   is the pathname of a segment to be written on the tape. The manner in which the segment is written is determined by the control arguments immediately preceding namei. If either or both of the -text or -link control arguments precede namei, namei must specify a standard object segment.

3. control_args
   may be chosen from the following:

   -collection, -col
   writes sequential collection marks every time it appears in the command line.

   -text, -tx
   writes only the text of the segment specified by namei rather than the whole segment.

   -link, -lk
   writes two separate segments for each namei argument:
   name>.link contains the separated linkage
   name>.defs contains the separated definitions

   -stop, -sp
   calls debug before the segment is written out so that the user may modify the SLT entry in any desired way. A message giving the location of the segment and the SLT entry is printed out before such a call is made.

Note

If a namei argument is not preceded by any control arguments, the segment specified by namei is written on the tape in its entirety.

Example

A common use of this command is the preparation of tapes for loading via the BOS LOADDM command. For example, to write new versions of the DUMP and PATCH programs on tape 26105, type:

```
write_mst 26105 -tx >udd>Opr>bos_dir>dump -tx >udd>Opr>bos_dir>patch
```

Notice that only the -text (-tx) control argument is used; BOS expects only the text of each segment and no collection marks.

SECTION 3

SUBROUTINE DESCRIPTIONS

This section contains descriptions of Multics subroutines, arranged in alphabetical order. The format of each subroutine description is the same as the format of the Multics subroutines described in the MPM Subroutines. See Section 2 of the MPM Subroutines for detailed information regarding this format.

Name: abbrev_

The abbrev_ subroutine provides a means of expanding abbreviations in command lines and changing data in and extracting data from the profile segments used by the abbrev command. All of the features of the command itself are available and a simple expand entry point is provided for returning expanded command lines.

Entry: abbrev_$abbrev_

This entry point is used to expand and execute a command line. The command line can be an abbrev request line, as recognized by the abbrev command documented in the MPM Commands. An abbrev request line can be used to add and delete abbreviations and change the modes of operation of abbrev. The abbrev command need not be invoked in the process before the abbrev_ subroutine can be called.

Usage

```
declare abbrev_$abbrev_ entry (ptr, fixed bin, fixed bin);

call abbrev_$abbrev_ (line_ptr, line_len, code);
```

where:

1. line_ptr (Input)
   is a pointer to an aligned character string to be interpreted as a command line or an abbrev request line.

2. line_len (Input)
   is the number of characters in the input line.

3. code (Output)
   is a standard status code returned by the command processor.

Entry: abbrev_$expanded_line

This entry point returns an expanded version of an input string. See the description of the abbrev command in the MPM Commands for a discussion of abbrev expansion.

Usage

```
declare abbrev_$expanded_line entry (ptr, fixed bin, ptr, fixed bin, ptr,
     fixed bin);

call abbrev_$expanded_line (in_ptr, in_len, space_ptr, space_len, out_ptr,
     out_len);
```

where:

1. in_ptr (Input)
   is a pointer to an aligned character string to be expanded.

2. in_len (Input)
   is the number of characters in the input string.

3. space_ptr (Input)
   is a pointer to a work space where the expanded character string can be placed.

4. space_len (Input)
   is the number of characters available in the work space.

5. out_ptr (Output)
   points to the expanded string.

6. out_len (Output)
   is the number of characters in the expanded string.

Notes

If the length of the expanded string exceeds the length of the work space provided, the expanded line is allocated in system_free_n_, where n is the current ring. It is the user's responsibility to free this storage when it is no longer needed.

The space_ptr pointer should not point to the same string as in_ptr since expansion is done directly into the work space.

Entry: abbrev_$set_cp

This entry point sets up a different command processor to be called by the abbrev_ subroutine after a command line is expanded. Its argument is an entry. If the first pointer in the entry is null, the command processor to be called is command_processor_.

Usage

```
declare abbrev_$set_cp entry (entry);

call abbrev_$set_cp (cp_entry);
```

where cp_entry is a command processor entry point.

Examples

The code:

```
chars = ".a ab1 " || char_string;
call abbrev_ (addr (chars), length (chars), code);
```

sets up ab1 as an abbreviation for the character string stored in chars.

The code:

```
chars = "delete foo; logout";
call abbrev_ (addr (chars), length (chars), code);
```

calls the command processor with the string arrived at by expanding the command line:

```
delete foo; logout
```

That is, if foo is an abbreviation for *.pl1, the command processor is given the line:

```
delete *.pl1; logout
```

to be executed.

The code:

```
chars = some_string;
cp    = addr(chars);
xcp   = addr(xchars);
call abbrev_$expanded_line (cp, length (chars),
     xcp, length (xchars), out_ptr, out_len);
```

copies some_string into chars and leaves the expanded version in xchars, unless the length of the expanded version is greater than length(chars). In that case the expanded version is in allocated storage. In either case, out_ptr points to the expanded version and out_len is its length.

Name: ask_

The ask_ subroutine provides a flexible terminal input facility for whole lines, strings delimited by blanks, or fixed-point and floating-point numbers. Special attention is given to prompting the terminal user.

Entry: ask_$ask_

This entry point returns the next string of characters delimited by blanks or tabs from the line typed by the user. If the line buffer is empty, the ask_ subroutine formats and types out a prompting message and reads a line from the user_input I/O switch.

Usage

```
declare ask_ entry options (variable);

call ask_ (ctl, ans, ioa_args);
```

where:

1. ctl (Input)
   is an ioa_ control string (char(*)) in the same format as that used by the ioa_ subroutine (described in the MPM Subroutines).

2. ans (Output)
   is the return value (char(*)).

3. ioa_args (Input)
   are any number of arguments to be converted according to ctl.

Entry: ask_$ask_clr

This entry point clears the internal line buffer. Because the buffer is internal static, the input of one program can accidentally be passed to another unless the second begins with a call to this entry point. If a value typed by the user is incorrect and if the program wishes to ask for the line to be retyped, the ask_$ask_clr entry point can also be called.

Usage

```
declare ask_$ask_clr entry;

call ask_$ask_clr;
```

There are no arguments.

Entry: ask_$ask_int

This entry point works the same as the ask_$ask_ entry point except that the next item on the line must be a number. An integer value is returned. Numbers can be fixed point or floating point, positive or negative. A leading dollar sign or a comma is ignored. If the value typed is not a number, the program types:

```
"string" nonnumeric.  Please retype:
```

and waits for the user to retype the line.

Usage

```
declare ask_$ask_int entry options (variable);

call ask_$ask_int (ctl, int, ioa_args);
```

where:

1. ctl (Input)
   is as above. If a period is typed, zero is returned.
2. int (Output)
   is the return value (fixed bin).
3. ioa_args (Input)
   are as above.

Entry: ask_$ask_flo

This entry point works like the ask_$ask_int entry point except that it returns a floating value.

Usage

```
declare ask_$ask_flo entry options (variable);

call ask_$ask_flo (ctl, flo, ioa_args);
```

where:

1. ctl (Input)
   is as above.

2. flo (Output)
   is the return value (float bin).

3. ioa_args (Input)
   are as above.

Entry: ask_$ask_yn

This entry point works like the ask_$ask_int entry point except that it returns a value of "yes" or "no." Its arguments are the same as those used with the ask_$ask_int entry point.

Usage

```
declare ask_$ask_yn entry options (variable);

call ask_$ask_yn (ctl, ans, ioa_args);
```

where:

1. ctl (Input)
   is as above.

2. ans (Input)
   is a value of "yes" or "no" if such a value was present.

3. ioa_args (Input)
   are as above.

Entry: ask_$ask_line

This entry returns the remainder of the line typed by the user. Leading blanks are removed. If there is nothing left on the line, the program prompts and reads a new line.

Usage

```
declare ask_$ask_line entry options (variable);

call ask_$ask_line (ctl, line, ioa_args);
```

where:

1. ctl (Input)
   is as above.

2. line (Output)
   is the return value (char(*)).

3. ioa_args (Input)
   are as above.

Entry: ask_$ask_c

This entry point tests to determine if there is anything left on the line. If so, it returns the next symbol, as in the ask_$ask_ entry point, and sets a flag to 1. Otherwise, it sets the flag to 0 and returns.

Usage

```
declare ask_$ask_c entry (char(*), fixed bin);

call ask_$ask_c (ans, flag);
```

where:

1. ans (Output)
   is the next symbol, if any.

2. flag (Output)
   is the symbol flag. Its value can be:
   1 if the symbol is returned
   0 if there is no symbol

Entry: ask_$ask_cint

This entry point is a conditional entry for integers. If an integer is available on the line, it is returned and the flag is set to 1. If the line is empty, the flag is set to 0. If there is a symbol on the line, but it is not a number, it is left on the line and the flag is set to -1.

Usage

```
declare ask_$ask_cint entry (fixed bin, fixed bin);

call ask_$ask_cint (int, flag);
```

where:

1. int (Output)
   is the returned value, if any.

2. flag (Output)
   is the int flag. Its value can be:
   1 if int is returned
   0 if the line is empty
   -1 if there is no number

Entry: ask_$ask_cflo

This entry point works like the ask_$ask_cint entry point but returns a floating value, if one is available.

Usage

```
declare ask_$ask_cflo entry (float bin, fixed bin);

call ask_$ask_cflo (flo, flag);
```

where:

1. flo (Output)
   is the returned value, if any.

2. flag (Output)
   is the flow flag. Its value can be:
   0 if the line is empty
   1 if the value is returned
   -1 if it is not a number

Entry: ask_$ask_cline

This entry point returns any part of the line that remains. A flag is set if the rest of the line is empty.

Usage

```
declare ask_$ask_cline entry (char(*), fixed bin);

call ask_$ask_cline (line, flag);
```

where:

1. line (Output)
   is the returned line, if any.

2. flag (Output)
   is the line flag. Its value can be:
   1 if the line is returned
   0 if the line is empty

Entry: ask_$ask_cyn

This entry point works like the ask_$ask_cint entry point except that it returns a value of "yes" or "no" if one is available.

Usage

```
declare ask_$ask_cyn (char(*), fixed bin);

call ask_$ask_cyn (ans, flag);
```

where:

1. ans (Output)
   is a value of "yes" or "no" if such a value is present.

2. flag (Output)
   is the yn flag. Its value can be:
   1 if a "yes" or "no" value is returned
   0 if the line is empty
   -1 if the next value on the line is not "yes" or "no"

Entry: ask_$ask_n

This entry point scans the line and returns the next symbol without changing the line pointer. A call to the ask_$ask_ entry point later returns the same value.

Usage

```
declare ask_$ask_n entry (char(*), fixed bin);

call ask_$ask_n (ans, flag);
```

where:

1. ans (Output)
   is the returned symbol, if any.

2. flag (Output)
   is the ans flag. Its value can be:
   0 if the line is empty
   1 if the symbol is returned

Entry: ask_$ask_nint

This entry point scans the line for integers. The second argument is returned as -1 if there is a symbol on the line but it is not a number, 1 if successful, and 0 if the line is empty.

Usage

```
declare ask_$ask_nint entry (fixed bin, fixed bin);

call ask_$ask_nint (int, flag);
```

where the arguments are the same as in the ask_$ask_cint entry point.

Entry: ask_$ask_nflo

This entry point scans the line for floating point numbers.

Usage

```
declare ask_$ask_nflo entry (float bin, fixed bin);

call ask_$ask_nflo (flo, flag);
```

where the arguments are the same as in the ask_$ask_cflo entry point.

Entry: ask_$ask_nline

This entry point initiates a scan of the rest of the line.

Usage

```
declare ask_$ask_nline entry (char(*), fixed bin);

call ask_$ask_nline (line, flag);
```

where the arguments are the same as the ask_$ask_cline entry point.

Entry: ask_$ask_nyn

This entry point returns the next symbol, if it is a "yes" or "no" value, without changing the line pointer. The arguments are the same as those used with the ask_$ask_cint entry point.

Usage

```
declare ask_$ask_nyn entry (char(*), fixed bin);

call ask_$ask_nyn (ans, flag);
```

where:

1. ans (Output)
   is a value of "yes" or "no" if such a value is present.

2. flag (Output)
   is the yn flag. Its value can be:
   1 if a "yes" or "no" value is returned
   0 if the line is empty
   -1 if the next value on the line is not "yes" or "no."

Entry: ask_$ask_setline

This entry point sets the internal static buffer for the ask_ subroutine to the given input line so that the line can be scanned.

Usage

```
declare ask_$ask_setline entry (char(*));

call ask_$ask_setline (line);
```

where line is the line to be placed in the ask_ buffer. Trailing blanks are removed from line. A carriage return is optional at the end of line. (Input)

Entry: ask_$ask_prompt

This entry point deletes the current contents of the internal line buffer and prompts for a new line. The line is read in and the entry returns.

Usage

```
declare ask_$ask_prompt entry options (variable);

call ask_$ask_prompt (ctl, ioa_args);
```

where:

1. ctl (Input)
   is a control string (char(*)) similar to that typed by the ioa_ subroutine.

2. ioa_args (Input)
   are any number of arguments to be converted according to ctl.

Name: copyright_notice_

The copyright_notice_ subroutine adds (and optionally deletes) copyright notices to source-program segments.

Usage

```
declare copyright_notice_ entry (char(*) aligned, char(*) aligned,
     fixed bin(35));

call copyright_notice_ (dir_name, entryname, code);
```

where:

1. dir_name (Input)
   is the pathname of the directory containing the segment to be modified.

2. entryname (Input)
   is the entryname of the segment.

3. code (Output)
   is a standard status code.

Operation

The copyright_notice_ subroutine extracts the language suffix from its second argument and searches the notice directory for segments named suffix.Z and suffix.Z_delete, where Z is the string copyright unless changed by a call to copyright_notice_$set_suffix.

If a delete notice exists and is in the segment, it is removed. If the segment does not contain a copy of the new notice, it is added at the top of the segment or following an initial percent-semicolon character string.

If no notice segments are found for the given language type, the error code error_table_$typename_not_found is returned.

Entry: copyright_notice_$set_suffix

This entry point sets the name of the copyright notice segments. The default is suffix.copyright and suffix.copyright_delete where suffix is the language suffix.

Usage

```
declare copyright_notice_$set_suffix entry (char(*));

call copyright_notice_$set_suffix (x);
```

where x (Input) is the new suffix.

Entry: copyright_notice_$test

This entry sets the directory to be searched for copyright notice segments. The default is >ldd>include.

Usage

```
declare copyright_notice_$test entry (char(*));

call copyright_notice_$test (d);
```

where d (Input) is the directory to be searched for copyright notices.

Name: create_ips_mask_

The create_ips_mask_ subroutine returns a bit string that can be used to disable specified ips interrupts (also known as ips signals).

Usage

```
declare create_ips_mask_ entry (ptr, fixed bin, bit(36) aligned);

call create_ips_mask_ (array_ptr, lng, mask);
```

where:

1. array_ptr (Input)
   is a pointer to an array of ips (interprocess signal) names that are char(32) aligned.

2. lng (Input)
   is the number of elements in the above array.

3. mask (Output)
   is a mask that disables all of the ips signals named in the array pointed to by array_ptr. (See "Notes" below.)

Notes

If any of the names are not valid ips signal names, the condition create_ips_mask_err is signalled.

If the first name in the array is -all, then a mask is returned that masks all interrupts.

Currently, the allowed ips names are:

```
quit
cput
alrm
neti
sus_
trm_
wkp_
```

The returned mask contains a "0"b in the bit position corresponding to each ips name in the array and a "1"b in all other bit positions. The bit positions are ordered as in the above list. It should be noted that it is necessary to complement this mask (using a statement of the form "mask = ^mask") in cases where the requirement is for a mask with "1" bits corresponding to specified interrupts. An ips mask is used as an argument to the following entry points: hcs_$reset_ips_mask, hcs_$set_automatic_ips_mask, and hcs_$set_ips_mask.

Name: datebin_

The datebin_ subroutine has several entry points to convert clock readings into binary integers (and vice versa) representing the year, month, day, hour, minute, second, current shift, day of the week, number of days since January 1, 1901, and the number of days since January 1 of the year indicated by the clock.

All the arguments found in the datebin_ entry points are listed and defined below. Clock readings are Multics Greenwich mean time (GMT) and all other arguments represent local time.

1. clock
   is a calendar clock reading with the number of microseconds since 0000 GMT January 1, 1901.

2. absda
   is the number of the days the clock reading represents (with January 1 = 1).

3. mo
   is the month (1 - 12).

4. da
   is the day of the month (1 - 31).

5. yr
   is the year (1901 - 1999).

6. hr
   is the hour of the day (0 - 23).

7. min
   is the minute of the hour (0 - 59).

8. sec
   is the second of the minute (0 - 59).

9. wkday
   is the day of the week (1 = Monday, 7 = Sunday).

10. s
    is the shift, as defined in installation_parms.

11. dayr
    is the day of the year (1 - 366).

12. datofirst
    is the number of days since January 1, 1901, up to, but not including January 1 of the year specified.

13. oldclock
    is a calendar clock reading in microseconds since January 1, 1901, 0000 GMT.

14. ZZ
    is the desired hour and minutes expressed as hhmm in decimal (e.g., 1351).

15. newclock
    is the time the shift changes next after clock.

16. shift
    is the current shift at time clock.

17. newshift
    is the shift that begins at time newclock.

If arguments passed to datebin_ are not in the valid range, the returned arguments are generally 0 (in certain cases, no checking should be done).

Entry: datebin_$datebin

This entry point returns the month, day, year, hour, minute, second, weekday, shift, and number of days since January 1, 1901, given a calendar clock reading.

Usage

```
declare datebin_ entry (fixed bin(71), fixed bin, fixed bin, fixed bin,
     fixed bin, fixed bin, fixed bin, fixed bin, fixed bin, fixed bin);

call datebin_ (clock, absda, mo, da, yr, hr, min, sec, wkday, s);
```

where clock is an input argument and the remaining arguments are output arguments.

Entry: datebin_$shift

This entry point returns the shift given a calendar clock reading. If clock is invalid, -1 is returned.

Usage

```
declare datebin_$shift (fixed bin(71), fixed bin);

call datebin_$shift (clock, s);
```

where clock is an input argument and s is an output argument.

Entry: datebin_$time

This entry point returns the hour, minute and second given a calendar clock reading. If clock is invalid, hr, min, and sec are -1.

Usage

```
declare datebin_$time entry (fixed bin(71), fixed bin, fixed bin,
     fixed bin);

call datebin_$time (clock, hr, min, sec);
```

where clock is an input argument and the remaining arguments are output arguments.

Entry: datebin_$wkday

This entry point returns the day of the week (Monday = 1 ... Sunday = 7) given a calendar clock reading. If clock is invalid, 0 is returned.

Usage

```
declare datebin_$wkday entry (fixed bin(71), fixed bin);

call datebin_$wkday (clock, wkday);
```

where clock is an input argument and wkday is an output argument.

Entry: datebin_$dayr_clk

This entry point returns the day of the year (1 - 366) given a calendar clock reading. If clock is invalid, -1 is returned.

Usage

```
declare datebin_$dayr_clk entry (fixed bin(71), fixed bin);

call datebin_$dayr_clk (clock, dayr);
```

where clock is an input argument and dayr is an output argument.

Entry: datebin_$revert

This entry point returns a calendar clock reading for the month, day, year, hour, minute, and second specified.

Usage

```
declare datebin_$revert entry (fixed bin, fixed bin, fixed bin, fixed bin,
     fixed bin, fixed bin, fixed bin(71));

call datebin_$revert (mo, da, yr, hr, min, sec, clock);
```

where clock is an output argument and the remaining arguments are input arguments.

Entry: datebin_$revertabs

This entry point returns a calendar clock reading given the number of days since January 1, 1901.

Usage

```
declare datebin_$revertabs entry (fixed bin, fixed bin(71));

call datebin_$revertabs (absda, clock);
```

where absda is an input argument and clock is an output argument.

Entry: datebin_$datofirst

This entry point returns the number of days since January 1, 1901, up to but not including January 1 of the year specified.

Usage

```
declare datebin_$datofirst entry (fixed bin, fixed bin);

call datebin_$datofirst (yr, datofirst);
```

Entry: datebin_$dayr_mc

This entry point returns the day of the year when given a month, day, and year.

Usage

```
declare datebin_$dayr_mo entry (fixed bin, fixed bin, fixed bin,
     fixed bin);

call datebin_$dayr_mo (mo, da, yr, dayr);
```

where dayr is an output argument and the remaining arguments are input arguments.

Entry: datebin_$clockathr

This entry point returns a clock reading for the next time the given hour occurs.

Usage

```
declare datebin_$clockathr entry (fixed bin, fixed bin(71));

call datebin_$clockathr (zz, clock);
```

where zz is an input argument and clock is an output argument.

Entry: datebin_$last_midnight

This entry point returns a clock reading for the midnight (local time) preceding the current day.

Usage

```
declare datebin_$last_midnight entry (fixed bin(71));

call datebin_$last_midnight (clock);
```

where clock is an output argument.

Entry: datebin_$this_midnight

This entry point returns a clock reading for midnight (local time) of the current day.

Usage

```
declare datebin_$this_midnight entry (fixed bin(71));
call datebin_$this_midnight (clock);
```

where clock is an output argument.

Entry: datebin_$preceding_midnight

This entry point, given a clock reading, returns a clock reading for midnight (local time) of the preceding day.

Usage

```
declare datebin_$preceding_midnight entry (fixed bin(71), fixed bin(71));
call datebin_$preceding_midnight (oldclock, clock);
```

where oldclock is an input argument and clock is an output argument.

Entry: datebin_$following_midnight

This entry point, given a clock reading, returns a clock reading for midnight (local time) of that day.

Usage

```
declare datebin_$following_midnight entry (fixed bin(71), fixed bin(71));
call datebin_$following_midnight (oldclock, clock);
```

where old clock is an input argument and clock is an output argument.

Entry: datebin_$next_shift_change

This entry, given a clock reading, returns the time of the next shift change, the current shift, and the new shift.

Usage

```
declare datebin_$next_shift_change entry (fixed bin(71), fixed bin(71),
     fixed bin, fixed bin);

call datebin_$next_shift_change (clock, newclock, shift, newshift);
```

where clock is an input argument and the remaining arguments are output arguments.

Name: decode_definition_

The decode_definition_ subroutine, given a pointer to an object segment definition, returns the decoded information of that definition in a structured, directly accessible format. This subroutine can only be used on one segment at a time because it uses internal static storage.

Usage

```
declare decode_definition_ entry (ptr, ptr, bit 1 aligned)

call decode_definition_ (def_ptr, structure_ptr, eof)
```

where:

1. def_ptr (Input)
   is a pointer to the selected definition. (The caller extracts this from the previously returned information.) The initial pointer with which decode_definition_ can be called is a pointer to the base of the object segment (i.e., with a zero offset), unless the decode_definition_$init entry point has been called, in which case the initial pointer can be a pointer to the beginning of the definition section (as returned by the object_info_ subroutine).

2. structure_ptr (Input)
   is a pointer to the provided structure in which decode_definition_ returns the desired information. (See "Notes" below.)

3. eof (Output)
   is a binary indicator that is "1"b if the current invocation of decode_definition_ causes the search to go beyond the end of the definition list. If that is the case, the returned information in the structure is null. It may also be "1"b if any error occurs.

Notes

The structure, contained in the decode_definition_str.incl.pl1 structure, has the following format:

```
dcl  1 decode_definition_common_header based aligned,
       2 next_def    ptr,
       2 prev_def    ptr,
       2 block_ptr   ptr,
       2 section     char (4) aligned,
       2 offset      fixed bin,
       2 entrypoint  fixed bin;
dcl  1 decode_definition_str based aligned,
       2 header      like decode_definition_common_header,
       2 symbol      char (32) aligned;
```

where:

1. next_def
   is a forward pointer to the next definition in the list. It can be used to make a subsequent call to decode_definition_.

2. prev_def
   is a backward pointer to the preceding definition on the list. This pointer may be null if the definition is of the old format.

3. block_ptr
   is a pointer to the head of the definition block if this is a segn definition and to the head of a segname list if this is not a segn definition. This pointer may be null if the definition is of the old format.

4. section
   is a symbolic code defining the type of definition. It can assume one of the following values: text, link, stat, symb, or segn.

5. offset
   is the offset of the definition within the given section. This is set to 0 if section is segn.

6. entrypoint
   is nonzero, if this definition is an entry point. The value of this item is the entry point's offset in the text section.

7. symbol
   is the character-string representation of the definition.

Entry: decode_definition_$decode_cref

This entry point, given a pointer to an object segment definition, returns the decoded information of that definition in a structure similar to that returned by decode_definition_, but with a pointer to the symbol name instead the name itself. It is used only by cross ref.

Usage

```
dcl  decode_definition_$decode_cref entry (ptr, ptr, bit (1) aligned, ptr);

call decode_definition_$decode_cref (def_ptr, decode_def_acc_ptr, eof,
     link_ptr);
```

where:

1. def_ptr (Input)
   must be a pointer to the beginning of the definition section.

2. decode_def_acc_ptr (Input)
   is a pointer to a structure in which the entry point is to return information. (See "Notes" below.)

3. eof (Input)
is the same as for the decode_definition_ entry point.

4. link_ptr (Input)
is a pointer to the base of the linkage section of the object segment the first time this entry is called for a given object segment. It is to be null for subsequent calls.

Notes

The structure filled in by this entry point has the following format. It may be found in decode_descriptor_str.incl.pl1

```
dcl 1 decode_definition_acc based aligned,
    2 header   like decode_definition_common_header,
    2 acc_ptr  ptr;
```

where:

1. header
all items in this substructure are the same as for the decode_definition_str substructure header.

2. acc_ptr
is a pointer to the ACC string that is the symbolic name of this definition.

Entry: decode_definition_$init

This entry point is used for initialization and is especially useful when the object segment does not begin at offset 0 (as for an archive component). This entry point has no effect when the decode_definition_$full entry point is being used.

Usage

```
declare decode_definition_$init entry (ptr, fixed bin(24));

call decode_definition_$init (seg_ptr, bit_count);
```

where:

1. seg_ptr (Input)
is a pointer to the beginning of an object segment (not necessarily with an offset of 0).

2. bit_count (Input)
is the bit count of the object segment.

Entry: decode_definition_$full

This entry point, given a pointer to an object segment definition, returns more complete information about that definition. The symbolic name returned by this entry point can contain up to 256 characters. This entry point does not use internal static storage.

Usage

```
declare decode_definition_$full entry (ptr, ptr, ptr, bit (1) aligned)
     returns (bit(1) aligned);

call decode_definition_$full (def_ptr, structure_ptr, oi_ptr, eof);
```

where:

1. def_ptr (Input)
   is a pointer to the selected definition and is extracted from previously returned information. The initial pointer with which the decode_definition_$full entry point can be called is a pointer to the base of the definition section of the object segment.

2. structure_ptr (Input)
   is a pointer to the provided structure into which the decode_definition_$full entry point returns the desired information. (See "Notes" below.)

3. oi_ptr (Input)
   is a pointer to the structure returned by any entry point of the object_info subroutine.

4. eof (Output)
   same as for the decode_definition_ entry point.

Notes

The structure, contained in the decode_definition_str.incl.pl1 structure, has the following format:

```
dcl 1 decode_definition_full based aligned
    2 header           like decode_definition_common_header,
    2 symbol           char (256) aligned,
    2 symbol_lng       fixed bin,
    2 flags,
      3 new_format     bit (1) unaligned,
      3 ignore         bit (1) unaligned,
      3 entrypt_flag   bit (1) unaligned,
      3 retain         bit (1) unaligned,
      3 arg_count      bit (1) unaligned,
      3 desc_sw        bit (1) unaligned,
      3 unused         bit (30) unaligned,
    2 nargs            fixed bin,
    2 desc_ptr         ptr;
```

where:

1. next_def
   is the same as for the decode_definition_ entry point.

2. prev_def
   is the same as for the decode_definition_ entry point.

3. block_ptr
   is the same as for the decode_definition_ entry point.

4. section
   is the same as for the decode_definition_ entry point.

5. offset
   is the same as for the decode_definition_ entry point.

6. entrypoint
   is the same as for the decode_definition_ entry point.

7. symbol
   is the same as for the decode_definition_ entry point.

8. symbol_lng
   is the relevant length of the symbol in characters.

9. new_format
   indicates that the definition is in the new format.

10. ignore
    is the linker ignore switch.
    "1"b the linker should ignore this definition.
    "0"b the linker should not ignore this definition.

11. entrypt_flag
    is the entry-point switch.
    "1"b the definition is for an entry point
    "0"b the definition is for a segdef.

12. retain
    is the retain switch.
    "1"b the definition should be retained.
    "0"b the definition should not be retained.

13. arg_count
    is the arg_count switch.
    "1"b there is an arg_count for this definition.
    "0"b there is no arg_count for this definition.

14. desc_sw
    is the descriptor switch.
    "1"b there are descriptors for this definition.
    "0"b there are no descriptor for this definition.

15. unused
    is padding.

16. nargs
    indicates the number of arguments expected by this entry, if descr_sw equals "1"b.

17. desc_ptr
    points to an array of 18-bit pointers to the descriptors for the entry, if descr_sw equals "1"b.

Name: display_file_value_

The display_file_value_ subroutine outputs information about a file on a user-supplied switch.

Usage

```
dcl display_file_value_ entry (ptr, file, fixed bin (35));

call display_file_value_ (switch, a_file, code);
```

where:

1. switch (Input)
   is a pointer to the iocb of the switch on which output is to be written. If it is null, then iox_$user_output is used.

2. a_file (Input)
   is the file, variable, or constant whose value is to be displayed.

3. code (Output)
   is a standard status code.

Notes

The output produced is, first, the values of the two pointers that comprise a file. If the file is closed, then a note to that effect is produced, and the values of the file attribute block are given, and that is all.

For all open files, the file name, address of its iocb, and pathname are given. If the file is neither stream nor record type, or if it is both, then a note to the effect that the fsb is inconsistent is given. Attributes relevant to the type of file (stream or record) are given. For stream input files, the current input buffer is printed, with a circumflex above the next character that will be parsed.

Name: find_include_file_

The primary entry point of the find_include_file_ subroutine searches for an include file on behalf of a translator. If the include file is found, additional information about the found segment is returned in the parameters. The "translator" search list is used to locate the include file.

Entry: find_include_file_$initiate_count

This entry point is the interface presented to translators. A translator calls this entry point to invoke a search for a single segment include file using the "translator" search list. For more information about search lists, see the search facility commands, and in particular the add_search_paths command in the MPM Commands manual, Order No. AG92.

Usage

```
declare find_include_file_$initiate_count entry (char(*), ptr, char(*),
     fixed bin(24), ptr, fixed bin(35));

call find_include_file_$initiate_count (translator, referencing_ptr,
     file_name, bit_count, seg_ptr, code);
```

where:

1. translator (Input)
   is the name of the translator that is calling this procedure (e.g., pl1, alm).

2. referencing_ptr (Input)
   is a pointer into the segment (normally a pointer to the source line) that caused the invocation of this instance of this procedure.

3. file_name (Input)
   is the complete entryname of the include file this procedure is to locate (e.g., include.incl.pl1).

4. bit_count (Output)
   is the bit count as obtained from the storage system of the found include file. If an include file is not found, this parameter is set to 0.

5. seg_ptr (Output)
   is a pointer to the first character of the include file, if found; if not found, this parameter is set to the null pointer value.

6. code (Output)
is a standard status code. The code may be:

0
The requested file was found normally. All output parameters have been set normally.

error_table_$zero_length_seg
The requested file was found, but the bit count was zero. All output parameters have been set normally.

error_table_$noentry
The requested file was not found in any of the search directories.

other storage system error codes
The requested file was not found because of some error.

Note

If this procedure finds an include file by a link, the seg_ptr parameter correctly designates the actual location of the include file. It is possible, however, that the name of the actual include file is not the same as the file_name argument passed to this procedure. It is the responsibility of the translator to determine if the file_name passed to this procedure is also on the include file actually found. It is also the responsibility of the translator to call the hcs_$terminate_noname entry point (described in the MPM Subroutines, Order No. AG93) on the include file when processing is complete.

Name: find_partition_

The find_partition_ subroutine is used to ascertain information about a disk partition located on some mounted storage-system disk. It reads the label and locates the partition, returning information about its size and location, as well as returning the PVID of the volume, for use in a later call to one of the hardcore entries for partition reading and writing. Use of this subroutine requires access to phcs_.

Usage

```
dcl find_partition_ entry (char (*), char (*), bit (36) aligned,
     fixed bin (35), fixed bin (35), fixed bin (35));

call find_partition_ (pvname, partition_name, pvid, first_record,
     partition_size, code);
```

where:

1. pvname (Input)
   is the name of the physical volume on which the partition is located. The volume must be a presently mounted, storage system disk volume.

2. partition_name (Input)
   is the name of the disk partition to be located. It must be four characters long or shorter.

3. pvid (Output)
   is the physical volume ID of the volume the partition is located on. This is returned as a convenience, for use in a later call to one of the hardcore entries for partition I/O.

4. first_record (Output)
   is the number (zero origin, from the beginning of the volume) of the first record in the partition.

5. partition_size (Output)
   is the number of words in the partition.

6. code (Output)
is a nonstandard status code. It will be one of the following:

0
indicates that the partition exists and that the returned parameters are all correct.

error_table_$pvid_not_found
indicates that the specified physical volume is not presently mounted.

error_table_$entry_not_found
indicates that the specified partition could not be found.

an integer between 1 and 10
indicates that a physical disk error occurred while trying to read the label. Error messages for physical disk errors are declared in the include file fsdisk_errors.incl.pl1, in the array fsdisk_error_message.

Name: get_bound_seg_info_

The get_bound_seg_info_ subroutine is used by several object-display programs concerned with bound segments to obtain information about a segment as a bound segment as well as general object information.

Usage

```
declare get_bound_seg_info_ entry (ptr, fixed bin(24), ptr, ptr, ptr,
     fixed bin(35));

call get_bound_seg_info_ (obj_ptr, bit_count, oi_ptr, bm_ptr, sblk_ptr,
     code);
```

where:

1. obj_ptr (Input)
   is a pointer to the beginning of the segment.

2. bit_count (Input)
   is the bit count of the segment.

3. oi_ptr (Input)
   is a pointer to the object format structure to be filled in by the object_info_$display entry point (see structure declaration in the description of the object_info_ subroutine).

4. bm_ptr (Output)
   is a pointer to the bind map.

5. sblk_ptr (Output)
   is a pointer to the base of the symbol block containing the bindmap.

6. code (Output)
   is a standard status code.

Note

If obj_ptr points to an object segment but no bindmap is found, two possible codes are returned. One is error_table_$not_bound, indicating that the segment is not bound. The other is error_table_$oldobj, indicating that the segment was bound before the binder produced internal bind maps. If either one of these is returned, the structure pointed to by oi_ptr contains valid information.

Name: get_initial_ring_

The get_initial_ring_ subroutine returns the current value of the ring number in which the process was initialized.

Usage

```
declare get_initial_ring_ entry (fixed bin);

call get_initial_ring_ (i_ring);
```

where i_ring is the initial ring for the process. (Output)

Name: hash_

The hash_ subroutine is used to maintain a hash table. It contains entry points that initialize a hash table and insert, delete, and search for entries in the table.

A hash table is used to locate entries in another data table when the length of the data table or the frequency with which its entries are referenced makes linear searching uneconomical.

A hash table entry contains a name and a value. The name is a character string (of up to 32 characters) that is associated in some way with a data table entry. The value is a fixed binary number that can be used to locate that data table entry (for example, an array index or an offset within a segment). The entries in the hash table are arranged so that the location of any entry can be computed by applying a hash function to the corresponding name.

It is possible for several names to hash to the same location. When this occurs, a linear search from the hash location to the first free entry is required, to find a place for a new entry (if adding), or to find out whether an entry corresponding to the name exists (if searching). The more densely packed the hash table, the more likely this occurrence is. To maintain a balance between efficiency and table size, hash_ keeps a hash table approximately 75 percent full, by rehashing it (i.e. rebuilding it in a larger space) when it becomes too full.

The number of entries is limited only by the available space. The table uses eight words per entry plus ten words for a header. If an entire segment is available to hold the table, it may have over 32,000 entries.

Entry: hash_$make

This entry point initializes an empty hash table. The caller must provide a segment to hold it, and must specify its initial size (see hash_$opt_size).

Usage

```
declare hash_$make entry (ptr, fixed bin, fixed bin(35));

call hash_$make (table_ptr, size, code);
```

where:

1. table_ptr (Input)
   is a pointer to the table to be initialized.

2. size (Input)
is the initial number of entries. It is recommended that the value returned by hash_$opt_size be used.

3. code (Output)
is a standard status code. It will be zero if there is no error, or error_table_$invalid_elsize if size is too large.

Entry: hash_$opt_size

This entry point, given the number of entries to be placed in a new hash table initially, returns the optimal size for the new table. This function is used when rehashing a full hash table, and should be used when making a new hash table.

Usage

```
declare hash_$opt_size entry (fixed bin) returns (fixed bin);

size=hash_$opt_size (n_entries);
```

where:

1. n_entries (Input)
is the number of entries to be added.

2. size (Output)
is the optimal table size for that number of entries.

Entry: hash_$in

This entry point adds an entry to a hash table. If the additional entry would make the table too full, the table will be rehashed before the new entry is added (see the description of the rehash_ subroutine).

Usage

```
declare hash_$in entry (ptr, char(*), fixed bin, fixed bin(35));

call hash_$in (table_ptr, name, value, code);
```

where:

1. table_ptr (Input)
is a pointer to the hash table.

2. name (Input)
is a name associated with a data table entry. It may be up to 32 characters long.

3. value (Input)
is the locator (e.g., index or offset) of the data table entry associated with name.

4. code (Output)
is a standard system error code with the following values:

0
entry added successfully

error_table_$segnamdup
entry already exists, with same value

error_table_$namedup
entry already exists, with different values

error_table_$full_hashtbl
hash table is full and there is no room to rehash it into a larger space.

Entry: hash_$inagain

This entry point adds an entry to a hash table. It is identical to the hash_$in entry except that it will never try to rehash the table. The new entry will be added unless the table is completely full. This entry point is used by the rehash_ subroutine to avoid loops. It can also be used by an application that has a hash table embedded in a larger data base, where automatic rehashing would damage the data base.

Usage

```
declare hash_$inagain entry (ptr, char(*), fixed bin, fixed bin(35));

call hash_$inagain (table_ptr, name, value, code);
```

where:

the arguments are the same as those for the hash_$in entry point, above.

Entry: hash_$search

This entry point searches a hash table for a given name and returns the corresponding locator value.

Usage

```
declare hash_$search entry (ptr, char(*), fixed bin, fixed bin(35));

call hash_$in (table_ptr, name, value, code);
```

where:

1. table_ptr (Input)
   is a pointer to the hash table.

2. name (Input)
   is the name to be searched for. It may be up to 32 characters long.

3. value (Output)
   is the locator value corresponding to name.

4. code (Output)
   is a standard status code. It can be:

   0
   name was found

   error_table_$noentry
   name was not found in the hash table

Entry: hash_$out

This entry point deletes a name from the hash table.

Usage

```
declare hash_$out entry (ptr, char(*), fixed bin, fixed bin(35));

call hash_$out (table_ptr, name, value, code);
```

where:

1. table_ptr (Input)
   is a pointer to the hash table.

2. name (Input)
is the name to be deleted. Its maximum length is 32 characters.

3. value (Input)
is the locator value corresponding to name.

4. code (Output)
is a standard status code. It can be:

0
name was found and deleted

error_table_$noentry
name was not found in the hash table

hcs_$get_page_trace

Name: hcs_$get_page_trace

The hcs_$get_page_trace entry point returns information about recent paging activity.

Usage

```
declare hcs_$get_page_trace entry (ptr);

call hcs_$get_page_trace (data_ptr);
```

where data_ptr is a pointer to a user data space where return information is stored. (Input)

Notes

The format of the data structure returned by hcs_$get_page_trace is described below. The amount of data returned cannot be known in advance other than that there are less than 1024 words returned.

```
dcl 1 trace          aligned based(tp)
    2 next_available bit(18) aligned,
    2 size           bit(18) aligned,
    2 time           fixed bin(71),
    2 pad1           fixed bin(35),
    2 index          bit(17),
    2 pad2           fixed bin(71),
    2 data           (512 refer(divide (trace.size,2,17,0))),
      3 info         bit(36) aligned,
      3 type         bit(6) unaligned
      3 pageno       bit(12) unaligned,
      3 time_delta   bit(18) unaligned;
```

where:

1. next_available
   is a relative pointer (relative to the first trace entry) to the next entry to be used in the trace list.

2. size
   is the number of words in the trace array and, hence, twice the number of entries in the array.

3. time
   is the real-time clock reading at the time the last trace entry was entered in the list.

4. pad1
   is unused.

hcs_$get_page_trace

5. index
   is a relative pointer to the first trace entry entered in the last quantum. Thus, all events traced in the last quantum can be determined by scanning from trace.index to trace.next_available (minus 1) with the obvious check for wrap-around.

6. pad2
   is unused.

7. info
   is information about the particular trace entry.

8. type
   specifies what kind of a trace entry it is. The following types are currently defined:

   0 page fault
   2 segment fault begin
   3 segment fault end
   4 linkage fault begin
   5 linkage fault end
   6 bound fault begin
   7 bound fault end
   8 signaller event
   9 restarted signal
   10 reschedule
   11 user marker
   12 interrupt

9. pageno
   is the page number associated with the fault. Certain trace entries do not fill in this field.

10. time_delta
    is the amount of real time elapsed between the time this entry was entered and the previous entry was entered. The time value is in units of 64 microseconds.

hphcs_$ips_wakeup

Name: hphcs_$ips_wakeup

The hphcs_$ips_wakeup entry point sends a specified IPS signal to a specified process. That process is interrupted immediately unless it has the specified IPS signal masked off. See the description of the hcs_$get_ips_mask, hcs_$reset_ips_mask, and hcs$set_ips_mask entry points in the MPM Subsystem Writers' Guide (Order No. AK92) for a discussion of ips masking.

Usage

```
declare hphcs_$ips_wakeup entry (bit(36) aligned, char(4) aligned);

call hphcs_$ips_wakeup (process_id, eps_name);
```

where:

1. process_id (Input)
   is the process identifier of the target process.

2. ips_name (Input)
   is the name of the ips signal to be sent to the target process.

Notes

See the description of the set_ips_mask command in Section 2 of this manual for a list of valid ips signal names.

If the arguments are invalid (nonexistent process, undefined ips signal name) or are not properly aligned, the call is ignored; i.e., no signal is sent, and no error indication is given.

hphcs_$read_partition

Name: hphcs_$read_partition

This entry point is used to read words of data from a specified disk partition on some mounted physical storage-system disk.

Usage

```
dcl hphcs_$read_partition entry (bit (36) aligned, char(*), fixed bin (35),
     pointer, fixed bin (19), fixed bin (35));

call hphcs_$read_partition (pvid, partition_name, offset, data_pointer,
     word_count, code);
```

where:

1. pvid (Input)
   is the physical volume id of the disk from which to read. The physical volume id is used instead of the volume name because this is a ring zero interface, and volume names are not accessible by ring zero; hence, all ring-zero interfaces that reference physical volumes use the pvid. A pvname may be converted to a pvid by a call to mdc_$find_pvname, or the pvid may have been returned by a previous call to find_partition_.

2. partition_name (Input)
   is the name of the disk partition to be read from. It must be four characters long or shorter.

3. offset (Input)
   is the offset in words, from the first word of the partition, of the first location to be read. It must be nonnegative and less than the number of words in the partition.

4. data_ptr (Input)
   is a pointer to the user-supplied buffer into which the data is to be read. It must be aligned on a word boundary.

5. word_count (Input)
   is the number of words to be read. The sum of offset and word_count must be less than or equal to the number of words in the partition. The sum of word_count and binary (rel (data_ptr)) must also be less than or equal to sys_info$max_seg_size, in order to avoid accessing past the end of the segment pointed to by data_ptr.

hphcs_$read_partition

6. code (Output)
is a nonstandard status code. It will be one of the following:

0
indicates that the data was successfully read.

error_table_$pvid_not_found
indicates that the specified physical volume is not presently mounted.

error_table_$entry_not_found
indicates that the specified partition could not be found.

error_table_$out_of_bounds
indicates that read request attempts to access data outside the partition; that is, the sum of offset and word_count is too large.

an integer between 1 and 10
indicates that a physical disk error occurred while trying to read the label. Error messages for physical disk errors are declared in the include file fsdisk_errors.incl.pl1, in the array fsdisk_error_message.

Name: hphcs_$write_partition

This entry point is used to write words of data into a specified disk partition on some mounted physical storage-system disk. No protection is provided against simultaneous use of this entry point by several processes writing to the same partition; thus, care must be exercised when using it.

Usage

```
dlc hphcs_$write_partition entry (bit (36) aligned, char (*),
     fixed bin (35), pointer, fixed bin (18), fixed bin (35));

call hphcs_$write_partition (pvid, partition_name, offset, data_pointer,
     word_count, code);
```

where:

1. pvid (Input)
   is the physical volume id of the disk from which to read. The physical volume id is used instead of the volume name because this is a ring zero interface, and volume names are not accessible by ring zero; hence, all ring-zero interfaces that reference physical volumes use the pvid. A pvname may be converted to a pvid by a call to mdc_$find_pvname, or the pvid may have been returned by a previous call to find_partition_.

2. partition_name (Input)
   is the name of the disk partition to be read from. It must be four characters long or shorter.

3. offset (Input)
   is the offset in words, from the first word of the partition, of the first location to be read. It must be nonnegative and less than the number of words in the partition.

4. data_ptr (Input)
   is written into the partition from the user-supplied buffer. It must be aligned on a word boundary.

5. word_count (Input)
   is the number of words to be read. The sum of offset and word_count must be less than or equal to the number of words in the partition. The sum of word_count and binary (rel (data_ptr)) must also be less than or equal to sys_info$max_seg_size, in order to avoid accessing past the end of the segment pointed to by data_ptr.

hphcs_$write_partition

6. code (Output)
is a nonstandard status code. It will be one of the following:

0
indicates that the data was successfully read.

error_table_$pvid_not_found
indicates that the specified physical volume is not presently mounted.

error_table_$entry_not_found
indicates that the specified partition could not be found.

error_table_$out_of_bounds
indicates that read request attempts to access data outside the partition; that is, the sum of offset and word_count is too large.

an integer between 1 and 10
indicates that a physical disk error occurred while trying to read the label. Error messages for physical disk errors are declared in the include file fsdisk_errors.incl.pl1, in the array fsdisk_error_message.

Name: lex_error_

The lex_error_ subroutine generates compiler-style error messages on the error_output I/O switch for translators generated by the reduction_compiler command and for other procedures that process tokens generated by the lex_string_ subroutine. See "Notes" below for a description of the error-message format.

Usage

```
declare lex_error_ entry options (variable);

call lex_error_ (error_number, Serror_printed, severity_no,
     max_severity_no, Pstmt, Ptoken, Scontrol, message, brief_message,
     arg1, ..., argn);
```

where:

1. error_number (Input)
   is the error number (fixed bin), as it should appear in the error message.

2. Serror_printed (Input/Output)
   is a switch (bit(1) unaligned) that is "1"b if the text of the error message has been printed in a previous error and "0"b, otherwise. If Serror_printed is "1"b, the text is omitted from the error message. Otherwise, text is included and the switch is set to "1"b to suppress this text in any subsequent occurrence of the same error.

3. severity_no (Input)
   is the severity number (fixed bin) of the error. It must have a value from 0 through 4. See "Notes" below for an interpretation of the severity_no value.

4. max_severity_no (Input/Output)
   is the severity number (fixed bin) of the highest-severity error message that has been printed by the lex_error_ subroutine. Before the lex_error_ is invoked by a translator, max_severity_no should be initialized to 0. Each time it is called, the lex_error_ subroutine compares this value with the severity_no of the current message and sets max_severity_no to the higher of these two numbers.

5. Pstmt (Input)
   is a pointer to the statement descriptor generated by the lex_string_ subroutine for the statement that is to be printed after the error message. The line number and statement number given in this statement descriptor are included in the error message.

6. Ptoken (Input)
   is a pointer to the token descriptor of the token that is in error. If Pstmt is null, then the number of the line that contains the token described by the descriptor is included in the error message. If both Pstmt and Ptoken are null, then no line number is included in the error message.

7. Scontrol (Input)
is a control bit string (bit(*)) that determines whether the message character string or the brief_message character string is used in the error message. The interpretation of the bits in this string is described in "Notes" below.

8. error_message_text (Input)
is an ioa_ control string (char(*) or char(*) varying) that contains the long form of the error message text.

9. brief_message_text (Input)
is an ioa_ control string (char(*) or char(*) varying) that contains the brief form of the error message text.

10. argi (Input)
are optional arguments that are substituted into the ioa_ message texts, in place of the ioa_ control characters.

Notes

The error messages that are generated by the lex_error_ subroutine have the form shown below.

```
prefix error_number, SEVERITY severity_no IN STATEMENT k of LINE l.
error_message_text
SOURCE:
statement_in_error
```

For example,

```
ERROR 7, SEVERITY 2 in STATEMENT 2 OF LINE 2.
A bad track specification was given in a Volume statement.
9track has been assumed.
SOURCE:
Volume: 70082, 8track;
```

The severity_no associated with an error controls the prefix that is placed in the error message, as shown in the table below.

SEVERITY	PREFIX	EXPLANATION
0	COMMENT	Comment. The error message is a comment, which does not indicate that an error has occurred, but merely provides information for the user.
1	WARNING	Warning only. The error message warns of a statement that may or may not be in error, but compilation continues without ill effect.
2	ERROR	Correctable error. The message diagnoses an error that the translator can correct, probably without ill effect. Compilation continues, but correct results cannot be guaranteed.
3	FATAL ERROR	An uncorrectable but recoverable error. The translator has detected an error that it cannot correct. Translation continues in an attempt to diagnose further errors, but no output is produced by the translation.
4	TRANSLATOR ERROR	An unrecoverable error. The translator cannot continue beyond this error. The translation is aborted after the error message is printed.

The phrase "IN STATEMENT k OF LINE l" appears in the error message only if Pstmt is a nonnull pointer. Pstmt is assumed to point to a statement descriptor generated by the lex_string_ subroutine. The values for k and l come from this descriptor. If the error occurred in the first statement of line l, then the phrase "STATEMENT k OF" is omitted from the error message.

If Pstmt is null, then "STATEMENT k OF" is omitted from the error message, and l is the line number on which the token described by Ptoken appears. If Ptoken is a null pointer, "IN STATEMENT k OF LINE l" is omitted altogether.

Currently, only the first two bits of the Scontrol bit string have meaning, as shown in the table below.

Scontrol	INTERPRETATION
"00"b	The printed error contains the error_message_text the first time the error occurs, and the brief_message_text for subsequent occurrences of that error during a given translation.
"10"b	The printed error always contains the error_message_text.
"11"b	The printed error always contains the error_message_text.
"01"b	The printed error always contains the brief_message_text.

If Serror_printed is "1"b, then the lex_error_ subroutine assumes the text of the error message has already been printed in a previous message. It uses the long or brief error message text, according to the value of Scontrol.

If Pstmt points to a statement descriptor, then the lex_error_ subroutine sets the error_in_stmt switch in the statement descriptor. It also checks the value of the output_in_err_msg switch in the descriptor. If this switch is "0"b, the lex_error_ subroutine sets it to "1"b and prints the character string representation of the statement in the error message. If it is already "1"b, then the lex_error_ subroutine assumes that the statement has already appeared in another error message and omits the "SOURCE:" phrase from the error message.

If max_severity_no is less than severity_no, then the lex_error_ subroutine sets max_severity_no equal to severity_no.

Refer to the lex_string_ subroutine for a description of statement and token descriptors.

Name: lex_string_

The lex_string_ subroutine provides a facility for parsing an ASCII character string into tokens (character strings delimited by break characters) and statements (groups of tokens). It supports the parsing of comments and quoted strings. It parses an entire character string during one invocation, creating a chain of descriptors for the tokens and statements in a temporary segment. The cost per token of lex_string_ is significantly lower than that of parse_file_ because the overhead of calling parse_file_ to obtain each token is eliminated. Therefore, the lex_string_ subroutine is recommended for translators that deal with moderate to large amounts of input.

The descriptors generated when the lex_string_ subroutine parses a character string can be used as input to translators generated by the reduction_compiler command, as well as in other applications. In addition, the information in the statement and token descriptors can be used in error messages printed by the lex_error_ subroutine.

Refer to the the reduction_compiler and lex_error_ descriptions for details on the use of these facilities.

Entry: lex_string_$init_lex_delims

This entry point constructs two character strings from the set of break characters and comment, quoting, and statement delimiters: one string contains the first character of every delimiter or break character defined by the language to be parsed; the second string contains a character of control information for each character in the first string. These two character strings form the break tables that the lex_string_ subroutine uses to parse an input string. It is intended that these two (delimiter and control) character strings be internal static variables of the program that calls lex_string_, and that they be initialized only once per process. They can then be used in successive calls to lex_string_$lex, as described below.

Usage

```
declare lex_string_$init_lex_delims entry (char(*), char(*), char(*),
     char(*), char(*), bit(*), char(*) varying aligned,
     char(*) varying aligned, char(*) varying aligned,
     char(*) varying aligned);

call lex_string_$init_lex_delims (quote_open, quote_close, comment_open,
     comment_close, statement_delim, Sinit, break_chars,
     ignored_break_chars, lex_delims, lex_control_chars);
```

where:

1. quote_open (Input)
   is the character string delimiter that begins a quoted string. It may contain up to four characters. If it is a null character string, then quoted strings are not supported during the parsing of an input string.

2. quote_close (Input)
   is the character string delimiter that ends a quoted string. It may be the same character string as quote_open, and may contain up to four characters.

3. comment_open (Input)
   is the character string delimiter that begins a comment. It may contain up to four characters. If it is a null character string, then comments are not supported during the parsing of a character string.

4. comment_close (Input)
   is the character string delimiter that ends a comment. It may be the same character string as comment_open, and may contain up to four characters.

5. statement_delim (Input)
   is the character string delimiter that ends a statement. It may contain up to four characters. If it is a null character string, then statements are not delimited during the parsing of a character string.

6. Sinit (Input)
   is a bit string that controls the creation of statement descriptors, and the creation of token descriptors for quoting delimiters. The bit string consists of two bits in the order listed below.

   Ssuppress_quoting_delims
   is "1"b if token descriptors for the quote opening and closing delimiters of a quoted string are to be suppressed. A token descriptor is still created for the quoted string itself, and the quoted_string switch in this descriptor is turned on. If Ssuppress_quoting_delims is "0"b, then token descriptors are returned for the quote opening and closing delimiters, as well as for the quoted string.

   Ssuppress_stmt_delims
   is "1"b if the token descriptor for a statement delimiter is to be suppressed. The end_of_stmt switch in the descriptor of the token that precedes the statement delimiter is turned on, instead. If Ssuppress_stmt_delims is "0"b, then a token descriptor is returned for a statement delimiter, and the end_of_stmt switch in this descriptor is turned on.

7. break_chars (Input)
   is a character string containing all of the characters that may be used to delimit tokens. The string may include characters used also in the quoting, comment, or statement delimiters, and should include any ASCII control characters that are to be treated as delimiters.

8. ignored_break_chars (Input)
   is a character string containing all of the break_chars that may be used to delimit tokens but that are not tokens themselves. No token descriptors are created for these characters.

9. lex_delims (Output)
is an output character string containing all of the delimiters that the lex_string_ subroutine will use to parse an input string. This string is constructed by the init_lex_delims entry from the preceding arguments. It must be long enough to contain all of the break_chars, plus the first character of the quote_open delimiter, the comment_open delimiter, and the statement_delim delimiter, plus 30 additional characters. This length will not exceed 128 characters, the number of characters in the ASCII character set.

10. lex_control_chars (Output)
is an output character string containing one character of control information for each character in lex_delims. This string is also constructed by init_lex_delims from the preceding arguments. It must be as long as lex_delims.

Entry: lex_string_$lex

This entry point parses an input string, according to the delimiters, break characters, and control information given as its arguments. The input string consists of two parts: the first part is a set of characters, which are to be ignored by the parser except for the counting of lines; the second part is the characters to be parsed. It is necessary to count lines in the part that is otherwise ignored so that accurate line numbers can be stored in the token and statement descriptors for the parsed section of the string.

Usage

```
declare lex_string_$lex entry (ptr, fixed bin(21), fixed bin(21), ptr,
     bit(*), char(*), char(*), char(*), char(*), char(*),
     char(*) varying aligned, char(*) varying aligned,
     char(*) varying aligned, char(*) varying aligned, ptr, ptr,
     fixed bin(35));

call lex_string_$lex entry (Pinput, Linput, Lignored_input, Psegment, Slex,
     quote_open, quote_close, comment_open, comment_close, statement_delim,
     break_chars, ignored_break_chars, lex_delims, lex_control_chars,
     Pfirst_stmt_desc, Pfirst_token_desc, code);
```

where:

1. Pinput (Input)
is a pointer to the string to be parsed.

2. Linput (Input)
is the length (in characters) of the second part of the input string, the part that is actually to be parsed.

3. Lignored_input (Input)
is the length (in characters) of the first part of the input string, the part that is ignored except for line counting. This length may be 0 if none of the input characters are to be ignored.

4. Psegment (Input)
   is a pointer to a temporary segment created by the translator_temp_ subroutine.

5. SLex (Input)
   is a bit string that controls the creation of statement and comment descriptors, the handling of doubled quotes within a quoted string, and the interpretation of a comment_close delimiter that equals the statement_delim. The bit string consists of four bits in the order listed below.

   Sstatement_desc
   is "1"b if statement descriptors are to be created along with the token descriptors. If Sstatement_desc is "0"b, or if the statement delimiter is a null character string, then no statement descriptors are created.

   Sscomment_desc
   is "1"b if comment descriptors are to be created for any comments that appear in the input string. When Scomment_desc is "0"b, comment_open is a null character string, or statement descriptors are not being created, then no comment descriptors are created.

   Sretain_doubled_quotes
   is "1"b if doubled quote_close delimiters that appear within a quoted string are to be retained. If Sretain_doubled_quotes is "0"b, then a copy of each quoted string containing doubled quote_close delimiters is created in the temporary segment with all doubled quote_close delimiters changed to single quote_close delimiters.

   Sequate_comment_close_stmt_delim
   is "1"b if the comment_close and statement_delim character strings are the same, and if the closing of a comment is to be treated as the ending of the statement containing the comment. It could be used when parsing line-oriented languages that have only one statement per line and one comment per statement.

6. quote_open (Input)
   is the character string delimiter that begins a quoted string. It may contain up to four characters. If it is a null character string, then quoted strings are not supported during the parsing of an input string.

7. quote_close (Input)
   is the character string delimiter that ends a quoted string. It may be the same character string as quote_open, and may contain up to four characters.

8. comment_open (Input)
   is the character string delimiter that begins a comment. It may contain up to four characters. If it is a null character string, then comments are not supported during the parsing of a character string.

9. comment_close (Input)
   is the character string delimiter that ends a comment. It may be the same character string as comment_open, and may contain up to four characters.

10. statement_delim (Input)
is the character string delimiter that ends a statement. It may contain up to four characters. If it is a null character string, then statements are not delimited during the parsing of a character string.

11. break_chars (Input)
is a character string containing all of the characters that may be used to delimit tokens. The string may include characters used also in the quoting, comment, or statement delimiters, and should include any ASCII control characters that are to be treated as delimiters.

12. ignored_break_chars (Input)
is a character string containing all of the break_chars that may be used to delimit tokens but that are not tokens themselves. No token descriptors are created for these characters.

13. lex_delims (Input)
is the character string initialized by lex_string_$init_lex_delims.

14. lex_control_chars (Input)
is the character string initialized by lex_string_$init_lex_delims.

15. Pfirst_stmt_desc (Output)
is a pointer to the first in the chain of statement descriptors. This is a null pointer on return if no statement descriptors have been created.

16. Pfirst_token_desc (Output)
is a pointer to the first in the chain of token descriptors. This is a null pointer on return if no tokens were found in the input string.

17. code (Output)
is one of the following status codes:

0
the parsing was completed successfully.

error_table_$zero_length_seg
no tokens were found in the input string.

error_table_$no_stmt_delim
the input string did not end with a statement delimiter, when statement delimiters were used in the parsing.

error_table_$unbalanced_quotes
the input string ended with a quoted string that was not terminated by a quote_close delimiter.

Notes

Any character may be used in the quoting, comment, and statement delimiter character strings, including such characters as new line and the space character.

A quoted string is defined in the PL/I sense, as a string of characters that is treated as a single token, even though some of the characters may be delimiters or break characters. The string must begin with a quote_open delimiter, and must end with a quote_close delimiter. Two consecutive quote_close delimiters may be used to represent a quote_close delimiter within the quoted string. entry point lex_string_$entry point lex provides the option of retaining any doubled quote_close delimiters in the quoted string token, or of copying the quoted string into the temporary segment, changing double quote_close to single quote_close delimiters, and treating the modified copy as the quoted string token. Switches in the token descriptor of a quoted string are turned on: to indicate that the token is a quoted string; to indicate whether any quote_close delimiters appear within the quoted string; and to indicate whether doubled quote_close delimiters have been retained in the token.

Statements are defined as groups of tokens that are terminated by a statement delimiter token. The lex_string_$lex subroutine can optionally return a token descriptor for the statement delimiter or it can suppress the token descriptor of the statement delimiter. It always turns on the end_of_stmt switch in the final token descriptor of each statement, even if the token descriptor of the statement delimiter has been suppressed. Also, it can optionally return a statement descriptor that points to the descriptors for the first and last tokens of a statement, contains a pointer to and the length of the part of the input string containing the entire statement, and describes various other characteristics of the statement. These descriptors are described in the next section.

Comments are defined in the PL/I sense, as a string of characters that begin with a comment_open delimiter, and that end with a comment_close delimiter. Comments that appear in the input string act as breaks between tokens. The lex_string_$lex entry point can optionally create descriptors for each comment that appears in a statement. These descriptors are chained off of the statement descriptor for that statement. Switches are set in each comment descriptor of a given statement to indicate whether the comment appears before any of the tokens in that statement, and whether any tokens intervene between this comment and any previous comments in that statement.

The lex_string_ subroutine uses the translator_temp_ facility to allocate space for the descriptors in the temporary segment. Refer to the translator_temp_ subroutine description for details on the use of these temporary segments.

Descriptors

If the lex_string_$lex entry point were invoked using standard PL/I parsing conventions to parse the input shown in Figure 3-1, then tokens and token descriptors created by the lex_string_ subroutine would have the form shown in Figure 3-2.

```
Volume:  70092;
Write;
File 4; /* Process 4th file on the tape.  */
/* END */
```

Figure 3-1. Sample Input to lex_string_

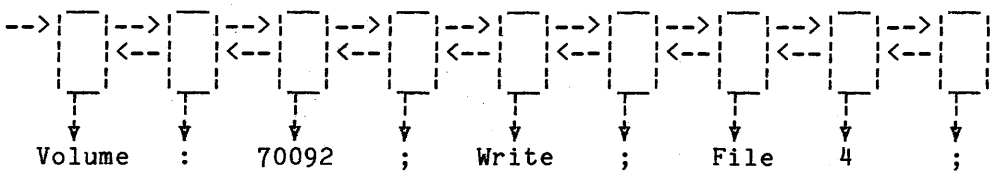


Figure 3-2. Input Tokens and Their Descriptors

If statement descriptors were being created by the lex_string_ subroutine, then the output would have the form shown in Figure 3-3.

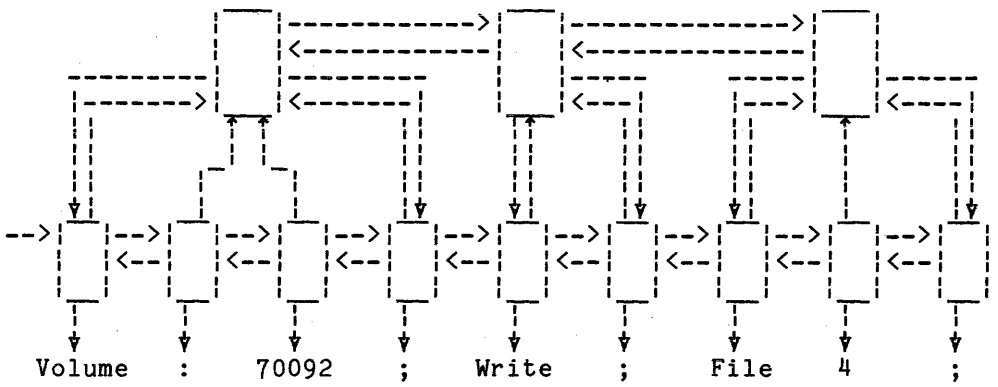


Figure 3-3. Tokens, Token Descriptors, and Statement Descriptors

Below is a declaration for the token descriptor structure.

```
dcl 1 token                 aligned based (Ptoken),
    2 group1                unaligned,
      3 version             fixed bin(17),
      3 size                fixed bin(17),
    2 Pnext                 ptr unal,
    2 Plast                 ptr unal,
    2 Pvalue                ptr unal,
    2 Lvalue                fixed bin(18),
    2 Pstmt                 ptr unal,
    2 Psemant               ptr unal,
    2 group2                unaligned,
```

```
        3 Itoken_in_stmt      fixed bin(17),
        3 line_no             fixed bin(17),
        3 Nvalue              fixed bin(35),
        3 S,
          4 end_of_stmt       bit(1),
          4 quoted_string     bit(1),
          4 quotes_in_string  bit(1),
          4 quotes_doubled    bit(1),
          4 pad2              bit(32);

dcl Ptoken                    ptr;
dcl token_value               char(token.Lvalue) based (token.Pvalue);
```

1. version
   is the version number of the structure. The structure shown above is version 1.

2. size
   is the size of the structure, in words.

3. Pnext
   is a pointer to the descriptor for the next token in the input. If this is the last token descriptor, then the pointer is null.

4. Plast
   is a pointer to the descriptor for the previous token in the input. If this is the first token descriptor, then the pointer is null.

5. Pvalue
   is a pointer to the token character string.

6. Lvalue
   is the length of the token character string, in characters.

7. Pstmt
   is a pointer to the statement descriptor for the statement that contains this token. If statement descriptors are not being created, then this pointer is null.

8. Psemant
   is a pointer available for use by the caller of lex_string_. It is initialized as a null pointer. It might be used to chain a structure defining the semantic value of the token to the token's descriptor.

9. Itoken_in_stmt
   is the position of the token with respect to the other tokens in the statement containing this token. If no statement delimiters are being used in the parsing, then this is the position of the token with respect to all other tokens in the input.

10. line_no
    is the line_no on which this token appears.

11. Nvalue
    is a number available for use by the caller of lex_string_. It is initialized to 0. It might be set to the numeric value of a token that is the character string representation of an integer.

12. end_of_stmt
    is "1"b if this is the last token of a statement.

13. quoted_string
    is "1"b if this token appeared in the input as a quoted string.

14. quotes_in_string
    is "1"b is quote_close delimiters appear within this quoted string token.

15. quotes_doubled
    is "1"b if quote_close delimiters that appear in a quoted string token are still represented by doubled quote_close delimiters, rather than having been converted to single quote_close delimiters.

16. pad2
    is available for use by the caller of lex_string_. It is initialized to ""b by lex_string_.

17. Ptoken
    is a pointer to a token descriptor.

18. token_value
    is the character string representation of the token described by the token descriptor pointed to by Ptoken.

Statement descriptors are declared by the structure shown below.

```
dcl 1 stmt                  aligned based (Pstmt),
    2 group1                unaligned,
      3 version             fixed bin(17),
      3 size                fixed bin(17),
    2 Pnext                 ptr unal,
    2 Plast                 ptr unal,
    2 Pvalue                ptr unal,
    2 Lvalue                fixed bin(18),
    2 Pfirst                ptr unal,
    2 Plast_token           ptr unal,
    2 Pcomments             ptr unal,
    2 Puser                 ptr unal,
    2 group2                unaligned,
      3 Ntokens             fixed bin(17),
      3 line_no             fixed bin(17),
      3 Istmt_in_line       fixed bin(17),
      3 semant_type         fixed bin(17),
      3 S,
        4 error_in_stmt     bit(1),
        4 output_in_err_msg bit(1),
        4 pad               bit(34);

dcl Pstmt                   ptr;
dcl stmt_value              char(stmt.Lvalue) based (stmt.Pvalue);
```

1. version
   is the version number of this structure. The structure declared above is version 1.

2. size
   is the size of this structure, in words.

3. Pnext
   is a pointer to the statement descriptor for the next statement. If this is the descriptor for the last statement, then this pointer is null.

4. Plast
   is a pointer to the descriptor for the previous statement. If this is the descriptor for the first statement, then the pointer is null.

5. Pvalue
   is a pointer to the character string representation of the statement as it appears in the input, excluding any leading newline characters or leading comments.

6. Lvalue
   is the length of the character string representation of the statement, in characters.

7. Pfirst_token
   is a pointer to the descriptor of the first token in the statement.

8. Plast_token
   is a pointer to the descriptor of the last token in the statement.

9. Pcomments
   is a pointer to a chain of comment descriptors associated with this statement.

10. Puser
    is a pointer available for use by the caller of lex_string_.

11. Ntokens
    is a count of the tokens in this statement.

12. line_no
    is the line number on which the first token of this statement appears in the input.

13. semant_type
    is a number available for use by the caller of lex_string_. It is initialized to 0 by lex_string_. It might be used to classify the statement by its semantic type.

14. error_in_stmt
    is "1"b if an error has occurred while processing this statement. This switch is never set by lex_string_, but it is set by lex_error_ when a statement descriptor is used to generate an error message.

15. output_in_err_msg
    is "1"b if the statement has already been output in another error message. This switch is referenced and set by lex_error_ to prevent a statement from being printed in more than one error message.

16. pad
    is available for use by the caller of lex_string_. It is initialized to ""b by lex_string_.

17. Pstmt
    is a pointer to a statement descriptor.

18. stmt_value
is the character string value of the statement, as it appears in the input, excluding any leading newline characters or leading comments.

Comment descriptors are declared as follows.

```
dcl 1 comment              aligned based (Pcomment),
    2 group1               unaligned,
      3 version            fixed bin(17),
      3 size               fixed bin(17),
    2 Pnext                ptr unal,
    2 Plast                ptr unal,
    2 Pvalue               ptr unal,
    2 Lvalue               fixed bin(18),
    2 group2               unaligned
      3 line_no            fixed bin(17),
      3 S,
        4 before_stmt      bit(1),
        4 contiguous       bit(1),
        4 pad              bit(16);

dcl Pcomment               ptr;
dcl comment_value          char(comment.Lvalue) based (comment.Pvalue);
```

1. version
is the version number of this structure. The structure declared above is version 1.

2. size
is the size of this structure, in words.

3. Pnext
is a pointer to the descriptor for the next comment associated with the statement containing this comment. If there are no more comments associated with that statement, then the pointer is null.

4. Plast
is a pointer to the descriptor for the previous comment associated with the statement containing this comment. If this is the first comment associated with the statement, then the pointer is null.

5. Pvalue
is a pointer to the character string value of the comment string, exactly as it appears in the input, excluding the comment_open and comment_close delimiters.

6. Lvalue
is the length of the character string value of the comment, in characters.

7. line_no
is the line number on that the comment begins.

8. before_stmt
is "1"b if the comment appears in its statement before any tokens.

9. contiguous
   is "1"b if no tokens appear between this comment and the previous comment associated with this statement.

10. pad
    is available for use by lex_string_'s caller.

11. Pcomment
    is a pointer to a comment descriptor structure.

12. comment_value
    is the character string value of a comment.

The above declarations are available for inclusion in PL/I programs in lex_descriptors_.incl.pl1.

Name: link_unsnap_

The link_unsnap_ subroutine restores snapped links pointing to a given segment or its linkage section. Such links then appear as if they had never been snapped (changed into ITS pairs). This is accomplished by sequentially indexing through the linkage offset table (LOT) and for each linkage section listed there, searching for links to be restored.

Usage

```
declare link_unsnap_ entry (ptr, ptr, ptr, fixed bin(17), fixed bin(17));

call link_unsnap_ (lot_ptr, isot_ptr, linkage_ptr, hcsc, high_seg);
```

where:

1. lot_ptr (Input)
   is a pointer to the LOT.

2. isot_ptr (Input)
   is a printer to the ISOT.

3. linkage_ptr (Input)
   is a pointer to the linkage section to be discarded.

4. hscs (Input)
   is one less than the segment number of the first segment that can be unsnapped.

5. high_seg (Input)
   is the number of LOT slots used in searching for links to be restored.

Name: list_dir_info_

The list_dir_info_ subroutine is used by the list_dir_info, rebuild_dir, and comp_dir_info commands to list the values in a single entry in a directory information segment created by the save_dir_info command.

Usage

```
declare list_dir_info_ entry (ptr, fixed bin, char(1));

call list_dir_info_ (ptr, mode, prefix);
```

where:

1. ptr (Input)
   points to an entry in the dir_info segment (created by invoking the save_dir_info command).

2. mode (Input)
   is the verbosity desired. It can be 0, 1, or 2 (where 0 is the least verbose).

3. prefix (Input)
   is a one-character prefix for every line printed.

Notes

Output from the list_dir_info_ subroutine is written on the user_output I/O switch. It consists of a series of lines, each of the form:

```
item_name:  value
```

The prefix character is appended to the beginning of each line.

The list below gives the output items for each verbosity level, for segments, directories, and links. Verbosity level 1 returns information listed in 0 and 1; verbosity level two returns information listed in 0, 1, and 2.

For segments:

0. names
   type
   date used
   date modified

1. date branch modified
   records used
   bit count
   bit count author
   max length
   safety switch
   property list

2. ACL
   data dumped
   current length
   device ID
   move device ID
   copy switch
   ring brackets
   unique ID
   author

For directories:

0. names
   type
   date used
   date modified

1. date branch modified
   bit count
   records used
   quota
   date dumped
   current length
   device ID
   move device ID
   copy switch
   ring brackets
   unique ID
   author
   bit count author
   max length
   safety switch
   property list

2. ACL
   initial seg ACL
   initial dir ACL

For links:

0. names
   type
   target

1. date link modified

2. date link dumped

mdc_$pvname_info

Name: mdc_$pvname_info

This entry point gets various kinds of information about a specified storage-system physical volume.

Usage

```
dcl  mdc_$pvname_info entry (char (*), bit (36) aligned, char (*),
     bit (36) aligned, fixed bin, fixed bin (35));

call mdc_$pvname_info (pvname, pvid, lvname, lvid, device_type, code);
```

where:

1. pvname (Input)
   is the name of the physical volume about which information is to be returned.

2. pvid (Output)
   is the physical volume id of the specified volume. It can be used as a parameter to ring-zero volume and partition interfaces.

3. lvname (Output)
   is the name of the logical volume to which the physical volume belongs.

4. lvid (Output)
   is the logical volume id of the logical volume to which the physical volume belongs.

5. device_type (Output)
   is a number indicating what type of device the specified physical volume is mounted on. The names and characteristics of these devices are listed in various arrays declared in the include file fs_dev_types.incl.pl1.

6. code (Output)
   is a standard system-status code. It is nonzero if the information about the volume cannot be obtained or if the volume does not exist.

Name: parse_channel_name_

The parse_channel_name_ subroutine parses a character string that is intended to be an IOM channel number.

Usage

```
dcl parse_channel_name_ entry (char (*), fixed bin (3), fixed bin (6),
     fixed bin (35));

call parse_channel_name_ (arg, iom, channel, code);
```

where:

1. arg
   is the character string to be parsed. It must be of the format:

   {tag}number
   where tag is IOM tag (a through d) and number is an octal channel number from 0 to 77. If tag is specified, the IOM selected must appear in the configuration deck. The tag must be specified if multiple IOMs are configured.

2. iom (Output)
   is the iom the channel is connected.

3. channel (Output)
   is the channel number.

4. code
   is 0 if arg is a valid representation of a channel; otherwise, error_table_$bad_channel.

Name: parse_file_

The parse_file_ subroutine provides a facility for parsing ASCII text into symbols and break characters. It is recommended for occasionally used text-scanning applications. In applications where speed or frequent use are important, in-line PL/I code is recommended (to do parsing) instead.

A restriction of the subroutine is that the text to be parsed must be an aligned character string.

The initialization entry points, parse_file_$parse_file_init_name and parse_file_$parse_file_init_ptr, save a pointer to the text to be scanned and a character count in internal static storage. Thus, only one text can be parsed at one time.

Entry: parse_file_$parse_file_init_name

This entry point initializes the subroutine given a directory and an entry-point name. It gets a pointer to the desired segment and saves it for subsequent calls in internal static.

Usage

```
declare parse_file_$parse_file_init_name entry (char(*), char(*), ptr,
     fixed bin(35));

call parse_file_$parse_file_init_name (dir_name, entryname, ptr, code);
```

where:

1. dir_name (Input)
   is the directory name portion of the pathname of the segment to be parsed.

2. entryname (Input)
   is the entryname of the segment to be parsed.

3. ptr (Output)
   is a pointer to the segment.

4. code (Output)
   is a standard status code. It is zero if the segment is initiated. If nonzero, the segment cannot be initiated. It can return any code from hcs_$initiate except error_table_$segknown.

Entry: parse_file_$parse_file_init_ptr

This entry point initializes the parse_file_ subroutine with a supplied pointer and character count. It is used in cases where a pointer to the segment to be parsed is already available.

Usage

```
declare parse_file_$parse_file_init_ptr entry (ptr, fixed bin);

call parse_file_$parse_file_init_ptr (ptr, cc);
```

where:

1. ptr (Input)
   is a pointer to a segment or an aligned character string.

2. cc (Input)
   is the character count of the ASCII text to be scanned.

Entry: parse_file_$parse_file_set_break

This entry point is used to define break characters. Normally, all nonalphanumeric characters are break characters (including blank and newline).

Usage

```
declare parse_file_$parse_file_set_break entry (char(*));

call parse_file_$parse_file_set_break (cs);
```

where cs is a control string. Each character found in cs is made a break character. (Input)

Entry: parse_file_$parse_file_unset_break

This entry point renders break characters as normal alphanumeric characters. It is not possible to unset blank, newline, or comment delimiters, however. These are always treated as break characters.

Usage

```
declare parse_file_$parse_file_unset_break entry (char(*));

call parse_file_$parse_file_unset_break (cs);
```

where cs is a control string, each character of which is made a nonbreaking character. (Input)

Entry: parse_file_$parse_file_

This entry point scans the text file and returns the next break character or symbol. Blanks, newline characters, and comments enclosed by /* and */, however, are skipped over.

Usage

```
declare parse_file_ entry (fixed  bin, fixed  bin, fixed  bin(1),
     fixed bin(1));

call parse_file_ (ci, cc, break, eof);
```

where:

1. ci (Output)
   is an index to the first character of the symbol or break character. (The first character of the text is considered to be character 1.)

2. cc (Output)
   is the number of characters in the symbol.

3. break (Output)
   is set to 1 if the returned item is a break character; otherwise, it is 0.

4. eof (Output)
   fin
   is set to 1 if the end of text has been reached; otherwise, it is 0.

Entry: parse_file_$parse_file_ptr

This entry point is identical to the parse_file$parse_file_ entry point except that a pointer (with bit offset) to the break character or the symbol is returned instead of a character index.

Usage

```
declare parse_file_$parse_file_ptr entry (ptr, fixed bin, fixed bin(1),
     fixed bin(1));

call parse_file_$parse_file_ptr (ptr, cc, break, eof);
```

where:

1. ptr (Output)
   is a pointer to the symbol or the break character.

2. cc (Output)
   is the same as above.

3. break (Output)
   is the same as above.

4. eof (Output)
   is the same as above.

Entry: parse_file_$parse_file_cur_line

This entry point returns to the caller the current line of text being scanned. This entry is useful in printing diagnostic error messages.

Usage

```
declare parse_file_$parse_file_cur_line entry (fixed bin, fixed bin);

call parse_file_$parse_file_cur_line (ci, cc);
```

where ci and cc are the same as in the parse_file_$parse_file_ above.

Entry: parse_file_$parse_file_line_no

This entry point returns to the caller the current line number of text being scanned. This entry is useful in printing diagnostic error messages.

Usage

```
declare parse_file_$parse_file_line_no entry (fixed bin);

call parse_file_$parse_file_line_no (cl);
```

where cl is the number of the current line. (Output)

Examples

Suppose the file zilch in the directory dir_name contains the following text:

```
name:  foo;  /*foo program*/
pathname:  >bar;
linkage;
end;
fini;
```

The following calls could be made to initialize the parsing of zilch:

```
call parse_file_$parse_file_init_name (dir_name, zilch, ptr, code);
call parse_file_$parse_file_unset_break (">_");
declare atom char (cc) unaligned based (p);
```

Subsequent calls to the parse_file_$parse_file_ptr entry point would then yield the following:

atom	break	eof
name	0	0
:	1	0
foo	0	0
;	1	0
pathname	0	0
:	1	0
>bar	0	0
;	1	0
linkage	0	0
;	1	0
end	0	0
;	1	0
fini	0	0
;	1	0
-	-	-

phcs_$read_disk_label

Name: phcs_$read_disk_label

This entry point is used to read the label of a storage-system disk drive. The label is described by the structure "label," in the include file fs_vol_label.incl.pl1.

Usage

```
dcl  phcs_$read_disk_label entry (bit (36) aligned, pointer,
     fixed bin (35));

call phcs_$read_disk_label (pvid, label_ptr, code);
```

where:

1. pvid (Input)
   is the physical volume id of the disk whose label is to be read. The physical volume id is used instead of the volume name because this is a ring-zero interface, and volume names are not accessible by ring zero; hence, all ring-zero interfaces that reference physical volumes use the pvid. A pvname may be converted to a pvid by calling the subroutine mdc_$find_volname or may have been returned by a previous call to find_partition_.

2. label_ptr (Input)
   is a pointer to the user-supplied buffer in which to read the label. The label is 1024 words long and is described in fs_vol_label.incl.pl1.

3. code (Output)
   is a nonstandard status code. It will be one of the following:

   zero
   indicates that the label was successfully read.

   error_table_$pvid_not_found
   indicates that the specified physical volume is not presently mounted.

   an integer between 1 and 10
   indicates that a physical disk error occurred while trying to read the label. Error messages for physical disk errors are declared in the include file fsdisk_errors.incl.pl1, in the array fsdisk_error_message.

Name: rehash_

This subroutine rehashes (reformats into a different size) a hash table of the form that is maintained by the hash_ subroutine. In most cases, hash_ calls rehash_ automatically when a table becomes too full. For hash tables that are embedded in larger data bases, the data base maintainer must monitor the density of the hash table and call rehash_ when necessary to maintain the optimal table size. See the description of the hash_ subroutine for more information.

Usage

```
declare rehash_ entry (ptr, fixed bin, fixed bin(35));

call rehash_ (table_ptr, size, code);
```

where:

1. table_ptr (Input)
   is a pointer to the table to be rehashed.

2. size (Input)
   is the new size of the hash table. See the description of hash_$opt_size.

3. code (Output)
   is a standard status code. It may be:

   0
   table rehashed successfully

   error_table_$invalid_elsize
   size is too large

   error_table_$full_hashtbl
   size is not large enough to hold all the entries in the current hash table.

Name: ring0_get_

The ring0_get_ subroutine returns the name and pointer information about hardcore segments.

Entry: ring0_get_$segptr

This entry point returns a pointer to a specified ring 0 segment. Only the name is used to determine the pointer.

Usage

```
declare ring0_get_$segptr entry (char(*), char(*), ptr, fixed bin);

call ring0_get_$segptr (dir_name, entryname, seg_ptr, code);
```

where:

1. dir_name (Input)
   is ignored.

2. entryname (Input)
   is the name of the ring 0 segment for which a pointer is desired.

3. seg_ptr (Output)
   is a pointer to the segment.

4. code (Output)
   is a standard status code. It is nonzero if, and only if, the entry is not found.

Notes

If the entry is not found, seg_ptr is returned null.

Entry: ring0_get_$segptr_given_slt

This entry point is analogous to the segptr entry point except that external SLT (Segment and Loading Table) name tables are used, instead of the versions of these tables currently being used by the system.

Usage

```
declare ring0_get_$segptr_given_slt entry (char(*), char(*), ptr,
     fixed bin);

call ring0_get_$segptr_given_slt (dir_name, entryname, seg_ptr, code, sltp,
     namep);
```

where:

1. dir_name (Input)
   is ignored.

2. entryname (Input)
   is the name of the ring 0 segment for which a pointer is desired.

3. seg_ptr (Output)
   is a pointer to the segment.

4. code (Output)
   is a standard status code. It is nonzero if, and only if, the entry is not found.

5. sltp (Input)
   is a pointer to an SLT that contains information about the segment.

6. namep (Input)
   is a pointer to a name table (associated with the above SLT) containing the names of segments.

Entry: ring0_get_$name

This entry point returns the primary name and directory name of a ring 0 segment when given a pointer to the segment.

Usage

```
declare ring0_get_$name entry (char(*), char(*), ptr, fixed bin);

call ring0_get_$name (dir_name, entryname, seg_ptr, code);
```

where:

1. dir_name (Output)
   is the pathname of the directory of the segment. If the segment does not have a pathname (as is the case for most hardcore segments), this is returned as a null string.

2. entryname (Output)
   is the primary name of the segment.

3. seg_ptr (Input)
   is a pointer to the ring 0 segment.

4. code (Output)
is a standard status code. It is nonzero if, and only if, seg_ptr does not point to a ring 0 segment.

Entry: ring0_get_$name_given_slt

This entry point is analogous to the name entry point except that external SLT (Segment Loading Table) and name tables are used, instead of the versions of these tables currently being used by the system.

Usage

```
declare ring0_get_$name_given_slt entry (char(*), char(*), ptr, fixed bin);

call ring0_get_$name_given_slt (dir_name, entryname, seg_ptr, code, sltp,
     namep);
```

where:

1. dir_name (Output)
is the pathname of the directory of the segment. If the segment does not have a pathname (as is the case for most hardcore segments), this is returned as a null string.

2. entryname (Output)
is the primary name of the segment.

3. seg_ptr (Input)
is a pointer to the ring 0 segment.

4. code (Output)
is a standard status code. It is nonzero if, and only if, seg_ptr does not point to a ring 0 segment.

5. sltp (Input)
is a pointer to an SLT that contains information about the segment.

6. namep (Input)
is a pointer to a name table (associated with the above SLT) containing the names of segments.

Entry: ring0_get_$names

This entry point returns all the names and the directory name of a ring 0 segment when given a pointer to the segment.

Usage

```
declare ring0_get_$names entry (char(*), ptr, ptr, fixed bin);

call ring0_get_$names (dir_name, names_ptr, seg_ptr, code);
```

where:

1. dir_name (Output)
   is the pathname of the directory of the segment.

2. names_ptr (Output)
   is a pointer to a structure (described in "Notes" below) containing the names of the segment.

3. seg_ptr (Input)
   is a pointer to the ring 0 segment.

4. code (Output)
   is nonzero if, and only if, seg_ptr does not point to a ring 0 segment.

Notes

The following structure is used:

```
dcl 1 segnames based (namesptr) aligned,
    2 count    fixed bin,
    2 names    (50 refer (segnames.count)),
      3 length fixed bin,
      3 name   (char(32);
```

where:

1. count
   is the number of names.

2. names
   is a substructure containing an array of segment names.

3. length
   is the length of the name in characters.

4. name
   is the space for the name.

Entry: ring0_get_$definition

This entry point is used to ascertain the offset of a symbol in a hardcore se in the running Multics Supervisor.

Usage

```
declare ring0_get_$definition entry (ptr, char(*), char(*), fixed bin(18),
     fixed bin, fixed bin(35));

call ring0_get_$definition (seg_ptr, component_name,
     sym_name, offset, type, code);
```

where:

1. seg_ptr (Input/Output)
   is a pointer to the base of the segment in which it is desired to obtain a symbol offset. If supplied as null, the segment which bears the name component_name in the SLT will be used, and seg_ptr will be returned as output as a pointer to the base of this segment.

2. component_name (Input)
   is the name of the segment or segment bound component in which the symbol, sym_name, is to be found. If sym_name is an unambiguous reference in the segment defined by seg_ptr, this parameter may be given as a null string. If seg_ptr is given as null, this parameter must be supplied, and specifies the segment name as well.

3. sym_name (Input)
   is the name of the external symbol in the segment specified by seg_ptr or component_name. If more than one external symbol of this name appears in this segment, component_name is used to select the correct component.

4. offset (Output)
   is the offset of this definition, if found, into the section of the specified segment as specified by type.

5. type (Output)
   is the definition type of this definition, as specified in the MPM Subsystem Writer's Guide, detailing in which section of the specified segment this definition resides.

6. code (Output)
   is a standard status code. If the the segment specified has no definitions, error_table_$no_defs is returned.

Entry: ring0_get_$definition_given_slt

This entry point is used to ascertain the offset of a symbol in a hardcore segment in other than the running Multics supervisor. Copies of the SLT, SLT name table, and hardcore-definitions segment are supplied.

Usage

```
declare ring0_get_$definition_given_slt entry (ptr, char(*), char(*),
     fixed bin(18), fixed bin, fixed bin(35), ptr, ptr, ptr);

call ring0_get_$definition_given_slt (seg_ptr, component_name, syn_name,
     offset, type, code, slt_ptr, nametbl_ptr, deftbl_ptr):
```

where:

1. seg_ptr (Input/Output)
   is as above.

2. component_name (Input)
   is as above.

3. sym_name (Input)
   is as above.

4. offset (Output)
   is as above.

5. type (Output)
   is as above.

6. code (Output)
   is as above.

7. slt_ptr (Input)
   is a pointer to the copy of the segment loading table (SLT) to be used.

8. nametbl_ptr (Input)
   is a pointer to the corresponding copy of the SLT name table.

9. deftbl_ptr (Input)
   is a pointer to the corresponding copy of the hardcore definitions segment (definitions_).

Name: ring_zero_peek_

The ring_zero_peek_ subroutine is used to copy information out of an inner-ring segment. The user must have access to either the phcs_ gate or the metering_ring_zero_peek_ gate in order to use any of the entry points in this subroutine. The phcs_ gate allows unrestricted access to all inner-ring segments; metering_ring_zero_peek_ allows the user to examine specifically those data bases that are useful for metering the system. The program chooses the appropriate gate depending on the user's access and the segments being examined.

Usage

```
declare ring_zero_peek_ entry (ptr, ptr, fixed bin(19), fixed bin(35));

call ring_zero_peek_ (ptr0, ptr_user, nwords, code);
```

where:

1. ptr0 (Input)
   is a pointer to the data in ring 0 that is to be copied out.

2. ptr_user (Input)
   is a pointer to the region in the user's address space where the data is to be copied.

3. nwords (Input)
   is the number of words to be copied.

4. code (Output)
   is the standard status code that is nonzero if the user did not have access to the requested data.

Entry: ring_zero_peek_$by_name

This entry point is used to copy information out of a named segment in the Multics supervisor. It is like ring_zero_peek_, except that the name of the ring zero segment is provided, rather than a pointer to it.

Usage

```
dcl ring_zero_peek_$by_name entry (char(*), fixed bin(18), pointer,
     fixed bin(19), fixed bin(35));

call ring_zero_peek_$by_name (segment_name, offset, copy_ptr, word_count,
     code);
```

where:

1. segment_name (Input)
   is the name of the supervisor segment from which data is to be copied. It may not be a pathname.

2. offset (Input)
is the offset from the beginning of the segment at which copying is to start. It may be specified as zero to cause copying to start from the base of the segment.

3. copy_ptr (Input)
is a pointer to the area in the outer ring where the data is to be copied.

4. word_count (Input)
is the number of words to be copied.

5. code (Output)
is a standard status code. It is nonzero if the segment cannot be found, or if the user does not have sufficient access to copy the requested data from it.

Notes

This entry point can be used to avoid a call to ring0_get_. For examining segments in the supervisor, this entry point and the by_definition entry point are recommended because they are much simpler to use than ring0_get_, and they are only minimally less efficient. Generally, it will be nearly as efficient to use this entry point as it would be to save static pointers to inner-ring objects.

Entry: ring_zero_peek_$by_definition

This entry point is used to copy information out of a named segment in the Multics supervisor, starting at a named symbol. It is like ring_zero_peek_$by_name, except that the copying is done from the specified definition, rather than from the base of the segment.

Usage

```
dcl ring_zero_peek_$by_definition entry (char(*), char(*), fixed bin(18),
     pointer, fixed bin(19), fixed bin(35));

call ring_zero_peek_$by_definition (segment_name, symbol_name, offset,
     ptr_user, word_count, code);
```

where:

1. segment_name (Input)
is the name of the supervisor segment from which words are to be copied. It may not be a pathname.

2. symbol_name (Input)
is the name of the external symbol in the specified segment at which copying is to start.

3. offset (Input)
is the offset from the specified definition at which copying is to start. It may be specified as zero to cause copying to start at the specified definition.

4. ptr_user (Input)
is a pointer to the area in the outer ring where the data is to be copied.

5. word_count (Input)
is the number of words to be copied.

6. code (Output)
is a standard status code. It is nonzero if the segment cannot be found, if the specified external symbol does not exist or is ambiguous, or if the user does not have sufficient access to copy the requested data.

Notes

See "Notes" to ring_zero_peek_$by_name entry point.

Entry: ring_zero_peek_$get_max_length

This entry point is used to determine the maximum length of a named ring-zero segment.

Usage

```
dcl ring_zero_peek_$get_max_length entry (char(*), fixed bin(19),
     fixed bin(35));

call ring_zero_peek_$get_max_length (seg_name, max_length, code);
```

Arguments:

1. seg_name (Input)
is the name of the ring-zero segment.

2. max_length (Output)
is the maximum length (in words) of the segment.

3. code (Output)
is a standard status code. It is nonzero if the user does not have sufficient access to copy the requested data, or if the segment does not exist.

Entry: ring_zero_peek_$get_max_length_ptr

This entry point is used to determine the maximum length of a specified segment by examining its SDW. The user must have sufficient access to examine the SDW for the segment.

Usage

```
dcl ring_zero_peek_$get_max_length_ptr entry (pointer, fixed bin(19),
     fixed bin(35));

call ring_zero_peek_$get_max_length_ptr (seg_ptr, max_length, code);
```

where:

1. seg_ptr (Input)
   is a pointer to the segment for which the max length is to be returned. If the segment is not active at the time of the call, the user must have sufficient access to reference the segment, and this reference will cause a segment fault.

2. max_length (Output)
   is the maximum length (in words) of the segment.

3. code (Output)
   is a standard status code. It is nonzero if the user does not have sufficient access to copy the requested data, or if the segment does not exist.

Name: sort_items_

The sort_items_ subroutine provides a generalized, yet highly efficient, sorting facility. Entry points are provided for sorting fixed binary (35) numbers, float binary (63) numbers, fixed-length character strings, varying character strings, and fixed-length bit strings. A generalized entry point is provided for sorting other data types (including data structures and data aggregates) and for sorting data into a user-defined order.

The procedure implements the QUICKSORT algorithm of M. H. van Emden, including the Wheeler modification to detect ordered sequences.

The subroutine takes a vector of unaligned pointers to the data items to be sorted and rearranges the elements of this vector to point to the data items in correct order. Only the pointers are moved or copied into temporary storage; the data items remain where they were when sort_items_ was invoked.

Entry: sort_items_$fixed_bin

This entry point sorts a group of aligned fixed binary (35,0) numbers into numerical order by reordering a pointer array whose elements point to the numbers in the group.

Usage

```
declare sort_items_$fixed_bin entry (ptr);

call sort_items_$fixed_bin (v_ptr);
```

where v_ptr points to a structure containing an array of unaligned pointers to the aligned fixed binary (35,0) numbers to be sorted. (Input)

Notes

The structure pointed to by v_ptr is to be declared as follows, where n is the value of v.n:

```
dcl 1 v aligned,
    2 n fixed bin (18),
    2 vector (n) ptr unaligned;
```

Entry: sort_items_$float_bin

This entry point sorts a group of aligned float binary (63) numbers into numerical order by reordering a pointer array whose elements point to the numbers in the group.

Usage

```
declare sort_items_$float_bin entry (ptr);

call sort_items_$float_bin (v_ptr);
```

where:

1. v_ptr (Input)
   points to the above structure containing an array of unaligned pointers to the aligned float binary (63) numbers to be sorted.

Entry: sort_items_$char

This entry point sorts a group of fixed-length unaligned character strings into ASCII collating sequence by reordering a pointer array whose elements point to the character strings in the group.

Usage

```
declare sort_items_$char entry (ptr, fixed bin (24));

call sort_items_$char (v_ptr, string_lth);
```

where:

1. v_ptr (Input)
   points to the structure (described in "Notes" above) containing an array of unaligned pointers to the varying character strings to be sorted.

2. string_lth (Input)
   is the length of each character string.

Entry: sort_items_$varying_char

This entry point sorts a group of varying character strings into ASCII collating sequence by reordering a pointer array whose elements point to the character strings in the group.

Usage

```
declare sort_items_$varying_char entry (ptr);

call sort_items_$varying_char (v_ptr);
```

where v_ptr points to the structure (described in "Note" above) containing an array of unaligned pointers to the varying character strings to be sorted. (Input)

Entry: sort_items_$bit

This entry point sorts a group of fixed-length unaligned bit strings into bit string order by reordering a pointer array whose elements point to the bit strings in the group. Bit string ordering guarantees that, if each ordered bit string were converted to a binary natural number, the binary value would be less than or equal to the value of its successors.

Usage

```
declare sort_items_$bit entry (ptr, fixed bin (24));

call sort_items_$bit (v_ptr, length);
```

where:

1. v_ptr (Input)
   points the structure (described in "Note" above) containing an array of unaligned pointers to the fixed-length unaligned bit strings to be sorted.

2. length (Input)
   is the number of bits in each string.

Entry: sort_items_$general

This entry point sorts a group of arbitrary data elements, structures, or other aggregates into a user-defined order by reordering a pointer array whose elements point to the data items in the group. The structure of data items, the information field or fields within each item by which items are sorted, and the data ordering principle are all decoupled from the sorting algorithm by calling a user-supplied function to order pairs of data items. The function is called with pointers to a pair of items. It must compare the items and return a value that indicates whether the first item of the pair is less than, equal to, or greater than the second item. The sorting algorithm reorders the elements of the pointer array based upon the results of the item comparisons.

Usage

```
declare sort_items_$general entry (ptr, entry);

call sort_items_$general (v_ptr, function);
```

where:

1. v_ptr (Input)
   points the structure (described in "Note" above containing an array of unaligned pointers to the data items to be sorted.

2. function (Input)
   is a user-supplied ordering function. Its calling sequence is shown under "Note" below.

Note

The sort_items_$general entry point calls a user-supplied function to compare pairs of data items. This function must know the structure of the data items being compared, the field or fields within each item that are to be compared, and the ordering principle to be used in performing the comparisons. The function returns a relationship code as its value. The calling sequence of the function is shown below.

```
declare function entry (ptr unaligned, ptr unaligned)
        returns (fixed bin(1));

value = function (ptr_first_item, ptr_second_item);
```

where:

1. ptr_first_item (Input)
   is an unaligned pointer to the first data item.

2. ptr_second_item (Input)
   is an unaligned pointer to a data item to be compared with the first data item.

3. value (Output)
is the value of the first data item compared to the second data item. It can be:

-1 the first data item is less than the second.
 0 the first data item is equal to the second.
+1 the first data item is greater than the second.

Example

A simple example of a user-supplied ordering function is shown below. It compares pairs of fixed binary (35,0) numbers. If this function is passed to the sort_items_$general entry point, it performs the same function as a call to the sort_items_$fixed_bin entry point, but with less efficiency because of the overhead involved in calling the function.

```
function: procedure (p1, p2) returns (fixed bin(1));

    declare (p1, p2) ptr unaligned,
         datum fixed bin(35,0) based;

         if p1 -> datum < p2 -> datum then
              return (-1);
         else if p1 -> datum = p2 -> datum then
              return ( 0);
         else
              return (+1);

         end function;
```

Name: sort_items_indirect_

The sort_items_indirect_ subroutine is a variation of the sort_items_$general entry point. It provides a facility for sorting a group of data items, based upon the value of an information field that is logically associated with each item but resides at a varying offset from the beginning of each item. A name in the name list associated with the status block returned by the hcs_$status_ entry point is an example of such an information field.

The sort_items_indirect_ subroutine provides high performance entry points for sorting data items by the value of a single fixed binary (35) field, float binary (63) field, fixed-length bit string field, fixed-length character string field, or adjustable length character string field associated with each item. A generalized entry point is provided for sorting other types of information fields, for sorting aggregate information fields, or for sorting items into a user-defined order.

To use the sort_items_indirect_ subroutine, for some entries the caller must create three arrays: a vector of pointers to the data items being sorted (the item vector), a vector of pointers to the single information field within each item on which the sort is based (the field vector), and an array of indices into these two vectors. For other entries, only two arrays are required: a vector of pointers to the data items being sorted and an array of indices into the vectors. This index array is initialized sequentially with integers by sort_items_indirect_, which then reorders these indices to index the pointer vectors to the data items in correct order.

Only indices are moved or copied into temporary storage. Vector elements and data items remain where they were when sort_items_indirect_ was invoked.

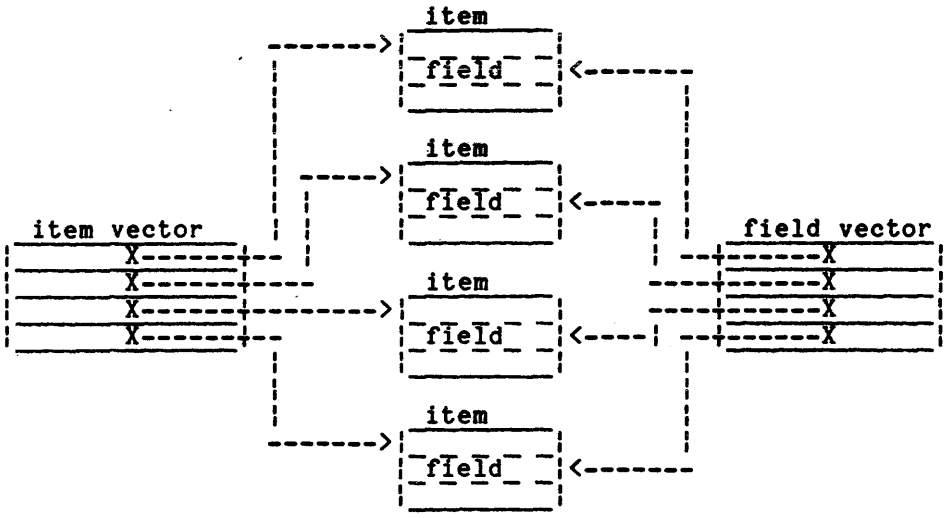
This procedure differs from that used in the sort_items_ subroutine in that an array of indices into the vector is sorted rather than the vector itself. This allows the caller to create two vectors of pointers: one containing pointers to the data items to be sorted and one containing pointers to the particular data field within each item on which the item is to be sorted. There is a one-to-one correspondence between the elements of the data items vector and the elements of the data field within each item vector. This correspondence is maintained across the reordering of the index array. Thus, the index array provides indices into the sorted list of data fields and also into the sorted list of data items containing these fields.

Notes

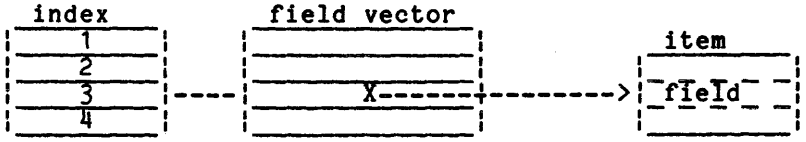
To use the sort_items_indirect_$adj_char entry point, one additional array must be created: an array of lengths of the adjustable length character string information fields on which the sort is based.

For the sake of simplicity, the sort information field is shown as part of the items being sorted in each of the diagrams below. A more general application might show each item containing a locator variable that addresses the sort field(s) associated with that item.

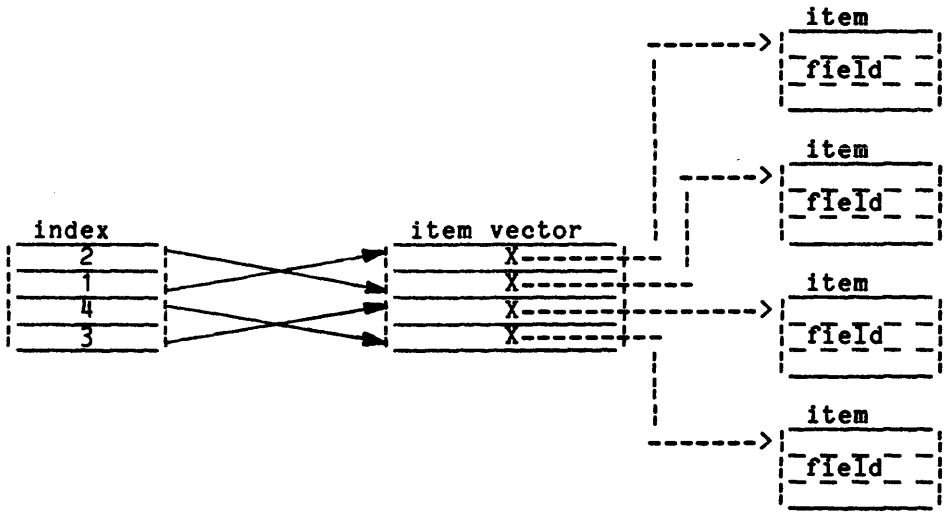
The one-to-one correspondence between elements of the item vector and elements of the field vector is shown below.



The array of indices can be used to reference elements of both vectors. The field vector and index array are passed to the sort_items_indirect_ subroutine, which references the sorting field in each item through elements of these two arrays, as shown below.



The sort_items_indirect_ subroutine reorders the index values so that values selected sequentially from the index array reference pointer to the elements of a sorted list of information fields. Because the sorting process involves only the interchange of index values, there is still a correspondence between the elements of the item vector and the elements of the field vector after the sort is complete.



If the information field upon which the sort is based is located at a known offset from the beginning of each item, then the calling program can avoid creating the index array and the item vector by using the sort_items_ subroutine. (This subroutine cannot process adjustable length fields.) The field vector is passed to the sort_items_ subroutine, and then the elements of the item vector are computed by applying the appropriate offset to the corresponding field vector elements.

The QUICKSORT algorithm of M. H. van Emden (including the Wheeler modification to detect ordered sequences) is used to perform the sort.

Entry: sort_items_indirect_$fixed_bin

This entry point sorts a group of information fields, which are aligned fixed binary (35,0) numbers, into numerical order by reordering an index array. The elements of this index array are indices into an array of unaligned pointers to the numbers in the group.

Usage

```
declare sort_items_indirect_$fixed_bin entry (ptr, ptr);

call sort_items_indirect_$fixed_bin (v_ptr, i_ptr);
```

where:

1. v_ptr (Input)
   points to a structure containing an array of unaligned pointers to the aligned fixed binary (35,0) numbers to be sorted. The structure pointed to by v_ptr is to be declared as follows, where n is the number of elements to be sorted.

```
dcl 1 v      aligned,
    2 n      fixed bin (18),
    2 vector (n) ptr unaligned;
```

2. i_ptr (Input)
   points to a structure containing an ordered array of fixed binary (24) indices into the unaligned pointer array. The structure pointed to by i_ptr is to be declared as follows, where n is the number of elements to be sorted. Since sort_items_indirect_ sets the i.n and i.array elements, the user needs not set them prior to calling the subroutine.

```
dcl 1 i aligned,
    2 n fixed bin (18),
    2 array (n) fixed bin (18);
```

Entry: sort_items_indirect_$float_bin

This entry point sorts a group of information fields, which are aligned float binary (63,0) numbers, into numerical order by reordering an index array. The elements of this index array are indices into an array of unaligned pointers to the numbers in the group.

Usage

```
declare sort_items_indirect_$float_bin entry (ptr, ptr);

call sort_items_indirect_$float_bin (v_ptr, i_ptr);
```

where:

1. v_ptr (Input)
   points to the above structure v containing an array of unaligned pointers to the aligned float binary (63,0) numbers to be sorted.

2. i_ptr (Input)
   points to the above structure i containing an ordered array of fixed binary (18) indices into the unaligned pointer array.

Entry: sort_items_indirect_$char

This entry point sorts fixed-length unaligned character strings into ASCII collating sequence by reordering an index array whose elements are indices into a pointer array that points to the strings. All the strings must be the same length.

Usage

```
declare sort_items_indirect_$char entry (ptr, ptr, fixed bin (21));

call sort_items_indirect_$char (v_ptr, i_ptr, string_lth);
```

where:

1. v_ptr (Input)
   points to the above structure v containing an array of unaligned pointers to the fixed-length unaligned character string to be sorted.

2. i_ptr (Input)
   points to the above structure i of fixed bin indices into the unaligned pointer array.

3. string_lth (Input)
   indicates the length of each character string.

Entry: sort_items_indirect_$varying_char

This entry point sorts a group of information fields, which are varying unaligned character strings, into ASCII collating sequence by reordering an index array. The elements of this index array are indices into an array of pointers to the character strings in the group.

Usage

```
declare sort_items_indirect_$varying_char entry (ptr, ptr);

call sort_items_indirect_$varying_char (v_ptr, i_ptr);
```

where:

1. v_ptr (Input)
   points to the above structure v containing an array of unaligned pointers to the varying fixed-length character strings to be sorted.

2. i_ptr (Input)
   points to the above structure i containing an ordered array of fixed binary (18) indices into the unaligned pointer array.

Entry: sort_items_indirect_$bit

This entry point sorts a group of information fields, which are fixed-length unaligned bit strings into bit-string order by reordering an index array. The elements of this index array are indices into an array of pointers to the bit strings in the group. Bit string ordering guarantees that, if each ordered bit string is converted to a binary natural number, the binary value is less than or equal to the value of each of its successors.

Usage

```
declare sort_items_indirect_$bit entry (ptr, ptr, fixed bin (24));

call sort_items_indirect_$bit (v_ptr, i_ptr, length);
```

where:

1. v_ptr (Input)
   points to the above structure v containing an array of unaligned pointers to the fixed-length unaligned bit strings to be sorted.

2. i_ptr (Input)
   points to the above structure i containing an ordered array of fixed binary (24) indices into the unaligned pointer array.

3. length (Input)
   is the number of bits in each string.

Entry: sort_items_indirect_$general

This entry point sorts a group of information fields (which are arbitrary data elements, structures, or other aggregates) into a user-defined order. It does this by reordering an array of indices into a pointer array. The elements of this index array point to the sort information field within the data items of the group. The structure and data type of the information field and the data ordering principle are decoupled from the sorting algorithm by calling a user-supplied function to order pairs of information fields. The function is called with pointers to a pair of fields. It must compare the fields and return a value that indicates whether the first field of the pair is less than, equal to, or greater than the second field. The sorting algorithm reorders the elements of the index array based upon the results of the information field comparisons.

Usage

```
declare sort_items_indirect_$general entry (ptr, ptr, entry);

call sort_items_indirect_$general (v_ptr, i_ptr, function):
```

where:

1. v_ptr (Input)
   points to the above structure v containing an array of unaligned pointers to the information fields to be sorted.

2. i_ptr (Input)
   points to the above structure i containing an ordered array of fixed bin (18) indices into the unaligned pointer array.

3. function (Input)
   is a user-supplied ordering function. (See "Note" below.)

Note

The sort_items_indirect_$general entry point calls a user-supplied function to compare pairs of data items. This function must know the structure and data type of the information fields, and it must know the ordering principle to be used to compare a pair of information fields. The function returns a relationship code as its value. The calling sequence of the function is shown below.

```
declare function entry (ptr unaligned, ptr unaligned) returns
     (fixed bin(1));

value = function (ptr_1st_field, ptr_2nd_field);
```

where:

1. ptr_1st_field (Input)
   is an unaligned pointer to the first information field.

2. ptr_2nd_field (Input)
   is an unaligned pointer to an information field to be compared with the first information field.

3. value (Output)
is the value of the first information field compared to the second information field. It can be:

-1 first information field is less than the second.
 0 first information field is equal to the second.
+1 first information field is greater than the second.

A simple example of a user-supplied ordering function is shown in the sort_items_ subroutine.

Entry: sort_items_indirect_$adj_char

This entry point sorts a group of information fields, which are unaligned adjustable length character strings, into ASCII collating sequence order by reordering an index array. The elements in this index array are indices into an array of unaligned pointers to the character strings in the group.

Usage

```
declare sort_items_indirect_$adj_char (ptr, ptr, ptr);

call sort_items_indirect_$adj_char (v_ptr, i_ptr, l_ptr);
```

where:

1. v_ptr (Input)
points to the above structure v containing an array of unaligned pointers to the unaligned adjustable length character strings to be sorted.

2. i_ptr (Input)
points to the above structure i containing an ordered array of indices into the unaligned pointer array.

3. l_ptr (Input)
points to a structure containing an array of lengths of the unaligned adjustable length character strings to be sorted. The structure pointed to by l_ptr is to be declared as follows, where n is the number of elements to be sorted.

```
dcl 1 l      aligned,
    2 n      fixed bin (18),
    2 vector (n) fixed bin (21);
```

Name: stu_

The stu_ (symbol table utility) subroutine provides a number of entry points for retrieving information from the runtime symbol table section of an object segment generated by the PL/I, FORTRAN, or COBOL compilers. (See the pl1, fortran, and cobol commands in the MPM Commands, Order No. AG92.) A runtime symbol table is produced when a program is compiled with the -table control argument or when a runtime symbol table is required to support a feature of the language such as PL/I data-directed or FORTRAN NAMELIST input/output statements. A partial symbol table, containing only a statement map, is produced when a program is compiled with the -brief_table control argument.

It is anticipated that the format of the symbol table will change sometime in the future. In that case, the entry points described below will not work with the new format.

Entry: stu_$decode_runtime_value

This entry point is called to decode encoded values (e.g., string length or arithmetic precision) stored in a runtime_symbol node.

Usage

```
declare stu_$decode_runtime_value entry (fixed bin(35), ptr, ptr, ptr, ptr,
     ptr, fixed bin) returns (fixed bin(35));

value = stu_$decode_runtime_value (v, block_ptr, stack_ptr, link_ptr,
     text_ptr, ref_ptr, code);
```

where:

1. v (Input)
   is an encoded value from a runtime_symbol node, e.g., runtime_symbol.size.

2. block_ptr (Input)
   points to the runtime_block node that corresponds to the block that contains the declaration of the identifier whose runtime_symbol node contains the encoded value. Normally, the value of block_ptr is obtained from a call to the stu_$find_runtime_symbol entry point described above.

3. stack_ptr (Input)
   is a pointer to the active stack frame associated with the procedure or begin block that corresponds to the specified runtime_block node. If the specified block node is quick, stack_ptr should point to the stack frame in which the quick block is placing its automatic storage. If the specified block is not active and does not have a current stack frame, stack_ptr can be null.

4. link_ptr (Input)
   is a pointer to the linkage section of the specified block. If link_ptr is null, the stu_$decode_runtime_value entry point attempts to obtain the linkage pointer, if it is needed, from the linkage offset table (LOT); decoding fails if a pointer to the linkage section is needed and text_ptr, block_ptr, and link_ptr are all null or if the segment has never been executed.

5. text_ptr (Input)
   is a pointer to the base of the object segment that contains the specified block. If text_ptr is null, the stu_$decode_runtime_value entry point attempts to obtain the text pointer, if it is needed, from the active stack frame or the block_ptr; decoding fails if a pointer to the object segment is needed and stack_ptr, block_ptr, and text_ptr are all null.

6. ref_ptr (Input)
   is the value of the pointer to be used as locator qualifier if the variable that corresponds to the runtime_symbol node that contains the encoded value is based. The value of ref_ptr can often be determined by means of the stu_$get_implicit_qualifier entry point described below.

7. code (Output)
   is a status code. It is:
   0 if the encoded value was successfully decoded
   1 if the value could not be decoded

8. value (Output)
   is the decoded value if the value of code is 0.

Entry: stu_$find_block

This entry point, given a pointer to the symbol table header of an object segment, searches the runtime symbol table of the object segment for the runtime_block node that corresponds to a given procedure block in the object program.

Usage

```
declare stu_$find_block entry (ptr, char(*) aligned) returns (ptr);

block_ptr = stu_$find_block (header_ptr, name);
```

where:

1. header_ptr (Input)
   points to a symbol table header.

2. name (Input)
   is the ASCII name of the runtime_block node to be found. The name of a runtime_block node is the same as the first name written on the procedure statement that corresponds to the runtime_block node.

3. block_ptr (Output)
is set to point to the runtime_block node if it is found or is null if the block is not found.

Entry: stu_$find_containing_block

This entry point, given a pointer to the symbol table header of a standard object segment and an offset into the text section, returns a pointer to the runtime_block node corresponding to the smallest procedure or begin block that lexically contains the source line for the instruction pointed to, or null if none could be found.

Usage

```
declare stu_$find_containing_block entry (ptr, fixed bin (18) unsigned)
     returns (ptr);

bp = stu_$find_containing_block (hp, offset);
```

where:

1. hp (Input)
is a pointer to the symbol table header.

2. offset (Input)
is the offset from the base of the segment of an instruction.

3. bp (Output)
is the returned pointer to the runtime_block node, or null.

Entry: stu_$find_header

This entry point, given an ASCII name or a pointer or both to any location in a (possibly bound) object segment, searches the given segment for the symbol table header corresponding to the designated program.

Usage

```
declare stu_$find_header entry (ptr, char(32) aligned, fixed bin(24))
     returns (ptr);

header_ptr = stu_$find_header (seg_ptr, name, bc);
```

where:

1. seg_ptr (Input)
points to any location in the object segment.

2. name (Input)
is the ASCII name of the program whose symbol header is to be found. If seg_ptr is null, name is treated as a reference name and the segment is determined according to the user's search rules. If the designated segment is bound, name specifies the component.

3. bc (Input)
is the bit count of the object segment; if 0, the stu_$find_header entry point determines the bit count itself.

4. header_ptr (Output)
points to the symbol table header if it is found or is null if the header is not found.

Note

Since determining the bit count of a segment is relatively expensive, the user should provide the bit count if he has it available (e.g., as a result of a call to hcs_$initiate_count, described in the MPM Subroutines, Order No. AG93).

Entry: stu_$find_runtime_symbol

This entry point, given a pointer to the runtime_block node that corresponds to a procedure or begin block, searches for the runtime_symbol node that corresponds to a specified identifier name. If the name is not found in the given block, the parent block is searched. This is repeated until the name is found or the root block of the symbol structure is reached, in which case a null pointer is returned.

Usage

```
declare stu_$find_runtime_symbol entry (ptr, char(*) aligned, ptr,
     fixed bin) returns (ptr);

symbol_ptr = stu_$find_symbol (block_ptr, name, found_ptr, steps);
```

where:

1. block_ptr (Input)
points to the runtime_block node in which the search is to begin.

2. name (Input)
is the ASCII name of the runtime_symbol node to be found. A name can be a fully or partially qualified structure name (e.g., "a.b.c"), in which the runtime_symbol node that corresponds to the lowest level item is located.

3. found_ptr (Output)
is set to point to the runtime_block node in which the specified identifier is found.

4. steps (Output)
is set to the number of steps that must be taken along the pl1_stack_frame.display_ptr chain to locate the stack_frame associated with the block designated by found_ptr starting at the stack frame for the block designated by block_ptr. (See "Example" below.) If the given identifier is found in the specified block, the value of steps is 0.

If the search fails, the value of steps indicates the reason for the failure as follows:

-1 block_ptr is null
-2 more than 64 structure levels
-3 name too long
-4 no declaration found
-5 symbol reference is ambiguous

5. symbol_ptr (Output)
is set to point to the runtime_symbol node if it is found or is null if an error occurs.

Entry: stu_$get_block

Given a pointer to the stack frame, gets a pointer to the runtime_block for the entry that created the frame and to the header for the object segment. This entry point is equivalent to stu_$get_runtime_block, except that the location is determined by the information in the stack frame.

Usage

```
dcl stu_$get_block entry (ptr, ptr, ptr);

call stu_$get_block (sp, hp, bp);
```

where:

1. sp (Input)
points to the stack frame in question.

2. hp (Output)
points to the header for the runtime symbol table of the object segment that contains the entry that created the frame. It will be set to null if the object segment has no symbol table, or if the object segment cannot be interpreted.

3. bp (Output)
points to the runtime_block node for the entry that created the frame. It will be set to null if the object segment has no symbol table or could not be interpreted.

Entry: stu_$get_implicit_qualifier

This entry point, given a pointer to the symbol node that corresponds to a PL/I based variable, attempts to return the value of the pointer variable that appeared in the based declaration (e.g., the value of "p" in "dcl a based (p);"). A null pointer is returned if the declaration does not have the proper form or if the value of the pointer cannot be determined.

Usage

```
declare stu_$get_implicit_qualifier entry (ptr, ptr, ptr, ptr, ptr) returns
     (ptr);

ref_ptr = stu_$get_implicit_qualifier (block_ptr, symbol_ptr, stack_ptr,
     link_ptr, text_ptr);
```

where:

1. block_ptr (Input)
   points to the runtime_block node that corresponds to the procedure or begin block in which the based variable is declared.

2. symbol_ptr (Input)
   points to the runtime_symbol node that corresponds to the based variable.

3. stack_ptr (Input)
   is a pointer to the active stack frame associated with the block in which the based variable is declared. If the specified block node is quick, stack_ptr should point to the stack frame in which the quick block is placing its automatic storage. If the specified block is not active and does not have a current stack frame, stack_ptr can be null.

4. link_ptr (Input)
   is a pointer to the linkage section of the specified block. If link_ptr is null, the stu_$get_implicit_qualifier entry point attempts to obtain the linkage pointer, if it is needed, from the active stack frame; the implicit qualifier cannot be determined if a pointer to the linkage section is needed and stack_ptr and link_ptr are both null.

5. text_ptr (Input)
   is a pointer to the base of the object segment that contains the specified block. If text_ptr is null, the stu_$get_implicit_qualifier entry point attempts to obtain the text pointer, if it is needed, from the active stack frame; the implicit qualifier cannot be determined if a pointer to the object section is needed and stack_ptr and text_ptr are both null.

6. ref_ptr (Output)
   is set to the value of the implicit qualifier or is null if the value cannot be determined.

Notes

A null pointer is returned for any one of a number of reasons. Some of these are:

1. The based variable was declared without an implicit qualifier, e.g.,

```
dcl a based;
```

2. Determining the implicit qualifier involves evaluating an expression, for example, the based variable was declared as:

```
dcl a based(p(i));
```

3. The based variable was declared with an implicit qualifier, but it is not possible to obtain the address of the qualifier (e.g., it is an authentic pointer, and stack_ptr is null).

Entry: stu_$get_line

This entry point, given a pointer to the symbol header of a standard object segment and an offset in the text section of the object segment, returns information that allows the source line that generated the specified location to be accessed. This entry point can be used with programs that have only a partial runtime symbol table.

Usage

```
declare stu_$get_line entry (ptr, fixed bin(18), fixed bin, fixed bin(18),
     fixed bin(18), fixed bin, fixed bin);

call stu_$get_line (head_ptr, offset, n_stms, line_no, line_offset,
     line_length, file);
```

where:

1. head_ptr (Input)
   is a pointer to the symbol section header of a standard object segment.

2. offset (Input)
   is the offset of an instruction in the text section.

3. n_stms (Input)
   indicates the number of source statements about which information is desired; the string specified by file, line_offset, and line_length is the source for n_stms statements, starting with the statement that contains the given instruction.

4. line_no (Output)
   is set to the line number, in the file in which it is contained, of the statement that contains the specified instruction or is -1 if the given offset does not correspond to a statement in the object program.

5. line_offset (Output)
   is set to the number of characters that precede the first character of the source for the specified statement.

6. line_length (Output)
   is set to the number of characters occupied by the n_stms statements that start with the statement that contains the specified location; the source for these statements is assumed to be entirely contained within a single source file. Let S be the contents of the source file that contains the specified statements considered as a single string; then the source string for the n_stms statements is substr(S,line_offset+1,line_length).

7. file (Output)
   is the number of the source file in which the source for the desired statements is contained (see "Source Map" in Section 1 of the MPM Subsystem Writers' Guide, Order No. AK92).

Entry: stu_$get_line_no

This entry point, given a pointer to a runtime_block node and an offset in the text segment that corresponds to the block, determines the line number, starting location, and number of words in the source statement that contains the specified location.

Usage

```
declare stu_$get_line_no entry (ptr, fixed bin(18), fixed bin(18),
     fixed bin(18)) returns (fixed bin(18));

line_no = stu_$get_line_no (block_ptr, offset, start, num);
```

where:

1. block_ptr (Input)
   points to the runtime_block node that corresponds to the block in which the instruction offset exists.

2. offset (Input)
   is the offset of an instruction in the text segment.

3. start (Output)
   is set to the offset in the text segment of the first instruction generated for the source line that contains the specified instruction or is -1 if the line is not found.

4. num (Output)
   is set to the number of words generated for the specified source line.

5. line_no (Output)
   is set to the line number, in the main source file, of the statement that contains the specified instruction or is -1 if the specified offset does not correspond to a statement in the program.

Notes

All line numbers refer to the main source file and not to files accessed by means of the %include statement.

No distinction is made between several statements that occur on the same source line. The start argument is the starting location of the code generated for the first statement on the line and num is the total length of all the statements on the line.

Entry: stu_$get_location

This entry point, given a pointer to a runtime_block node and the line number of a source statement in the block, returns the location in the text segment of the first instruction generated by the specified source line.

Usage

```
declare stu_$get_location entry (ptr, fixed bin(18)) returns
     (fixed bin(18));

offset = stu_$get_location (block_ptr, line_no);
```

where:

1. block_ptr (Input)
   points to the runtime_block node.

2. line_no (Input)
   specifies the source line number, which must be in the main source file.

3. offset (Output)
   is set to the offset in the text segment of the first instruction generated for the given line or is -1 if no instructions are generated for the given line.

Entry: stu_$get_map_index

This entry point, given a pointer to the symbol header of a standard object segment and an offset into the text section, returns the index of the statement map entry for the source line that generated the instruction at the offset and a pointer to the map entry. This entry can be used with object segments that have only a partial runtime symbol table.

Usage

```
declare stu_$get_map_index entry (ptr, fixed bin (18) unsigned, fixed bin,
     ptr);

call stu_$get_map_index (header, offset, map_index, map_entry_ptr);
```

where:

1. header (Input)
   is a pointer to the symbol header for the object segment.

2. offset (Input)
   is the offset of an instruction, relative to the base of the segment.

3. map_index (Output)
   is the index in the statement map array of the statement map entry for the line corresponding to the instruction, or -1 if no such map entry could be found.

4. map_entry_ptr (Output)
   is a pointer to the map entry identified by map_index, or null if no such entry could be found.

Even though the map entry index and map entry pointer can be computed from each other, both are supplied to the user for convenience.

Entry: stu_$get_runtime_address

This entry point, given a pointer to a runtime_symbol node and information about the current environment of the block in which the symbol that corresponds to the runtime_symbol node is declared, determines the address of the specified variable.

Usage

```
declare stu_$get_runtime_address entry (ptr, ptr, ptr, ptr, ptr, ptr, ptr)
     returns (ptr);

add_ptr = stu_$get_runtime_address (block_ptr, symbol_ptr, stack_ptr,
     link_ptr, text_ptr, ref_ptr, subs_ptr);
```

where:

1. block_ptr (Input)
   points to the runtime_block node that corresponds to the block in which the symbol, whose address is to be determined, is declared.

2. symbol_ptr (Input)
   points to the runtime_symbol node that corresponds to the symbol whose address is to be determined.

3. stack_ptr (Input)
   is a pointer to the active stack frame associated with the procedure or begin block that corresponds to the specified runtime_block node. If the specified block is quick, stack_ptr should point to the stack frame in which the quick block is placing its automatic storage. If the specified block is not active and does not have a current stack frame, stack_ptr can be null.

4. link_ptr (Input)
   is a pointer to the linkage section of the specified block. If link_ptr is null, the stu_$get_runtime_address entry point attempts to obtain the linkage pointer, if it is needed, from the LOT; the address of the specified symbol cannot be determined if a pointer to the linkage section is needed and text_ptr, block_ptr, and link_ptr are all null or the segment has never been executed.

5. text_ptr (Input)
   is a pointer to the base of the object segment that contains the specified block. If text_ptr is null, the stu_$get_runtime_address entry point attempts to obtain the text pointer, if it is needed, from the active stack frame or the block_ptr; the address of the specified symbol cannot be determined if a pointer to the object segment is needed and stack_ptr, block_ptr, and text_ptr are all null.

6. ref_ptr (Input)
   is the value of the reference pointer to be used if the runtime_symbol node corresponds to a based variable. If ref_ptr is null, the stu_$get_runtime_address entry point calls the stu_$get_implicit_qualifier entry point (described above) to determine the value of the pointer that was used in the declaration of the based variable.

7. subs_ptr (Input)
   points to a vector of single-precision fixed-point binary subscripts. The number of subscripts is assumed to match the number required by the declaration. This argument can be null if the runtime_symbol node does not correspond to an array.

8. add_ptr (Output)
   is set to the full bit address (with full bit offset) of the variable that corresponds to the symbol node or is null if the address cannot be determined.

Entry: stu_$get_runtime_block

This entry point, given a pointer to an active stack frame and a location within the object segment that created the frame, returns pointers to the symbol table header of the object segment and the runtime_block node that corresponds to the procedure or begin block associated with the stack frame. Null pointers are returned if the stack frame does not belong to a PL/I, FORTRAN, or COBOL program or if the object segment does not have a runtime symbol table.

Usage

```
declare stu_$get_runtime_block entry (ptr, ptr, ptr, fixed bin(18));

call stu_$get_runtime_block (stack_ptr, header_ptr, block_ptr, loc);
```

where:

1. stack_ptr (Input)
   points to an active stack frame.

2. header_ptr (Output)
   is set to point to the symbol table header or is null if the object segment does not have a runtime symbol table.

3. block_ptr (Output)
   is set to point to the runtime_block node that corresponds to the procedure or begin block associated with the stack frame or is null if the object segment does not have a runtime symbol table.

4. loc (Input)
   is an address within the object segment (e.g., where execution was interrupted); a negative value for loc means no location information is specified. The additional information provided by loc enables the stu_$get_runtime_block entry point to return the runtime_block node that corresponds to the quick PL/I procedure or begin block that is sharing the designated stack frame and was active at the time execution was interrupted.

Entry: stu_$get_runtime_line_no

This entry point, given a pointer to the symbol header of a standard object segment and an offset in the text section of the object segment, returns information about the line that caused the specified instruction to be generated. Since the symbol header is used to locate the statement map, this entry point can be used with object segments that have only a partial runtime symbol table.

Usage

```
declare stu_$get_runtime_line_no entry (ptr, fixed bin(18), fixed bin(18),
     fixed bin(18), fixed bin(18));

call stu_$get_runtime_line_no (head_ptr, offset, start, num, line_no);
```

where:

1. head_ptr (Input)
   is a pointer to the symbol section header of a standard object segment.

2. offset (Input)
   is the offset of an instruction in the text section.

3. start (Output)
is set to the offset in the text segment of the first instruction generated for the source line that contains the specified instruction or is -1 if the line is not found.

4. num (Output)
is set to the number of words in the object code generated for the specified source line.

5. line_no (Output)
is set to the line number, in the main source file, of the statement that contains the specified instruction or is -1 if the specified offset does not correspond to a statement in the program.

Notes

All line numbers refer to the main source file and not to files accessed by means of the %include statement.

No distinction is made between several statements that occur on the same source line. The start argument is the starting location of the code generated for the first statement on the line and num is the total length of all the statements on the line.

Entry: stu_$get_runtime_location

This entry point, given a pointer to the symbol header of a standard object segment and a line number in the main source file, returns the starting location in the text section of the object code generated for the line. This entry point can be used with object segments that have only a partial runtime symbol table.

Usage

```
declare stu_$get_runtime_location entry (ptr, fixed bin) returns
     (fixed bin(18));

offset = stu_$get_runtime_location (head_ptr, line_no);
```

where:

1. head_ptr (Input)
is a pointer to the symbol section header of a standard object segment.

2. line_no (Input)
is the line number of a statement in the main source file.

3. offset (Output)
is set to the location in the text segment where the object code generated for the specified line begins or is -1 if no code is generated for the given line.

Entry: stu_$get_statement_map

This entry point, given a pointer to the symbol header of a standard object segment, returns information about the statement map of the object segment. This entry point can be used with object segments that have only a partial runtime symbol table.

Usage

```
declare stu_$get_statement_map entry (ptr, ptr, ptr, fixed bin);

call stu_$get_statement_map (head_ptr, first_ptr, last_ptr, map_size);
```

where:

1. head_ptr (Input)
   is a pointer to the symbol section header of a standard object segment.

2. first_ptr (Output)
   is set to point to the first entry in the statement map of the object segment or is null if the object segment does not have a statement map.

3. last_ptr (Output)
   is set to point to the location following the last entry in the statement map of the object segment or is null if the object segment does not have a statement map.

4. map_size (Output)
   is set to the number of words in an entry in the statement map.

Entry: stu_$offset_to_pointer

This entry point attempts to convert an offset variable to a pointer value using the area, if any, on which the offset was declared.

Usage

```
declare stu_$offset_to_pointer entry (ptr, ptr, ptr, ptr, ptr, ptr) returns
     (ptr);

off_ptr = stu_$offset_to_pointer (block_ptr, symbol_ptr, data_ptr,
     stack_ptr, link_ptr, text_ptr);
```

where:

1. block_ptr (Input)
   points to the runtime_block node that corresponds to the procedure or begin block in which the offset variable is declared.

2. symbol_ptr (Input)
   points to the runtime_symbol node that corresponds to the offset variable.

3. data_ptr (Input)
   points to the offset value to be converted to a pointer.

4. stack_ptr (Input)
   is a pointer to the active stack frame associated with the block in which the offset variable is declared. If the specified block node is quick, stack_ptr should point to the stack frame in which the quick block is placing its automatic storage. If the specified block is not active and does not have a current stack frame, stack_ptr can be null.

5. link_ptr (Input)
   is a pointer to the linkage section of the specified block. If link_ptr is null, the stu_$offset_to_pointer entry point attempts to obtain the linkage pointer, if it is needed, from the stack frame; conversion fails if a pointer to the linkage section is needed and stack_ptr and link_ptr are both null.

6. text_ptr (Input)
   is a pointer to the base of the object segment that contains the specified block. If text_ptr is null, the stu_$offset_to_pointer entry point attempts to obtain the text pointer, if it is needed, from the active stack frame; conversion fails if a pointer to the text section is needed and stack_ptr and link_ptr are both null.

7. off_ptr (Output)
   is set to the pointer value that corresponds to the offset value; it is null if the conversion fails or if the offset value is itself null.

Entry: stu_$pointer_to_offset

This entry point attempts to convert a pointer value to an offset variable using the area, if any, on which the offset was declared.

Usage

```
declare stu_$pointer_to_offset entry (ptr, ptr, ptr, ptr, ptr, ptr) returns
     (offset);

off_val = stu_$pointer_to_offset (block_ptr, symbol_ptr, data_ptr,
     stack_ptr, link_ptr, text_ptr);
```

where:

1. block_ptr (Input)
   is as above.

2. symbol_ptr (Input)
   is as above.

3. data_ptr (Input)
   points at the pointer value to be converted to an offset. This pointer value must be an unpacked pointer value.

4. stack_ptr (Input)
   is as above.

5. link_ptr (Input)
   is as above.

6. text_ptr (Input)
   is as above.

7. off_val (Output)
   is set to the offset value that corresponds to the pointer value; it is null if the conversion fails or if the pointer value is itself null.

Entry: stu_$remote_format

This entry point decodes a remote format specification.

Usage

```
declare stu_$remote_format entry (fixed bin(35), ptr, ptr, label) returns
     (fixed bin);

code = stu_$remote_format (value, stack_ptr, ref_ptr, format);
```

where:

1. value (Input)
   is the remote format value to be decoded.

2. stack_ptr (Input)
   is a pointer to the active stack frame of the block that contains the format being decoded.

3. ref_ptr (Input)
   is the pointer value to be used if the format value being decoded requires pointer qualification.

4. format (Output)
   is set to the format value if decoding is successful.

5. code (Output)
   is a status code. It is:
   0 if decoding is successful
   1 if decoding is not successful

Example

The use of some of the entry points documented above is illustrated by the following sample program, which is called with:

stack_ptr
a pointer to the stack frame of a PL/I block

symbol
an ASCII string giving the name of a user symbol in the PL/I program

subs_ptr
a pointer to an array of binary integers that give subscript values

The procedure determines the address and size of the specified symbol. If any errors occur, the returned address is null.

```
example:  proc (stack_ptr, symbol, subs_ptr, size) returns (ptr);


declare   stack_ptr ptr,
          symbol    char(*) aligned,
          subs_ptr  ptr,
          size      fixed bin(35);

declare   (header_ptr, block_ptr, symbol_ptr, ref_ptr, sp, blk_ptr,
           stack_ptr, add_ptr) ptr,
          (i, steps) fixed bin,
          code fixed bin(35),
          stu_$get_runtime_block entry(ptr, ptr, ptr, fixed bin(18)),
          stu_$find_runtime_symbol entry(ptr,char(*) aligned,ptr,fixed bin)
           returns(ptr),
          stu_$get_runtime_address entry(ptr,ptr,ptr,ptr,ptr,ptr,ptr)
           returns(ptr),
          stu_$decode_runtime_value entry(fixed bin(35),ptr,ptr,ptr,ptr,ptr,
           fixed bin) returns(fixed bin(35));

%include pl1_stack_frame;
%include runtime_symbol;

          /* determine header and block pointers */

          call stu_$get_runtime_block(stack_ptr,header_ptr,block_ptr,-1);

          if block_ptr = null then return(null);

          /* search for specified symbol */

          symbol_ptr = stu_$find_runtime_symbol(block_ptr,symbol,blk_ptr,steps);

          if symbol_ptr = null then return(null);

          /* determine stack frame of block owning symbol */

          sp = stack_ptr;
          do i = 1 to steps;
               sp = sp -> pl1_stack_frame.display_ptr;
               end;
```

```
/* determine address of symbol */

ref_ptr = null;
add_ptr = stu_$get_runtime_address(blk_ptr,symbol_ptr,sp,null,null,
 ref_ptr,subs_ptr);

if add_ptr = null then return(null);

/* determine size */

size = symbol_ptr -> runtime_symbol.size;

if size < 0
then do;
     size = stu_$decode_runtime_value(size,blk_ptr,sp,null,null,
      ref_ptr,code);
     if code ^= 0 then return(null);
     end;

return(add_ptr);
end example;
```

Name: sweep_disk_

The sweep_disk_ subroutine walks through the subtree below a specified node of the directory hierarchy, calling a user-supplied subroutine once for every entry in every directory in the subtree.

Usage

```
declare sweep_disk_ entry (char(168) aligned, entry);

call sweep_disk_ (base_path, subroutine);
```

where:

1. base_path (Input)
   is the pathname of the directory that is the base node of the subtree to be scanned.

2. subroutine (Input)
   is an entry point called for each branch or link in the subtree (see "User-Supplied Subroutines" below).

User-Supplied Subroutines

The subroutine is assumed to have the following declaration and call:

```
declare subroutine entry (char(168) aligned, char(32) aligned, fixed bin,
     char(32) aligned, ptr, ptr);

call subroutine (path, dir_name, level, entryname, b_ptr, n_ptr);
```

where:

1. path (Input)
   is the pathname of the directory immediately superior to the directory that contains the current entry.

2. dir_name (Input)
   is the name of the directory that contains the current entry.

3. level (Input)
   is the number of levels deep from the base_path directory of the subtree.

4. entryname (Input)
   is the primary name on the current entry.

5. b_ptr (Input)
   is a pointer to the branch structure returned by hcs_$star_list for the current entry.

6. n_ptr (Input)
   is a pointer to the names area for the immediately superior directory of the current entry returned by hcs_$star_list.

Entry: sweep_disk_$dir_list

This entry point operates in the same way as sweep_disk_ but is much less expensive to use and does not return date_time_contents_modified, date_time_used, or bit_count.

Usage

```
declare sweep_disk_$dir_list entry (char(168) aligned, entry);

call sweep_disk_$dir_list (base_path, subroutine);
```

The user-supplied subroutine is called in the same way as sweep_disk_, but b_ptr points instead to the branch structure returned by hcs_$star_dir_list. See the hcs_$star_ subroutine in the MPM Subsystem Writers' Guide (Order No. AK92).

Notes

If the base_path argument to the sweep_disk_ subroutine is the root (">"), the directory >process_dir_dir is omitted from the tree walk.

The sweep_disk_ subroutine attempts to force access to the directories in the subtree by adding an ACL term of the form "sma Person.Project.tag" to each directory ACL, and deleting that ACL term when finished processing the directory. If the user does not have sufficient access to add this ACL term for a given directory, the subroutine will process those parts of the subtree under it where the user already has sufficient access to list the directories.

Entry: sweep_disk_$loud

This entry point is used for debugging subsystems that use the sweep_disk_ subroutine. It sets an internal static flag in sweep_disk_ that causes sweep_disk_ to call com_err_ and report any errors encountered in listing directories or setting ACLs. Since sweep_disk_$loud takes no arguments, and should only be used for debugging, it can readily be invoked as a command ("sweep_disk_$loud") to cause sweep_disk_ to exhibit this debugging behavior for the rest of the process. There is no corresponding entry point to turn the switch off. Because this is a static switch, and affects all callers of sweep_disk_, it should not be turned on, except to debug, when it is important to understand the exact nature of any errors encountered. Normally, sweep_disk_ ignores errors and continues as best it can.

Usage

```
declare sweep_disk_$loud entry ();

call sweep_disk_$loud ();
```

Name: teco_get_macro_

The teco_get_macro_ subroutine is called by teco to search for an external macro.

By default the following directories are searched:

1. working directory
2. home directory
3. system_library_tools

Usage

```
declare teco_get_macro_ entry (char(*) aligned, ptr, fixed bin,
     fixed bin(35));

call teco_get_macro_ (mname, mptr, mlen, code);
```

where:

1. name (Input)
   is the name of the macro to be found.

2. mptr (Output)
   is a pointer to the macro.

3. mlen (Output)
   is the length of the macro.

4. code (Output)
   is a standard Multics status code.

Name: translator_info_

The translator_info_ subroutine contains utility routines needed by the various system translators. They are centralized here to avoid repetitions in each of the individual translators.

Entry: translator_info_$get_source_info

This entry point returns the information about a specified source segment that is needed for the standard object segment: storage-system location, date-time last modified, unique ID.

Usage

```
declare translator_info_$get_source_info entry (ptr, char(*), char(*),
     fixed bin(71), bit(36) aligned, fixed bin(35));

call translator_info_$get_source_info entry (source_ptr, dir_name,
     entryname, date_time_mod, unique_id, code);
```

where:

1. source_ptr (Input)
   is a pointer to the source segment about which information is desired.

2. dir_name (Output)
   is a pathname of the directory in which the source segment is located.

3. entryname (Output)
   fin
   is the primary name of the source segment.

4. date_time_mod (Output)
   is the date-time modified of the source segment as obtained from the storage system.

5. unique_id (Output)
   is the unique ID of the source segment as obtained from the storage system.

6. code (Output)
   is a storage system status code.

Status Code

A status code of zero indicates that all information has been returned normally.

A nonzero status code returned by this entry is a storage-system status code. Because the interface to this procedure is a pointer to the source segment, the presence of a nonzero status code probably indicates that the storage-system entry for the source segment has been altered since the segment was initiated, i.e., the segment has been deleted, or this process no longer has access to the segment.

Note

The entryname returned by this procedure is the primary name on the source segment. It is not necessarily the same name as that by which the translator initiated it.

Name: translator_temp_

This subroutine provides an inexpensive temporary storage management facility for translators in the Tools Library. It uses the get_temp_segment_ subroutine to obtain temporary segments in the user's process directory. Each segment begins with a header that defines the amount of free space remaining in the segment. An entry is provided for allocating space in temporary segments, but once allocated, the space can never be freed.

Entry: translator_temp_$get_segment

This entry point should be called by each program activation to obtain the first temporary segment to be used during that activation. Before the activation ends, the program should release the temporary segment for use by other programs. (See the translator_temp_$release_all_segments entry point below.)

Usage

```
declare translator_temp_$get_segment entry (char(*) aligned, ptr,
     fixed bin (35));

call translator_temp_$get_segment (program_id, Psegment, code);
```

where:

1. program_id (Input)
   is the name of the program that is using the temporary segment. This name is printed out by the list_temporary_ segments command.

2. Psegment (Output)
   is a pointer to the temporary segment that was created.

3. code (Output)
   is a status code.

Entry: translator_temp_$get_next_segment

This entry point may be called by a program activation to obtain additional temporary segments.

Usage

```
declare translator_temp_$get_next_segment entry (ptr, ptr, fixed bin(35));

call translator_temp_$get_next_segment (Psegment, Pnew_segment, code);
```

where:

1. Psegment (Input)
   is a pointer to one of the temporary segments that the program has previously obtained during its current activation.

2. Pnew_segment (Output)
   is a pointer to the new temporary segment.

3. code (Output)
   is a status code.

Entry: translator_temp_$allocate

This entry point can be called to allocate a block of space within a temporary segment.

Usage

```
declare translator_temp_$allocate entry (ptr, fixed bin) returns (ptr);

Pspace = translator_temp_$allocate (Psegment, Nwords);
```

where:

1. Psegment (Input/Output)
   is a pointer to the temporary segment in which space is to be allocated. Psegment must be passed by reference rather than by value, because the allocation routine may change its value if there is insufficient space in the current temporary segment to perform the allocation.

2. Nwords (Input)
   is the number of words to be allocated. It must not be greater than sys_info_$max_seg_size-32.

3. Pspace (Output)
   is a pointer to the space that was allocated. If Nwords > sys_info$max_seg_size-32, then Pspace will be a null pointer on return.

Notes

As an alternative to calling translator_temp_$allocate, a procedure that must perform many allocations can include translator_temp_alloc.incl.pl1. This include segment contains the program definition of an "allocate" function that can be called like the $allocate entry point above. The allocate function is a quick internal PL1 procedure that adds about 60 words to the external procedure and that shares its stack frame. Use of the allocate internal procedure can significantly reduce the cost of performing many allocations.

Entry: translator_temp_$release_all_segments

This entry point releases all of the temporary segments used by a program activation for use by other programs. It truncates these segments to conserve space in the process directory. It should be called by each program activation that uses temporary segments before the activation is terminated.

Usage

```
declare translator_temp_$release_all_segments entry (ptr, fixed bin(35));

call translator_temp_$release_all_segments (Psegment, code);
```

where:

1. Psegment (Input)
   is a pointer to any one of the temporary segments.

2. code (Output)
   is a status code.

Entry: translator_temp_$release_segment

This entry point releases one of the temporary segments used by a program activation. It truncates the temporary segment to conserve space in the process directory.

Usage

```
declare translator_temp_$release_segment entry (ptr, fixed bin(35));

call translator_temp_$release_segment (Psegment, code);
```

where:

1. Psegment (Input)
   is a pointer to the temporary segment to be released.

2. code (Output)
   is a status code.

SECTION 4

whotab DATA BASE

The >sc1>whotab segment is the public information base for the system. All logged-in users, except those with the nolist attribute, have an entry in this table. These entries are listed by the who command. In addition, various system parameters of interest to all users are recorded in whotab. Many of these parameters are returned by the system_info_ subroutine (described in the MPM Subsystem Writers' Guide, Order No. AK92) and the system active function (described in the MPM Commands, Order No. AG92). Only the initializer process can modify the segment.

The structure of the whotab data base is given below.

```
dcl 1 whotab                  based aligned
    2 mxusers                 fixed bin,
    2 n_users                 fixed bin,
    2 mxunits                 fixed bin,
    2 n_units                 fixed bin,
    2 timeup                  fixed bin (71),
    2 obsolete_sysid          char (8)
    2 nextsd                  fixed bin (71),
    2 until                   fixed bin (71),
    2 lastsd                  fixed bin (71),
    2 erfno                   char (8),
    2 obsolete_why            char (32),
    2 installation_id         char (32),
    2 obsolete_message        char (32),
    2 abs_event               fixed bin (71),
    2 abs_procid              bit (36),
    2 max_abs_users           fixed bin,
    2 abs_users               fixed bin,
    2 n_daemons               fixed bin
    2 request_channel         fixed bin (71),
    2 request_process_id      bit (36),
    2 shift                   fixed bin,
    2 next_shift_change_time  fixed bin (71),
    2 last_shift_change_time  fixed bin (71),
    2 fg_abs_users            fixed bin (17) unal,
    2 n_rate_structures       fixed bin (9) unsigned, unaligned,
    2 pad1                    bit (9) unaligned,
    2 pad (3)                 fixed bin,
    2 version                 fixed bin,
    2 header_size             fixed bin,
    2 entry_size              fixed bin,
    2 laste_adjust            fixed bin,
    2 laste                   fixed bin,
    2 freep                   fixed bin,
    2 header_extension_mbzl   fixed bin,
    2 n_abs (4)               fixed fin,
    2 abs_qres (4)            fixed fin,
    2 abs_cpu_limit (4)       fixed bin (35),
```

```
        2 abs_control,
          3 mbz                    bit (1) unaligned,
          3 abs_maxu_auto          bit (1) unaligned,
          3 abs_maxq_auto          bit (1) unaligned,
          3 abs_qres_auto          bit (1) unaligned,
          3 abs_cpu_limit_auto     bit (1) unaligned,
          3 queue_dropped (-1:4)   bit (1) unaligned,
          3 abs_up                 bit (1) unaligned,
          3 abs_stopped            bit (1) unaligned,
          3 control_pad            bit (24) unaligned,
        2 installation_request_channel
                                   fixed bin (71),
        2 installation_request_pid
                                   bit (36),
        2 sysid                    char (32),
        2 header_extension_pad1 (7)
                                   fixed bin,
        2 header_extension_mbz2    fixed bin,
        2 message                  char (124),
        2 header_extension_mbz3    fixed bin,
        2 why                      char (124),
        2 e (1000),
          3 active                 fixed bin,
          3 person                 char (28),
          3 project                char (28),
          3 anon                   fixed bin,
          3 padding                fixed bin (71)
          3 timeon                 fixed bin (71),
          3 units                  fixed bin,
          3 stby                   fixed bin,
          3 idcode                 char (4),
          3 chain                  fixed bin,
          3 proc_type              fixed bin,
          3 group                  char (8),
          3 fg_abs                 bit (1) unaligned,
          3 disconnected           bit (1) unaligned,
          3 suspended              bit (1) unaligned,
          3 pad2                   bit (33) unaligned,
          3 cant_bump_until        fixed bin (71),
          3 process_authorization  bit (72);
```

Header variables:

mxusers
: is the maximum number of users allowed on the system.

n_users
: is the current number of users.

mxunits
: is the maximum number of load units allowed.

n_units
: is the current load.

timeup
: is the time the system was started.

obsolete_sysid
: is obsolete; use the field sysid instead.

nextsd
: is the time the system will be shutdown, if nonzero.

until
: is the projected time of the next system start-up.

lastsd
is the time of last crash or shutdown.

erfno
is the error number of the last crash, if known.

obsolete_why
is obsolete; use why instead.

installation_id
is the name of the installation.

obsolete_message
is obsolete; use message instead.

abs_event
is the event channel for signalling absentee requests.

abs_procid
is the process identifier of the absentee user manager.

max_abs_users
is the current maximum number of absentee users.

abs_users
is the current number of absentee users.

n_daemons
is the number of daemons logged in via the message coordinator.

request_channel
is the event channel over which requests to the answering service should be sent.

request_processid
is the identifier of the process to which answering service requests should be sent.

shift
is the number of the current shift.

next_shift_change_time
is the time the current shift is scheduled to end.

last_shift_change_time
is the time the current shift started.

fg_abs_users
is the current number of foreground absentee users.

n_rate_structures
is the number of rate structures defined at the site.

pad1
is unused.

pad
is unused.

version
is the structure version.

header_size
is the length of the header (in words).

entry_size
is the length of the entry (in words).

laste_adjust
is used only by answering service programs. It gives the count of 32-word blocks in the header from header_extension_mbz1.

laste
is the index of the last entry in use.

freep
is the index of the first free entry chained through "chain."

header_extension_mbz1
offset 100o.

n_abs (4)
gives the number of processes from each background queue.

abs_qres (4)
gives the number of absentee positions reserved for each queue.

abs_cpu_limit (4)
gives the current absentee cpu limits.

abs_control
see absentee_user_table.

mnbz
must not be zero.

abs_maxu_auto
is one if automatic.

abs_maxq_auto
is one if automatic.

abs_qres_auto
is one if automatic.

abs_cpu_limit_auto
is one if automatic.

abs_cpu_limit_auto
is one if automatic.

queue_dropped (-1:4)
is one if queue is dropped. Queue -1 is the foreground; 0-4 are respective background queue numbers.

abs_up
is one if the absentee facility is running.

abs_stopped
is one if the absentee facility is stopped.

control_pad

installation_request_channel
is the IPC channel for the install command.

installation_request_pid
is the installation process identifier.

sysid
is the current system name.

header_extension_pad1
is not used at present.

header_extension_mbz2
offset 140o.

message
is the message for all users.

header_extension_mbz3
offset 200o.

why
is the reason for the next shutdown.

User entry variables, with whotab.e(i):

active
is nonzero if this entry describes a logged-in user.

person
is the person name (Person_id).

project
is the project identifier (Project_id).

anon
indicates whether the user is an anonymous user:
1 yes
0 no

padding
is unused.

timeon
is the time of login.

units
is the number of load units for the user.

stby
indicates whether the user has secondary status:
1 yes
0 no

idcode
is the terminal identifier.

chain
is a chain for the free list.

proc_type
indicates the process type:
1 interactive
2 absentee
3 daemon

group
is the user's load-control group identifier.

fg_abs
is "1"b if this entry describes a foreground absentee user.

disconnected
is "1"b if the process is disconnected.

suspended
is "1"b if the process is suspended.

pad2
is unused.

cant_bump_until
is the time at which the user will (or did) become subject to preemption.

process_authorization
Is the AIM authorization of the user's process.

MULTICS SYSTEM PROGRAMMING TOOLS ADDENDUM A

SUBJECT

Additions and Changes to the Manual

SPECIAL INSTRUCTIONS

This is the first addendum to AZ03, Revision 2, dated January 1982.

Insert the attached pages into the manual according to the collating instructions on the back of this cover.

Throughout the manual, change bars in the margin indicate technical additions and changes; deletions are marked by an asterisk in the margin. Commands that are entirely new do not contain change bars (see Preface for a list of the new commands). These changes will be incorporated into the next revision of this manual.

Note:

Insert this cover behind the manual cover to indicate the updating of the document with Addendum A.

SOFTWARE SUPPORTED

Multics Software Release 10.0

ORDER NUMBER

AZ03-02A

July 1982

34910
7.5C682
Printed in U.S.A.

Honeywell

COLLATING INSTRUCTIONS

To update the manual, remove old pages and insert new pages as follows:

Remove	Insert
title page, preface	title page, preface
iii through viii	iii through vii
1-1 through 1-4	1-1 through 1-4.1, blank
2-3, 2-4	2-3, blank 2-4, 2-4.1
2-25 through 2-30	2-25, 2-25.1 2-25.2, 2-26 2-27, 2-28 2-29, 2-30
	2-44.1, 2-44.2 2-44.3, blank
2-45, 2-46	2-45, 2-45.1 2-45.2, 2-46
2-55 through 2-58	2-55, 2-56 2-57, blank 2-57.1, 2-58
2-61 through 2-64	2-61, 2-62 2-63, blank 2-63.1, 2-64
	2-86.1, blank
2-99 through 2-102	2-99, 2-100 2-101, blank 2-101.1, 2-101.2, 2-101.3, 2-101.4 2-101.5, 2-101.6 2-101.7, blank 2-101.8, 2-102
	2-104.1, 2-104.2
2-115, 2-116	2-115, 2-115.1 2-115.2, 2-116
2-171, 2-172	2-171, 2-172

File No.: 1L13, 1U13

7/82

AZ03-02A

3-87, 3-88	3-87, 3-88
3-91 through 3-94	3-91 through 3-94
3-97, 3-98	3-97, 3-98
4-1 through 4-4	4-1 through 4-6
i-1 through i-6	i-1 through i-5, blank

INDEX

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE: MULTICS SYSTEM PROGRAMMING TOOLS
ADDENDUM A

ORDER NO. AZ03-02A

DATED JULY 1982

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

FROM: NAME _____ DATE _____

TITLE _____

COMPANY _____

ADDRESS _____

_____

CUT ALONG LINE

FOLD ALONG LINE

PLEASE FOLD AND TAPE–
NOTE: U. S. Postal Service will not deliver stapled forms

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

FOLD ALONG LINE

Honeywell

Honeywell

Honeywell Information Systems
in the U.S.A.: 200 Smith Street, MS 486, Waltham, Massachusetts 02154
In Canada: 155 Gordon Baker Road, Willowdale, Ontario M2H 3N7
In the U.K.: Great West Road, Brentford, Middlesex TW8 9DH
In Australia: 124 Walker Street, North Sydney, N.S.W. 2060
In Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F.

33889, 7.5C282, Printed in U.S.A.

AZ03-02