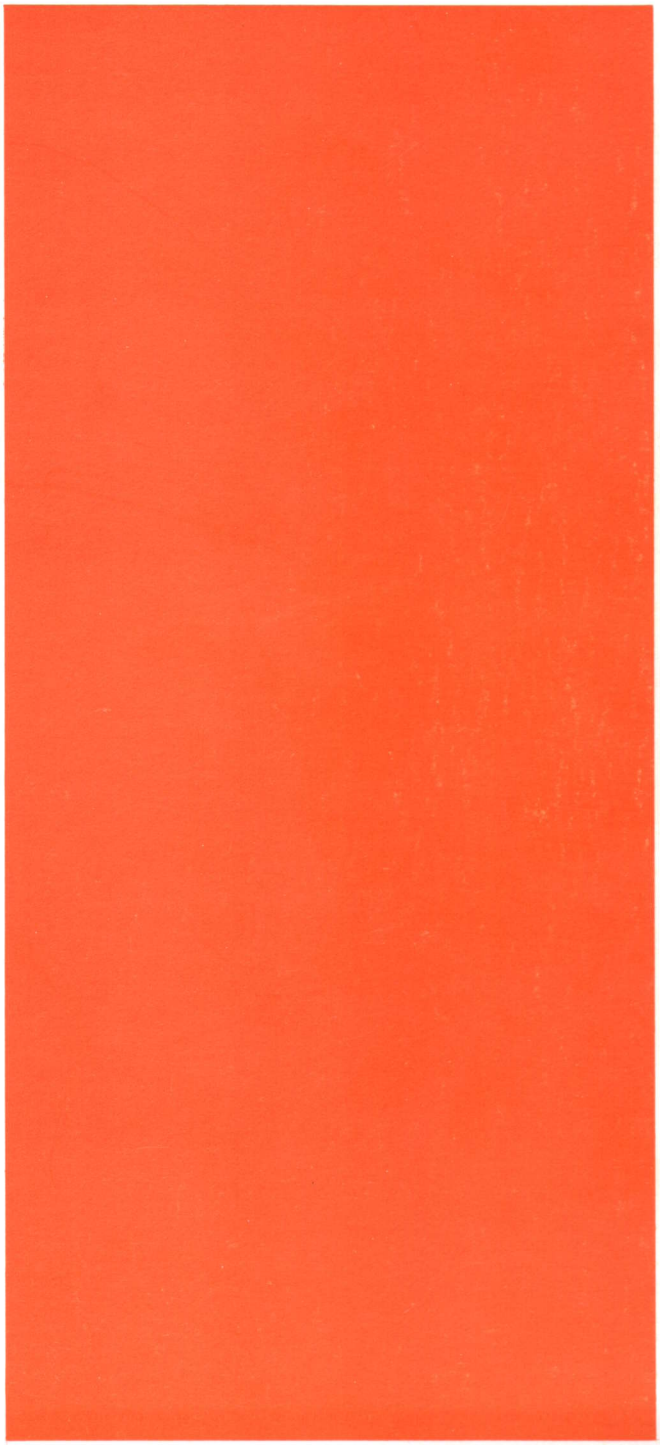
Honeywell

SYSTEM TOOLS
PROGRAM LOGIC MANUAL

SERIES 60 (LEVEL 68)

MULTICS

SOFTWARE



RESTRICTED DISTRIBUTION

Honeywell

SYSTEM TOOLS
PROGRAM LOGIC MANUAL

SERIES 60 (LEVEL 68)

MULTICS

RESTRICTED DISTRIBUTION

SUBJECT:

Descriptions of Internal Interfaces for Use by Multics System Programmers.

SPECIAL INSTRUCTIONS:

This Program Logic Manual (PLM) describes certain internal modules constituting the Multics System. It is intended as a reference for only those who are thoroughly familiar with the implementation details of the Multics operating system; interfaces described herein should not be used by application programmers or subsystem writers; such programmers and writers are concerned with the external interfaces only. The external interfaces are described in the Multics Programmers' Manual, Commands and Active Functions (Order No. AG92), Subroutines (Order No. AG93), and Subsystem Writers' Guide (Order No. AK92).

As Multics evolves, Honeywell will add, delete, and modify module descriptions in subsequent PLM updates. Honeywell does not ensure that the internal functions and internal module interfaces will remain compatible with previous versions.

This PLM is one of a set, which when complete, will supersede the System Programmers' Supplement to the Multics Programmers' Manual (Order No. AK96).

DATE:

February 1975

ORDER NUMBER:

AN51, Rev. 0

PREFACE

Multics Program Logic Manuals (PLMs) are intended for use by Multics system maintenance personnel, development personnel, and others who are thoroughly familiar with Multics internal system operation. They are not intended for application programmers or subsystem writers.

The PLMs contain descriptions of modules that serve as internal interfaces and perform special system functions. These documents do not describe external interfaces, which are used by application and system programmers.

Since internal interfaces are added, deleted, and modified as design improvements are introduced, Honeywell does not ensure that the internal functions and internal module interfaces will remain compatible with previous versions. To help maintain accurate PLM documentation, Honeywell publishes a special status bulletin containing a list of the PLMs currently available and identifying updates to existing PLMs. This status bulletin is distributed automatically to all holders of the System Programmers' Supplement to the Multics Programmers' Manual (Order No. AK96) and to others on request. To get on the mailing list for this status bulletin, write to:

Large Systems Sales Support
Multics Project Office
Honeywell Information Systems Inc.
Post Office Box 6000 (MS A-85)
Phoenix, Arizona 85005

 File No.: 2L13

CONTENTS

CONTENTS

CONTENTS

SECTION I

COMMANDS

This Program Logic Manual (PLM) is not structured in the same manner as most others in this series. The System Tools PLM consists only of a number of command and subroutine descriptions with no design motivation, implementation description, or data structure description except what is needed to describe the use of the command or subroutine as a tool. If design and implementation documentation is desired for a particular command, it should be available in the Command Implementation PLM, Order No. AN67, in this series. The Index PLM, Order No. AN50, may be of use in trying to find out which command or subroutine may be wanted and in which PLM a detailed description can be found.

This section, command descriptions, is arranged alphabetically. For programs that consist of a set of several related commands, the set may be documented within one command description. Also, the set is arranged according to the order in which the commands are used rather than alphabetically.

add_copyright

Name: add_copyright

The add_copyright command adds a copyright notice to a source program. If the program already contains a copyright notice no change is made to the segment. Different notices are used for each different language type suffix.

Usage

add_copyright path

where path is the name of the source program to be modified.

Consult the description of the copyright_notice_ subroutine for details of the operation of this command.

Name: as_who

The as_who command is a privileged version of the who command used by the answering service. It extracts information directly from the answer_table rather than from whotab.

Entry: as_who$as_who_init

This entry must be invoked prior to the other entries if an answer_table other than the standard one (found in >system_control_1) is to be used.

Usage

as_who$as_who_init path

where path is a pathname of a nonstandard answer_table.

Entry: as_who$how_many_users

This entry prints the number of users, the time of system initialization, and the time of the last shutdown (or crash).

Usage

as_who$how_many_users

Entry: as_who$hmu

This entry prints the number of users currently logged in.

Usage

as_who$hmu

Entry: as_who

This entry lists the selected users and prints other information specified in the -long control argument below.

Usage

as_who -control_args- User_ids

where:

1. control_args — are selected from the following:

   -long, -lg — specifies that the long form of output including login time, tty ID, etc., are printed.

   -name, -nm — specifies that the users are sorted by name.

   -project, pj — specifies that the users are sorted by project.

2. User_ids — are of the form Person_id.Project_id.tag.

Note

If an argument is not one of the above, it is assumed to be a User_id in one of the following formats:

Person_id.Project_id	lists all users logged in with the specified name and project.
Person_id	lists all users logged in with the specified name.
.Project_id	lists all users logged in with the specified project.

The default sort is by login time.

If no arguments are specified, name and project for each user are returned. Anonymous users' true login names are shown, preceded by an asterisk (*).

Specification of -long returns time of login, tty ID, device channel, weight, Person_id and Project_id for each user, as well as flags indicating special variables.

A flag of N indicates that the user has the nolist bit and will not be listed on an ordinary who.

A flag of + identifies a user with the nobump attribute.

A flag of > indicates that a user whose grace period has not yet run out is subject to bumping by other members of the project.

A flag of X indicates that a user has been bumped but has not yet logged out.

backup_dump

Name: backup_dump

The backup_dump module can be called as a command or as a subroutine. It sets up the requested control arguments and dumps the appropriate directories and segments.

Usage

backup_dump -control_args-

where control_args represents the current arguments, selected from the list shown in the description of the bk_arg_reader_ subroutine in this document.

Notes

If a pathname is not given (see below), dumping begins with the current working directory. Unless flags have been set by previous calls in the process, the defaults are: -all, -err_offl, -map, -nodebug, -nohold, -1tape, -sweep, and -tape.

Option settings, except for the pathname, are retained (in static storage) from one invocation to the next unless overridden by a supplied argument. Errors are handled by the condition handler in the following manner. If the error can be ignored, idump_signal writes the condition name and the erroneous pathname offline, terminates the current segment, and proceeds to the next entry. If the error cannot be ignored, the condition name is written online and the listener is called. At this point, commands can be typed at the terminal in an attempt to correct the problem.

Name: backup_load

The backup_load command prompts the user for the names of tapes that are to be loaded. This command is the primary procedure of the reloader for the backup system. It can be called directly, but is usually invoked by the operational interface procedure reload.

Usage

backup_load -control_args- pathname

where control_args can be one or more of the following:

-debug	enables unprivileged operation for debugging and unprivileged users.
-first	loads the first copy only of the branch encountered on the tape (applicable to retrievals only).
-last	loads all copies encountered on the tape so that the last one remains after the tape has been loaded.
-map	enables map writing.
-nodebug	enables normal operation.
-nomap	disables map writing.
-noquota	does not reload quotas.
-quota	enables reloading of quotas on directories.
-notrim	disables pruning of excess entries.
-trim	enables pruning of excess entries in a directory.

If a control argument without a hyphen is encountered, it is taken to be the pathname of a retrieval control segment and the load is therein controlled. If no pathname is given, everything on the tape is reloaded.

Notes

For compatibility with previous systems, backup_load is temporarily accepting control arguments without hyphens. For as long as hyphens remain optional, the first unrecognized argument is taken to be the name of the retrieval control file.

Format of the Retrieval Control File

The retrieval is controlled by an ASCII segment containing one line for each entity to be reloaded. A line can contain a single pathname or two pathnames separated by an equal sign. The left-hand-side specifies the object sought and the right-hand-side, if present, specifies the new name under which that entity is to be reloaded. The sought pathname must begin with a > and end with either an entryname or the characters >**. If an entryname is specified, a single entity by that name is retrieved. If >** is specified, the entire directory hierarchy, beginning at the point indicated in the pathname, is retrieved.

If a new name is specified on the right, it can be either a pathname or an entryname. Components of pathnames are replaced one for one; that is, a subtree can only be moved to a new point at the same level in the hierarchy. If an entryname is given, the single object found is loaded with its former pathname and the new entryname.

If two pathnames are specified, both are checked against the current hierarchy and a new pathname consisting only of the primary entryname is created. This new pathname, as well as the original, is then used in searching the hierarchy. For example, >udd>m is translated into >user_dir_dir>Multics and both versions are sought.

A retrieval control file can contain a maximum of 50 lines. When the retrieval is complete, backup_load attempts to delete satisfied requests from the retrieval control file.

Examples

A retrieval control file containing the line:

```
>udd>Multics>**
```

causes the tape to be searched for directories and segments whose first two pathname components are >user_dir_dir>Multics. These items are reloaded as found.

A retrieval control file containing the line:

```
>ldd>a>b>c
```

causes the tape to be searched for the segment >library_dir_dir>a>b>c. This item is reloaded as found.

A retrieval control file containing the line:

```
>ldd>a>b=c
```

causes the tape to be searched for the segment >library_dir_dir>a>b. This item is reloaded under the name >ldd>a>c.

A retrieval control file containing the line:

```
>ldd>x>y>**=>ldd>z>y
```

causes the tape to be searched for directories and segments whose first three pathname components are >library_dir_dir>x>y. These items are reloaded in the subtree >ldd>z>y.

Name: check_mst, ckm

The check_mst command is used to read one or more tapes that define a Multics System Tape (MST) and provides information about improper combinations of attributes and missing procedures. This command also provides information about the segment numbers assigned for supervisor and initialization segments, their names, attributes, and the number of references.

Usage

check_mst args

where args can be either keywords or numbers whose interpretation depends on the most recently specified keyword.

The possible keywords are:

col — Succeeding numeric arguments define the collections after which a cross-reference check is desired.

tape — Succeeding numeric arguments identify the reel numbers of the tapes to be read.

debug — This keyword sets a switch that preserves the tables constructed during checking and should not normally be used.

Notes

When invoked, check_mst assumes that the keyword col has already been specified. In addition, if no values for a particular keyword are encountered, the values used in the last invocation in the same process are reused. When invoked for the first time in a process, the default settings are equivalent to:

check_mst col 2 tape 9999

The debug keyword must be specified for each invocation in which it is to be active.

Keywords can be specified in any order and can be respecified as often as desired.

check_mst

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

```
check_mst 3 tape NNN
```

where NNN is the reel number desired.

Provision is made for up to five collections to be loaded and checked for cross-references. Currently, though, Multics uses only three collections.

Diagnostics and Results

Output produced is directed through the I/O switch to the segment NNN.ckrout where NNN is the reel ID of the first tape specified.

When the tape reel has been mounted by the operator, an asterisk (*) is printed on the user's terminal. After each collection is read, the number of words to be loaded into segments directly from the tape is printed. When the tape is dismounted, the total number of words read from the tape is printed, 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 be printed 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 i-th collection from user tapes, extracting names and linkage information.

2. "Collection-mark K": A collection mark with a value of K was read from the user tape, completing the loading phase for the collection. A simulated collection mark of 0 is generated if not enough tapes were specified.

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 legal in the context in which it was encountered; an

check_mst

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. (This diagnostic also occurs, without the "Illegal Control Word" message, if the "Collection Mark 0" mentioned above was generated.)

4. "Possible tape format error or incorrect switch setting": The physical tape can not 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 do not exist on the tape.

5. "Name <z> alsc on seg No.i": Duplicated names appear on the tape. This occurs normally only in collection 3 for two segments that are different in ring 0 and outside it.

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.

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. Refs-Count — the number of times the name was encountered as external reference by some other segments.

4. Acc — the access to the segment as specified on the tape (REWP).

5. Switches (miscellaneous attributes)

   Segment Status

   W — Wired down.

   G — The segment is paged.

   P — The segment is a per-process segment.

   D — The segment is the descriptor 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.

6. Ring brackets

7. Length (a multiple of 16 words)

8. Pathname

check_mst

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.

comp_dir_info

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 seg1 seg2 -control_arg-

where:

1. seg1 is the pathname of the old directory information segment. If the suffix dir_info is not supplied, it is assumed.

2. seg2 is the pathname of the new directory information segment. If the suffix dir_info is not supplied, it is assumed.

3. control_arg is an optional control argument. It can be one of the following:

   -verbose, -vb compares and prints maximum information.

   -long, -lg compares almost all items and prints all information.

   -brief, -bf compares and prints minimum information.

If no control argument is specified, an intermediate amount of information is compared and printed.

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

Unless -brief was specified, a heading identifying the directories being compared and the times the information was saved is printed.

Output is in three sections:

modified entries
deleted entries
added entries

comp_dir_info

For deletions and additions, a heading of the form:

deleted: seg xxx

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

item_name: value

Consult the description of the subroutine list_dir_info_ in this document for information concerning what items are listed for a given verbosity.

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

modified: dir yyy

is printed, followed by lines of the form:

item_name changed from value1 to value2

and:

item_name added: value

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

When looking for a match between the old and new dir_info segments, comp_dir_info looks first for a match on unique ID. 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, comp_dir_info checks a set of items that depend on the verbosity requested, to determine whether to report the entry as modified.

The names item is always checked. Other checking is dependent upon the control argument. The following table lists the items and the verbosity level at which they are first checked.

Segments:

-bf names
deletion of ACL
truncation

comp_dir_info

```
          -lg safety switch
              author
              bit count author
              ACL
              date branch modified
              records used
              max length

          -vb date modified
              bit count

     Links:

          -bf names
              link target

          -lg date link modified

     Directories:

          -bf names
              deletion of ACL

              quota
              bit count

          -lg safety switch
              author
              bit count author
              ACL
              initial seg ACL
              initial dir ACL

          -vb date branch modified
              date modified

     The following items are never compared:

          date dumped
          date used
```

If comp_dir_info completes a pass without finding any modifications, deletions, or additions; it prints "Identical." Invoking the command with a higher verbosity level may detect some changes.

copy_mst

Name: copy_mst, cpm

The copy_mst command is used to create copies of Multics System Tapes, or to coalesce multiple tape reels into one output reel.

Usage

copy_mst

All arguments are requested interactively from the user's terminal.

Summary statistics are printed on the user's terminal for each collection copied, as well as for the end of all tapes/collections requested.

copyright_archive

Name: copyright_archive

The copyright_archive command adds a copyright notice to each component of an archive of source commands. If a program already contains a copyright notice no change is made to it. Different notices are used for each different language type suffix. If adding the copyright notices would cause the archive to overflow, an error message is typed and no modifications are made to the archive.

Usage

copyright_archive path -control_args- -n1- -n2-...

where:

1. path — is the pathname of the archive to be modified.
2. ni — are component names to be modified. If no ni are given, the entire archive is processed.
3. control_args — are selected from the following:

   -check — no modification is made to the archive. The archive overflow is tested for, and if -long is specified, comments are printed describing what would be done if archive were modified.

   -long, -lg — messages are printed describing what is done to the archive.

   -suffix Z — the copyright messages are named type.Z where type is pl1, alm, etc. If this argument is not specified, the default suffix is copyright.

Operation

The copyright_archive command makes two passes over the archive: the first checks to see what change in length would be made to the archive, and the second makes the changes.

The copyright messages are inserted at the beginning of each component unless the component begins with the string %; -- in this case the notice is inserted right after the percent and semicolon characters.

copyright_archive

The copyright messages are assumed to reside in >ldd>include unless the command:

```
copyright_archive$test dirname
```

has been executed. Both the directory name and suffix are effective for all subsequent invocations until reset.

If a language type has no corresponding copyright notice segment, an error message is printed and no change is made to that component. The dates in the archive header are left undisturbed by this command.

If a segment named type.Z_delete is found in the notice directory (where Z is usually copyright and type is the language suffix), each component selected is checked to see if the notice in the delete segment is present, and if so, the old notice is deleted before the new notice is added.

Name: cross_reference, cref

The cross-reference command is used to create a cross-reference table composed of program names and names of programs that call them, as determined by a driving list.

Usage

```
cross_reference arg
```

where arg is either a segment name, the name of an object archive, or specifies a driving list named arg.crl.

Notes

The output is directed to a segment (in the same directory as the driving file or segment) named arg.cr. All programs called by any module specified in the driving list appear in the left hand column with their primary name first and their entrynames following. To the right of the entrynames appears the names of the modules specified in the driving list that call them. A star to the left of a called program's entryname indicates that it is an entry in a program not included in the scope of the driving list. Entries included in the scope of the list not called by any other such entry is listed in the left hand column. Since it is possible for a component in an archive to refer to another component by a synonym stored in the bindfile, it is probable that any such component is reported as calling something not included in the scope of the driving list.

If the argument to cross_reference is a segment or archive name rather than a driving file, the output file contains a list of all calls made by the segment, or in the case of an archive, all calls made by all of the components.

Format of a Driving File

The driving file consists of three parts. The first part consists of absolute pathnames of directories to be searched in finding the desired object segments and archives. Up to and including nine pathnames, separated by blanks, tabs, or new lines, are accepted. If less than nine are specified, the directory containing the driving file is also included in the search list. This first part is optional, and, if omitted, causes the search rules to default to the directory containing the driving file.

The second part of the driving file consists of a list of aliases in the following format:

segment_name alias1 alias2 alias3...

If it is desired that a segment be known by its own name as well as its aliases, the segment name must be included in the list of aliases. This section is optional, and can be omitted entirely.

The third part is a list of the segments, both stand-alone and archive, that are to be cross-referenced. Only one segment name is allowed per line.

Example

In >udd>Project_id>Person_id, there resides three segments; loose, bound_p_.archive, and test.crl. The segment bound_p_.archive contains p1, p2, and bound_p_.bind. An entry in loose, called loosent, is to be made synonymous with loose.

```
loose calls p2, alloc_, and loose.
p1 calls p2, p1$recur, and loosent
p2 calls loose, p2, and p1
```

test.crl contains the following driving file:

```
>udd>Project_id>Person_id
loose loose loosent
loose
bound_p_.archive
```

Executing the command "cref test" produces the output segment test.cr:

```
alloc_
*alloc_                 loosent loose

loose
 loosent                p1
 loose                  p2 loosent loose

loosent
 loosent                p1
 loose                  p2 loosent loose
```

cross_reference

```
p1
 recur                p1
 p1                   p2

p2
 p2                   p2 p1 loosent loose
```

date_deleter

Name: date_deleter

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

Usage

date_deleter dir_path n_days

where:

1. dir_path is the pathname of the directory in which the deletions are to occur.

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.

Example

date_deleter >ldd>old 7

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

edit_mst_header

Name: edit_mst_header, emh

The edit_mst_header command takes an MST header file and produces a new MST header file. The nature of the editing is specified in a third edit_header file.

Editing is performed by header entry. A header entry is a fini or collection header statement or a series of statements describing a single segment on the MST. An example of the latter is a series of statements beginning with a name statement and ending with an end statement. In the edit_header file, header entries can be preceded by a control line. Following is a list of recognized control lines and their effect:

1. skip_to: -- the old header file is scanned starting at the current position until a header entry is found whose first line matches the first line of the header entry immediately following this control line in the edit_header file. The current position in the old header file is set to the beginning of the matching header entry. If no match is found, a message is printed on the terminal and the command returns.

2. skip_thru: -- same as skip_to:, only the current position in the old header file is placed at the beginning of the header entry immediately following the matching header entry.

3. copy_to: -- the old header file is copied starting at the current position to the new header file until a header entry in the old header file whose first line matches the first line of the header entry following the control line in the edit_header file is found. The current position is set to the beginning of the matching header entry and the matching header entry is not copied. If no match is found, an error message is printed on the terminal and the command returns.

4. copy_thru: -- same as copy_to:, except the current position is set to the header entry in the old header file following the matching header entry and the matching header entry is copied.

5 copy_to, replace: -- same as copy_thru:, except the header entry following the control line in the edit_header file is copied into the new header file instead of the matching header entry in the old header file.

If no control line precedes a header entry in the edit_header file, then this header entry is simply copied into the new header file.

edit_mst_header

All comparisons are based on the first line of the header entry, the first line being those characters occurring before a comma (,) or a semicolon (;). All blanks, tabs, newlines, and all characters occurring between /* and */ are ignored.

Example

```
name:  abc, cde, fgh;
end
```

matches:

```
name:  /* comment */ abc, fgh, g;

end;
```

because the first line in both cases is:

```
name:abc
```

however,

```
name:  abc;
end;
```

does not match:

```
object:  abc;
end;
```

Usage

```
emh edit old new
```

where:

1. edit — identifies the edit header file whose pathname is edit.edit_header.

2. old — identifies the old header file whose pathname is old.header. old can also be -hard for the current hardcore header file or -soft for the current softcore header file.

3. new — identifies the new header file whose pathname is new.header If new is omitted, the new header file is assumed to have the entryname of edit.header in the user's working directory. If the new header segment does not exist, it is created.

expand

Name: expand

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

Usage

expand paths

where paths are the relative pathnames of files to be expanded.

Notes

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

Expand 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.la n g. An include statement such as: % include a; looks for a file called a.incl.lang, first in the working directory and then in >library_dir_dir>include. 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;.

gen_sst_card

Name: gen_sst_card, gsc

The gen_sst_card command is used to calculate the sizes of the Active Segment Table (AST) pools and to print what the System Segment Table (SST) card should look like.

Usage

```
gen_sst_card n1 n2 n3 n4 -sst_size- -number_of_memories-
     -pd_records-
```

where:

1. n1-n4	are the number of AST entries to be assigned to the corresponding AST list. (The lists correspond to sizes of 4k, 16k, 64k, and 256k segments.)
2. sst_size	is used to specify the number of 1024 word blocks to be used in the SST segment. The size of the first pool (4k segments) is then calculated to use all of SST not used by the other lists or the SST header.
3. number_of_memories	specifies the number of 128K system controllers handled by the header of the SST. This control argument can only be used with the sst_size control argument. The default value is three memories.
4. pd_records	is the number of records on the paging device. This control argument can only be specified if all other arguments are also specified.

gen_tcd_card

Name: gen_tcd_card, gtc

The gen_tcd_card command is used to print the tcd configuration card that is needed to handle the specified input sizes.

Usage

gen_tcd_card no_apts no_itts no_dsts

where:

1. no_apts — is the number of Active Process Table (APT) entries needed.
2. no_itts — is the number of Interprocess Transmission Table (ITT) entries needed.
3. no_dsts — is the number of Device Signal Table (DST) entries needed.

generate_mst

Name: generate_mst, gm

The generate_mst command is used to generate a Multics System Tape (MST). This command uses an ASCII segment known as a header for a driving file in creating the tape.

Usage

generate_mst hdr_name tape_no -search_list-

where:

1. hdr_name — is the relative pathname of the full generate header from which it is desired to create an MST (without the suffix header).

2. tape_no — is the identification number of the tape to be generated.

3. search_list — is an optional control argument indicating that the user wishes to specify the search rules for the segments to be placed on the tape. It is specified as -dr or -directory. If used, the user must have in his working directory an ASCII list of relative pathnames of directories to be searched in the order in which the search is desired. This list should have the name hdr_name.search where hdr_name is the same as argument 1.

Notes

Default search rules:

1. Search the current working directory.

2. If the segment is not found search the directory >ldd>hard>object.

3. If the segment is not found in 1. or 2., it is missing.

get_device_status

Name: get_device_status, gds

The get_device_status command prints the current assignment of the specified device. If the calling process has hphcs_ privileges, the command prints the process_group_id of the process to which the device is assigned. Otherwise, it just reports the device as being assigned to another process.

Usage

get_device_status ionames

where ionames are the names of I/O devices about which information is desired.

get_library_segment

Name: get_library_segment, gls

The get_library_segment command can be used to find source 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 of control arguments.

   NOTE: All of the control arguments specified in the command are effective for each seg_name specified.

-sys lname specifies that get_library_segment should search in the library named lname for the segments. For the Multics System Libraries, lname can be one of the names shown below:

lname	directory path(s) searched
hardcore, hard, h	>ldd>hardcore>source
online_system, online, os	>ldd>standard>source >ldd>tools>source >ldd>auth_maint>source >ldd>network>source >ldd>languages>source
standard, sss	>ldd>standard>source
tools, t	>ldd>tools>source
auth_maint, am	>ldd>auth_maint>source
languages, lang	>ldd>languages>source
network, net	>ldd>network>source

get_library_segment

standard.object, sss.o	>ldd>standard>object
tools.object, t.o	>ldd>tools>object
auth_maint.object, am.o	>ldd>auth_maint>object
languages.object, lang.o	>ldd>languages>object
network.object, net.o	>ldd>network>object
info_files, info	>documentation>iml_info_segment >documentation>info
pt_files, pt	>documentation>pt_files
include, incl	>ldd>include

-long, -lg — specifies that the pathname of the segment from which each source segment is copied is to be printed.

-rename new_name — specifies that the immediately-preceding seg_name is copied into the user's current working directory, and then its name is changed to new_name.

-control control_dir, -ct control_dir — specifies that get_library_segment looks in the directory specified by control_dir to find its control segments. The control_dir argument may be -working_directory or -wd, in which case get_library_segment looks in the current working directory for its control segments. (See "Operation" below.) If this control argument is not specified, get_library_segment looks in the directory >ldd to find its control segments.

Notes

If the -sys control argument is not given, then get_library_segment searches the following default group of directories, in the order listed:

```
>ldd>hardcore>source
>ldd>standard>source
>ldd>tools>source
>ldd>auth_maint>source
>ldd>network>source
>ldd>languages>source
```

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

If the -control argument is given in the get_library_segment command, then one or more -sys arguments, specifying the names of the user libraries to be searched, must also be given.

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

Examples

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

copies foo.pl1 and random.alm from >ldd>tools>source or >ldd>sss>source if they exist in one of these standard directories.

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

copies foo.pl1 from one of the following directories. The directories are searched in the order listed below:

```
>ldd>languages>source
>ldd>standard>source
>ldd>tools>source
>ldd>auth_maint>source
>ldd>network>source
>ldd>hardcore>source
```

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

searches the default group of directories for segment gorp.pl1, copies it into the user's working directory with the name gorp.pl1, and then renames it to glop.pl1.

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

copies the object segment fortran_blast_ and the bind segment, bound_parse_.bind, from the directory >ldd>languages>object.

get_library_segment

Operation

If no -control argument is specified, then get_library_segment searches for segments in one or more of the Multics System Libraries. From each library name given in a -sys argument, get_library_segment constructs a pathname of the form >ldd>lname.control. It uses this as the pathname of a control segment. This control segment tells get_library_segment 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 the procedure that searches the directory to find seg_name. search_procedure can have the form:

```
segment_name
```

or:

```
segment_name$entry_name
```

For each directory_path specified in the control segment, get_library_segment 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_names,
     containing_seg, code);
```

where:

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

2. seg_names are the names of the segment to be found, including any language suffix. (Input)

get_library_segment

3. containing_seg — is the name of the segment in directory_path in which seg_name was found. Usually, this name is the same as seg_name, or it is the name of an archive that contains seg_name. (Output)

4. code — is either a standard storage system status code, or 0, or 1. If it is 0, then seg_name was found in directory_path>containing_seg. If it is 1, then seg_name was not found. (Output)

Notes

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

```
archive x directory_path>containing_seg seg_name
```

to copy the segment into the current working directory. If the -rename argument was specified for seg_name, the segment in the working directory is then renamed.

If the code returned by search_procedure is 0, and if the final eight nonblank characters of containing_seg are not the suffix archive, then get_library_segment calls copy_seg_ to copy directory_path>seg_name into the current directory, renaming the segment as it is copied if a -rename argument was specified.

If a code of 1 is returned by the search_procedure, then get_library_segment continues the search with the next directory_path in the current control segment. If the current control segment contains no more directory_paths, then 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, then get_library_segment prints an error message, and begins searching for next seg_name.

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

The procedures get_archive_file_$srchgls and get_primary_name are used to find segments in the Multics system libraries.

If no -sys argument is specified in the command, then get_library_segment uses a built-in control list to search the default group of directories listed above.

User Libraries

If the -control argument is specified, get_library_segment can be used to extract segments from a user library. This control argument causes get_library_segment to construct a control segment pathname of the form: control_dir>lname.control. Therefore, the -control argument 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 get_library_segment to extract a copy of source program alpha.pl1 from his 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 >udd>Project_id>Person_id>library>source.archive, which has, as one of its names, alpha.pl1.

grab_tape_drive

Name: grab_tape_drive, gtd

The grab_tape_drive command allows the user to request a number of nine and/or seven track tape drives and have a given command executed when the tape drives are simultaneously available. The grab_tape_drive command checks the tape drive command configuration every 10 seconds and when there are sufficient drives available, the given command is executed.

Usage

grab_tape_drive -control_args- command_line

where:

1. control_args may be taken from the following list:

   -t9 n — finds n free 9 track tape drives before executing the command line. The default if -t9 n and -t7 n are both omitted is one 9 track drive.

   -t7 n — finds n free 7 track tape drives before executing the command line.

   -info, in — lists the tape drive configuration and returns without further processing.

   -long, -lg — lists the tape drive configuration every minute.

2. command_line — is the (optional) command line to be executed when the specified number of tape drives are free.

Notes

It is possible for a tape drive to be attached by another process after grab_tape_drive finds the drive free and before the caller's program is executed and attaches the drive. If the command to be executed calls com_err_ with the system code "no device available", grab_tape_drive continues to check the tape drive configuration and retry the command. Such a retry is attempted five times.

This command requires access to certain privileged gates into the supervisor.

Name: if

The if command provides conditional execution of a command line if some specified condition is met.

Usage

if key keyargs -then c1 -else c2

where:

1. key — selects the type of test performed. See also the legal key section.
2. keyargs — are arguments depending on the choice of key.
3. -then — is a literal control argument.
4. c1 — is a command line executed if the key test succeeds.
5. -else — is a literal control argument.
6. c2 — is a command line executed if the test fails.

The -then c1 portion and the key must be supplied; all the other parts can be omitted. If the -else c2 portion is omitted, and the test fails, no action is taken. If the -else c2 portion is supplied, it must come after the -then c1 portion. Command lines to be executed are passed to the user's current command processor procedure via a call to cu_$cp. All keys can be preceded by the string -not to reverse the sense of the test. If either the c1 or the c2 argument is omitted, no action is taken.

Legal Key

Key	Succeeds
arg x	if x is an argument (i.e., if -then is not the second argument.)
noarg x	if no argument x (useful in exec_coms, where an unsupplied parameter like &5 yields no argument.)
is path	if there is a branch named by path. A directory will do, but a link to nowhere will fail.

isnt path	if there is no branch pointed to by path.
isdir path	if there is a directory branch named path.
islink path	if there is a link named path.
isfile path	if there is a segment branch named path.
isnzf path	if there is a segment pointed to by path (links ok) that is a segment with a nonzero bit count.
day xxx	if the current date is xxx. xxx can be a day of the week expressed as three letters, like Mon, or a day of the month, expressed as two digits, like 04.
argeq xxx yyy	if arguments xxx and yyy are equal. If both are omitted, then they are equal. If yyy is not supplied, then the test fails. Otherwise the test is by PL/I string comparison, so trailing blanks are not significant.
ask query	the user is asked a question consisting of the string query. If he answers yes, then the test succeeds. If query is omitted, the string answer is used. The question is asked by a call to command_query_.

Example

The following exec_com checks for errors before calling the change_wdir command:

```
&command_line off
if noarg &1 -then "ioa_ ERROR1; pi"
if -not isdir &1 -then "ioa_ ERROR2; pi"
if argeq wd &1 -then -else "cwd &1"
&quit
```

Notice, the arguments to the command processor contain blanks and, therefore, are in quotes.

Name: include_cross_reference, icref

The include_cross_reference command is a library tool that performs an include file cross-reference. This command uses an ASCII driving file containing the absolute pathnames of the directories to be searched. All archive files in the given directories are inspected for source procedures. Each source procedure in turn is inspected for its include file usage.

The output from this command is an ASCII segment that lists all include files alphabetically; below each include file name is printed a list of those source procedures that use it.

Usage

include_cross_reference search_list

where search_list is the pathname of the ASCII driving file. Any name can be specified. The output from include_cross_reference is an ASCII segment called search_list.icr, where search_list is the entryname.

Notes

The command does not check the include file libraries themselves. Therefore, the presence of include files listed is not verified. Also, include files included in include files are not found.

list_assigned_devices

Name: list_assigned_devices, lad

The list_assigned_devices command prints the device name of all devices assigned to the calling process as reflected by the assignment table in ring 0. It does not use the per-process attach table and can be useful if that table has been rendered incorrect.

Usage

```
list_assigned_devices
```

Name: list_dir_info

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

Usage

list_dir_info segpath -control_arg-

where:

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

2. control_arg is an optional control argument. It can be:

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

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

Notes

If neither -long nor -brief is selected, an intermediate verbosity 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 subroutine list_dir_info_ for information on the items printed printed for each verbosity level.

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 -control_args-

where control_args can be chosen from the following list:

-all, -a	specifies that all the names of a segment will be printed. The default is to print only the primary names.
-depth, -dh	specifies the depth in the hierarchy that is to be scanned. The depth is relative to the base of the specified subtree. This control argument requires a decimal integer specifying the depth immediately following it in the command line.
pathname	If an argument is specified that is neither of the above, it is assumed to be the relative pathname of the subtree to search. The last such pathname specified is the only one listed. If no pathname is given, then the working directory is assumed.

Notes

For each level in the hierarchy that is listed, the names are indented three more spaces making it possible to see exactly which segments exist at which depth in the hierarchy.

For each segment printed, two numbers are printed out. The first number is the number of records used by the segment and the second number is the device ID of the secondary storage device on which the segment resides.

Name: listing_tape_print, ltp

The listing_tape_print command is the driving module for the listing tape system. It has as entries the various commands that govern the functions of the listing tape system.

A listing tape is a single logical entity containing files in a sorted order. The files consist of single segments or multisegment files. The files are sorted according to the ASCII collating sequence by name, area name and system number in that order. A listing tape can exist physically as one reel or several reels of tape taken in a specific order. It can also exist as a segment or a multisegment file in the hierarchy.

At the beginning of a listing tape is a dictionary or list of all the files on the tape in sorted order. The list consists of the name of the file together with other information such as area name, system number, unique identifier, date the file was last modified and last dumped, and the size of the file in words.

When a listing tape is to be created from the contents of a directory, the names are first sorted and a dictionary is constructed. This dictionary is then used to place the files on the listing tape in the sorted order. Similarly, when merging tapes and/or directories, a dictionary is read or created for each input source. A merged output dictionary is created and the output listing tape is constructed using this resulting dictionary.

Entry: listing_tape_print, ltp

This command causes some or all of the files on a listing tape to be printed.

Usage

listing_tape_print -control_args-

where control_args are one or more of the following:

-ip*n* reel_name*m* indicates the names of the reels of an input listing tape. At least one input listing tape must be specified although more than one can be specified by repeating the -ip*n* reel_name sequence. Drive type 7 or 9 track is indicated by making *n* either 7 or 9. If *n* is not given, 9 is assumed. reel_name*m* is the name of the mth reel of a listing tape. The reels must be specified in their

proper order since a listing tape is a single logical entity and the files it contains are in sorted order. Also, for the same reason, all reels must be specified if any are to be used.

-device device_name, -dv device_name
: is the name of the device to be attached for output. If the -pd argument is not used, then the device name is assumed to be a printer device name such as prta or prtb. If the -pd argument is present with a Device Interface Module (DIM) name other than that for the printer, then a device name appropriate to that DIM should be used. Thus, if the DIM name is tape_, then an appropriate tape name should be given, i.e., listing_tape_23 or m1692.

-printer_dim dim_name, -pd dim_name
: indicates that the following argument is the name of a DIM such as tape_ or file_. The printer DIM (prtdim_) is the default, but this control argument allows for the user's own DIM or output onto a tape or into a segment or multisegment file.

-file_input file_name, -fi file_name
: indicates that file_name is the pathname of a control file.

-area_name aname, -an aname
: indicates that aname is an area name of the area to be considered.

-first file_name, -ft file_name
: indicates that file_name is the name of the first file on the listing tape to be printed.

-copy n, -cp n
: indicates that n is the number of copies to be printed of each file output. The maximum is 2.

Notes

The listing_tape_print command first checks for an input print file. If there is no input print file and if an area name is given, it prints all files with that area name. If there is neither an input print file nor an area name, then all files on the tape are printed. In any of the above cases, if there is a -first file name, then all files prior to the given file name are skipped, remembering that the tape is sorted, before any printing begins.

Entry: listing_tape_dictionary, ltd

This command causes the dictionary of the files on the listing tape to be copied from the listing tape into a file in the user's working directory. The name of the file is name_of_first_reel.dict.

Usage

listing_tape_dictionary -ip reel_name1

where:

1. -ip — indicates that the names of the reels of an input listing tape follow. At least one input listing tape must be specified although more then one may be specified by repeating the -ipn reel_name sequence. Drive type 7 or 9 track is indicated by making n either 7 or 9. If n is not given, 9 is assumed.

2. reel_name1 — is the name of the first reel of a listing tape. Since only the first reel is used, it is not necessary to specify the succeeding reels even though each reel has a dictionary before any other data. Only the first reel can be specified separately in this manner.

Entry: listing_tape_merge, ltm

This command causes one or more input listing tapes and possibly the contents of one or more dictionaries to be merged. An output listing tape is generated as the result of this merge operation.

Usage

listing_tape_merge -control_args-

where control_args can be one or more of the following:

-outputn output_reel_namem, -opn output_reel_namem — is a control argument indicating the names of the output reels. One output listing tape must be specified, i.e., at least one reel. Drive type 7 or 9 track is given by making n either 7 or 9. If n is not given, 9 is assumed.

listing_tape_print

output_reel_name*m* is the name of the mth output reel. All reels must be specified and in their proper order. Enough reels must be specified to hold all of the output generated or the current command is aborted with an error. If the first four characters of the reel name are "file" then the output is placed in a segment or multisegment file with the given reel name.

-input*n* reel_name*m*,
-ip*n* reel_name*m*

indicates the names of the reels of an input listing tape. At least one input listing tape must be specified although more than one can be specified by repeating the -ip*n* reel_name sequence. Drive type 7 or 9 track is indicated by making *n* either 7 or 9. If n is not given, 9 is assumed. reel_name*m* is the name of the mth reel of a listing tape. The reels must be specified in their proper order since a listing tape is a single logical entity and the files it contains are in sorted order. Also, for the same reason all reels must be specified if any are to be used.

-directory*i* dir_name,
-dr*i* dir_name

indicates that the ith directory pathname follows. A directory must be specified. Up to ten directories, 0 through 9, are allowed.

-area_name aname,
-an aname

indicates that aname is the area name to be associated with the segments in the ith directory. An area name is any string of up to 16 characters. If no area name is given for the ith directory a default area name of "hardcore" is associated with that directory.

-system_number snumber,
-sn snumber

indicates that snumber is the system number to be associated with the segments in the ith

listing_tape_print

directory. A system number has four fields and is a maximum of 16 characters. The four fields are:

1. a numeric field of 1 to 8 digits
2. a single character nonnumeric field
3. a second numeric field of 1 to 8 digits
4. any ASCII characters the first being nonnumeric

In sorting, the second field is ignored. If no system number is given for the ith directory, then the segment name of the directory is used as the system number. Thus, for directory >ldd>listings>18-36, the default system number is 18-36.

-file_input file_name, -fi file_name — indicates that file_name is the pathname of a segment containing a list of segments and their associated area names. For the ltm command this list is used to delete segments. Using the input file in another way, all files with dates earlier than a begin date are deleted. (See "Notes" below.)

-link yes_no, -lk yes_no — indicates that yes_no, which is either yes or no, tells this command whether or not to process links in the directories given above.

Notes

For the ltm command, at least one input listing tape and one output listing tape are required.

The file input segment has the following format for deletion. Each line has one or two arguments. Each argument begins with a minus sign (-) and is terminated by a blank or a newline character. The first argument is always present, and is the name of the segment to be operated upon. The second argument, if present, is an area name. If no area name is given,

then the last area name given is the default area name. The initial default area name is blank.

The file input segment format for use as a begin date has a plus sign (+) as the first character followed by a date and time expressed as mm/dd/yy tttt.

Entry: listing_tape_create, ltc

The ltc command exists for historical and aesthetic reasons. It differs from the ltm command only in that no input listing tape need be specified.

Examples

To create a listing tape from a single directory, type:

```
ltc -dr0 >ldd>listings>17-36 -an0 hardcore -sn0 17-36 -op tape1
```

or:

```
ltc -dr0 >ldd>listings>17-36 -op tape1
```

To create a listing tape from several directories, type:

```
ltc -dr0 >ldd>listings>17-3b -dr1 >ldd>listings>17-3c
-dr2 >ldd>listings>14-43x -an2 soft -op tape1 tape2
```

To output into the file file_foo, type:

```
ltc -dr0 >ldd>listings>18-36 -op file_foo
```

To merge several tapes and directories, type:

```
ltm -ip tape1 tape2 -ip xtape13 xtape14 xtape15
-dr0 >ldd>listings>18-0a -dr1 >ldd>listings>new_dims
-an1 dims -sn1 31-4c -op otape4 otape5 otape6 otape7
```

To delete from a listing tape, type:

```
ltm -ip tape33 tape34 tape35 -fi delete_list -op tape1 tape2
tape3
```

To get a dictionary, type:

```
ltd -ip xtape13
```

where xtape13 is the first reel of a listing tape.

listing_tape_print

To print all, type:

```
ltp -ip xtape13 xtape14 xtape15 -dv prta
```

To output all into a file zilch, type:

```
ltp -ip xtape72 file_ -dv zilch
```

To print by area name, type:

```
ltp -ip tape27 tape28 -dv prta -an hardcore
```

To print according to an input file, type:

```
ltp -ip tape43 -fi print_list -dv prtb
```

An example of an input list for printing or deleting is:

```
-acc.list      -hardcore
-ab.list
-zero.list     -command
-foo.list
-aaagh.list    -active
```

In this example, ab.list has the default area name hardcore, while foo.list has the default area name command.

Name: mexp

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 iteration 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 definitions 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 invoking 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 &i 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             ..._00000: lda       a
                                          tnz       ..._00000
                                          sta       b
     test      c,d             ..._00001: lda       c
                                          tnz       ..._00001
                                          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
          lls       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/&iifend
&)
            &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.

nothing

Name: nothing, nt

The nothing command does nothing more than return, thereby allowing timing tests to be made at command level.

Usage

nothing

pause

Name: pause

The pause command is an interface to the timer_manager_$sleep entry allowing the caller to "sleep" for a given number of seconds.

Usage

pause -time-

where time is the number of seconds (decimal integer) to sleep. (If not specified, a time of 10 seconds is used.)

print_configuration_deck

Name: print_configuration_deck, pcd

The print_configuration_deck command prints the contents of the current configuration deck as kept in ring 0. The data is kept up-to-date by the reconfiguration commands and, hence, reflects the current configuration being used.

Usage

print_configuration_deck -args-

where args specifies the names of the cards to be printed. They are printed as they are punched. If no arguments are specified, all cards are printed. If more than one card exists with a specified name, all such cards are printed. Up to 32 arguments are processed.

Note

No action is taken for misspelled arguments or valid arguments for which there are no corresponding configuration cards.

Name: print_error_message
pem
peo
pel
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.

print_error_message

Usage

peol code

print_gen_info

Name: print_gen_info, pgi

The print_gen_info command prints out some general information about an object segment. If the object segment is bound, it prints out information about the components of the bound segment. The items printed are name (of segment or component), date time created (compiled), author, translator name, and directory in which the object segment exists.

Usage

print_gen_info objseg -componentname-

where objseg is the pathname of the object segment of interest and componentname is an optional control argument that, if given, causes printing of information about that component only (appropriate only if objseg is a bound segment).

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 which 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 -brief

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. -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.

print_text_boundary

Name: print_text_boundary, ptb

The print_text_boundary command prints out the decimal value of the text boundary field of the symbol section of a standard object segment.

Usage

print_text_boundary names

where names are the relative pathnames of the object segments whose text boundaries are to be displayed.

Example

```
print_text_boundary test_ci

test_ci                 1024
```

Name: print_translator_search_rules, ptsr

The print_translator_search_rules command prints the current translator search rules in effect for the calling process.

Usage

print_translator_search_rules

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 re-create 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 segpath -control_arg-

where:

1. segpath is the pathname of a directory information segment. If segpath does not have the suffix dir_info, the suffix is assumed.

2. control_arg is an optional control argument. It can be:

   -brief, -bf suppresses the comments "creating directory X" and "appending link X".

   -long, -lg prints full information about any missing segments.

See the description of the list_dir_info_ subroutine for an explanation of what is printed for missing segments for a given verbosity.

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-

Both arguments are optional; n is the number of times the message is to be printed, and string is the message to be printed (command-language quotes can be used to permit embedded blanks in the message). If string is an asterisk (*), the previous message is reused. The first time repeat 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 HT character plus ASCII graphics in ascending numeric sequence, is used. If n is not specified, or is 0, its previous value is used; the default first-time value is 10. If string was 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 was 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 lower case), followed by <NL>, causes repeat_line to return to its caller. Any other line is interpreted as a new message to be printed n times.

resetcopysw

Name: resetcopysw

The resetcopysw command turns off the copy switch for the designated segments.

Usage

resetcopysw paths

where paths are the pathnames of segments whose copy switches are to be turned off.

Note

The current state of a segment's copy switch can be determined by issuing the command:

status path -all

where path is the pathname of the segment. The copy switch is not mentioned if it is off.

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. name 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.

save_dir_info

Name: save_dir_info

The save_dir_info command creates a segment containing all information available from the file 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 manipulated by the list_dir_info, rebuild_dir, and comp_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 suffix dir_info, it is assumed.

send_admin_command

Name: send_admin_command, sac

The send_admin_command command can be used by authorized system administrators to request execution of commands by the initializer process, and to change the admin mode password.

This command can be used to send a command to the initializer.

Usage

sac commandline

where commandline is the desired command line. It must be 80 characters or less. If it contains embedded blanks, they must be enclosed in quotes.

Notes

The command line is stored in the segment, >system_control_dir>communications, and a wake-up is sent to the initializer. When the wake-up is received, the initializer types a message, executes the command, and stores the message in log.

To print any unexecuted command in system_control_dir>communications type:

sac -print

To cancel an unexecuted command in system_control_dir>communications type:

sac -cancel

To change the operator password for admin mode (also stored in system_control_dir>communications) type:

sac -chpass
or
sac -cp.

The command requests both the old password and the new password.

The new password is stored in the segment and an online message sent to the operator explaining that the password has been changed.

set_proc_required

Name: set_proc_required, sprq

The set_proc_required command is primarily used as a processor T&D aid when it is desired to run a particular (set of) program(s) on a particular CPU. It instructs the scheduler to restrict the calling process to run only on the specified processor. The current configuration can be determined with the print_configuration_deck command.

Usage

set_proc_required cpu_tag

where cpu_tag is the character string representation of the processor tag for the CPU to be selected.

Example

set_proc_required a

set_text_boundary

Name: set_text_boundary, stb

The set_text_boundary command is used to fill in the text boundary field of the symbol section of a standard object segment. This field's primary use is by the binder that forces the component to begin on the specified text boundary.

Usage

set_text_boundary name1 bndry1...namen bndryn

where:

1. namei	is the relative pathname of the object segment whose text boundary is to be changed.
2. bndryi	is a decimal value for the text boundary. This value must be even.

Example

```
set_text_boundary test_ci 1024
```

set_timax

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 of less than or equal to zero causes it to use the default timax from tc_data$timax.

Examples

```
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's console.

```
set_timax  0
```

sets timax to the default timax (currently eight seconds) and prints messages on the user's terminal and operator's console.

Name: set_translator_search_rules, stsr

The set_translator_search_rules command is a command interface to the find_include_file_ subroutine which allows manipulation of the search rules used to find include files by the language translators.

Usage

set_translator_search_rules paths

where paths are pathnames that are to be searched, in the order given, when searching for an include file. The default search rules are:

```
working_dir
>user_dir_dir>Project_id>include
>library_dir_dir>include
```

The command recognizes the following keywords which can be used instead of pathnames:

```
-working_dir
-home_dir
-referencing_dir
-default
```

Example

```
stsr >udd>m>include >ldd>bos>include >ldd>include
```

setcopysw

Name: setcopysw

The setcopysw command allows the user to set or reset the copy switch for designated segments.

Entry: setcopysw

This entry turns on the copy switch for the designated segments.

Usage

setcopysw paths

where paths are the pathnames of segments whose copy switches are to be turned on.

Name: setquota, sq

The setquota command requires access to the highly privileged gate and places an arbitrary secondary storage quota on a specified directory.

Usage

setquota pathname1 quota1 ... pathnamen quotan

where:

1. pathnamei is the name of the directory on which the quota is to be placed. The active function wd can be used to specify the working directory.

2. quotai is the quota in 1024 word pages to be placed on the directory.

Examples

```
setquota > 29902

setquota >udd>Multics>Jones 0
```

Note

No permission in the directory is required to use this command. It is not necessary that the new quota be greater than the current number of pages being used by this directory. This command causes the directory to have a terminal quota even if it is set to zero. This command does not cause the inferior counts of the superior directory to be updated.

Name: teco, TECO

INTRODUCTION

TECO (Text Editor and COrrector) is a character oriented text editor modeled after the TECO in general use on the Digital Equipment Corp. PDP-10, which was originally written at MIT's Artificial Intelligence project. TECO allows many simple editing requests, macro definitions, iterations, and conditional statements. 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 in general use, many new commands and features have been added that make the macro facility really 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, and allowing macros that reside in files to be called directly from TECO.

TECO is basically a character oriented editor, whereas editors like edm and qedx are line oriented editors. In edm and qedx it is only possible to position the pointer to the beginning of the line. The pointer is then considered to point at the whole line. These editors then supply commands (the substitute or change command) to edit the current line. In TECO such a complicated command is unnecessary because the pointer can point between any two characters in the buffer. The fundamental character-oriented commands are insert, delete, search, and moving the pointer. With these commands it is very easy to do what would be complicated operations in a line oriented editor. The concept of a line as an important entity is not unknown in TECO, however. There are many commands that can be line oriented. These are the L, T, K, X, and S commands.

TECO reads command lines from the user's terminal (actually it reads from the switch user_input) line by line until a line ending with $ is typed. Execution of the complete command string is started when this last line is read. TECO will type "*" when it is waiting for a new command string.

ENTERING TECO

TECO can be called from the Multics command level by the Multics command:

teco -pathname-

If pathname is specified, TECO automatically reads in the file by effectively executing the string "EI/pathname/J" upon entry. If no pathname is specified, the buffer is initialized to empty. To create a new file, type TECO (without specifying a pathname) and then use the I request to insert text.

EXITING FROM TECO

To exit from TECO, type the EQ command (followed by $ and a newline).

TECO DEFINITIONS

TECO uses four storage areas:

1. The buffer is the 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. Commands to TECO are written as a character string that is read into the Command String Area. TECO interprets the characters in the command string as a series of commands. Upper and lower case letters can be used interchangeably in commands.

3. The Q-Registers are 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. The Q-Register pushdown list is 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 command level is returned to, i.e. a "ESC" is typed.

TECO uses numeric expressions for many of its operations. These can consist of any combination of decimal or octal numbers, the unary operator hyphen (-), the binary operators +, -, *, /, ! (boolean or), & (boolean and), and the special valued commands and symbols. All operators are of equal precedence and expressions are evaluated from left to right. Note, however, that parentheses can be used in their normal manner. Spaces are ignored (except to terminate decimal numbers). If two numeric quantities are given with no operator between them, the default operator + is used. Note 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

. (pointer) equivalent to the number of characters to the left of the pointer, i.e. the current value of the pointer.

There is another special symbol related to the symbols above and this is the H (wHole) symbol. This symbol is equivalent to 0,Z. It is the only symbol in TECO that has two values. It is useful for referring to the whole buffer.

Commands that return values can also be used in expressions, but they cannot appear immediately to the right of an operator. This is because the command assumes that everything to its left is part of one of its arguments. If a command appears within parentheses, it assumes that its arguments are entirely between the last parenthesis and the command. Therefore a command does not read parts of an expression outside the parentheses in which it appears.

The plus and minus binary operators (this does not include the unary minus) assume a right operand of 1 if none is given.

EXAMPLES

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 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.

EXAMPLES

(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

TECO types out error messages in one of two modes, long or short. Short error messages are less than nine 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 command:

```
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 types out the long error message:

```
TECO_error "/: ?"
```

The above holds for all error messages except those informing the user that a file could not be found.

IMPLEMENTATION RESTRICTIONS

The maximum number of characters allowed in a Q-register is 262143. The maximum number of characters allowed in a quoted string is 262143, as is the maximum number of characters in a TECO command line. Note 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.

LEARNING TECO

This description of TECO contains three additional parts. In the second part, commands are described that:

1. read and write files
2. examine text within a file
3. make deletions and insertions
4. search for strings of text

Examples of using the commands are given at the end of the commands part of the description. After reading the second and third parts, the reader should be able to use most of the common editing requests.

In the third part of this description, more sophisticated TECO commands are described, including use of Q-registers, macros, iterations, conditionals. The commands listed in Section III transform TECO from just another editor to one of the most powerful general purpose text editors in existence.

The fourth part of the description contains a summary of all the TECO commands in alphabetical order. This is intended to be used as a reference section.

BASIC TECO COMMANDS

The most general form of a TECO command is:

m,nX/string/

where m and n are optional numeric arguments, X is the command to be executed, and /string/ is a quoted string. In most cases, the command is just one character, though in some cases, it may be two characters. Not all of the commands take arguments. Those that do generally have default values for missing arguments. Only a few commands expect quoted strings. The string must not be ommitted if the command expects one. Some commands also return values; this is discussed later in "Advanced TECO Commands."

The letters chosen for commands generally have some mnemonic meanings, which are indicated in the description of each command. Unfortunately, TECO has a fairly long history, having originally been developed for editing paper tapes, and so some of the mnemonic meanings are almost lost now. As many commands as one wishes can be typed at a time. Execution of the commands does not start until after the "$" followed by a newline character is typed. Spaces can be inserted anywhere (except in the middle of numbers) and newline characters can be inserted anywhere except between a command and its arguments.

Remember that uppercase and lowercase letters can be used interchangeably as commands.

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 just inserted.

Writing a File: - EO (External Output)

EO/pathname/ writes the contents of the buffer to the file specified by pathname. This command takes arguments similar to the T command; it writes out that part of the buffer which would be

typed by T. i Note, however, that if no arguments are given, EO assumes B,Z as the default rather than 1.

NOTE: The pointer is never moved by the EO command.

Typing the Buffer - T (Type)

T		equivalent to 1T
nT	+n	types out the string of characters beginning at the current pointer position and terminating after n newline characters have been encountered. T types out the rest of the current line, and 2T types out the rest of the current line and the next line. The last character typed by T is a newline unless there aren't that many lines in the file.
	-n	types out starting just after the (n+1)th newline to the left of the pointer and finishing at the pointer. 0T types out the beginning of the line up to the current pointer. Usually two T commands are given at once, such as 0TT, which types out the entire line the pointer is in. When 0T is useful, the last character it types out is not a newline. -T types out the previous line and the beginning of the current line. If the pointer is at the beginning of a line, -T types out the previous line, the newline at the end of that line, and nothing more.
m,nT		Types out the (m+1)th through the nth characters of the buffer.

NOTE: The pointer is never moved by the T command.

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.

nC — Moves the pointer n characters to the right of its current position (equivalent to .+nJ). If n is ommitted, 1 is assumed.

nR — Like nC except it moves the pointer to the left (equivalent to -nC). If n is ommitted, 1 is assumed.

nL +n — Moves the pointer to the right, stopping after it has passed over n newline characters. If n is ommitted, 1 is assumed. L moves the pointer to the beginning of the next line.

-n — 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, and -L moves the pointer to the beginning of the previous line.

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.

K — Takes arguments like the T command except it deletes that text T types. The pointer is moved to where the deletion took place. If no arguments are specified, 1K is assumed.

+n — deletes all the characters beginning at the current pointer position and terminating after n newline characters have been encountered. 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.

-n — deletes all the characters starting just after the (n+1)th newline to the left of the current pointer and ending at the current 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 command sequence 0L-K can be used.

m,nK — Deletes the (m+1)th through the nth characters of the 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.

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/ — 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 nth 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/ — Instead of searching the entire buffer for n occurrences of the quoted string, only m lines from the current pointer are searched. If m is positive, 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, 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. If m is less than or equal to 0, n must be negative. If m is greater than zero, n must be positive.

Typing Out Values - = (Equals)

n=	or m,n=	types out the decimal value of all the arguments separated by spaces and ending with a newline.

Leaving TECO - EQ (External Quit)

EQ	returns to the caller of TECO (e.g. Multics command level). (Don't forget to do an EO command before the EQ, if the editing is to be saved.)

Restarting TECO after a Quit

If quit is used to abort a command string, the program_interrupt (pi) command can be used to restart the TECO command. It does not abort the entire command string; only those commands not yet executed. The current command is aborted if the effect of doing so is identical to that of not starting the command in the first place. The TECO command keeps track of what it is doing, so that if the sequence:

```
(quit)
program_interrupt (or pi)
```

is given, it does not abort the current operation if it would leave TECO in an inconsistent state. In other words, the sequence only interrupts between TECO commands, not in the middle of a command.

At times it is desirable to get around this feature. When doing an EO, for instance, TECO does not allow the user to pi back to TECO command level, once the EO has started until, it has completed writing the file. To get around this, type:

```
(quit)
TECO$abort          or          TECO$ABORT
```

When TECO$ABORT is called, the most recent invocation of TECO aborts its current operation without checking for consistency of states. Note 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 a runaway E command. Under other circumstances, however, it is wise for the user to type:

```
-5t5t
```

to ensure control is maintained. Except for the case of a runaway EO command, this feature is probably totally unnecessary in normal use.

STAND ALONE EXAMPLES

Entering Teco

TECO source.pl1	enter TECO and read in the file source.pl1 from the working directory.
TECO <x>y>z>a.ec	enter TECO and read in the file specified.
TECO	enter the buffer initially empty.

Reading a File

EI/source.pl1/	Insert the text contained in source.pl1 at the current point in the buffer.

Writing a File

EO/new_source.pl1/	Write the whole buffer out into new_source.pl1.
.,zEO/bottom/	Write out the buffer from the current pointer to the end into the file "bottom".
2EO/lines/	Write out two lines starting at the current pointer position to the file "lines".

Typing Text

2T	Type out from "." to the end of the next line.
0T	Type out the current line from its beginning to ".".
0TT	Type out all of the current line.

teco

25,100T	Type out the 25+1 (26th) through the 100th character of the buffer.

Moving the Pointer

J	Position the pointer at the beginning of the buffer.
ZJ	Position the pointer at the end of the buffer.
L	Position the pointer at the beginning of the next line in the buffer.
0L	Position the pointer at the beginning of the current line.
-L	Position the pointer at the beginning of the current line.
R	Backup the pointer by one character position.
812-388C	Move the pointer ahead 812-388 (424) character positions.

Deleting Text

19,22K	Delete the 19+1 (20th) through the 22nd character of the file. Set the pointer to 19.
19J 3D	Move the pointer to the right of the 19th character and then delete the next three characters (20-22).
HK	Delete the whole buffer.
-D	Delete the character just to the left of the pointer.

Inserting Text

I/abc /	Insert the line abc followed by a newline character at the current pointer position.

teco

I.abc.	Insert the string abc without a newline character.
65I	Insert the character with ASCII code 65 (A) at the current pointer position.

Typing Values

Z =	Type out how many characters are in the buffer.
Z,.=	Type out how many characters are in the buffer followed by the current pointer position.
=	Type just a blank line.
Q6+53 =	Type out 53 plus the value contained in Q-register 6.

Searching for Text

J S/Hello/	Position the pointer just to the right of the first occurrence of the string Hello in the buffer.
ZJ -S"Hello"	Position the pointer just to the left of the last occurrence of the string Hello in the buffer.
J 3S"* "	Position the pointer just after the third occurrence of a line ending with an asterisk (*).
J 1,1S/Hello /	Position the pointer just after the first line in the buffer if it ends in Hello. If the first line does not end in Hello, type out an error message.

EXAMPLES OF BASIC EDITING REQUESTS

In the following examples, underlined text is produced by TECO.

TECO abc.pl1	Enter TECO and read in the segment abc.pl1.
⌘5LT$	Move to the 6th line and type it out.
dcl a fixed bin; ⌘S/a/-DI/b/0LT$	Change the "a" to a "b" and retype the line.
dcl b fixed bin; ⌘S/dcl d/0LKT$	Search for the declaration of d and delete the line that contains it. Then type out the next line.
dcl f fixed bin; ⌘KI/dcl g char(2); /$	Delete this line and then insert a declaration of g.
⌘EO/abc.pl1/EQ$	Write the edited text out to the file and then return from TECO.

teco

ADVANCED TECO COMMANDS

In "Basic TECO Commands" the general form of a TECO command was given. Some items were left out, however. The actual format is:

m,nXq/string1//string2/.../stringn/

The q indicates a Q-register on which the command is to act.

It should also be noted that more than one string can be given. Although no TECO command 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 command.

In the "Introduction" we specified that expressions can be built from numbers, special valued commands, and symbols. Examples of valued commands are given in this section. Care should be taken to notice that commands with values appear only on the left side of the first operator, or within parentheses. Otherwise, the part of the expression preceding the command is considered to be an argument to the command.

The effect of many commands can have their function changed by preceding the command with a colon (:). The colon has no fixed meaning--it is defined for each command individually. The following commands given earlier can be used as follows.

:Iq/string/ or n:Iq	like the I command except that the specified string is inserted into Q-register q. The former contents of Q-register q are lost.
n:L	Equivalent to nLR. Thus TECO moves to the end of the line rather than the beginning.
:S/string/, n:S/string/, or m,n:S/string/	like 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.
:T/string/	types the specified string on the user's terminal.

Numeric Q-Registers

Q-Registers can be used, as mentioned in "Introduction", to hold numeric values. These values can be used in expressions that are arguments to other commands.

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	Return the number stored in Q-register q as the value. Note that Q is not really a command--it is a special symbol. (See "Introduction".) Thus, in the expression 5+Q3 the 5+ is not considered an argument to Q; the result is the sum of Q3 and 5. Note if Q-register q contains text, the length of the text in characters is returned.

INCREMENTING Q-REGISTERS - %

%q	Add 1 to Q-register q and return the new number as the value. Q-register q cannot contain text. Note that %, like Q, is a special symbol, not a command.

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 command lines for execution as macros, or to provide quoted strings for commands that expect them.

EXTRACTING TEXT TO A Q-REGISTER - X (eXtract)

Xq		takes arguments like the T command, but copies the text that T would type 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	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	copies all the text from just to the right of the (n+1)th newline, that is, to the left of the current pointer, to the current pointer into Q-register q. 0X/ copies the beginning of the current line into Q-register /. No ewline characters are put into Q-register /. -Xa puts the previous line and the beginning of the current line into Q-register a.
m,nXq		copies character number (m+1) through character number n into Q-register q.

INSERTING TEXT DIRECTLY INTO A Q-REGISTER - :I (Insert)

:Iq/string/	This command is identical to the normal "I" command except that the text is inserted into Q-register q rather than the buffer. The former contents of Q-register q are deleted. The main text buffer is not affected.
n:Iq	is like :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 illegal 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 command, it is loaded into Q-register ". As an example, SQ", immediately after another search, searches again for the same string. This notation is illegal 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 command 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] command if the stack is empty.

teco

Loops

TECO has the ability to execute a command string repeatedly, much as FORTRAN or PL/I provides do-loops.

BEGINNING A LOOP - < and > (opposite of each other)

<	is equivalent to n< except that n is set to a very large number that is for all practical purposes infinite.
n<	causes TECO to take note of the fact that a loop is beginning. The value of n and the position of the < in the command string are saved.
>	causes execution to return to just after < if the string has not yet been executed n times.
n<...>	this causes the string between the angle brackets to be executed 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 command is used (terminate loop if search failed). The ; command cannot appear outside of a loop.

Goto's

TECO provides the ability to transfer control to a different part of the command string.

GOTO - O (gOto)

causes the TECO command to search the current macro (or, if we are not in a macro, the command line) for the string " string ". If it is found, TECO begins interpreting commands just after the label found. 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 command line typed by the user. Note that although TECO

checked all the way down to the command line typed by the user. Note that although TECO can exit a macro using an O command, it cannot use that command to exit a loop. Only a semicolon (;) can be used to terminate a loop.

Macros

TECO has the ability to execute strings of text (macros) other than those read from the user's terminal. The associated commands are listed below:

EXECUTING A MACRO IN A Q-REGISTER - M (Macro)

Mq — causes the contents of Q-register q to be executed as a command string. Note that if the M command is given any numeric arguments, they are passed to the first command inside the macro. String arguments can be fetched by the :X command.

EXECUTING A MACRO IN A FILE - EM (External Macro)

EM/string/ — is just like the M command except that the command string is found in a file named string.TECO. This file is looked for in three places: 1) the working directory, 2) the user's login directory, 3) the TECO library.

OBTAINING A STRING ARGUMENT TO A MACRO

:Xq — causes TECO to suspend execution of the current macro, return to its caller to fetch a quoted string into Q-register q, and then restore the macro that was being executed. Note that each :X command in a macro fetches another quoted string. Note that the U command(s) should be the first command 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 command from being used within a loop.

2. A macro can modify itself if it is in a Q-register. Note, 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 command, the results of modifying the file are hard to predict: TECO reads the command string directly from the file.

3. When a macro is invoked by the EM command, 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 command in the segment being OQ". (Don't forget to put all the appropriate names on the segment).

4. If an M or EM command is given as the last command in one macro, the command 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 command 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. If no start_up macro is found, the string EI/filename/J is executed, where filename is the first argument to TECO. At the present time, there is a start_up macro in the TECO library. When the start_up macro is called, the first thing on the pushdown list is the number of arguments TECO was called with. The remaining items in the list are the actual string arguments to TECO going from left to right on the command line.

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 combatted. 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 macro writers now 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 command is much more expensive, however. This leads to the practice of creating a macro in a macro library that only loads a Q-register with a useful macro. When the user realizes that he wants the macro, he gives the EM command 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 lower case 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 on a 2741 are good, otherwise other special characters can be used. This leaves the upper case 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 lower case letters on both a 2741 and a Model 37 teletype.

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

TECO 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. An insert merely adds text to the end of the first buffer segment and increases the number of legitimate characters in the first buffer segment. A D or X command merely changes the number of legitimate characters in one of the buffer segments. In

order to move the pointer, a string copy from one buffer segment to the other must be performed. It does not matter to TECO which direction the pointer is moved, although a reverse search is somewhat slower than a forward search, since the PL/I index built-in function can only be used for a forward search.

Any operation that does not move text is less expensive than an operation that does move text, where the cost of the operation that does move text is proportional to the amount of text moved. For the most part, performing input or output is the major cost involved in editing. This cost can be decreased by using more sophisticated commands, such as loops or macros, and performing the same editing operation with fewer interactions. The cost of I/O operations is comparable to a medium length search (5,000 characters).

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, then entering TECO for the first time in a process is somewhat slow since all these segments must be created. TECO 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 > command, the ; command, the " or :´ command, and the O command. Of these, the > command is the fastest since TECO already knows exactly where to transfer it. The ;, ", and :´ commands are next, since they merely search from where they are forward. Although the > command and the ; command cannot change macro levels, the ", and :´ commands can. This adds a small expense. The ;, ", and :´ commands all have to check so that a ; command completely skips over another nested loop and looks beyond it for a >.

Similarly the " transfer skips over nested if statements, as does the :' command. Usually the matching ' or > is not far from the transfer, so this only causes a short search. 0 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 command line and gives an error. Although this is the most expensive transfer, its cost is proportional to the distance of the label from the beginning of the macro.

Conditionals

TECO has the ability to conditionally execute strings. The " command corresponds to the PL/I statement "if ... then do;". The ' command corresponds to the PL/I statement "end;". " and ' are matched much like (and) and can be nested. The letter following the " determines what test is made.

NUMERIC COMPARISONS - "E (Equals), "N (Not equal), "G (Greater than),"L (Less than)

m,n"E	if m=n, then execution continues; otherwise execution skips to just after the corresponding '.
n"E	identical to n,0"E
m,n"N	like m,n"E except it tests for m^=n
n"N	identical to n,0"N
m,n"G	like m,n"E except it tests for m>n
n"G	identical to n,0"G
m,n"L	like m,n"E except it tests for m<n
n"L	identical to n,0"L

TESTING FOR A SYMBOL CONSTITUENT - "C (symbol Constituent)

n"C	if n is the ASCII code for either a letter, a digit, or one of the characters ., _, or $; then execution continues. Otherwise, execution skips to the corresponding '.

TERMINATING A CONDITIONAL DO - ´ (matches ")

´ is ignored when executed in normal execution. It is used to close a conditional statement.

:´ This command causes a transfer to the next ´, just as a 1"e does. Since this command 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, the then expression is terminated by the :´ command and the else expression is terminated by the ´ command.

Reading Input from the User´s Terminal - VW (V then Wait for input)

VW does a V command (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 command. Multics escape/kill processing is not effected because only one character is read at a time.

:VWq does a V command 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 command processor for execution.

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 command. n must be specified. (Note 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 command 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 command. 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.

Null Command - W

W	does nothing. It is most useful for throwing away unneeded numeric arguments.
new_line	has the same effect as W.
$	has the same effect as W.

EXAMPLES OF MACROS

A Writing Macro

This macro writes out the entire buffer into a file whose name is in Q-register *. The file being edited can be changed merely 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 zeroes 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*J	reads the new file into the buffer and put the pointer at the beginning of the buffer.

A Start Up Macro

This macro only uses the first 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*| q1"n ]* eiq*j '
```

]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 writing macro given in the above example.
q1"n	if the contents of Q-register 1 are not zero, then execute the following statements; otherwise transfer to the ' that ends the macro.
]*	pops the first argument to TECO off the pushdown list and into Q-register *.
eiq*j	reads the file into the buffer and moves the pointer to the beginning of the buffer.
'	this point is transferred to if there are no arguments given to TECO.

A 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.
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.

teco

A TECO SUMMARY

NAME	MNEMONIC	USE AND EXPLANATION
a	Ascii	nA The value of the command is the ASCII code for the (.+n)th character in the buffer.
b	Beginning	B The value of this symbol is always zero.
c	Characters	nC Moves the pointer n characters to the right. If n is omitted, 1 is assumed.
d	Delete	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.
ec	External Command	EC/command/ passes the string to the Multics command processor.
ei	External Input	EI/file/ reads the file into the buffer to the left of the current pointer.
em	External Macro	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	External Output	EO/file_name/ writes out the entire buffer into the file specified.

NAME	MNEMONIC	USE AND EXPLANATION
		+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	External Quit	EQ TECO returns to its caller after zeroing out all Q-registers.
g	Get Q-register	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	wHole	H This symbol is equivalent to 0,Z. It is the only symbol that has two values.
i	Insert	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	Jump	nJ moves the pointer to the right of the nth character in the buffer. If n is omitted, 0 is assumed.
k	Kill buffer	K deletes the rest of the current line from the buffer.

NAME	MNEMONIC	USE AND EXPLANATION
		+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	Lines	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 lst (n+1)th line.
m	Macro	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 command in the macro. string1 through stringn are string arguments to the macro that can be retrieved with the :X command. EM also takes all these arguments.
o	gOto	o/label/ transfers control to just after label in the current macro, its caller, etc., or the command string.

NAME	MNEMONIC	USE AND EXPLANATION
q	Q-register	Qq the value of this command 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 command can also replace any quoted string if Q-register q contains text. The contents of the Q-register are used as the quoted string. (See also sections 3.1.2.2 and 3.1.4.)
r	Reverse	R moves the pointer one character to the left. nR moves the pointer n characters to the left.
s	Search	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. (0 or-m),-nS/string/ only searches from the current pointer to the beginning of the last mth line. :s takes arguments in all the ways S

NAME	MNEMONIC	USE AND EXPLANATION
		does, except that if S does not find the string, it types out an error message and returns to TECO command level. :S does not. Instead, :S has the value -1 if the search succeeds and 0 if the search fails.
t	Type	T types out the rest of the current line of the terminal. +nT types out the buffer from the current pointer to the beginning of the next nth line. (0 or-n)T types out the buffer from the beginning of the last nth line to the current pointer. m,nT types out the (m+1)th through the nth characters of the buffer.
:t		:T/string/ types out the quoted string on the terminal.
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. This may be used inside a macro to get the numeric arguments to the macro.
vw	VieW	VW when this command is executed, one character is read from the terminal. The ASCII code for the character read is the value of the VW command.
:vw		:VWq reads in an entire line from the

teco

NAME	MNEMONIC	USE AND EXPLANATION
		terminal and puts it into Q-register q. The newline is the last character in the register.
w	Wipe out	W this command is used for throwing away unwanted numeric arguments.
x	eXtract from buffer	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. 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	Last Letter	Z this symbol's value is the total number of characters in the buffer. ZJ moves the pointer to the right of the last character in the buffer.
%	Increment	%q if Q-register q contains a numeric value, this command increments the register by1. The value of the command is the new value of the Q-register.
$		$ throws away its arguments and does nothing.

teco

NAME	MNEMONIC	USE AND EXPLANATION
newline		newline throws away its arguments and does nothing.
?	What's happening?	? turns tracing on.
??		?? turns tracing off.
\	Number-character	\ the value of this command 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.
[	Push	[q pushes the contents of Q-register q onto the pushdown list.
]	Pop	]q pops the top element off the pushdown list and into Q-register q.
<	Begin a loop	< this marks the place in the command string that is transferred to by the > command. This loop can only be exited by the ; command.
>	End a loop	> transfers control to just after the last < command executed and decrements the loop count. If we have looped enough times, this command does nothing. Nested loops are allowed.
;	Terminate if positive	; if the last :S command was

NAME	MNEMONIC	USE AND EXPLANATION
		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 > command and exits the present loop; otherwise does nothing.
"C	If Symbol Constituent	n"C if n is the ASCII code for either a letter, a digit, .,_, or $; does nothing. Otherwise, transfers to just after the next '.
"e	If Equal	m,n"E if m=n, then does nothing; otherwise goes to just after the next '. n"E if n=0.
g	If Greater	m,n"G if m>n. n"G if n>0.
"l	If Less than	m,n"L if m<n. n"L if n<0.
"n	If Not Equal	m,n"N if m^=n. n"N if n^=0.
'	Matches "	' marks the location a " command can transfer to. If executed, as a command, it does nothing.
:"		:" marks the location a " command can transfer to. If executed as a command, it transfers to just after

NAME	MNEMONIC	USE AND EXPLANATION
		the next '. If statements may be nested, but " characters in the command string are only matched with one ' character.
o	gOto	o/label/ transfers control to just after label.
	Label	label this entire construct is ignored if it is executed.
.	Pointer	. the value of this command is the value of the current pointer.
=	Equals	= types out a newline.
		n= types out n on the console followed by a newline.
		m,n= types out m followed by a space, followed by n, followed by a newline.

teco_error

Name: teco_error

The teco_error subroutine, which is most frequently used as a 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)

teco_ssd

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.

test_archive

Name: test_archive, ta

The test_archive command is a library maintenance tool that checks an archive segment for archive format errors or 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).

test_tape

Name: test_tape

The test_tape command invokes a Device Interface Module (DIM) whose only function is to write and read a tape to determine its reliability. It reports all hardware detected errors (parity, read after write, etc.) and the software detected incorrect data condition. The test_tape command can also be used to do compatibility checks between drives (i.e., write a tape on one, read it on another).

Usage

test_tape mode pattern count volumeid density

where:

1. mode — is the mode of the test. If the character -w is present in the mode string, then the tape is written. If -r is present, then the tape is read. Either -wr or -rw are also legal, but note that the tape is always written first and read second.

2. pattern — specifies the word of octal data to be used to fill the buffers. It should take the form -ptrn followed by a space and up to 12 octal digits. If less than 12 digits are given, the field is padded on the left with zeros.

3. count — indicates the number of writes to be performed in creating the tape. Each write operation creates three 272-word physical records.

Notes

If no arguments are specified, the tape is both written and read, with a data pattern of 525252525252, for the entire length of the tape.

The test_tape command senses the end of tape mark (EOT) and stops even if the record count has not been exhausted.

This command has four control arguments that control what happens when an error is encountered. They are:

print, noprint — controls printing of an error when it occurs. Statistics are kept of all errors. The default is print.

halt, nohalt	tells test_tape to terminate when an error is encountered. The default is nohalt.
retry, noretry	specifies that if the I/O is in error that it should be retried. test_tape retries 25 times before giving up. Statistics are kept of all recoverable errors. The default is retry.
sleep, nosleep	specifies that test_tape should sleep when a nonrecoverable error is found. This allows the operator to mark the tape. The program pauses for two minutes and then continues. The default is nosleep.

These control arguments can be changed by using the change entry point of test_tape. The change control argument accepts up to 99 arguments, each of which can be one of the above or the string options. The latter causes test_tape to print the control arguments in effect.

Examples

```
test_tape$change sleep noprint options
```

results in the sleep control argument being enabled, the print control argument being disabled, and a list of the current control arguments being printed.

The change entry offers another advantage in that the user can quit out of an executing test and change the current control arguments. In the example the user has decided that he is getting too much output and wishes to stop the printing. He presses the QUIT button and types:

```
test_tape$change noprint; start
```

The change takes effect immediately.

unassign_device

Name: unassign_device

The unassign_device command causes the specified I/O device to become unassigned using the facilities of the Hardcore Ring I/O Assignment Manager (IOAM). If the caller has the ability to call hphcs_ entries, this command works for a device attached to any process; otherwise it works only for devices attached to the calling process.

This command bypasses the attach table of the relevant process, thus leaving it in an inconsistent state, with possible dire consequences for the process if it should use the switch again. Normally, a user should never need to use this. The io_call command should be used to detach a device from command level.

Usage

unassign_device ionames

where ionames are the names of I/O devices to be freed.

Name: value

The value command returns a character string associated with a named item in a user symbol table segment. This enables administrative exec_com segments to reference variables.

Command entry points are provided to set entries in the symbol table and to list the symbol table.

Entry: value

This entry is meant to be called as an active function. It looks up its argument in the segment value_seg in the user's current working directory, and returns the value associated with the argument. If no value is found, the string undefined is returned.

Usage

[value ident]

Note

This call returns the value associated with ident to the command processor.

Entry: value$set

This command entry can set entries in value_seg or delete entries from value_seg.

Usage

value$set ident val

Sets identifier ident to have the value val.

value$set ident

Removes the definition of ident from value_seg.

Note

Both ident and val can be up to 32 characters long.

Entry: value$dump

This command entry lists particular values in value_seg or lists the whole segment.

Usage

value$dump ident

lists the value of ident.

value$dump

lists the contents of value_seg.

Entry: value$set_seg

This command entry causes the value command and active function to use a value_seg given by pathname.

Usage

value$set_seg path

where path causes the value command to use the segment identified by path instead of value_seg for the rest of the process or until another call to value$set_seg.

Note

For this entry only, if the symbol table segment does not exist, it is created.

Example:

```
value$set it output_file
value$set name1  """John Smith"""

dprint -he [value name1] [value it]
```

SECTION II

SUBROUTINES

This section consists only of subroutine descriptions; no design motivation, implementation description, or data structure description is included except what is needed to describe the use of the subroutine as a tool. If a more detailed description is desired, check the Index PLM, Order No. AN50, to find out which PLM contains more material on a specific subroutine.

This section, subroutines, is arranged alphabetically. For programs that consist of a set of several related subroutines, the set may be documented within one subroutine description. Also, the set is arranged according to the order in which the subroutines are used rather than alphabetically.

Name: abbrev_

The abbrev_ subroutine provides a means of 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 is provided for expanding abbreviations or command lines.

Entry: abbrev_$abbrev_

This entry is used to expand and execute a command line. The command line passed to abbrev_ can be an abbrev request line (as in the abbrev command), which is treated as if the abbrev command itself had intercepted the call. This means that the user can add and delete abbreviations as well as change the other control modes of abbrev. The abbrev command need not have been invoked in the process for abbrev_ to be called.

Usage

```
declare abbrev_$abbrev_ entry (ptr, fixed bin, fixed bin);

call abbrev_$abbrev_ (ptr, n, code);
```

where:

1. ptr is a pointer to an aligned character string that is to be interpreted as a command line. The character string can be an abbrev request line. (Input)

2. n is the number of characters in the above mentioned string. (Input)

3. code is a standard Multics status code returned from the procedure command_processor_. (Output)

Note

If the character string passed to abbrev_ is not an abbrev request line, the string is expanded and the expanded version is passed on to the Multics command processor for execution.

Entry: abbrev_$expanded_line

This entry is given a pointer to a character string and returns through another pointer an expanded version of the string. The string can contain abbrev break characters (see the abbrev command description in the MPM Commands and Active

Functions, Order No. AG92) that are treated exactly as in the abbrev command.

Usage

```
declare abbrev_$expanded_line entry (ptr, fixed bin, ptr,
     fixed bin, ptr, fixed bin);

call abbrev_$expanded_line (in_ptr, inl, vout_ptr, voutl,
     out_ptr, outl);
```

where:

1. in_ptr — is a pointer to the aligned character string to be expanded. (Input)
2. inl — is the number of characters in the input character string. (Input)
3. vout_ptr — is a pointer to a data area where the output (expanded) character string can be placed. (Input)
4. voutl — is the number of characters that can be placed in the area to which vout_ptr points. (Input)
5. out_ptr — points to the expanded string. (Output)
6. outl — is the number of characters in the expanded string. (Output)

Notes

If the length of the expanded string exceeds the length of the area provided (pointed to by vout_ptr), abbrev_$expanded_line allocates in system_free_n_ as much storage as is necessary to hold the entire expanded line. It is the user's responsibility to free this storage when it is no longer fYYyYr.

The vout_ptr pointer should not point to the same string as in_ptr since expansion is done directly into the area to which vout_ptr points.

Entry: abbrev_$set_cp_

This entry sets up a different command processor to be called by abbrev_ after a command line is expanded. Its argument is an entry. If the first pointer in the entry is null, abbrev_ calls command_processor_.

Usage

```
declare abbrev_$set_cp_ entry (entry);

call abbrev_$set_cp_ (cp_entry);
```

where cp_entry is the entry of the desired command processor.

Examples

The code:

```
chars = ".a ab1 " || string;
call abbrev_(addr(chars), length(chars), code);
```

sets up ab1 as an abbreviation for the character string stored in string.

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), outp, outl);
```

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 outp points to the expanded version and outl 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 returns the next string of characters delimited by blanks or tabs from the line typed by the user. If the line buffer is empty, ask_ formats and types out a prompting message and reads a line from user_input.

Usage

```
declare ask_ entry options (variable);

call ask_ (ctl, ans, ioa_args...);
```

where:

1. ctl — is a control string (char(*)) in the same format as that used by ioa_. (Input)
2. ans — is the return value (char(*)). (Output)
3. ioa_args — are any number of arguments to be converted according to ctl. (Input)

Entry: ask_$ask_clr

This entry clears the internal line buffer. Because the buffer is internal static, one program's input can accidentally be passed to another unless the second begins with a call to this entry. If a value typed by the user is incorrect and if the program wishes to ask for the line to be retyped, ask_$ask_clr can also be called.

Usage

```
declare ask_$ask_clr entry;

call ask_$ask_clr;
```

There are no arguments.

Entry: ask_$ask_int

This entry works the same as ask_ 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" non-numeric. 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 is as above. (Input)
2. int is the return value (fixed bin). (Output)
3. ioa_args are as above. (Input)

Entry: ask_$ask_flo

This works like ask_$ask_int 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 is as above. (Input)
2. flo is the return value (float bin). (Output)
3. ioa_args are as above. (Input)

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 — is as above. (Input)
2. line — is the return value (char(*)). (Output)
3. ioa_args — are as above. (Input)

Entry: ask_$ask_c

This entry tests to determine if there is anything left on the line. If so, it returns the next symbol, as in ask_, and sets a flag to nonzero. Otherwise, it sets the flag to zero and returns.

Usage

```
declare ask_$ask_c entry (char(*), fixed bin);

call ask_$ask_c (ans, flag);
```

where:

1. ans — is the next symbol, if any. (Output)
2. flag — =1 if the symbol is returned;
   =0 if there is no symbol. (Output)

Entry: ask_$ask_cint

This is a conditional entry for integers. If an integer is available on the line, it is returned and flag is set to 1. If the line is empty, 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 flag is set to -1.

Usage

```
declare ask_$ask_cint entry (fixed bin, fixed bin);

call ask_$ask_cint (int, flag);
```

where:

1. int — is the returned value, if any. (Output)

2. flag — =1 if int is returned;
=0 if the line is empty;
=-1 if there is no number. (Output)

Entry: ask_$ask_cflo

This entry works like ask_$ask_cint 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 — is the returned value, if any. (Output)

2. flag — =0 if the line is empty;
=1 if the value is returned;
=-1 if it is not a number. (Output)

Entry: ask_$ask_cline

This entry 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 — is the returned line, if any. (Output)

2. flag — =1 if the line is returned;
=0 if the line is empty. (Output)

Entry: ask_$ask_n

This entry scans the line and returns the next symbol without changing the line pointer. A call to ask_ later returns the same value.

Usage

```
declare ask_$ask_n entry (char(*), fixed bin);

call ask_$ask_n (ans, flag);
```

where:

1. ans — is the returned symbol, if any. (Output)

2. flag — =0 if the line is empty;
=1 if the symbol is returned. (Output)

Entry: ask_$ask_nint

This entry 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, as 1 if successful, and as 0 if the line is empty.

Usage

```
declare ask_$ask_nint entry (fixed bin, fixed bin);

call ask_$ask_nint (int, flag);
```

Same arguments as in ask_$ask_cint.

Entry: ask_$ask_nflo

This entry scans the line for floating point numbers.

Usage

```
declare ask_$ask_nflo entry (float bin, fixed bin);

call ask_$ask_nflo (flo, flag);
```

Same arguments as in ask_$ask_cflo.

Entry: ask_$ask_nline

This entry initiates a scan of the rest of the line.

Usage

```
declare ask_$ask_nline entry (char(*), fixed bin);

call ask_$ask_nline (line, flag);
```

Same arguments as ask_$ask_cline.

Entry: ask_$ask_setline

This entry sets the internal static buffer in ask_ 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 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 — is a control string (char(*)) similar to that typed by ioa_. (Input)

2. ioa_args — are any number of arguments to be converted according to ctl. (Input)

Name: bk_arg_reader_

The bk_arg_reader_ subroutine handles most argument reading for backup and the reloader. It is called from all of the backup command programs, start_dump, backup_dump, reload and backup_load. The main function of bk_arg_reader_ is to set flags and to enter data, such as pathnames, in the backup external static segment bk_ss_.

Usage

```
declare bk_arg_reader_ entry (fixed bin, ptr,
     fixed bin(35));

call bk_arg_reader_ (iac, ialp, ocode);
```

where:

1. iac is the index of the first argument to be processed. (Input)

2. ialp is a pointer to an argument list. (Input)

3. ocode is the standard status code (see "Notes"). (Output)

Entry: bk_arg_reader_$dump_arg_reader

This entry is called by start_dump, catchup_dump, complete_dump, and backup_dump to handle input arguments.

Usage

```
declare bk_arg_reader_$dump_arg_reader entry (fixed bin,
     ptr, fixed bin(35));

call bk_arg_reader_$dump_arg_reader (iac, ialp, ocode);
```

Same arguments as above.

Entry: bk_arg_reader_$reload_arg_reader

This entry is called by each of the reload commands (reload, iload, retrieve, and backup_load) to handle input arguments.

Usage

```
declare bk_arg_reader_$reload_arg_reader entry (fixed bin,
     ptr, fixed bin(35));

call bk_arg_reader_$reload_arg_reader (iap, ialp, ocode);
```

Same arguments as above.

Notes

The bk_arg_reader_ subroutine handles three classes of arguments: those common to both reloading and dumping, those related only to reloading, and those related only to dumping. In addition, there are arguments used only by a particular entry or by all but one entry. Finally, there are control arguments that merely direct the handling of an immediately following argument.

Control arguments and a few other arguments should be preceded by a minus sign (-). Arguments that immediately follow control arguments cannot be preceded by a minus sign (-).

Arguments Common to All Backup Commands

-all causes switches to be set indicating that no date testing is to be done except to ensure that reloading older copies of directories does not overwrite newer ones. Default varies with the particular entry called.

-control indicates that the following argument is a dump control file name if entry was through the dump_arg_reader entry or a retrieval control file name if entry was through the reload_arg_reader entry.

-debug disables those hphcs_ calls that set quotas and transparency switches.

-nodebug enables hphcs_ calls to set quotas and the transparency switches. This is the default.

-err_offl causes error messages to be output into a file rather than online. The name of the error file is given on the first error encountered. This is the default.

-err_onl causes error messages to be output onto the user's console.

-map causes a list of the segments and directories processed to be output into a file. This is the default.

-nomap inhibits listing of the names of processed segments and directories and turns on the tape switch if entry was through dump_arg_reader (see -tape).

-operator indicates that the next argument is the operator's (or user's) name or initials up to 16 characters in length.

->string if the user originally invoked reload$retrieve or backup_load, then >string must be the pathname of a retrieval control file. If entry to bk_arg_reader_ was through the dump_arg_reader entry then >string must be the pathname of a segment to be dumped or a directory where dumping of a subtree is to begin.

-x.... assumes this argument to be a date, where x is a decimal integer; an attempt is made to convert it to a storage system date using convert_date_to_binary_. (See date information shown below under "Otherwise Unrecognized Arguments".)

Arguments Recognized by the Dumper

-contin continues by starting incremental backup after the catchup pass. This is the default.

-nocontin ends the dump after the catchup pass.

-dtd causes each segment to be tested and dumped only if the segment or its branch has been modified since the last time it was dumped.

-hold causes the current dump tape or tapes to remain mounted and inhibits rewinding after the current dump cycle is completed. This is set by the dumper.

-nohold causes the dump tape or tapes to be rewound and unloaded at the end of the current dump cycle. This is the default.

-only indicates that only the requested segment or directory and its branch are to be dumped. (See ->string above.)

-sweep indicates that the whole subtree beginning with the given directory is to be dumped, subject to the dtd and date criteria if they have been invoked. (See ->string -dtd and -x... above). This is the default.

-output causes dump information to be output onto tape if the tape switch is on. This is the default.

-nooutput inhibits writing on tape even if tape switch is on. This is used for a dumper test run or debugging.

-restart indicates that the next argument is the pathname of a segment or directory where dumping is to be restarted. Use of this feature assumes that there is a dump control file. Its normal usage is in conjunction with catchup_dump or complete_dump where a failure of some nature has occurred.

-tape allows writing onto a dump output tape. This is the default.

-notape inhibits writing onto a dump output tape. This also causes a map to be output even if it was previously inhibited. (See -map above).

-tapes indicates that the next argument is the number of output tape copies to be made. Only 1 or 2 are allowed and the default is one.

-1tape sets the number of tape copies to 1 as an alternative to the -tapes argument.

-2tapes sets the number of tape copies to 1 as an alternative to the -tapes argument.

-wakeup indicates that the next argument is the wake-up interval between dump cycles given in minutes.

Arguments Recognized by the Reloader

-first prevents searching a tape for additional copies of a requested segment or subtree after the first copy has been retrieved.

-last indicates that the last copy of a given segment or subtree on a tape or set of tapes is to be retrieved. This is the default.

-quota causes quotas to be reset during reload and suspended during retrieval. This is the default.

-noquota inhibits resetting and suspension of quotas.

-reload enables actual reloading of segments into the hierarchy. This is the default.

-noreload inhibits actual reloading of segments into the hierarchy. This is a debugging tool and when map is enabled (see -map) it also causes the names and access control lists (ACLs) that would have been reloaded to be put into the map.

-qcheck disables suspension of quota checking.

-noqcheck enables suspension of quota checking. This is the default.

-trim enables deletion of all entries in a directory not found in the copy of that directory being reloaded. (Entries deleted since an earlier version of the directory existed are deleted when a later version is reloaded.) This is the default for reload and iload.

-notrim inhibits deletion of entries in a directory. Entries can only be added or modified. This is the default for retrieve.

Otherwise Unrecognized Arguments

-..... an attempt is made to translate otherwise unrecognized arguments into a storage system date using convert_date_to_binary_. Dates have special meaning for various entries and functions:

To the dumper a date means that only segments that have been modified or whose entries have been modified since the date given are to be dumped.

To the retriever a date means load the first copy of a segment or subtree dumped after the date given. Thus if several copies of a segment exist on a single dump tape, the user can indicate by this means which copy is to be retrieved.

bk_arg_reader_

The convert_date_to_binary_ subroutine is used to convert dates; any format of date acceptable to it is acceptable to bk_arg_reader_. An easy format to remember is MM/DD/YY tttt.t.

An example is 10/23/73 1405.6.

Name: canonicalizer_

The canonicalizer_ subroutine canonicalizes an input string according to standard Multics format. The string is placed into the caller's buffer in canonical form if its length is not greater than that of the buffer. If the canonical string does not fit into the caller's buffer, the error code error_table_$area_too_small is returned. The partial string is not returned.

Usage

```
declare canonicalizer_ entry (ptr, fixed bin, ptr, fixed bin,
     fixed bin, bit(4) aligned, fixed bin);

call canonicalizer= (inptr, inlength, outptr, maxlength,
     outlength, flags, code);
```

where:

1. inptr — is a pointer to the string to be canonicalized. (Input)

2. inlength — is the length of the input string. (Input)

3. outptr — is a pointer to the buffer in which the canonicalized string is to be placed. (Input)

4. maxlength — is the character length of the output buffer. (Input)

5. outlength — is the returned string length. (Output)

6. flags — is a set of bit flags, each controlling a canonicalization function. These bits are, from the left:

   Canonicalization control. This bit must be on for the string to be canonicalized.

   Erase/Kill control. If this bit is on, erase and kill characters in the string are processed with their erase and kill functions.

   Escape control. If this bit is on, escape characters in the string are processed with their escape functions.

canonicalizer_

TTY33 convention indicator. If this bit is on, special escape conventions for the TTY33-type character set are processed. The escape control bit must also be on to allow processing of escape characters if this mode is desired. (Input)

7. code — is a standard Multics status code. (Output)

Note

This routine does not alter the cases of letters, regardless of the value of the TTY33 convention indicator. It is assumed that case processing has already taken place, but without deleting the escape characters responsible for capitalization, since their initial presence is essential to correct canonicalization. In this mode, single escape characters preceding capital letters are simply deleted.

Name: command_processor_

The command_processor_ subroutine is the Multics command language interpreter. It parses a command line and invokes the specified command or commands. It should never be called directly but rather through the subroutine cu_$cp.

Usage

```
declare command_processor_ entry (ptr, fixed bin(17), fixed
     bin(17));

call command_processor_ (inp, inl, code);
```

where:

1. inp — is a pointer to the command line, which must be an aligned character string. (Input)

2. inl — is the length of the command line. (Input)

3. code — is a standard Multics status code that equals 0 if there are no errors in the command line; equals 100 if a null line was typed. code is nonzero if there is an error in the line. (Output)

Entry: command_processor_$return_val

This entry is used by the bracket handling routine, proc_brackets_, to evaluate a command line inside square brackets and to return a string that is the concatenation of all the values returned by all the commands in the command line. (Each value is separated from the next by a blank.)

Usage

```
declare command_processor_$return_val entry (ptr, fixed
     bin(17), fixed bin(17), char(*) varying, char(*)
     varying, fixed bin(17));

call command_processor_$return_val (inp, inl, flag,
     ret_string, workspace, code);
```

where:

1. inp	is a pointer to the command line. (Input)
2. inl	is the length of the command line. (Input)
3. flag	equals 0 if brackets are to be treated as special characters; otherwise equals 1. (Input)
4. ret_string	is the value to be returned. (Output)
5. workspace	is for intermediate storage. (Input)
6. code	is a standard Multics status code. (Output)

Entry: command_processor_$ignore_brackets

This entry is used by the bracket processing routine proc_brackets_ to process a command line in which brackets are not to be treated as special characters.

Usage

```
declare command_processor_$ignore_brackets entry (ptr,
     fixed bin(17), fixed bin(17));

call command_processor_$ignore_brackets (inp, inl, code);
```

where:

1. inp	is a pointer to the command line. (Input)
2. inl	is the length of the command line. (Input)
3. code	is a standard Multics status code. (Output)

Entry: command_processor_$set_line

This entry is used to set the maximum expanded command line size.

Usage

```
declare command_processor_$set_line entry (fixed bin (17));

call command_processor_$set_line (newsize);
```

where newsize is the maximum expanded command line size. (Input)

Entry: command_processor_$get_line

This entry is used to obtain the value of the maximum expanded command line size.

Usage

```
declare command_processor_$get_line entry (fixed bin(17));

call command_processor_$get_line (size);
```

where size is the current maximum expanded command line size. (Output)

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_ (dn, en, ec);
```

where:

1. dn is the directory name in which the segment to be modified resides. (Input)

2. en is the entryname of the segment. (Input)

3. ec is a standard Multics status code. (Output)

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, but following any percent-semicolon at the very beginning.

If no notice segments are found for a 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 T.copyright and T.copyright_delete where T is the language type.

copyright_notice_

Usage

```
declare copyright_notice_$set_suffix entry (char(*));

call copyright_notice_$set_suffix (x);
```

where x is the new suffix. (Input)

Entry: copyright_notice_$test

This entry sets the directory searched for copyright notice segments. The default is >ldd>include.

Usage

```
declare copyright_notice_$test entry (char(*));

call copyright_notice_$test (d);
```

where d is the directory to be searched for copyright notices. (Input)

Name: create_ips_mask_

The create_ips_mask_ subroutine returns a bit string that can be used to mask specified interrupts.

Usage

```
declare create_ips_mask_ entry (ptr, fixed bin, bit(36)
     aligned);

call create_ips_mask (p, lng, mask);
```

where:

1. p is a pointer to an array of ips names that are char(32) aligned. (Input)

2. lng is the number of elements in the above array. (Input)

3. mask is a mask that masks all of the ips signals named in the array pointed to by p when passed to the appropriate ring 0 entry point. (Output)

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.

Name: cu_

The cu_ subroutine description discusses only those entry points in cu_ that are of interest mainly to system programmers.

Entry: cu_$grow_stack_frame

This entry allows its caller to allocate temporary storage by extending the caller's current stack frame.

Usage

```
declare cu_$grow_stack_frame entry (fixed bin, ptr,
     fixed bin);

call cu_$grow_stack_frame (len, ptr, code);
```

where:

1. len is the length (in words) by which the caller's stack frame is to be extended. The standard Multics call, save, and return discipline requires that stack frames begin on mod 16 word boundaries. Therefore, if len is not a mod 16 number, the stack frame is grown by the next mod 16 quantity greater than len. (Input)

2. ptr is a pointer to the first location of len words allocated in the caller's stack frame. (Output)

3. code is a standard Multics status code. Zero indicates that the call is successful; nonzero indicates that the call is in error (e.g., len is too big). (Output)

Entry: cu_$shrink_stack_frame

The shrink_stack_frame entry allows its caller to deallocate temporary storage by reducing the caller's current stack frame.

Usage

```
declare cu_$shrink_stack_frame entry (ptr, fixed bin);

call cu_$shrink_stack_frame (ptr, code);
```

where:

1. ptr is a pointer to the first word of the storage to be deallocated. The stack frame from the word indicated by ptr to the end of the frame is deallocated. ptr must point to a mod 16 word boundary. (Input)

2. code is a standard Multics status code. When code is nonzero, the call is in error. (Output)

Entry: cu_$caller_ptr

This entry allows a routine to obtain a pointer to its caller. The pointer returned points to the instruction within the text section after the instruction that called out.

Usage

```
declare cu_$caller_ptr entry (ptr);

call cu_$caller_ptr (point);
```

where point is the pointer to the text section of the caller. If null, the invoker of cu_ has no caller. (Output)

Entry: cu_$set_ready_proc

This entry establishes the procedure called by cu_$ready_proc.

Usage

```
declare cu_$set_ready_proc entry (entry);

call cu_$set_ready_proc (ready_proc_ptr);
```

where ready_proc_ptr is the entry to be used as a ready procedure. (Input)

Entry: cu_$set_ready_mode

This entry is called to set the internal static ready flags that are passed to the ready procedure by cu_$ready_proc if it is called with no arguments.

Usage

```
declare cu_$set_ready_mode (1, 2 bit(1) unaligned,
     2 bit(35) unaligned);

call cu_$set_ready_mode (flags);
```

where flags is the structure defined in the description of cu_$ready_proc. (Input)

Entry: cu_$set_ready_mode

This entry returns the internal static ready flags.

Usage

```
declare cu_$set_ready_mode (1, 2 bit(1) unaligned,
     2 bit(35) unaligned);

call cu_$set_ready_mode (flags);
```

where flags is the structure defined in the description of cu_$ready_proc. (Output)

Entry: cu$get_ready_proc

This entry returns a pointer to the ready procedure.

Usage

```
declare cu_$get_ready_proc entry (ptr);

call cu_$get_ready_proc (ready_proc_ptr);
```

where ready_proc_ptr is a pointer to the entry of the current ready procedure. (Output)

Entry: cu_$ready_proc

This entry is called after each command line is processed. It calls print_ready_message_$print_ready_message unless the user has previously set up his own ready_message procedure via a call to cu_$set_ready_proc. In either case, the argument passed to the procedure called is the structure flags. If cu_$ready_proc is called with no arguments, then an internal set of flags is passed to the procedure instead. The internal static flags may be changed via cu_$set_ready_mode and obtained via cu_$get_ready_mode.

Usage

```
declare cu_$ready_proc entry options (variable);

call cu_$ready_proc (flags);

call cu_$ready_proc ();
```

where flags is the following structure. (Input)

```
declare 1 flags aligned,
          2 ready_sw bit(1) unaligned,
          2 pad bit(35) unaligned;
```

If ready_sw = "1"b then the procedure called is to print the ready message.

Notes

The following cu_ entry points can be found in the MPM Subroutines, Order No. AG93:

cu_$af_arg_ptr	cu_$get_cl
cu_$af_arg_count	cu_$get_cp
cu_$arg_count	cu_$level_get
cu_$arg_list_ptr	cu_$level_set
cu_$arg_ptr	cu_$ptr_call
cu_$arg_ptr_rel	cu_$set_cl
cu_$cl	cu_$set_cp
cu_$cp	cu_$stack_frame_ptr
cu_$gen_call	cu_$stack_frame_size

datebin_

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.

The arguments listed below are common to all entry points. 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.

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_

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:

1. clock is as above. (Input)

2-10. are as above. (Output)

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 entry (fixed bin(71), fixed bin);

call datebin_$shift (clock, s);
```

where:

1. clock is as above. (Input)

2. s is as above. (Output)

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:

1. clock is as above. (Input)

2-4. are as above. (Output)

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:

1. clock is as above. (Input)

2. wkday is as above. (Output)

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:

1. clock is as above. (Input)

2. dayr is as above. (Output)

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:

1-6. are as above. (Input)

7. clock is as above. (Output)

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:

1. absda is as above. (Input)

2. clock is as above. (Output)

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);
```

where:

1. yr is as above. (Input)

2. datofirst is as above. (Output)

Entry: datebin_$dayr_mo

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:

1-3. are as above. (Input)

4. dayr is as above. (Output)

Entry: datebin_$clockathr

This entry 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:

1. zz — is the desired hour and minute, expressed as hhmm in decimal (e.g. 1351). (Input)
2. clock — is as above. (Output)

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 as above. (Output)

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 as above. (Output)

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:

1. oldclock is as above. (Input)

2. clock is as above. (Output)

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:

1. oldclock is as above. (Input)

2. clock is as above. (Output)

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.

datebin_

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:

1. clock is as above. (Input)
2. newclock is the time the shift changes next after clock. (Output)
3. shift is the current shift at time clock. (Output)
4. newshift is the shift which begins at time newclock. (Ouput)

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.

Usage

```
declare decode_definition_ entry (ptr, ptr) returns
    (bit(1) aligned);

eof  = decode_definition_ (def_pointer, structure_pointer);
```

where:

1. def_pointer — is a pointer to the selected definition. This pointer is extracted 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 decode_definition_$init has been called, in which case the initial pointer can be a pointer to the beginning of the definition section (as returned by object_info_). (Input)

2. structure_pointer — is a pointer to the provided structure in which decode_definition_ returns the desired information. (See "Notes" below.) (Input)

3. eof — is a binary indicator that is "1"b if the current invocation of decode_definition_ caused the search to go beyond the end of the definition list. If that is the case, the returned information in the structure is null. (Output)

decode_definition_

Notes

The structure has the following format:

```
declare 1 structure aligned,
          2 next_def ptr,
          2 last_def ptr,
          2 block_ptr ptr,
          2 section char(4) aligned,
          2 offset fixed bin,
          2 entrypoint fixed bin,
          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. last_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, symb or segn.

5. offset — is the offset of the definition within the given section. This is set to 0 if section = 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_$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 has no effect when the decode_definition_$full entry is being used.

Usage

```
declare  decode_definition_$init entry (ptr, fixed bin(24));

call decode_definition_$init (segp, bitcnt);
```

where:

1. segp — is a pointer to the beginning of an object segment (not necessarily with an offset of 0). (Input)

2. bitcnt — is the bit count of the object segment. (Input)

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 can contain up to 256 characters.

Usage

```
declare decode_definition_$full entry (ptr, ptr, ptr)
     returns (bit(1) aligned);

eof = decode_definition_$full (def_pointer,
     structure_pointer, oi_pointer);
```

where:

1. def_pointer — is a pointer to the selected definition and is extracted from previously returned information. The initial pointer with which decode_definition_$full can be called is a pointer to the base of the definition section of the object segment. (Input)

2. structure_pointer — is a pointer to the provided structure into which decode_definition_$full

returns the desired information. (See "Notes" below.) (Input)

3. oi_pointer — is a pointer to the structure returned by any entry point of object_info_. (Input)

4. eof — same as for the decode_definition_ entry.

Notes

The structure has the following format:

```
declare 1 structure aligned,
          2 next_def ptr,
          2 last_def ptr,
          2 block_ptr ptr,
          2 section char(4) aligned,
          2 offset fixed bin,
          2 entrypoint fixed bin,
          2 symbol char(256) aligned,
          2 symbol_lng fixed bin,
          2 flags,
            3 new_format bit(1) unaligned,
            3 ignore bit(1) unaligned,
            3 entrypoint bit(1) unaligned,
            3 retain bit(1) unaligned,
            3 descr_sw bit(1) unaligned,
            3 unused bit(31) unaligned,
          2 n_args fixed bin,
          2 descr_ptr ptr;
```

where:

1-7. — are the same as for decode_definition_.

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 "1"b if the linker should ignore this definition.

11. entrypoint — is "1"b if this definition is for an entry point rather than for a segdef.

12. retain — is "1"b if this definition should be retained.

13. descr_sw — is "1"b if there are descriptors for this definition.

14. unused — is padding.

15. n_args — indicates the number of arguments expected by this entry, if descr_sw = "1"b.

16. descr_ptr — points to an array of 18-bit pointers to the descriptors for the entry, if descr_sw = "1"b.

Entry: decode_definition_$decode_cref

This entry point is used by the utility cross-referencer, cross_ref, to examine object segments possibly contained in archive segments, and to return a pointer to the ASCII Character with Count (ACC) format rather than to the string itself.

Usage

```
declare decode_definition_$decode_cref entry (ptr, ptr,
     bit(1) aligned, ptr);

call decode_definition_$decode_cref (def_pointer,
     structure_pointer, eof, linkbase_ptr);
```

where:

1-2. are as above. (Input)

3. is as above. (Output)

4. linkbase_ptr — is a pointer to the base of the linkage section of the object segment the first time the procedure is called for a given object segment (in which case def_pointer is assumed to point to the base of definitions for the segment), and null thereafter. (Input)

decode_definition_

Note

The structure to which structure_pointer points is the same as for the primary entry, except that:

```
2 symbol char(32) aligned;
```

is replaced by:

```
2 symb_ptr ptr;
```

where symb_ptr is a pointer to the ACC-format representation of the symbol.

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. In order to allow nondefault translator search rules, additional entry points are provided to allow a user to set and get the current search rules for locating include files.

The current default search rules are:

1. Look in the current working directory.

2. Look in the include directory in the user's current project directory. (e.g., For the user Person_id.Project_id, this would be the directory >user_dir_dir>Project_id>include).

3. Look in the directory >library_dir_dir>include.

Entry: find_include_file_$initiate_count

This entry is the interface presented to translators. A translator calls this entry point to invoke a search for a single segment include file using rules previously specified by the user.

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,
     error_code);
```

where:

1. translator — is the name of the translator that is calling this procedure (e.g., pl1, alm). (Input)

2. referencing_ptr — is a pointer into the segment (normally a pointer to the source line) that caused the invocation of this instance of this procedure. (Input)

3. file_name — is the complete entryname of the include file this procedure is to locate (e.g., include.incl.pl1). (Input)

4. bit_count 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. (Output)

5. seg_ptr 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. (Output)

6. error_code is a standard Multics status code. See "Status Codes" below. (Output)

Status Codes

Any of the following status codes may be returned by this entry point:

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 filename argument passed to this procedure. It is the responsibility of the translator to determine if the filename passed to this procedure is also on the include file actually found.

Translator Search Rules

When the entry point find_include_file_$initiate_count is invoked to find an include file, a set of search rules is used to specify where to look for the include file. These search rules should not be confused with those of the dynamic linking mechanism of Multics.

In order to allow the user to manipulate the translator search rules currently in effect, two entry points are provided to set and get the search rules. These entry points utilize the following structure for communication:

```
declare 1 tsr_strct based aligned,
          2 version_number fixed bin(17),
          2 num_valid_rules fixed bin(17),
          2 num_possible_rules fixed bin(17),
          2 rule (1:1 refer (tsr_strct.num_possible_rules))
                 char (168) unaligned;
```

(The based attribute is used here only to allow the refer option to indicate the rationale of the use of num_possible_rules.)

1. version_number — allows the structure declaration to change in possible later versions while retaining upward compatibility. (Input)

2. num_valid_rules — communicates either the number of valid rules currently stored in the rule array, or if there was not enough room in the rule array to store all the current search rules, how many rules are currently in effect. (Input/Output)

3. num_possible_rules — denotes the size of the rule array. (Input)

4. rule — is the array of search rules. Each rule is either a full pathname (indicated by the presence of an initial >) or a keyword. (Input/Output)

Search Rule Keywords

default	places the default search rules at this position.
home_dir	places the user's home directory at this position.
referencing_dir	when processing this position, the searching program looks in the same directory as the one that referenced the sought-after file.
working_dir	when processing this position, the searching program looks in the current working directory.

Entry: find_include_file_$set_search_rules

This entry is provided to allow the caller to change the translator search rules currently in effect. The search directories provided are not checked for existence, but must be full pathnames or known keywords.

Usage

```
declare find_include_file_$set_search_rules entry (ptr,
        fixed bin (35));

call find_include_file_$set_search_rules (addr (tsr_strct),
        error_code);
```

where:

1. addr (tsr_strct) is the address of the communication structure described above. (Input)
2. error_code is a standard Multics status code. See "Status Codes" below. (Output)

Status Codes

0	The user-specified search rules were processed normally and are now in effect.
error_table_$area_too_small	The allocation area used by this procedure does not have room for the new search rules. The previous search rules remain in effect.
error_table_$bad_string	One of the specified search rule keywords could not be recognized. The previous search rules remain in effect.
error_table_$id_already_exists	One of the special rules was multiply used. (The special rules are the working_dir and referencing_dir rules.) The previous search rules remain in effect.
error_table_$unimplemented_version	The program does not recognize the version number element of the communication structure. The previous search rules remain in effect.

Note

This procedure shares allocation storage with other system procedures. Thus, at any given time, the number of search rules that a user may set can range from some large number down to zero.

Entry: find_include_file_$get_search_rules

This entry is provided to allow the caller to get the translator search rules currently in effect.

Usage

```
declare find_include_file_$get_search_rules entry (ptr,
     fixed bin (35));

call find_include_file_$get_search_rules (addr (tsr_strct),
     error_code);
```

where:

1. addr (tsr_strct) is the address of the communication structure described above. (Input)

2. error_code is a standard Multics status code. See "Status Codes" below. (Output)

Status Codes

0	Normal processing has occurred--the rule array has been filled in and the num_valid_rules element of the communication structure contains the number of rules that are in effect.
error_table_$too_many_sr	The caller did not provide a sufficiently large rule array. No rules have been filled in, but the num_valid_rules element has been set to the necessary size for the rule array, if the user should call this entry again.
error_table_$unimplemented_version	The program does not recognize the version number element of the communication structure. The communication structure has not been altered.

Note

This entry point is designed primarily to allow the user to reset or to modify the current search rules. In particular, the rule array can not be identical to the rule array used to set the rules (e.g., the home_dir keyword is not returned, but rather the actual directory name).

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_ (objp, bitcount, oip,
     bmp, sblkp, code);
```

where:

1. objp is a pointer to the beginning of the segment. (Input)
2. bitcount is the segment's bit count. (Input)
3. oip is a pointer to the object format structure to be filled in by object_info_$display (see structure declaration in the description of object_info_). (Input)
4. bmp is a pointer to the bind map. (Output)
5. sblkp is a pointer to the base of the symbol block containing the bindmap. (Output)
6. code is a standard Multics status code. (Output)

Note

If objp 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 oip contains valid information.

get_initial_ring_

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_ (iring);
```

where iring is the initial ring for the process. (Output)

get_lock_id_

Name: get_lock_id_

The get_lock_id_ subroutine returns the 36-bit unique lock ID to be used by a process in setting locks. By using this lock ID a convention can be established so that a process wishing to lock a data base and finding it already locked can verify that the lock is set by an existing process.

Usage

```
declare get_lock_id_ entry (bit(36) aligned);

call get_lock_id_ (lock_id);
```

where lock_id is the unique identifier of this process. (Output)

get_primary_name_

Name: get_primary_name_

The get_primary_name_ subroutine locates a segment in a given directory, and returns the first, or primary, name of that segment. For example, if foo.archive is the primary name of an archive that contains component able.pl1, and if able.pl1 is another name on the archive, then get_primary_name_ returns the name foo.archive when called with the name able.pl1.

Usage

```
declare get_primary_name_ entry (char(*), char(*),
     char(*), fixed bin(35));

call get_primary_name_ (dir, seg, prime_name, code);
```

where:

1. dir — is the directory to be searched. (Input)
2. seg — is the name of the segment to be located in the search directory. (Input)
3. prime_name — is the primary name of the segment, if it was found. (Output)
4. code — is a standard Multics status code. (Output)

Note

On return, if code = 0, the segment was found, and prime_name is the primary name of the segment. If code = 1, then the segment was not found in the directory searched. Any other value of code is a standard status code returned from a call to hcs_$status_.

Name: get_seg_ptr_

The get_seg_ptr_ subroutine consists of entries to initiate and terminate data segments. It also is capable of creating, truncating, and setting the bit count on segments. It is more useful than the current Multics file primitives because it expands pathnames, creates segments, and initiates them all in one call by the user. Similarly, it sets the bit count, truncates the segment, and terminates them all in one call.

The primary entry initiates a segment given a relative pathname and checks access to the segment. If the segment does not exist, it is created if the user so requests.

Usage

```
declare get_seg_ptr_ entry (char(*), bit(6) aligned,
     fixed bin(24), ptr, fixed bin);

call get_seg_ptr_ (pathname, wanted_access, bit_count,
     return_ptr, return_code);
```

where:

1. pathname — is a relative pathname to the segment. (Input)

2. wanted_access — is the requested access to the segment. It can be described by the following overlay structure:

```
declare 1 wanted_access_overlay aligned,
          2 pad1 bit(1) unaligned,
          2 read bit(1) unaligned,
          2 execute bit(1) unaligned,
          2 write bit(1) unaligned,
          2 pad2 bit(1) unaligned,
          2 create bit(1) unaligned;
```

The read, execute, and write bits are interpreted as standard Multics access control bits. If the segment exists, then the error code error_table_$moderr is returned if the user does not have at least the access requested. Note, however, that return_ptr contains a valid pointer even if this error occurs. The create bit is interpreted, if on, as an indication that the segment should be created (with the specified access) if it does not exist.

3. bit_count is the bit count of the segment. (Output)

4. return_ptr is a pointer to the segment. If the segment is not initiated, this pointer is returned as null. (Output)

5. return_code is a standard Multics status code. The only condition under which this code is nonzero when the return_ptr is nonnull is if the error is error_table_$moderr. (Output)

Entry: get_seg_prt_$release_seg_ptr_

This entry terminates a segment initiated by one of the entries to get_seg_ptr_. If a bit count is specified, the bit count of the segment is set and the segment is truncated to the corresponding length.

Usage

```
declare release_seg_ptr_ entry (ptr, fixed bin(24),
     fixed bin);

call release_seg_ptr_ (seg_ptr, bit_count, return_code);
```

where:

1. seg_ptr is a pointer to the segment to be terminated. (Input)

2. bit_count is the bit count to be set on the segment. If this argument is negative, it is assumed that the bit count should remain the same and no truncation should take place. (Input)

3. return_code is a standard Multics status code. (Output)

Entry: get_seg_ptr_$get_seg_ptr_arg_

This entry is identical to get_seg_ptr_ except that it obtains the pathname of the segment to be initiated from the caller's argument list. It saves a call to cu_$arg_ptr.

Usage

```
declare get_seg_ptr_arg_ entry (fixed bin, bit(6)
     aligned, fixed bin(24), ptr, fixed bin);

call get_seg_ptr_arg_ (arg_number, wanted_access,
     bit_count, return_ptr, return_code);
```

where:

1. arg_number is the number of the caller's argument to be used. (Input)

2-5. are as above.

Entry: get_seg_ptr_$get_seg_ptr_full_path_

This entry is identical to get_seg_ptr_ except that the pathname is specified as an absolute pathname in directory/entry form.

Usage

```
declare get_seg_ptr_full_path_ entry (char(*), char(*),
     bit(6) aligned, fixed bin(24), ptr, fixed bin);

call get_seg_ptr_full_path_ (dir_name, entry_name,
     wanted_access, bit_count, return_ptr, return_code);
```

where:

1. dir_name is the absolute pathname of the directory of the segment. (Input)

2. entry_name is the entryname of the segment. (Input)

3-6. are as above.

Entry: get_seg_ptr_$get_seg_ptr_search_

This entry is identical to get_seg_ptr_ except that just an entryname is specified. The directory is determined by Multics search rules. If the segment is not found and if the c (create) bit is on, then the segment is created in the process directory. Note, however, that if the entryname is not known as a reference name before a call to get_seg_ptr_search_, this entry does not cause it to be made known. This procedure initiates the segment with a null reference name. This has the net effect that full search rules are followed each time this routine is called.

Usage

```
declare get_seg_ptr_search_ entry (char(*), bit(6) aligned,
     fixed bin(24), ptr, fixed bin);

call get_seg_ptr_search_ (entry_name, wanted_access,
     bit_count, return_ptr, return_code);
```

where:

1. entry_name is the entryname of the segment to be found. (Input)

2-5. are as above.

Name: get_temp_seg_

The get_temp_seg_ subroutine manages temporary segments in the process directory. For each segment it creates, it maintains information about the procedure and invocation level currently using the temporary segment. It allows a program that uses temporary segments to be called recursively or when one invocation has been suspended.

Usage

```
declare get_temp_seg_ entry (bit(36) aligned,
     bit(5) aligned, ptr, fixed bin(35));

call get_temp_seg_ (id, access, tsptr, code);
```

where:

1. id — is an identifier unique to the current invocation of the calling procedure. The calling procedure must previously have called get_temp_seg_$assign_temp_seg_id_ to get id. (Input)

2. access — is the access desired for the temporary segment. The bits correspond to the read, execute, and write attributes respectively. The first and last bits are not used. (Input)

3. tsptr — points to the temporary segment. (Output)

4. code — is a standard Multics status code. (Output)

Entry: get_temp_seg_$assign_temp_seg_id_

This procedure should be called before the main entry (above) is first called by a given invocation of a procedure wishing to use get_temp_seg_.

Usage

```
declare get_temp_seg_$assign_temp_seg_id_ entry (char(*)
     aligned, bit (36) aligned, fixed bin (35));

call get_temp_seg_$assign_temp_seg_id_ (procname, id, code);
```

where:

1. procname is the name of the calling procedure. (Input)

2. id is the identifier to be used by the current invocation of the calling procedure when calling the other entries. (Output)

3. code is a standard Multics status code. (Output)

Entry: get_temp_seg_$release_temp_seg_

This procedure releases a temporary segment that has been obtained by calling get_temp_seg_.

Usage

```
declare get_temp_seg_$release_temp_seg_ entry (bit(36) aligned,
     ptr, fixed bin(35));

call get_temp_seg_$release_temp_seg_ (id, tsptr, code);
```

where:

1. id is the same as above. (Input)

2. tsptr is the same as above. (Input)

3. code is the same as above. (Output)

Entry: get_temp_seg_$release_temp_segs_all_

This entry releases all temporary segments currently being used by the current invocation of the caller.

Usage

```
declare get_temp_seg_$release_temp_segs_all_ entry
     (bit(36) aligned, fixed bin(35));

call get_temp_seg_$release_temp_segs_all_ (id, code);
```

where:

1. id is the same as above. (Input)
2. code is the same as above. (Output)

Name: hash_

The hash_ subroutine can be used to initialize, insert, delete, and search for entries in a hash table. This is a fast alternative to searching for an entry in another unorganized data table or group of data. When an entry to a data table is created, hash_$in should be called with the identifier of the entry and the value that is used to locate that entry in the data table (i.e., array subscript). When it is necessary to access this entry later, this value is returned by hash_$search (given the identifier).

Entry: hash_$make

This entry point initializes a hash table pointed to by tableptr with the number of buckets or entries given by nb.

Usage

```
declare hash_$make entry (ptr, fixed bin, fixed bin);

call hash_$make (tableptr, nb, code);
```

where:

1. tableptr — is the pointer to the hash table. (Input)
2. nb — is the number of buckets in the hash table. There is a maximum of 6552 buckets. The table created occupies 10*nb+8 words. (Input)
3. code — is a standard Multics status code that equals error_table_$invalid_elsize if nb is too large. If code equals 0, there is no error. (Output)

Entry: hash_$in

This entry point inserts an entry in the hash table pointed to by tableptr. The input arguments name and value are the identifier and corresponding value that are used to access the entry, name, in the data table.

Usage

```
declare hash_$in entry (ptr, char(*), fixed bin, fixed bin);

call hash_$in (tableptr, name, value, code);
```

where:

1. tableptr is the pointer to the hash table. (Input)

2. name is the identifier of the entry in the data table. The maximum length is 32 characters. (Input)

3. value is the locator of the entry in the data table that corresponds to name. (Input)

4. code is a standard Multics status code that equals error_table_$segnamedup if the entry already exists with the same value. If code equals error_table_$namedup, the entry already exists with a different value. If code equals 0, there is no error. (Output)

Entry: hash_$search

This entry point, given name, returns the corresponding value (supplied by hash_$in) that locates the entry in the data table.

Usage

```
declare hash_$search entry (ptr, char(*), fixed bin,
     fixed bin);

call hash_$search (tableptr, name, value, code);
```

where:

1. tableptr is a pointer to the hash table. (Input)

2. name is the identifier of an entry. The maximum length is 32 characters. (Input)

3. value is the locator of an entry in the data table corresponding to name. (Output)

4. code is a standard Multics status code that equals error_table_$noentry if the entry is not in the hash table. If code equals 0, there is no error. (Output)

Entry: hash_$out

This entry point deletes an entryname from the hash table.

Usage

```
declare hash_$out entry (ptr, char(*), fixed bin,
     fixed bin);

call hash_$out (tableptr, name, value, code);
```

where:

1. tableptr — is a pointer to the hash table. (Input)
2. name — is the identifier of an entry. The maximum length is 32 characters. (Input)
3. value — is the locator associated with name. (Output)
4. code — is a status code that equals error_table_$noentry if name was not in the hash table. If code equals zero, there is no error. (Output)

Note

If a hash table becomes full or inefficient, it is rehashed with 400 more buckets (unless the maximum number of buckets has been reached, in which case the code returned is error_table_$full_hashtbl).

hcs_$get_page_trace

Name: hcs_$get_page_trace

The hcs_$get_page_trace subroutine returns information about recent paging activity.

Usage

```
declare hcs_$get_page_trace entry (ptr);

call hcs_$get_page_trace (datap);
```

where datap 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.

```
declare 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. 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.

5. info — is information about the particular trace entry.

6. type — specifies what kind of a trace entry it is. The following types are currently defined:

page fault	0
segment fault begin	2
segment fault end	3
linkage fault begin	4
linkage fault end	5
bound fault begin	6
ound fault end	7
signaller event	8
restarted signal	9
reschedule	10
user marker	11
interrupt	12

7. pageno — is the page number associated with the fault. Certain trace entries do not fill in this field.

8. 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.

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, fixed bin(17),
     fixed bin(17));

call link_unsnap_ (log_ptr, linkage_ptr, hcsc, high_seg);
```

where:

1. lot_ptr — is a pointer to the LOT. (Input)
2. linkage_ptr — is a pointer to the linkage section to be discarded. (Input)
3. hcsc — is one less than the segment number of the first segment that can be unsnapped. (Input)
4. high_seg — is the number of LOT slots used in searching for links to be restored. (Input)

list_dir_info_

Name: list_dir_info_

The list_dir_info_ subroutine is used by list_dir_info, rebuild_dir, and comp_dir_info to list the values in a single entry in a directory information segment created by save_dir_info.

Usage

```
declare list_dir_info_ entry (ptr, fixed bin, char(1));

call list_dir_info_ (p, mode, prefix);
```

where:

1. p — points to an entry in the dir_info segment. (Input)
2. mode — is the verbosity desired. It can be 0, 1, or 2. (Input)
3. prefix — is a one-character prefix for every line printed. (Input)

Output from list_dir_info_ is written on user_output. 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.

For segments:

0.
- names
- type
- date used
- date modified

1.
- date branch modified
- records used
- bit count
- bit count author
- max length
- safety switch

2.
- ACL
- data dumped
- current length
- device ID

list_dir_info_

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

2. ACL
initial seg ACL
initial dir ACL

For links:

0. names
type
target

1. date link modified

2. date link dumped

parse_file_

Name: parse_file_

The parse_file_ subroutine provides a facility for parsing an 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_init_name and parse_file_init_ptr, both 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 initializes the subroutine given a directory and an entryname. 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);

call parse_file_$parse_file_init_name (dir, entry, p,
      code);
```

where:

1. dir is the directory name portion of the pathname of the segment to be parsed. (Input)

2. entry is the entryname of the segment to be parsed. (Input)

3. p is a pointer to the segment. (Output)

4. code is a standard Multics 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 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 (p, cc);
```

where:

1. p is a pointer to a segment or an aligned character string. (Input)

2. cc is the character count of the ASCII text to be scanned. (Input)

Entry: parse_file_$parse_file_set_break

Break characters can be defined by use of this entry. 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 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 will be made a nonbreaking character. (Input)

Entry: parse_file_

The text file is scanned and the next break character or symbol is returned. Comments enclosed by /* and */, blanks, and newline characters, 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 — 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.) (Output)

2. cc — is the number of characters in the symbol. (Output)

3. break — is set to 1 if the returned item is a break character; otherwise it is 0. (Output)

4. eof — is set to 1 if the end of text has been reached; otherwise it is 0. (Output)

Entry: parse_file_$parse_file_ptr

This entry is identical to parse_file_ 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 (p, cc, break, eof);
```

where:

1. p is a pointer to the symbol or the break character. (Output)

2-4. are the same as above. (Output.

Entry: parse_file_$parse_file_cur_line

The current line of text being scanned is returned to the caller. 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:

1-2. are the same as in parse_file_ above.

Entry: parse_file_$parse_file_line_no

The current line number of text being scanned is returned to the caller. 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 contains the following text:

```
name:  foo;  /*foo program*/

path_name:  >bar;

linkage;

end;

fini;
```

The following calls could be made to initialize the parsing of zilch:

```
call parse_file_$parse_file_init_name (dir, zilch,
     p, code);

call parse_file_$parse_file_unset_break (">_");

declare atom char (cc) unaligned based (p);
```

Subsequent calls to parse_file_ptr would then yield the following:

atom	break	eof
name	0	0
:	1	0
foo	0	0
;	1	0
path_name	0	0
:	1	0
>bar	0	0
;	1	0
linkage	0	0
;	1	0

parse_file_

atom	break	eof
end	0	0
;	1	0
fini	0	0
;	1	0
-	-	1

Name: print_gen_info_

The print_gen_info_ subroutine is used to print out general information about an object segment. The format of the output is the same as the print_gen_info command.

Usage

```
declare print_gen_info_ entry (ptr, fixed bin(24),
     char(*), fixed bin(35));

call print_gen_info_ (p, bc, stream, code);
```

where:

1. p — is a pointer to the object segment of interest. (Input)
2. bc — is the bit count of the object segment. (Input)
3. stream — is the name of the I/O switch over which the information is to be output. (Input)
4. code — is a standard Multics status code. (Output)

Entry: print_gen_info_$component

This entry prints the information about a particular component of a bound segment over the I/O switch specified.

Usage

```
declare print_gen_info_$component entry (ptr,
     fixed bin(24), char(*), fixed bin(35), char(*));

call print_gen_info_$component (p, hc, stream, code,  name);
```

where:

1-4. are the same as the print_gen_info_ entry.

5. name — is the name of the component within the bound segment for which information is to be printed. (Input.

Name: ring0_get_

The ring0_get_ subroutine returns the name and pointer information about hardcore segments.

Entry: ring0_get_$segptr

This entry 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, entry, segptr, code);
```

where:

1. dir — is ignored. (Input)
2. entry — is the name of the ring 0 segment for which a pointer is desired. (Input)
3. segptr — is a pointer to the segment. (Output)
4. code — is nonzero if, and only if, the entry was not found. (Output)

Note

If the entry was not found, segptr is returned null.

Entry: ring0_get_$name

This entry 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, entry, segptr, code);
```

where:

1. dir — is the pathname of the directory of the segment (if the segment does not have a pathname, this is ""). (Output)

2. entry — is the primary name of the segment. (Output)

3. segptr — is a pointer to the ring 0 segment. (Input)

4. code — is nonzero if, and only if, segptr does not point to a ring 0 segment. (Output)

Entry: ring0_get_$names

This entry 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, names_ptr, segptr, code);
```

where:

1. dir — is the pathname of the directory of the segment. (Output)

2. names_ptr — is a pointer to a structure containing the names of the segment. (Output)

The following structure is used:

```
declare 1 segnames based (names_ptr) aligned,
          2 count fixed bin,
          2 names (50 refer (segnames.count)),
            3 length fixed bin,
            3 name char(32);
```

a. count — is the number of names.

b. names — is a substructure containing an array of segment names.

c. length — is the length of the name in characters.

d. name — is the space for the name.

ring0_get_

3. segptr — is a pointer to the ring 0 segment. (Input)

4. code — is nonzero if, and only if, segptr does not point to a ring 0 segment. (Output)

ring_zero_peek_

Name: ring_zero_peek_

The ring_zero_peek_ subroutine is used to extract data from the hardcore supervisor. Data that is not generally available to normal users is returned only to privileged users.

Usage

```
declare ring_zero_peek_ entry (ptr, ptr, fixed bin (18),
     fixed bin (35));

call ring_zero_peek_ (ptr0, ptr_user, nwords, status);
```

where:

1. ptr0	is a pointer to the data in ring 0 that is to be copied out. (Input).
2. ptr_user	is a pointer to the region in the user's address space where the data is to be copied. (Input).
3. nwords	is the number of words to be copied. (Input)
4. status	is a returned status code which is nonzero if the user did not have access to the requested data. (Output)

Name: set_lock_

The set_lock_ subroutine contains entry points for locking and unlocking user data bases. The following entry points are documented in the MPM Subsystem Writers' Guide (SWG), Order No. AK92: set_lock_$lock, set_lock_$unlock.

Entry: set_lock_$admin_lock

This entry performs the same function as set_lock_$lock. It, however, also grants increased priority scheduling to the executing process and keeps metering statistics about its use.

Usage

```
declare set_lock_$admin_lock entry (bit(36) aligned,
     fixed bin, fixed bin);

call set_lock_$admin_lock (lock, wait_time, status);
```

where:

1. lock — is the lock word to be locked. (Input/Output)
2. wait_time — is the number of seconds for which this procedure is to wait for the lock to be unlocked before giving up. (Input)
3. status — indicates the success of the call. See the SWG section on set_lock_. (Output)

Entry: set_lock_$admin_unlock

This entry performs the same function as set_lock_$unlock. It, however, also rescinds increased priority scheduling to the executing process and keeps metering statistics about its use.

set_lock_

Usage

```
declare set_lock_$admin_unlock entry (bit(36) aligned,
     fixed bin);

call set_lock_$admin_unlock (lock, code);
```

where:

1. lock is the lock word to be unlocked. (Input/Output)
2. code is a standard Multics status code. (Output)

Name: sort_items_

The sort_items_ subroutine provides a generalized, yet highly efficient, sorting facility. Entries are provided for sorting fixed binary (35) numbers, float binary (63) numbers, fixed-length character strings, and fixed-length bit strings. A generalized entry 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.

Entry: sort_items_$fixed_bin

This entry 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 (vP);
```

where vP points to a structure containing an array of unaligned pointers to the aligned fixed binary (35,0) numbers to be sorted. (Input)

Entry: sort_items_$float_bin

This entry 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 (vP);
```

where vP points to a structure containing an array of unaligned pointers to the aligned float binary (63) numbers to be sorted. (Input)

sort_items_

Entry: sort_items_$char

This entry 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 (vP, length);
```

where:

1. vP — points to a structure containing an array of unaligned pointers to the fixed-length unaligned character strings to be sorted. (Input)

2. length — is the number of characters in each string. (Input)

Entry: sort_items_$bit

This entry 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 (vP, length);
```

where:

1. vP — points to a structure containing an array of unaligned pointers to the fixed-length unaligned bit strings to be sorted. (Input)

2. length — is the number of bits in each string. (Input)

Entry: sort_items_$general

This entry 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 (vP, function);
```

where:

1. vP — points to a structure containing an array of unaligned pointers to the data items to be sorted. (Input)

2. function — is a user-supplied ordering function. Its calling sequence is shown in the "Note" below. (Input)

Note

The command sort_items_$general 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 — is an unaligned pointer to the first data item. (Input)

2. ptr_second_item — is an unaligned pointer to a data item to be compared with the first data item. (Input)

3. value — is -1 if the first data item is less than the second.

   is 0 if the first data item is equal to the second.

   is +1 if the first data item is greater than the second. (Output)

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 sort_items_$general, it performs the same function as a call to sort_items_$fixed_bin, 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;
```

sort_items_

Note

The structure pointed to by vP is to be declared as follows, where n is the value of v.n:

```
declare 1 v aligned,
          2 n fixed bin (24),
          2 vector (n) ptr unaligned;
```

sort_items_indirect_

Name: sort_items_indirect_

The sort_items_indirect_ subroutine is a variation of the sort_items_$general entry. It provides a facility for sorting a group of data items, based upon the value of an information field that is logically associated with each item, but resides at a varying offset from the beginning of each item. One of the names in the name list associated with the status block returned by hcs_$status_ is an example of such an information field.

The procedure sort_items_indirect_ provides high performance entries 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.

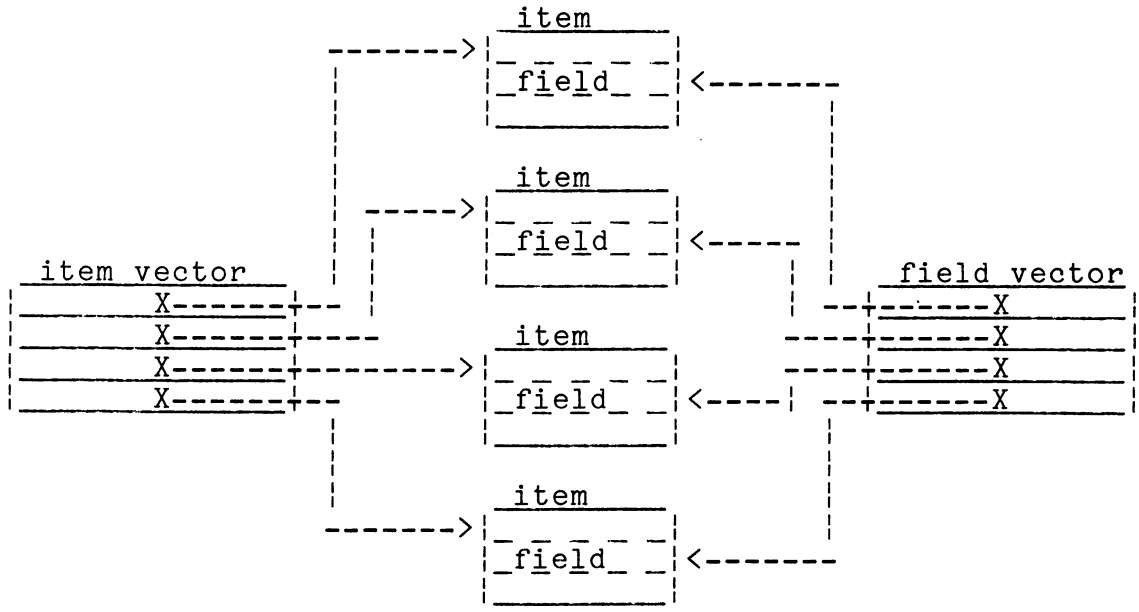
To use sort_items_indirect_, the caller must create three arrays: a vector of pointers to the data items being sorted (the item vector); a vector of pointers to the single information field within each item on which the sort is based (the field vector); and an array of indices into these two vectors.

Notes

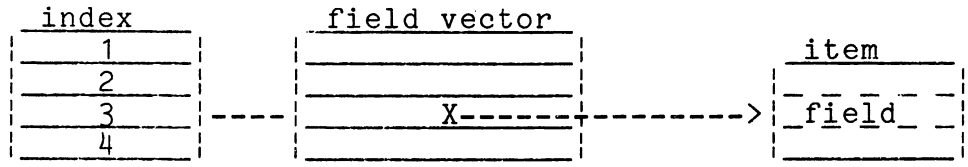
To use sort_items_indirect_$adj_char, one additional array must be created: an array of lengths of the adjustable-length character string information fields on which the sort is based.

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.

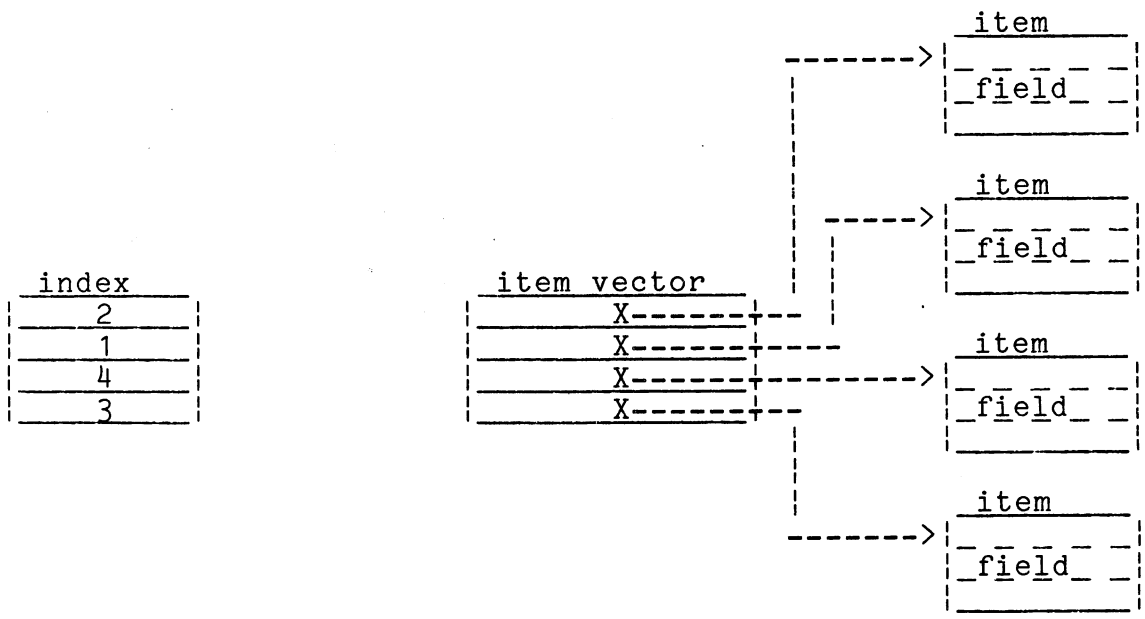


The array of indices can be used to reference elements of both vectors. The field vector and index array are passed to sort_items_indirect_, which references the sorting field in each item through elements of these two arrays, as shown below.



The procedure sort_items_indirect_ reorders the index values so that values selected sequentially from the index array reference points 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. Therefore, the index array can also be used to reference a sorted list of items, as shown below.

sort_items_indirect_



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. (sort_items_ cannot process adjustable-length fields.) The field vector is passed to sort_items_, 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 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 (vP, iP);
```

where:

1. vP — points to a structure containing an array of unaligned pointers to the aligned fixed binary (35,0) numbers to be sorted. (Input)

2. iP — points to the structure into which the ordered array of fixed binary (24) indices into the unaligned pointer array will be placed. (Input)

Entry: sort_items_indirect_$float_bin

This entry 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 (vP, iP);
```

where:

1. vP — points to a structure containing an array of unaligned pointers to the aligned float binary (63,0) numbers to be sorted. (Input)

2. iP — points to the structure into which the ordered array of fixed binary (24) indices into the unaligned pointer array will be placed. (Input)

Entry: sort_items_indirect_$char

This entry sorts a group of information fields, which are fixed-length 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_$char entry (ptr, ptr,
    fixed bin (24));

call sort_items_indirect_$char (vP, index, length);
```

where:

1. vP — points to a structure containing an array of unaligned pointers to the unaligned fixed-length character strings to be sorted. (Input)

2. iP — points to the structure into which the ordered array of fixed binary (24) indices into the unaligned pointer array will be placed. (Input)

3. length — is the number of characters in each string. (Input)

Entry: sort_items_indirect_$bit

This entry 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 was converted to a binary natural number, the binary value would be 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 (vP, iP, length);
```

where:

1. vP	points to a structure containing an array of unaligned pointers to the fixed-length unaligned bit strings to be sorted. (Input)
2. iP	points to the structure into which the ordered array of fixed binary (24) indices into the unaligned pointer array will be placed. (Input)
3. length	is the number of bits in each string. (Input)

Entry: sort_items_indirect_$general

This entry 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 (vP, iP, function);
```

where:

1. vP	points to a structure containing an array of unaligned pointers to the information fields to be sorted. (Input)
2. iP	points to the structure into which the ordered array of fixed bin (24) indices into the unaligned pointer array will be placed. (Input)
3. function	is a user-supplied ordering function. (See "Note" below.) (Input)

Note

The procedure sort_items_indirect_$general 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 — is an unaligned pointer to the first information field. (Input)

2. ptr_2nd_field — is an unaligned pointer to an information field to be compared with the first information field. (Input)

3. value — is -1 if the first information field is less than the second.

   is 0 if the first information field is equal to the second.

   is +1 if the first information field is greater than the second. (Output)

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 sort_items_indirect_$general, it performs the same function as a call to sort_items_indirect_$fixed_bin, but with less efficiency because of the overhead involved in calling the function.

sort_items_indirect_

```
function: procedure (p1, p2) returns (fixed bin(1));

    declare(p1, p2) ptr unaligned,
         field fixed bin(35,0) based;

         if p1 -> field < p2 -> field then
              return (-1);
         else if p1 -> field = p2 -> field then
              return ( 0);
         else
              return (+1);

         end function;
```

Entry: sort_items_indirect_$adj_char

This entry 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 (vP, iP, lP);
```

where:

1. vP — points to a structure containing an array of unaligned pointers to the unaligned adjustable-length character strings to be sorted. (Input)

2. iP — points to the structure into which the ordered array of indices into the unaligned pointer array will be placed. (Input)

3. lP — points to a structure containing an array of lengths of the unaligned adjustable-length character strings to be sorted. (Input)

Note

The structure pointed to by vP is to be declared as follows, where n is the value of v.n:

```
declare 1 v aligned,
          2 n fixed bin (24),
          2 vector (n) ptr unaligned;
```

The structure pointed to by iP or lP is to be declared as follows, where n is the value of a.n:

```
declare 1 a aligned,
          2 n fixed bin (24),
          2 array (n) fixed bin (24);
```

sweep_disk_

Name: sweep_disk_

The sweep_disk_ subroutine traverses the directory hierarchy below a specified node, calling a user-supplied routine at each entry of the subtree.

Usage

```
declare sweep_disk_ entry (char(168) aligned, entry);

call sweep_disk_ (path, counter);
```

where:

1. path — is the pathname of the base node of the subtree to be scanned. (Input)

2. counter — is an entry point called for each branch or link in the subtree. (Input)

Notes

The routine counter is assumed to have the following declaration and call:

```
declare counter entry (char(168) aligned, char(32) aligned,
     fixed bin, char(32) aligned, ptr, ptr);

call counter (ddn, een, lrl, ename, bptr, nptr);
```

where:

1. ddn — is the pathname of the directory immediately superior to the directory that contains the current entry. (Input)

2. een — is the entryname of the directory that contains the current entry. (Input)

3. lrl — is the number of levels deep from the given starting node. (Input)

4. ename is the primary name on the current entry. (Input)

5. bptr is a pointer to the branch structure returned by hcs_$star_list for the current entry. (Input)

6. nptr is a pointer to the names area for the current entry's parent directory, returned by hcs_$star_list. (Input)

The routine sweep_disk_ attempts to initiate and terminate directories to avoid a Known Segment Table (KST) overflow. If it has sufficient access, it attempts to give itself access. If unable to get access to a directory, it attempts to continue. The contents of >process_dir_dir are ignored. Access is cleaned up after a directory is processed.

Name: system_info_

The entry points discussed in this description are provided by the system_info_ subroutine for the use of system modules in addition to those described in the MPM Subroutines, Order No. AG93.

The following entry points are documented in the MPM Subroutines:

system_info_$device_prices
system_info_$device_rates
system_info_$installation_id
system_info_$next_shutdown
system_info_$prices
system_info_$rates
system_info_$shift_table
system_info_$sysid
system_info_$timeup
system_info_$titles
system_info_$users

Entry: system_info_$abs_chn

This entry returns the event channel and process ID for the process that is running the absentee user manager.

Usage

```
declare system_info_$abs_chn entry (fixed bin(71),
     bit(36) aligned);

call system_info_$abs_chn (ec, pid);
```

where:

1. ec is the event channel over which signals to absentee_user_manager_ should be sent. (Output)

2. pid is the process ID of the absentee manager process (currently the initializer). (Output)

Entry: system_info_$next_shift_change

This entry point returns the time of the next shift change.

Usage

```
declare system_info_$next_shift_change entry (fixed bin,
     fixed bin(71), fixed bin)

call system_info_$next_shift_change (nowshift, changetime,
     newshf);
```

where:

1. nowshift is the current shift number. (Output)
2. changetime is the time the shift changes. (Output)
3. newshf is the shift after changetime. (Output)

Entry: system_info_$shift_table

This entry point returns the system's local shift definition table.

Usage

```
declare system_info_$shift_table ((336) fixed bin);

call system_info_$shift_table (stt);
```

where stt is a table of shifts, indexed by half-hour within the week. stt(1) gives the shift for 0000-0030 Mondays, etc. (Output)

teco_get_macro_

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. mname is the name of the macro to be found. (Input)
2. mptr is a pointer to the macro. (Output)
3. mlen is the length of the macro. (Output)
4. code is a standard Multics status code. (Output)

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 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,
     dirname, entry_name, date_time_mod, unique_id,
     error_code);
```

where:

1. source_ptr — is a pointer to the source segment about which information is desired. (Input)
2. dirname — is a pathname of the directory in which the source segment is located. (Output)
3. entry_name — is the primary name of the source segment. (Output)
4. date_time_mod — is the date time modified of the source segment as obtained from the storage system. (Output)
5. unique_id — is the unique id of the source segment as obtained from the storage system. (Output)
6. error_code — is a standard Multics status code. (Output)

Status Codes

A zero status code 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 primaryname on the source segment. It is not necessarily the same name as that by which the translator initiated it.

virtual_cpu_time_

Name: virtual_cpu_time_

The virtual_cpu_time_ subroutine returns the CPU time used by the calling process not spent handling page faults, or system interrupts. It is therefore a measure of the CPU time within a process that is independent of other processes, current configuration, and overhead to implement the virtual memory for the calling process.

Usage

```
declare virtual_cpu_time_ entry returns (fixed bin(71));

time = virtual_cpu_time_ ();
```

where time is the virtual CPU time in microseconds, used by the calling process. (Output)

Name: whotab

The whotab subroutine is the public information base for the system. All users who are listed by the who command have an entry in this table which becomes active after they are logged in. In addition, various system parameters of interest to all users are recorded in whotab. The answering service module lg_ctl_ maintains most of the data; only the initializer process can modify the segment.

Usage

```
declare 1 whotab based (whoptr) 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 sysid char(8),
          2 nextsd fixed bin(71),
          2 until fixed bin(71),
          2 lastsd fixed bin(71),
          2 erfno char(8),
          2 why char(32),
          2 installation_id char(32),
          2 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 pad (17) fixed bin,
          2 laste fixed bin,
          2 freep fixed bin,
          2 e (1000),
            3 active fixed bin,
            3 person char(28) aligned,
            3 project char(28),
            3 anon fixed bin,
            3 alias char(8),
            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 pad1 (5) fixed bin;
```

where:

1. mxusers	is the maximum number of users allowed on the system.
2. n_users	is the current number of users.
3. mxunits	is the maximum number of load units allowed.
4. n_units	is the current load.
5. timeup	is the time the system was started.
6. sysid	is the current system name.
7. nextsd	is the time the system will be shutdown, if nonzero.
8. until	is the projected time of the next system start-up.
9. lastsd	is the time of last crash or shutdown.
10. erfno	is the error number of the last crash, if known.
11. why	is the reason for next shutdown, if known.
12. installation_id	is the name of the installation.
13. message	is a message for all users (not used).
14. abs_event	is the event channel for signalling absentee requests.
15. abs_procid	is the processid of the absentee user manager.
16. max_abs_users	is the current maximum number of absentee users.
17. abs_users	is the current number of absentee users.
18. pad	is padding.
19. laste	is the index of the last entry in use.
20. freep	is the index of the first free entry chained through "chain".

whotab

21. active	is nonzero if the user is logged in.
22. person	is the person name.
23. project	is the project ID.
24. anon	is 1 for an anonymous user, otherwise 0.
25. alias	is the user alias (not used).
26. timeon	is the time of login.
27. units	is the number of load units for the user.
28. stby	is 1 for a secondary user.
29. idcode	is the tty ID code.
30. chain	is a chain for the free list.
31. proc_type	is the process type: 1 = interactive; 2 = absentee.
32. group	is the user's load control group ID.
33. pad1	is unused.

INDEX

HONEYWELL INFORMATION SYSTEMS

Publications Remarks Form*

TITLE: SERIES 60 (LEVEL 68) MULTICS SYSTEM TOOLS PROGRAM LOGIC MANUAL

ORDER No.: AN51, REV. 0

DATED: FEBRUARY 1975

ERRORS IN PUBLICATION:

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION:

(Please Print)

FROM: NAME _____ DATE: _____

COMPANY _____

TITLE _____

_____

_____

CUT ALONG LINE

*Your comments will be promptly investigated by appropriate technical personnel, action will be taken as required, and you will receive a written reply. If you do not require a written reply, please check here. ☐

CUT ALONG LINE

FOLD ALONG LINE

FIRST CLASS
PERMIT NO. 39531
WELLESLEY HILLS,
MASS. 02181

Business Reply Mail
Postage Stamp Not Necessary if Mailed in the United States

POSTAGE WILL BE PAID BY:

HONEYWELL INFORMATION SYSTEMS
60 WALNUT STREET
WELLESLEY HILLS, MASS. 02181

ATTN: PUBLICATIONS, MS 050

FOLD ALONG LINE

Honeywell

The Other Computer Company:

Honeywell

HONEYWELL INFORMATION SYSTEMS

12770
2.5C375
Printed in U.S.A.

In the U.S.A.: 200 Smith Street, MS 061, Waltham, Massachusetts 02154
In Canada: 2025 Sheppard Avenue East, Willowdale, Ontario

AN51, Rev. 0