Fall, 1973

Description of Interfaces into Ring 0

hcs_: Undocumented Entries

get_link_target :	gets the absolute pathname (i.e. link-free) of the branch pointed to by a link.
total_cpu_time_ :	returns total (i.e. including page fault and interrupt time) cpu time used by a process.
tty_attach :	attachs a terminal to a process
virtual_cpu_time_ :	returns virtual cpu time used by a process.

hcs_ : Obsolete Entries

accept_alm_obj
acl_add
acl_add1
acl_delete
acl_list
acl_replace
chname
ex_acl_delete
ex_acl_list
ex_acl_replace
fs_get_brackets
fs_get_call_name
fs_get_dir_name
get_entry_name
get_usage_values
initiate_seg
initiate_seg_count
list_dir
makeunknown
pre_page_info
set_alarm
set_timer
status
usage_values

hcs_ : Entries which should be in hphcs_

set_backup_dump_time
set_backup_times
set_dates
set_dtd

how bizarre!

hcs_: Entries which can be removed from ring 0

assign_linkage
del_dir_tree
fs_search_get_wdir
fs_search_set_wdir
get_count_linkage
get_linkage
get_lp
get_rel_segment
get_search_rules
get_seg_count
get_segment
initiate_search_rules
link_force
make_ptr
rest_of_datmk_
set_lp
unsnap_service

removed? linkers did this.

1. Were there any new entries added?

Name: hcs_$add_acl_entries

This subroutine, given a list of Access Control List (ACL) entries, will add the given ACL entries, or change their modes if a corresponding entry already exists, to the ACL of the specified segment.

Usage

```
declare hcs_$add_acl_entries entry (char(*), char(*),
        ptr, fixed bin, fixed bin(35));

call hcs_$add_acl_entries (dirname, ename, acl_ptr,
     acl_count, code);
```

1) dirname — is the directory portion of the path name of the segment in question. (Input)

2) ename — is the entry name portion of the path name of the segment in question. (Input)

3) acl_ptr — points to a user-filled segment_acl structure. See Notes below. (Input)

4) acl_count — contains the number of ACL entries in the segment_acl structure. See Notes below. (Input)

5) code — is a standard status code. (Output)

Notes

The following structure is used:

```
dcl 1 segment_acl (acl_count) aligned based (acl_ptr),
    2 access_name char(32),
    2 modes bit(36),
    2 zero_pad bit(36),
    2 status_code fixed bin(35);
```

1) access_name — is the access name (in the form person.project.tag) which identifies the processes to which this ACL entry applies.

2) modes — contain the modes for this access name. The first three bits correspond to the modes read, execute, and write. The remaining bits must be zero.

3) zero_pad must contain zero. (This field is for use with extended access.)

4) status_code is a standard status code for this ACL entry only.

If code is returned as error_table$argerr then the offending ACL entries in segment_acl will have status_code set to an appropriate error and no processing will have been performed.

If the segment is a gate (see the MPM Subsystem Writers' Guide section, Intraprocess Access Control (Rings)), then if the validation level is greater than Ring 1, then only access names that contain the same project as the user, and "SysDaemon" and "sys_control" projects will be allowed. If the ACL to be added is in error then no processing will be performed and the code error_table_$invalid_project_for_gate will be returned.

 (END)

Subroutine Call
2/13/73

Name: hcs_$add_dir_acl_entries

This subroutine, given a list of Access Control List (ACL) entries, will add the given ACL entries, or change their directory modes if a corresponding entry already exists, to the ACL of the specified directory.

Usage

```
declare hcs_$add_dir_acl_entries entry (char(*), char(*),
        ptr, fixed bin, fixed bin(35));

call hcs_$add_dir_acl_entries (dirname, ename, acl_ptr,
     acl_count, code);
```

1) dirname — is the path name of the directory superior to the one in question. (Input)

2) ename — is the entry name of the directory in question. (Input)

3) acl_ptr — points to a user-filled dir_acl structure. See Notes below. (Input)

4) acl_count — contains the number of entries in the dir_acl structure. See Notes below. (Input)

5) code — is a standard status code. (Output)

Notes

The following structure is used:

```
declare 1 dir_acl (acl_count) aligned based (acl_ptr),
          2 access_name char(32),
          2 dir_modes bit(36),
          2 status_code fixed bin(35);
```

1) access_name — is the access name (in the form person.project.tag) which identifies the process to which this ACL entry applies.

2) dir_modes — contains the directory modes for this access name. The first three bits correspond to the modes status, modify, and append. The remaining bits must be zero.

3) status_code — is a standard status code for this ACL entry only.

If code is returned as error_table_$argerr then the offending ACL entries in the dir_acl structure will have status_code set to an appropriate error and no processing will have been performed.

 (END)

Subroutine Call
2/27/73

Name: hcs_$add_dir_inacl_entries

This subroutine, given a list of Initial Access Control List (Initial ACL) entries, will add the given Initial ACL entries, or change their directory modes if a corresponding entry already exists, to the Initial ACL for new directories within the specified directory.

Usage

```
declare hcs_$add_dir_inacl_entries entry (char(*), char(*),
        ptr, fixed bin, fixed bin, fixed bin(35));

call hcs_$add_dir_inacl_entries (dirname, ename, acl_ptr,
     acl_count, ring, code);
```

1) dirname — is the path name of the directory superior to the one in question. (Input)

2) ename — is the entry name of the directory in question. (Input)

3) acl_ptr — points to a user-filled dir_acl structure. See Notes below. (Input)

4) acl_count — contains the number of entries in the dir_acl structure. See Notes below. (Input)

5) ring — is the ring number of the Initial ACL. (Input)

6) code — is a standard status code. (Output)

Notes

The following structure is used:

```
declare 1 dir_acl (acl_count) aligned based (acl_ptr),
          2 access_name char(32),
          2 dir_modes bit(36),
          2 status_code fixed bin(35);
```

1) access_name — is the access name (in the form person.project.tag) which identifies the processes to which this Initial ACL entry applies.

2) dir_modes — contains the directory modes for this access name. The first three bits correspond to the

modes status, modify, and append. The remaining bits must be zero.

3) status_code — is a standard status code for this Initial ACL entry only.

If code is returned as error_table_$argerr then the offending Initial ACL entries in the dir_acl structure will have status_code set to an appropriate error and no processing will have been performed.

 (END)

Subroutine Call
2/27/73

Name: hcs_$add_inacl_entries

This subroutine, given a list of Initial Access Control List (Initial ACL) entries, will add the given Initial ACL entries, or change their modes if a corresponding entry already exists, to the Initial ACL for new segments within the specified directory.

Usage

```
declare hcs_$add_inacl_entries entry (char(*), char(*),
        ptr, fixed bin, fixed bin, fixed bin(35));

call hcs_$add_inacl_entries (dirname, ename, acl_ptr,
     acl_count, ring, code);
```

1) dirname — is the superior directory portion of the path name of the directory in question. (Input)

2) ename — is the entry name portion of the path name of the directory in question. (Input)

3) acl_ptr — points to a user-filled segment_acl structure. See Notes below. (Input)

4) acl_count — contains the number of Initial ACL entries in the segment_acl structure. See Notes below. (Input)

5) ring — is the ring number of the Initial ACL. (Input)

6) code — is a standard status code. (Output)

Notes

The following structure is used:

```
dcl 1 segment_acl (acl_count) aligned based (acl_ptr),
    2 access_name char(32),
    2 modes bit(36),
    2 zero_pad bit(36),
    2 status_code fixed bin(35);
```

1) access_name — is the access name (in the form person.project.tag) which identifies the processes to which this Initial ACL entry applies.

2) modes — contain the modes for this access name. The first three bits correspond to the modes read, execute, and write. The remaining bits must be

zero.

3) zero_pad must contain zero. (This field is for use with extended access.)

4) status_code is a standard status code for this initial ACL entry only.

If code is returned as error_table$argerr then the offending Initial ACL entries in segment_acl will have status_code set to an appropriate error and no processing will have been performed.

 (END)

Subroutine Call
3/19/73

Name: hcs_$append_branch

This entry creates a segment in the specified directory, initiates the segment's Access Control List (ACL) by copying the Initial ACL for segments found in the directory, and adds the user to the segment's ACL with the mode specified. ACLs and Initial ACLs are described in the MPM Reference Guide section, Access Control.

Usage

```
declare hcs_$append_branch entry (char(*), char(*),
        fixed bin (5), fixed bin (35));

call hcs_$append_branch (dirname, entryname, mode, code);
```

1) dirname is the path name of the directory in which segname is to be placed. (Input)

2) entryname is the entry name of the segment to be created. (Input)

3) mode is the user's access mode; see Notes below. (Input)

4) code is a standard storage system status code. (Output)

Notes

Append (a) access mode is required in the directory dirname to add an entry to that directory.

A number of attributes of the segment are set to default values.

1) Ring brackets are set to the user's current validation level. See the MPM Subsystem Writers' Guide section, Intraprocess Access Control (Rings).

2) The user ID is set to the name and project of the user, with the instance tag set to *.

3) The copy switch is set to 0.

4) The bit count is set to 0.

See the MPM write-up for hcs_$append_branchx to create a storage system entry with other values than the defaults listed above.

The mode argument is a fixed binary number where the desired mode is encoded with one access mode specified by each bit. For segments the modes are:

read	8-bit (i.e., 01000b)
execute	4-bit (i.e., 00100b)
write	2-bit (i.e., 00010b)

For directories, the modes are:

status	8-bit (i.e., 01000b)
modify	2-bit (i.e., 00010b)
append	1-bit (i.e., 00001b)

The unused bits are reserved for unimplemented attributes and must be zero. For example, rw access in bit form is 01010b, and is 10 in fixed binary form.

 (END)

Name: hcs_$append_branchx

This entry creates either a subdirectory or a segment in the specified directory. (The entry point is really nothing more than an extended and more general form of hcs_$append_branch.) If a subdirectory is created then the subdirectory's access control list (ACL) is initiated by copying the Initial ACL for directories that is stored in the specified directory; otherwise the segment's ACL is initialized by copying the Initial ACL for segments. The input userid and mode (See Usage below) are then added to the ACL of the subdirectory or segment.

Usage

```
declare hcs_$append_branchx entry (char(*), char(*), fixed
        bin (5), (3) fixed bin (6), char(*), fixed bin (1),
        fixed bin (1), fixed bin (24), fixed bin (35));

call hcs_$append_branchx (dirname, entryname, mode, rings,
     userid, dirsw, copysw, bitcnt, code);
```

1) dirname — is the path name of the directory in which entryname is to be placed. (Input)

2) entryname — is the name of the segment or subdirectory to be created. (Input)

3) mode — is the user's access mode; see Notes below. (Input)

4) rings — are the new segment's or subdirectory's ring brackets; see the MPM Subsystem Writers' Guide section, Intraprocess Access Control (Rings). (Input)

5) userid — is the user's access control name of the form Person.Project.Tag. (Input)

6) dirsw — is the branch's directory switch (= 1 if a directory is being created; = 0 otherwise). (Input)

7) copysw — is the segment copy switch (= 1 if a copy is wanted whenever the segment is initiated; = 0 if the original is wanted). (Input)

8) bitcnt — is the segment's length (in bits). (Input)

9) code is a standard storage system status code. (Output)

Notes

Append (a) access mode is required in the directory dirname to add an entry to that directory.

The mode argument is a fixed binary number where the desired mode is encoded with one access mode specified by each bit. For segments the modes are:

read	8-bit (i.e., 01000b)
execute	4-bit (i.e., 00100b)
write	2-bit (i.e., 00010b)

For directories, the modes are:

status	8-bit (i.e., 01000b)
modify	2-bit (i.e., 00010b)
append	1-bit (i.e., 00001b)

Note that if modify access is given for a directory, then status must also be given; i.e., 01010b. The unused bits are reserved for unimplemented attributes and must be zero. For example, rw access in bit form is 01010b, and is 10 in fixed binary form.

 (END)

Name: hcs_$append_link

This subroutine is provided to create a link in the storage system directory hierarchy to some other directory entry in the hierarchy. For a discussion of links see the MPM Reference Guide section Segment, Directory and Link Attributes.

Usage

```
declare hcs_$append_link entry (char(*), char(*) char(*),
        fixed bin(35));

call hcs_$append_link (dir_name, link_name, path, code);
```

1) dir_name is the directory path name in which the link is to be created. (Input)

2) link_name is the entry name of the link to be created. (Input)

3) path is the path name of the segment to which link_name is to point. (Input)

4) code is a standard storage system status code. (Output)

Notes

The user must have the append attribute with respect to the directory in which the link is being created.

The entry pointed to by the link need not exist at the time the link is created.

The subroutines hcs_$append_branch and hcs_$append_branchx may be used to create a segment or directory entry in the storage system hierarchy.

 (END)

hcs_$assign_channel

Entry: hc_ipc$assign_channel

This entry assigns a special event channel and returns its identifier to the caller.

Usage

```
declare hc_ipc$assign_channel entry (fixed bin(71),
        fixed bin(35));

call hc_ipc$assign_channel (channel, code);
```

1) channel is the identifier of the assigned channel. (Output)

2) code is a status code. (Output)

"Special" event channels were created as part of the development of the "fast command loop". Their main benefit is to reduce the number of pages touched when going blocked. (For example, they bypass the ECT). Their main limitation is that no message may be associated with a wakeup directed to them.

hcs_$block

Entry: fast_hc_ipc$ipc_block

This entry point causes the calling process to put itself into the blocked state until a wake-up is received. Then the process runs again, all entries in the Interprocess Transmission Table (ITT) are copied into the Event Channel Table (ECT) of the ring to which they are directed. Control is then returned to the caller of fast_hc_ipc$ipc_block.

Usage

```
declare fast_hc_ipc$ipc_block entry;

call fast_hc_ipc$ipc_block;
```

There are no arguments.

This entry is invoked by ipc_$block when a non-special (see hcs_$assign_channel) channel is specified in the list of channels passed as an argument to it.

hcs_$cpu_time_and_paging_

Subroutine Call
Development System
6/30/72

Name: cpu_time_and_paging_

This procedure returns the total CPU time used by the calling process since it was created as well as two measures of the paging activity of the process.

Usage

(*)
```
declare cpu_time_and_paging_ entry (fixed bin, fixed bin(71),
        fixed bin);

call cpu_time_and_paging_ (pf, time, pp);
```

1) pf is the total number of page faults taken by the calling process. (Output)

2) time is the total cpu time used by the calling process. (Output)

3) pp is the total number of pre-pagings for the calling process. (Output)

Note: The name cpu_time_and_paging_ has been added to the segment hcs_ so that this entry may be called as indicated at (*) above.

(END)

Name: hcs_$chname_file

This subroutine changes an entry name on a storage system entry specified by path name. If an old (i.e., already existing) name is specified, it is deleted from the entry; if an new name is specified it is added. Thus, if only an old name is specified, the effect is to delete a name; if only a new name is specified, the effect is to add a name; and if both are specified, the effect is to rename the entry.

Usage

```
declare hcs_$chname_file entry (char(*), char(*),
        char(*), char(*), fixed bin(35));

call hcs_$chname_file (dir_name, entry_name, oldname,
     newname, code);
```

1) dir_name — is the path name of the directory in which the entry to be manipulated is found. (Input)

2) entry_name — is the name of the entry to be manipulated. (Input)

3) oldname — is the name to be deleted from the entry. It may be a null character string ("") in which case no name is to be deleted. If oldname is null, then newname must not be null. (Input)

4) newname — is the name to be added to the entry. It must not already exist in the directory on this or another entry. It may be a null character string ("") in which case no name is added. If it is null, then oldname must not be the only name on the entry. (Input)

5) code — is a standard storage system status code. It may have the values:

error_table_$nonamerr
error_table_$namedup
error_table_$segnamedup (Output)

Notes

The subroutine hcs_$chname_seg performs the same function, given a pointer to the segment instead of its path name.

The user must have the modify attribute with respect to the directory in question.

Examples

Assume that the entry >my_dir>alpha exists and that it also has the entry name beta. Then the following calls to hcs_$chname_file would have the effects described.

```
call hcs_$chname_file (">my_dir", "alpha", "beta", "gamma",
     code);
```

This call would change the entry name beta to gamma.

```
call hcs_$chname_file (">my_dir", "gamma", "gamma", "",
     code);
```

This call would remove the entry name gamma. Note that any entry name may be used in the second argument position.

```
call hcs_$chname_file (">my_dir", "alpha", "", "delta",
     code);
```

This call would add the entry name delta. The entry now has the names alpha and delta.

 (END)

Subroutine Call
2/13/73

Name: hcs_$chname_seg

This subroutine changes an entry name on a storage system segment, given a pointer to the segment. If an old (i.e., already existing) name is specified, it is deleted from the entry; if a new name is specified, it is added. Thus, if only an old name is specified, the effect is to delete a name; if only a new name is specified, the effect is to add a name; and if both are specified, the effect is to rename the entry.

Usage

```
declare hcs_$chname_seg entry (ptr, char(*), char(*),
        fixed bin(35));

call hcs_$chname_seg (seg_ptr, oldname, newname, code);
```

1) seg_ptr is a pointer to the segment whose name will be changed. (Input)

2) oldname is the name to be deleted from the entry. It may be a null character string ("") in which case no name is to be deleted. If oldname is null, then newname must not be null. (Input)

3) newname is the name to be added to the entry. It must not already exist in the directory on this or another entry. If may be a null character string ("") in which case no name is added. If it is null, then oldname must not be the only name on the entry. (Input)

4) code is a standard storage system status code. It may have the values:

```
error_table_$namedup
error_tab_$nonemerr
error_table_$segnamedup (Output)
```

Notes

The subroutine hcs_$chname_file performs the same function, given the directory and entry names of the segment instead of the pointer.

The user must have the modify attribute with respect to the directory in question.

Examples

Assume that the user has a pointer, seg_ptr, to a segment which has two entry names, alpha and beta. Then the following calls to hcs_$chname_seg would have the effects described.

```
call hcs_$chname_seg (seg_ptr, "beta", "gamma", code);
```

This call would change the entry name beta to gamma.

```
call hcs_$chname_seg (seg_ptr, "gamma", "", code);
```

This call would remove the entry name gamma.

```
call hcs_$chname_seg (seg_ptr, "", "delta", code);
```

This call would add the entry name delta. The entry now has the names alpha and delta.

 (END)

Subroutine Call
2/21/73

Name: hcs_$del_dir_tree

This subroutine deletes a subtree of the storage system hierarchy, given the path name of a directory. All segments, links and directories inferior to that directory are deleted including the contents of any inferior directories. The specified directory is not itself deleted; to delete it, see the MPM write-ups for hcs_$delentry_file and hcs_$delentry_seg.

Usage

```
declare hcs_$del_dir_tree entry (char(*), char(*),
        fixed bin(35));

call hcs_$del_dir_tree (parent_name, dir_name, code);
```

1) parent_name is the path name of the parent directory of the directory whose subtree is to be deleted. (Input)

2) dir_name is the entry name of the directory whose subtree is to be deleted. (Input)

3) code is a standard storage system status code.

Note

The user must have the status and modify attributes with respect to the specified directory and the safety switch must be off in that directory. If the user does not have status and modify attributes on inferior directories, hcs_$del_dir_tree will provide them.

If an entry in an inferior directory gives the user access only in a ring lower than his validation level, that entry will not be deleted and no further processing will be done on the subtree. For those users who need to know about rings, they are discussed in the MPM Subsystem Writers' Guide section, Intraprocess Access Control (Rings).

 (END)

Subroutine Call
3/15/73

Name: hcs_$delentry_file

This subroutine, given a directory name and an entry name, deletes the given entry from its parent directory. If the entry is a segment the contents of the segment are deleted first. If the entry specifies a directory which contains entries the status code error_table_$fulldir is returned and hcs_$del_dir_tree must be called to remove the contents of the directory.

Usage

```
declare hcs_$delentry_file (char(*), char(*),
        fixed bin(35));

call hcs_$delentry_file (dirname, ename, code);
```

1) dirname is the parent directory name. (Input)

2) ename is the entry name to be deleted. (Input)

3) code is a standard storage system status code. (Output)

Notes

The subroutine hcs_$delentry_seg performs the same function, given pointer to the segment instead of the pathname.

The user must have modify permission with respect to dirname. If ename specifies a segment or directory rather than a link, the safety switch of the segment or directory must be off. For a temporary period the user must have write permission with respect to the segment or modify permission with respect to the directory being deleted.

 (END)

Name: hcs_$delentry_seg

This subroutine, given the pointer to a segment, deletes the corresponding entry from its parent directory. If the entry is a segment the contents of the segment are deleted first. If the entry specifies a directory which contains entries the status code error_table_$fulldir is returned and hcs_$del_dir_tree must be called to remove the contents of the directory.

Usage

```
declare hcs_$delentry_seg (ptr, fixed bin(35));

call hcs_$delentry_seg (sp, code);
```

1) sp is the pointer to the segment to be deleted. (Input)

2) code is a standard storage system status code. (Output)

Notes

The subroutine hcs_$delentry_file performs the same function, given the directory and entry names of the segment instead of the pointer.

The user must have modify permission with respect to the segment's parent directory. The safety switch of the segment must be off. For a temporary period the user must have write permission with respect to the segment.

 (END)

Name: hcs_$delete_acl_entries

This subroutine is called to delete specified entries from an Access Control List (ACL) for a segment.

Usage

```
declare hcs_$delete_acl_entries entry (char(*), char(*),
        ptr, fixed bin, fixed bin(35));

call hcs_$delete_acl_entries (dirname, ename, acl_ptr,
     acl_count, code);
```

1) dirname is the directory portion of the path name of the segment in question. (Input)

2) ename is the entry name portion of the path name of the segment in question. (Input)

3) acl_ptr points to a user-filled delete_acl structure. See Notes below. (Input)

4) acl_count contains the number of ACL entries in the delete_acl structure. See Notes below. (Input)

5) code is a standard status code. (Output)

Notes

The following structure is used:

```
declare 1 delete_acl (acl_count) aligned based (acl_ptr),
          2 access_name char(32),
          2 status_code fixed bin(35);
```

1) access_name is the access name (in the form of person.project.tag) which identifies the ACL entry to be deleted.

2) status_code is a standard status code for this ACL entry only.

If code is returned as error_table_$argerr then the offending ACL entries in the delete_acl structure will have status_code set to an appropriate error and no processing will have been performed.

If an access name cannot be matched to one existing on the segment's ACL then the status code of that ACL entry is set to error_table_$user_not_found, processing continues to the end of the delete_acl structure and code is returned as zero.

 (END)

hcs_$delete_channel

Entry: hc_ipc$delete_channel

This entry frees a special event channel for use by another procedure.

Usage

```
declare hc_ipc$delete_channel entry (fixed bin(71),
        fixed bin(35));

call hc_ipc$delete_channel (channel, code);
```

1) channel — is the identifier of the channel to be freed. (Input)

2) code — is a status code. (Output)

See hcs_$assign_channel

Name: hcs_$delete_dir_acl_entries

This subroutine is used to delete specified entries from an Access Control List (ACL) for a directory. The delete_acl structure used by this subroutine is described in the MPM write-up for hcs_$delete_acl_entries.

Usage

```
declare hcs_$delete_dir_acl_entries entry (char(*),
        char(*), ptr, fixed bin, fixed bin(35));

call hcs_$delete_dir_acl_entries (dirname, ename, acl_ptr,
     acl_count, code);
```

1) dirname — is the path name of the directory superior to the one in question. (Input)

2) ename — is the entry name of the directory in question. (Input)

3) acl_ptr — points to a user-filled delete_acl structure. (Input)

4) acl_count — is the number of ACL entries in the delete_acl structure. (Input)

5) code — is a standard status code. (Output)

Note

The status code is interpreted as described in hcs_$delete_acl_entries.

 (END)

Name: hcs_$delete_dir_inacl_entries

This subroutine is used to delete specified entries from an Initial Access Control List (Initial ACL) for new directories within the specified directory. The delete_acl structure used by this subroutine is described in the MPM write-up for hcs_$delete_inacl_entries.

Usage

```
declare hcs_$delete_dir_inacl_entries entry (char(*),
        char(*), ptr, fixed bin, fixed bin, fixed bin(35));

call hcs_$delete_dir_inacl_entries (dirname, ename, acl_ptr,
     acl_count, ring, code);
```

1) dirname — is the path name of the directory superior to the one in question. (Input)

2) ename — is the entry name of the directory in question. (Input)

3) acl_ptr — points to a user-filled delete_acl structure. (Input)

4) acl_count — is the number of Initial ACL entries in the delete_acl structure. (Input)

5) ring — is the ring number of the Initial ACL. (Input)

6) code — is a standard status code. (Output)

Note

The status code is interpreted as described in hcs_$delete_inacl_entries.

 (END)

Subroutine Call
2/27/73

Name: hcs_$delete_inacl_entries

This subroutine is called to delete specified entries from an Initial Access Control List (Initial ACL) for new segments within the specified directory.

Usage

```
declare hcs_$delete_inacl_entries entry (char(*), char(*),
        ptr, fixed bin, fixed bin, fixed bin(35));

call hcs_$delete_inacl_entries (dirname, ename, acl_ptr,
     acl_count, ring, code);
```

1) dirname is the superior directory portion of the path name of the directory in question. (Input)

2) ename is the entry name portion of the path name of the directory in question. (Input)

3) acl_ptr points to a user-filled delete_acl structure. See Notes below. (Input)

4) acl_count contains the number of ACL entries in the delete_acl structure. See Notes below. (Input)

5) ring is the ring number of the Initial ACL. (Input)

6) code is a standard status code. (Output)

Notes

The following structure is used:

```
declare 1 delete_acl (acl_count) aligned based (acl_ptr),
          2 access_name char(32),
          2 status_code fixed bin(35);
```

1) access_name is the access name (in the form of person.project.tag) which identifies the Initial ACL entry to be deleted.

2) status_code is a standard status code for this Initial ACL entry only.

If code is returned as error_table_$argerr then the offending Initial ACL entries in the delete_acl structure will have status_code set to an appropriate error and no processing will have been performed.

If an access name cannot be matched to one existing on the Initial ACL then the status code of that Initial ACL entry is set to error_table_$user_not_found, processing continues to the end of the delete_acl structure and code is returned as zero.

 (END)

hcs_$fblock

Entry: fast_hc_ipc$ipc_f_block

This entry causes the process to put itself into the blocked state until a wake-up is received and to empty the ITT. It also returns information about special wake-ups.

Usage

```
declare fast_hc_ipc$ipc_f_block entry (bit(36) aligned,
        bit(1) aligned);

call fast_hc_ipc$ipc_f_block (special_events, call_regular);
```

1) special_events — is a string of bits indicating on which special channel wake-ups are pending. The bits corresponding to channels with wake-ups pending are turned on, the others are left unchanged. (Input/Output)

2) call_regular — is set to "1"b if any messages for the calling ring were emptied from the ITT. (Output)

This entry is called by ipc_$block if only special channels are specified in the list of channels passed to it as an argument (see hcs_$assign_channel)

To avoid the loss of wakeups, this entry should not be called by any procedure other than ipc_$fblock

Name: hcs_$fs_get_mode

This subroutine returns the access mode of the user, at the current validation level, with respect to a specified segment. For a discussion of access modes, see the MPM Reference Guide section, Access Control.

Usage

```
declare hcs_$fs_get_mode entry (ptr, fixed bin(5),
        fixed bin(35));

call hcs_$fs_get_mode (segptr, mode, code);
```

1) segptr is an pointer to the segment in question. (Input)

2) mode is the mode (see Notes below). (Output)

3) code is a standard storage system status code. (Output)

Notes

The mode and ring brackets for the segment in the user's address space are used in combination with the user's current validation level to determine the mode the user would have if he accessed this segment. For a discussion of ring brackets and validation level, see the MPM Subsystem Writers' Guide section, Intraprocess Access Control (Rings).

The mode argument is a fixed binary number where the desired mode is encoded with one access mode specified by each bit. For segments the modes are:

read	8-bit (i.e., 01000b)
execute	4-bit (i.e., 00100b)
write	2-bit (i.e., 00010b)

For directories, the modes are:

status	8-bit (i.e., 01000b)
modify	2-bit (i.e., 00010b)
append	1-bit (i.e., 00001b)

Note that if modify access is given for a directory, then status must also be given; i.e., 01010b). The high-order bit is reserved for an unimplemented attribute and must be zero. For example, rw access in bit form is 01010b, and is 10 in fixed binary form.

 (END)

Subroutine Call
2/28/73

Name: hcs_$fs_get_path_name

This entry, given a pointer to a segment, returns a path name for the segment, with the directory and entry name portions of the path name separated. The entry name returned is the primary name on the entry; see the MPM Reference Guide section, Segment, Directory and Link Attributes for a discussion of primary names.

Usage

```
declare hcs_$fs_get_path_name entry (ptr, char(*),
        fixed bin, char(*), fixed bin(35));

call hcs_$fs_get_path_name (segptr, dirname, ldn, ename,
     code);
```

1) segptr — is a pointer to the segment in question. (Input)

2) dirname — is the path name of the directory superior to the segment pointed to by segptr. If the length of the path name to be returned is greater than the length of dirname, the path name will be truncated. To avoid this problem, the length of dirname should be 168 characters. (Output)

3) ldn — is the number of nonblank characters in dirname. (Output)

4) ename — is the primary entry name of the segment pointed to by segptr. If the length of the entry name to be returned is greater than the length of ename, the entry name will be truncated. To avoid this problem, the length of ename should be 32 characters. (Output)

5) code — is a standard storage system status code. (Output)

 (END)

Name: hcs_$fs_get_ref_name

This entry point returns a specified (i.e., first, second, etc.) reference name for a specified segment.

Usage

```
declare hcs_$fs_get_ref_name entry (ptr, fixed bin,
        char(*), fixed bin);

call hcs_$fs_get_ref_name (segptr, count, rname, code);
```

1) segptr — is a pointer to the segment in question. (Input)

2) count — specifies which reference name is to be returned. See Notes. (Input)

3) rname — is the desired reference name. (Output)

4) code — is a standard file system status code. (Output)

Notes

If "count" = 1, the name by which the segment has most recently been made known will be returned. If "count" = 2, the second most recently added name is returned and so on. If "count" is larger than the total number of names, the name by which the segment was originally made known is returned and "code" is set to error_table_$ref_count_too_big.

See the MPM Reference Guide Section on naming conventions.

(END)

Entry: hcs_$fs_get_seg_ptr

Given a reference name of a segment, hcs_$fs_get_seg_ptr returns a pointer to the base of the segment. For a discussion of reference names, see the MPM Reference Guide section, Constructing and Interpreting Names.

Usage

```
declare hcs_$fs_get_seg_ptr entry (char(*), ptr,
        fixed bin(35));

call hcs_$fs_get_seg_ptr (rname, segptr, code);
```

1) rname is the reference name of a segment for which a pointer is to be returned. (Input)

2) segptr is a pointer to the base of the segment. (Output)

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

Note

If the reference name is accessible from the user's current validation level, segptr is returned pointing to the segment; otherwise, it is null. The user who needs to know about rings and validation levels can find a discussion of them in the Subsystem Writers' Guide section, Intraprocess Access Control (Rings).

 (END)

Name: hcs_$fs_move_file

This subroutine moves the data associated with one segment in the storage system hierarchy to another segment given the path names of the segments in question. The old segment remains, with a zero length.

Usage

```
declare hcs_$fs_move_file entry (char(*), char(*),
        fixed bin(2), char(*), char(*), fixed bin(35));

call hcs_$fs_move_file (from_dir, from_entry, at_sw, to_dir,
     to_entry, code);
```

1) from_dir — is the path name of the directory in which from_entry resides. (Input)

2) from_entry — is the entry name of the segment from which data is to be moved. (Input)

3) at_sw — see Notes below. (Input)

4) to_dir — is the path name of the directory in which to_entry resides. (Input)

5) to_entry — is the entry name of the segment to which data is to be moved. (Input)

6) code — is a standard storage system status code. It may have the value error_table_$no_move if either entry is not a segment, or one of the values described in Notes below.

Notes

The input argument at_sw is a 2-indicator switch which directs the procedure to use certain options. The two options specified are append option and truncate option. If the append option (high-order bit) is on, then append to_entry to to_dir if it does not already exist. If the append option is off and the destination entry can not be found the status code error_table_$noentry is returned.

If the truncate option (low-order bit) is on, to_entry is truncated if it is not zero length. Otherwise (i.e., if the option is off and the length of to_entry is not zero) the status code error_table_$clnzero is returned. In both of the cases where the move is not completed, the procedure will attempt to return the data to the original segment.

The subroutine hcs_$fs_move_seg performs the same function given pointers to the segments in question instead of path names.

 (END)

Name: hcs_$fs_move_seg

This subroutine moves the data associated with one segment in the hierarchy to another segment, given pointers to the segments in question. The old segment remains, with a zero length.

Usage

```
declare hcs_$fs_move_seg entry (ptr, ptr, fixed bin(1),
        fixed bin(35));

call hcs_$fs_move_seg (from_ptr, to_ptr, trunsw, code);
```

1) from_ptr — is the pointer to the segment from which data is to be moved. (Input)

2) to_ptr — is the pointer to the target segment. (Input)

3) trunsw — if equal to 1, then truncate the segment specified by to_ptr (if it is not already zero-length) before performing the move;
if equal to 0, then reflect the status code error_table_$clnzero if that segment is not already zero-length. (Input)

4) code — is a standard storage system status code. Besides the value given under trunsw above, it may also have the value error_table_$no_move. (Output)

Note

The subroutine hcs_$fs_move_file performs the same function given the path names of the segments in question instead of the pointers.

 (END)

hcs_$get_alarm_timer

Entry: set_alarm_timer$get_alarm_timer

This entry is called to determine the setting of the calling process' alarm clock. (real time clock)

Usage

```
declare set_alarm_timer$get_alarm_timer entry
        (fixed bin(71), fixed bin(71));
```

```
call set_alarm_timer$get_alarm_timer (time, channel);
```

1) time — Is the absolute (calendar clock time) of the alarm clock setting for the calling process. If this time is zero, the alarm clock is not set up for the calling process. (Output)

2) channel — Is the event channel over which the alarm clock wakeup will be sent. (Output)

This is an internal interface to the procedure timer_manager_

Subroutine Call
3/15/73

Name: hcs_$get_author

This subroutine returns the author of a segment or a link.

Usage

```
declare hcs_$get_author entry (char(*), char(*), fixed
        bin(1), char(*), fixed bin(35));

call hcs_$get_author (dirname, entry, chase, author, code);
```

1) dirname is the path name of the directory containing entry. The path name can have a maximum length of 168 characters. (Input)

2) entry is the name of the entry. It can have a maximum length of 32 characters. (Input)

3) chase if entry refers to a link, this flag indicates whether to return the author of the link or the author of the segment to which the link points:

   0 = return link author;
   1 = return segment author. (Input)

4) author is the author of the segment or link in the form of Doe.Student.a with a maximum length of 32 characters. (Output)

5) code is a standard storage system status code. (Output)

Note

The user must have status permission on the parent directory.

 (END)

Name: hcs_$get_bc_author

This subroutine returns the bit count author of a segment or directory. The bit count author is the name of the user who last set the bit count of the segment or directory.

Usage

```
declare hcs_$get_bc_author entry (char(*), char(*),
        char(*), fixed bin(35));

call hcs_$get_bc_author (dirname, ename, bc_author, code);
```

1) dirname — is the directory name of the segment whose bit count author is wanted. (Input)

2) ename — is the entry name of the segment whose bit count author is wanted. (Input)

3) bc_author — is the bit count author of the segment in the form of Doe.Student.a. (Output)

4) code — is a standard storage system status code. (Output)

Note

The user must have status permission on the directory containing the segment.

 (END)

Subroutine Call
3/1/73

Name: hcs_$get_dir_ring_brackets

This subroutine, given the path name of the superior directory and the name of the directory, will return that directory's ring brackets.

Usage

```
declare hcs_$get_dir_ring_brackets entry (char(*), char(*),
        (2) fixed bin(3), fixed bin(35));

call hcs_$get_dir_ring_brackets (dirname, ename, drb, code);
```

1) dirname is the path name of the superior directory. (Input)

2) ename is the entry name of the directory in question. (Input)

3) drb is a 2-element array to contain the directory's ring brackets. (Output)

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

Notes

The user must have status permission to dirname in order to list the directory's ring brackets.

Ring brackets are discussed in the MPM Subsystem Writers' Guide section, Intraprocess Access Control (Rings).

 (END)

hcs_$get_initial_ring

Entry: proc_info$get_initial_ring

This entry point returns the value of the ring number in which the process was initialized.

Usage

```
declare proc_info$get_initial_ring entry (fixed bin);

call proc_info$get_initial_ring (iring);
```

1) iring is the value of the process's initial ring. (Output)

The initial ring is the ring to which control is tranfered to when a newly created process transfers out of ring 0.

hcs_$get_ips_mask

Entry: ips_$get_ips_mask

This entry is used to find the current value of the IPS mask.

Usage

```
declare ips_$get_ips_mask entry (bit(36) aligned);

call ips_$get_ips_mask (oldmask);
```

1) oldmask is the current value of the IPS mask. (Output)

IPS stands for inter-process signal. It is a means by which a ~~current~~ process may be interrupted by a (potentially) different process. There are currently 3 ips interrupts:

quit - normally occurs because a user has hit the ATTN button on his console

alrm - occurs when the real time timer goes off (if the user has specified that he wants to be interrupted when this occurs)

cput - occurs when the cpu time timer goes off. (if the user has specified that he wants to be interrupted when this occurs)

The ips mask is a user-settable entity which allows one to selectively mask ips interrupts. It exists mainly as an internal interface to timer_manager_. The interface is not well thought out and should probably be redone (integrated with a new ~~ips~~ IPC facility)

Name: hcs_$get_max_length

This subroutine returns the max length of a segment given a directory name and entry name. The max length is the length beyond which the segment may not grow.

Usage

```
declare hcs_$get_max_length entry (char(*), char(*),
        fixed bin(18), fixed bin(35));

call hcs_$get_max_length (dirname, ename, max_length, code);
```

1) dirname is the directory name of the segment whose max length is wanted. (Input)

2) ename is the entry name of the segment whose max length is wanted. (Input)

3) max_length is the max length of the segment in words. (Output)

4) code is a standard storage system status code. (Output)

Note

The user must have status permission on the directory containing the segment.

 (END)

hcs_$get_page_trace

Internal Interface
Hardcore Ring
02/16/71

Entry: get_page_trace

This entry returns information about recent paging activity.

Usage

```
declare get_page_trace entry (ptr, fixed bin);

call get_page_trace (datap, count);
```

1) datap — Is a pointer to a user area where return information will be stored. (Input)

2) count — Is the number of page faults to be traced. (Input)

Notes

If count is less than or equal to 0 or greater than 200, then 200 page faults will be traced, i.e., the information for the last 200 page faults will be returned. Otherwise the last "count" page faults will be traced. The trace information is stored with the following data structure declaration:

```
declare 1 trace (200) based (datap) aligned,
          2 page_no fixed bin,
          2 seg_no fixed bin,
          2 delta_time fixed bin(71);
```

1) page_no — Is the page number of the segment which caused the page fault. Counting starts at 0, i.e., the first page is page 0.

2) seg_no — Is the segment number of the segment which caused the page fault.

3) delta_time — Is the real time in microseconds since the last page fault, i.e., delta_time(i) is the time from page fault i-1 to page fault i.

Note that the pointer datap should point at an even location. The trace information is stored starting at "trace(1)" and continues to "trace(count)", i.e., the most recent page fault is reflected in "trace(count)".

Name: hcs_$get_process_usage

This subroutine returns information about a process's usage of Multics since it was created. It provides data about processor and memory usage.

Usage

```
declare hcs_$get_process_usage entry (ptr, fixed bin(35));

call hcs_$get_process_usage (info_pointer, code);
```

1) info_pointer — is a pointer to the structure in which process information is returned (see Notes below). (Input)

2) code — is a standard status code. (Output)

Notes

The format of the structure based on info_pointer is:

```
declare 1 process_usage,
          2 number_wanted fixed bin,
          2 cpu_time_used fixed bin(71),
          2 memory_usage fixed bin(71),
          2 number_of_page_faults fixed bin(35),
          2 amount_of_prepaging fixed bin(35),
          2 process_virtual_time fixed bin(71);
```

1) number_wanted — is set by the calling program to specify the number of other entries in the structure to be filled in. The entry itself (the numbers wanted) is not included in this count. The value 5 would cause five entries listed below to be filled in. A smaller number, n, will cause the first n entries to be filled in. (Input)

2) cpu_time_used — is set to the amount of processor time (in microseconds) used by the calling process. (Output)

3) memory_usage — is a measure of the primary (core) memory used by this process. The units of memory usage are page-seconds, normalized to account for the size of primary memory actually in use. (Output)

4) number_of_page_faults — is set to the number of demand page faults this process has taken. (Output)

5) amount_of_prepaging — is the number of pages prepaged for this process. (Output)

6) process_virtual_time — is the amount of processor time (in microseconds) used exclusive of page fault and system interrupt processing time. (Output)

 (END)

Name: hcs_$get_ring_brackets

This subroutine, given the directory name and entry name of a nondirectory segment will return that segment's ring brackets.

Usage

```
declare hcs_$get_ring_brackets entry (char(*), char(*),
        (3) fixed bin(3), fixed bin(35));

call hcs_$get_ring_brackets (dirname, ename, rb, code);
```

1) dirname — is the directory portion of the path name of the segment in question. (Input)

2) ename — is the entry name of the segment in question. (Input)

3) rb — is a 3-element array to contain the segment ring brackets. (Output)

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

Notes

The user must have status permission to dirname in order to list a segment's ring brackets.

Ring brackets are discussed in the MPM Subsystem Writers' Guide section, Intraprocess Access Control (Rings).

 (END)

Name: hcs_$get_safety_sw

This subroutine returns the safety switch of a directory or a segment, given a directory name and an entry name.

Usage

```
declare hcs_$get_safety_sw entry (char(*), char(*), bit(1),
        fixed bin(35));

call hcs_$get_safety_sw entry (dirname, ename, safety_sw,
     code);
```

1) dirname — is the directory name of the segment whose safety switch is wanted. (Input)

2) ename — is the entry name of the segment whose safety switch is wanted. (Input)

3) safety_sw — is the value of the segment's safety switch.

= "0"b if the segment may be deleted.
= "1"b if the segment may not be deleted. (Output)

4) code — is a standard storage system status code. (Output)

Note

The user must have status permission with respect to the directory containing the segment.

 (END)

Subroutine Call
Development System
06/14/71

Name: hcs_$get_search_rules

This entry returns the search rules currently in use in the caller's process.

Usage

```
declare hcs_$get_search_rules entry (ptr);

call hcs_$get_search_rules (search_rules_ptr);
```

1) search_rules_ptr is a pointer to a user supplied search rules structure. (Input)

Notes

The search rule structure is declared as follows:

```
declare 1 search_rules,
          2 number fixed bin,
          2 names (21) char(168) aligned;
```

1) number is the number of search rules.

2) names are the names of the search rules.

(END)

hcs_$get_user_effmode

Entry: status_$get_user_effmode

This entry point returns the effective mode of a segment for a user in a specified ring. The segment, user, and ring are all specified by arguments.

Usage

```
declare status_$get_user_effmode entry (char(*), char(*),
        char(*), fixed bin, fixed bin(5), fixed bin(35));

call status_$get_user_effmode (dirname, entry, user,
     ring, mode, code);
```

1) dirname — is the path name of the directory containing entry. The path name can have a maximum length of 168 characters. (Input)

2) entry — is the entry name of the segment about which information is requested. The entry name can have a maximum length of 32 characters. (Input)

3) user — is the name of the user whose effective access to dirname>entry is requested. The name should be of the form "Doe.Multics.a". The name can have a maximum length of 32 characters. (Input)

4) ring — is the ring number for which the effective mode is to be computed. (Input)

5) mode — is the effective mode of the user in the ring requested. (Output)

6) code — is a returned status code. (Output)

hcs_$high_low_seg_count

Entry: unsnap_service$high_low_seg_count

This entry merely returns the highest segment number used and the hardcore segment count.

Usage

```
declare hcs_$unsnap_service_high_low entry (fixed bin,
        fixed bin);

call hcs_$unsnap_service_high_low (high_seg, hcscnt);
```

1) high_seg — highest segment number used. (Output)

2) hcscnt — hardcore-segment count. (Output)

Name: hcs_$initiate

This subroutine is used to search for a segment, make a copy of it if the copy switch so indicates, and make the segment or its copy known to the process. The reference name specified is entered in the address space of the process and a pointer to the segment is returned. If segsw is on, then the segment pointer is input and the segment is made known with that segment number.

Usage

```
declare hcs_$initiate entry (char(*), char(*), char(*),
        fixed bin(1), fixed bin(2), ptr, fixed bin(35));

call hcs_$initiate (pname, ename, rname, segsw, copysw,
     segptr, code);
```

1) pname — is the path name of the directory containing the segment. (Input)

2) ename — is the entry name of the segment. (Input)

3) rname — is the reference name. If it is zero in length, the segment is initiated by a null name. (Input)

4) segsw — is the reserved segment switch:

   = 0 if no segment number has been reserved;
   = 1 if a segment number was reserved. (Input)

5) copysw — is the copy switch:

   = 0 if it is desired to go by the setting in the directory entry;
   = 1 if no copy is wanted;
   = 2 if a copy is always wanted. (Input)

6) segptr — is a pointer to the segment. (It is Input if segsw = 1. Otherwise, it is Output.)

7) code — is a standard storage system status code. (Output)

Notes

The user must have non-null access on the segment ename in order to initiate it.

If ename cannot be initiated, a null pointer is returned for segptr and the returned value of code indicates the reason for failure. If ename is already known to the user's process, code is returned as error_table_$segknown and the the argument segptr will contain a valid pointer to ename. If ename is not already known, and no problems are encountered, segptr will contain a valid pointer and code will be zero.

 (END)

Name: hcs_$initiate_count

This subroutine, given a path name and a reference name, causes the segment defined by the path name (or a copy of it, depending upon the copysw option) to be made known by the given reference name. A segment number is assigned and returned as a pointer and the bit count of the segment is returned.

Usage

```
declare hcs_$initiate_count entry char(*), char(*), char(*),
        fixed bin(24), fixed bin(2), ptr, fixed bin(35));

call hcs_$initiate_count (pname, ename, rname, bitcount,
     copysw, segptr, code);
```

1) pname — is the directory portion of the path name of the segment in question. (Input)

2) ename — is the entry name portion of the path name of the segment in question. (Input)

3) rname — is the desired reference name. If it is zero in length, the segment is initiated by a null name. (Input)

4) bitcount — is the bit count of the segment. (Output)

5) copysw — is the copy switch:

   = 0 if it is desired to go by the setting in the hierarchy entry;
   = 1 if no copy is wanted;
   = 2 if a copy is always wanted. (Input)

6) segptr — is a pointer to the segment in question. (Output)

7) code — is a standard storage system status code. (Output)

Notes

The user must have non-null access on the segment ename in order to initiate it.

If ename cannot be initiated, a null pointer is returned for segptr and the returned value of code indicates the reason for failure. If ename is already known to the user's process, code is returned as error_table_$segknown and the argument segptr will contain a valid pointer to ename. If ename is not already known, and no problems are encountered, segptr will contain a valid pointer and code will be zero.

See also the MPM Reference Guide section, Constructing and Interpreting Names.

 (END)

Subroutine Call
Development System
12/15/71

Name: hcs_$initiate_search_rules

This is a supervisor entry which is mainly used by the set_search_rules and set_search_dirs commands. It also provides the user with a means of specifying the search rules which he wishes to use in his process. (For more information on search rules, see the appropriate MPM Reference Guide Section.)

Usage

```
declare hcs_$initiate_search_rules entry (ptr, fixed bin);

call hcs_$initiate_search_rules (search_rule_pointer,
     code);
```

1) search_rule_pointer — is a pointer to a structure containing the new search rules. (Input)

2) code — is a standard return status code. (Output)

Notes

The structure pointed to by search_rule_pointer is declared as follows:

```
declare 1 sr aligned,
          2 num fixed bin,
          2 names (21) char(168) aligned;
```

1) num — is the number of entries. The current maximum is 21 but the user need only disclose the maximum that he will use.

2) names — are the names of the search rules. They may be absolute pathnames or key words.

Search rules may be either absolute pathnames of directories or key words. The allowed search rules are:

pathname — the absolute pathname of a directory to be searched;

(key words)

initiated_segments — search for the already initiated segment;

referencing_dir	search the parent directory of the module making the reference;
working_dir	search the working directory;
process_dir	search the process directory;
home_dir	search the login or home directory;
default	return to the default search rules;
system_libraries	insert the default system libraries at this point in the search rules;
set_search_directories	insert the following directories after working_dir in the default search rules and make the result the current search rules.

The key word "default" cannot be used with any other code word or pathname as it returns the default rules and exits immediately.

The search rules can be changed when the procedure is called with different rules or the process is terminated.

Errors returned from this routine are:

error_table_$bad_string (not a pathname or code word)
error_table_$notadir
error_table_$too_many_sr

Additional file system errors may be returned from other routines which are called from hcs_$initiate_search_rules.

(END)

hcs_$ ioam_list

Name: ioam_

This procedure implements the pageable portion of the hardcore I/O Assignment Manager (IOAM). (See also dstm_.) It contains entry points which are called by Device Interface Modules (DIMs) to assign, unassign, and verify devices and is called from the user ring to list the device status and free assigned devices.

Entry: ioam_$ioam_device_list

This entry returns a list of all devices currently assigned to the calling process. One or more device names will be given for each device.

The format of the list is a block of data for each device as follows:

```
declare 1 device_info based (p) aligned,
          2 number_of_names fixed bin,
          2 name (number_of_names) char(32);
```

The blocks are packed adjacently. The first block is pointed to by datap.

Usage

```
declare ioam_$ioam_device_list entry (ptr, ptr,
        fixed bin);

call ioam_$ioam_device_list (areap, datap,
     device_count);
```

1) areap — points to an area in which allocation for the list will be done. (Input)

2) datap — points to the allocated list of devices. (Output)

3) device_count — is the number of devices. (Output)

hcs_$ioam_release

Entry: ioam_$ioam_release

This entry forces a specified device to become free if assigned to the calling process.

Usage

```
declare ioam_$ioam_release entry (char(*), fixed bin);

call ioam_$ioam_release (device_name, code);
```

1) device_name — is the name of the device to be released. (Input)

2) code — is a returned error code:

error_table_$ioname_not_found: device name unknown;

error_table_$io_no_permission: device not assigned to this process;

or any codes returned from the device detach handler. (Output)

hcs_$ioam_status

Entry: ioam_$ioam_status

This entry returns information about a specified device.

Usage

```
declare ioam_$ioam_status entry (char(*), char(*),
        fixed bin);

call ioam_$ioam_status (device_name, message, code);
```

1) device_name — is the name of the device in question. (Input)

2) message — is a character string description of the status of the device. It can have a maximum length of 32 characters. (Output)

3) code — is a numerical representation of the device status. (Output)

Notes

The status of the device and the returned values are as follows:

1) No meaning can be attached to the given device_name.

```
message = "unknown"; code = error_table_$ioname_not_found;
```

2) Device not assigned to any process.

```
message = "unassigned"; code = error_table_$dev_nt_assnd;
```

3) Device assigned to this process.

```
message = "assigned"; code = 0;
```

4) Device assigned to some other process.

```
message = "assigned to other process";
code = error_table_$already_assigned;
```

hcs_$ipc_init

Entry: hc_ipc$ipc_init

This entry is called once per-process per-ring to inform hc_ipc of the location of the ECT for that ring.

Usage

```
declare hc_ipc$ipc_init entry (ptr);

call hc_ipc$ipc_init (ect_ptr);
```

1) ect_ptr points to the ECT for the ring from which it is called. (input)

hcs_$level_get

Entry: level$get

This entry obtains the current validation level.

Usage

```
declare level$get entry (fixed bin);

call level$get (ring_no);
```

1) ring_no will contain the current validation level. (Output)

hcs_$ level_set

Entry: level$set

This entry points sets the validation level.

Usage

```
declare level$set entry (fixed bin);

call level$set (ring_no);
```

1) ring_no contains the value to be used as the new validation level. (Input)

Name: hcs_$list_acl

This subroutine is used to either list the entire Access Control List (ACL) of a segment or to return the access modes from specified entries. The segment_acl structure used by this subroutine is described in the MPM write-up for hcs_$add_acl_entries.

Usage

```
declare hcs_$list_acl entry(char(*), char(*), ptr, ptr,
        ptr, fixed bin, fixed bin(35));

call hcs_$list_acl (dirname, ename, area_ptr, area_ret_ptr,
     acl_ptr, acl_count, code)
```

1) dirname — is the directory portion of the path name of the segment in question. (Input)

2) ename — is the entry name portion of the path name of the segment in question. (Input)

3) area_ptr — points to an area into which the list of ACL entries is to be allocated. (Input)

4) area_ret_ptr — points to the start of the allocated list of ACL entries. (Output)

5) acl_ptr — if area_ptr is null then acl_ptr is assumed to point to an ACL structure, segment_acl, into which mode information is to be placed for the access names specified in that same structure. (Input)

6) acl_count — is the number of entries in the ACL structure identified by acl_ptr (Input); or is set to the number of entries in the segment_acl structure allocated in the area pointed to by area_ptr, if area_ptr is not null. (Output)

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

Note

If acl_ptr is used to obtain modes for specified access names (rather than obtaining modes for all access names on a segment), then each ACL entry will either have a zero code and will contain the segment's mode or will have code set to error_table_$user_not_found and will contain a zero mode.

 (END)

Name: hcs_$list_dir_acl

This subroutine is used to either list the entire Access Control List (ACL) of a directory or to return the access modes for specified entries. The dir_acl structure described in hcs_$add_dir_acl_entries is used by this subroutine.

Usage

```
declare hcs_$list_dir_acl entry (char(*), char(*), ptr,
        ptr, ptr, fixed bin, fixed bin(35));

call hcs_$list_dir_acl (dirname, ename, area_ptr,
     area_ret_ptr, acl_ptr, acl_count, code);
```

1) dirname — is the path name of the directory superior to the one in question. (Input)

2) ename — is the entry name of the directory in question. (Input)

3) area_ptr — points to an area into which the list of ACL entries is to be allocated. (Input)

4) area_ret_ptr — points to the start of the list of the ACL entries. (Output)

5) acl_ptr — if area_ptr is null then acl_ptr is assumed to point to an ACL structure, dir_acl, into which mode information is to be placed for the access names specified in that same structure. (Input)

6) acl_count — is either the number of entries in the ACL structure identified by acl_ptr (Input); or if area_ptr is not null, then it is set to the number of entries in the dir_acl structure that has been allocated. (Output)

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

Note

If acl_ptr is used to obtain modes for specified access names (rather than obtaining modes for all access names on a segment), then each ACL entry will either have a zero code and will contain the directory's mode or will have code set to error_table_$user_not_found and will contain a zero mode.

 (END)

Note

If acl_ptr is used to obtain modes for specified access names (rather than obtaining modes for all access names on the Initial ACL), then each Initial ACL entry will either have a zero status_code and will contain the directory's mode or will have status_code set to error_table_$user_not_found and will contain a zero mode.

 (END)

Subroutine Call
2/27/73

Name: hcs_$list_inacl

This subroutine is used to either list the entire Initial Access Control List (Initial ACL) for new segments within the specified directory, or to return the access modes from specified entries. The segment_acl structure used by this subroutine is described in the MPM write-up for hcs_$add_inacl_entries.

Usage

```
declare hcs_$list_inacl entry(char(*), char(*), ptr, ptr,
        ptr, fixed bin, fixed bin, fixed bin(35));

call hcs_$list_inacl (dirname, ename, area_ptr, area_ret_ptr,
     acl_ptr, acl_count, ring, code)
```

1) dirname — is the superior directory portion of the path name of the directory in question. (Input)

2) ename — is the entry name portion of the path name of the directory in question. (Input)

3) area_ptr — points to an area into which the list of Initial ACL entries is to be allocated. (Input)

4) area_ret_ptr — points to the start of the allocated list of Initial ACL entries. (Output)

5) acl_ptr — if area_ptr is null then acl_ptr is assumed to point to an Initial ACL structure, segment_acl, into which mode information is to be placed for the access names specified in that same structure. (Input)

6) acl_count — is the number of entries in the Initial ACL structure identified by acl_ptr (Input); or is set to the number of entries in the segment_acl structure allocated in the area pointed to by area_ptr, if area_ptr is not null. (Output)

7) ring — is the ring number of the Initial ACL. (Input)

8) code — is a standard status code. (Output)

Note

If acl_ptr is used to obtain modes for specified access names (rather than obtaining modes for all access names on the Initial ACL), then each Initial ACL entry will either have a zero status_code and will contain the segment's mode or will have status_code set to error_table_$user_not_found and will contain a zero mode.

 (END)

Name: hcs_$make_ptr

This entry, when given a segment name and an entry point name, returns a pointer to a segment entry point. It uses the search rules to find the required segment.

Usage

```
declare hcs_$make_ptr entry (ptr, char(*), char(*), ptr,
        fixed bin);

call hcs_$make_ptr (caller_ptr, seg_name, entry_point_name,
     entry_point_ptr, error_code);
```

1) caller_ptr — is a pointer to the "calling procedure" (see Notes below). (Input)

2) seg_name — is the name of segment to be located. (Input)

3) entry_point_name — is the name of the entry point to be located. (Input)

4) entry_point_ptr — is the pointer to the segment entry point specified by seg_name and entry_point_name. (Output)

5) error_code — is the returned error code. (Output)

Notes

The directory in which the procedure pointed to by caller_ptr is located is used as the calling directory for the standard search rules. If it is null, then rule 1 of the standard search rules is skipped. See the MPM Reference Guide section on The System Libraries and Search Rules. The standard usage is to have caller_ptr null.

The seg_name and entry_point_name arguments are nonvarying character strings of length ≤ 32. They need not be aligned and may be blank padded.

If a null string is given for the entry_point_name argument, then a pointer to the base of the segment is returned. In either case, the segment seg_name is made known to the process with the reference name seg_name. If an error was encountered upon return, the entry_point_ptr argument is null and a nonzero error code is given.

To invoke the procedure entry point pointed to by entry_point_ptr, use cu_$gen_call or cu_$ptr_call. (See cu_ in the MPM.)

Example

The following PL/I statements will generate a call to the procedure fred$foo passing as arguments the integer 17, the pointer p, and the character string "treat".

```
call hcs_$make_ptr (null, "fred", "foo", ep, code);

call cu_$ptr_call (ep, 17, p, "treat");
```


(END)

Name: hcs_$make_seg

This procedure creates a segment in a specified directory with a specified entry name. Once the segment is created, it is made known to the process by a call to hcs_$initiate and a pointer to the segment is returned to the caller. If the segment already exists, an error code is returned. However, a pointer to the segment is still returned.

Usage

```
declare hcs_$make_seg entry (char(*), char(*), char(*),
     fixed bin(5), ptr, fixed bin);

call hcs_$make_seg (dirname, entry, rname, mode
     segptr, code);
```

1) dirname — is the directory in which to create the segment. (Input)

2) entry — is the entry name of the segment to be created. (Input)

3) rname — is the desired reference name, or "". (Input)

4) mode — specifies the mode for this user. See Notes in hcs_$append_branch for more information on modes. (Input)

5) segptr — is a pointer to the created segment. (Output)

6) code — is a standard file system status code. (Output)

Notes

If dirname is null, the process directory is used. If the entry argument is null, a unique name is provided. The rname argument is passed directly to hcs_$initiate and should normally be null.

See also the MPM Reference Guide section on Constructing and Interpreting Names.

(END)

hcs_$mask_ips

Name: ips_

The procedure ips_ controls the enabling and disabling of interprocess signals (IPS). The entries modify the per-process IPS mask and automatic IPS mask register.

Entry: ips_$mask_ips

This entry is used to mask one or more (IPS) interrupts from going off.

Usage

```
declare ips_$mask_ips entry (bit(36) aligned, bit(36)
        aligned);

call ips_$mask_ips (mask, oldmask);
```

1) mask — contains a bit ON for each IPS interrupt to be masked. (Input)

2) oldmask — is the old IPS mask. (Output)

Notes

There is a one-to-one correspondence between bits of the mask word and IPS interrupts. The correspondence between bits of the mask word and IPS interrupts can be found with the use of the "create_ips_mask_" subroutine.

See hcs_$get_ips_mask.

hcs_$printer_attach

Entry: printer_dcm$printer_attach

This is the ring 0 printer DIM attach call. It finds a free per-printer structure, calls the mini-gim to attach the printer and initializes the DCW's.

Usage

```
call hcs_$printer_attach (channel_name, ev_chan, index,
                          ercode);
```

or

```
call printer_dcm$printer_attach ('');
```

1) channel_name(character(*)) is the device index

2) ev_chan(fixed binary(71)) is the channel to use in signalling events

3) index(fixed binary(17)) is the index into the attach list (got at attach time and used for further calls) (returned)

4) ercode(fixed binary(17)) error code (returned)

- =0 means everything is OK
- =1 means all printers are busy
- =2 the mini_gim could not assign the device
- =3 the per-printer structure was wrested from this process
- =4 there was an error in the device configuration

hcs_$printer_detach

Entry: printer_dcm$printer_detach

This is the ring 0 printer DIM detach call.

Usage

```
call hcs_$printer_detach (index, ercode);
```

or

```
call printer_dcm$printer_detach (index, ercode);
```

1) index(fixed binary(17)) is the device index

2) ercode(fixed binary(17)) is the error code

=0 nothing wrong

=1 index is out of bounds

=2 the device was not in use

=3 this was not attached to the named printer

External References

```
check$device_name
dct_seg
prt_ccnv
pds$processid
   $interaction_switch
mini_gim$cur_status
   $tdcw
   $list_connect
   $assign
   $unassign
   $set_list
```

hcs_$printer_write

Entry: printer_dcm$printer_write

This program handles the ring 0 operation of the printers. User calls are forwarded to this module which gets free buffer space, then code converts the data sets up the DCW's and initiates printer I/O through use of the mini-gim.

Usage

```
call hcs_$printer_write (index, workspace, offset,
                         nelem, nelemt, rcode);
```

or

```
call printer_dcm$printer_write (index, workspace, offset,
                                nelem, nelemt, rcode);
```

1) index(fixed binary(17))	identifies the printer channel (see $printer_attach)
2) workspace(pointer)	is a pointer to the beginning of the caller's workspace from which write-behind data is to be transferred
3) offset(fixed binary(17))	is the offset in characters from the start of the user's workspace and indicates where the data transfer should begin
4) nelem(fixed binary(17))	is the number of characters which should be transferred from the user's workspace
5) nelemt(fixed binary(17))	the number of characters actually transmitted by the write call (returned)
6) rcode(fixed binary(17))	error code means (returned) =0 A-OK =1 ioname is incorrect

=2 printer is not attached

=3 this process is not attached to this printer

If all the data was not written (i.e., nelem ≠ nelemt), the calling routing should first wait for a while, then try to write out the rest of the data. A proper sequence of instructions is:

```
loop:  call hcs_$printer_write (--);
       if nelem = nelemt then do;
       call ipc$block (--);          /* go to sleep */
       offset = offset + nelemt;
       nelem = nelem - nelemt;
       nelemt = 0;
       go to loop;
       end;
```

hcs_$proc_info

Name: proc_info

This procedure returns to the caller selected per-process data that may be of interest.

Usage

```
declare proc_info entry (bit(36) aligned, char(32) aligned,
        char(32) aligned, bit(36) aligned);

call proc_info (process_id, process_group_id,
     process_dir_name, lock_id);
```

1) process_id	is the variable used to identify this process. (Output)
2) process_group_id	is the character string representation of this process' access rights. (Output)
3) process_dir_name	is the path name of the process directory of this process. (Output)
4) lock_id	is the unique identifier used by this process in the standard locking procedures. (Output)

This is an internal interface to various user ring procedures such as get_process_id_.

Name: hcs_$quota_get

This subroutine returns the record quota and accounting information for a directory.

Usage

```
declare hcs_$quota_get entry (char(*), fixed bin(18),
        fixed bin(35), bit(36) aligned, fixed bin, fixed
        bin(1), fixed bin, fixed bin(35));

call hcs_$quota_get (dirname, quota, trp, tup, infqcnt,
     taccsw, used, code);
```

1) dirname — is the path name of the directory for which quota information is desired. (Input)

2) quota — is the record quota in the directory. (Output)

3) trp — is the time-record product charged to the directory. This number is in units of record-seconds. (Output)

4) tup — is the time that the trp was last updated in storage system time format (the high-order 36 bits of the 52-bit time returned by clock_). (Output)

5) infqcnt — is the number of immediately inferior directories (i.e., directories in this directory) which contain terminal accounts. (Output)

6) taccsw — is the terminal account switch. If the switch is on, the records are charged against the quota in this directory. If the switch is off, the records are charged against the quota in the first superior directory with a terminal account. (Output)

7) used — is the number of records used by segments in this directory and by non-terminal inferior directories. (Output)

8) code — is a standard storage system status code. (Output)

Notes

. THE USER MUST HAVE status PERMISSION ON THE directory:

If the account is currently active, this call will cause the account information in the directory header to be updated from the Active Segment Table (AST) entry before this information is returned to the caller. If the directory contains a non-terminal account, the quota, trp, and tup variables are all zero. The variable used, however, is kept up-to-date and represents the number of pages of segments in this directory and inferior non-terminal directories. If a quota were to be placed in this directory, it should be greater than this used value.

 (END)

Name: hcs_$quota_move

This subroutine is callable by any user and moves all or part of a quota between two directories, one of which is immediately inferior to the other.

Usage

```
declare hcs_$quota_move entry (char(*), char(*),
        fixed bin(18), fixed bin(35));

call hcs_$quota_move (dirname, entry, quota_change, code);
```

1) dirname — is the path name of the parent directory. (Input)

2) entry — is the entry name of the inferior directory. (Input)

3) quota_change — is the number of 1024-word pages of secondary storage quota to be subtracted from the parent directory and added to the inferior directory. (Input)

4) code — is a standard storage system status code. (Output)

Notes

The entry specified by entry must be a directory.

The user must have modify permission in both directories.

After the quota change, the remaining quota in each directory must be greater than the number of pages used in that directory.

The argument quota_change may be either a positive or negative number. If it is positive, the quota will be moved from dirname to entry. If it is negative, the move will be from entry to dirname. If the change results in zero quota left on entry, that directory is assumed to no longer contain a terminal quota and all of its used pages are reflected up to the used pages on dirname. There is a restriction on quotas such that all quotas in the chain from the root to (but not including) the terminal directory must be nonzero. This restriction means that hcs_$quota_move cannot leave behind a quota of zero in the superior directory.

 (END)

Name: hcs_$replace_acl

This subroutine replaces an entire Access Control List (ACL) for a segment with a user-provided ACL, and can optionally add an entry for *.SysDaemon.* with mode rw to the new ACL. The segment_acl structure described in hcs_$add_acl_entries is used by this subroutine.

Usage

```
declare hcs_$replace_acl entry (char(*), char(*), ptr,
        fixed bin, bit(1), fixed bin(35));

call hcs_$replace_acl (dirname, ename, acl_ptr, acl_count,
     no_sysdaemon_sw, code);
```

1) dirname — is the directory portion of the path name of the segment in question. (Input)

2) ename — is the entry name portion of the path name of the segment in question. (Input)

3) acl_ptr — points to the user supplied segment_acl structure that is to replace the current ACL. (Input)

4) acl_count — is the number of entries in the segment_acl structure. (Input)

5) no_sysdaemon_sw — if "0"b, then a *.SysDaemon.* rw entry will be put on the segment's ACL after the existing ACL has been deleted and before the user supplied segment_acl entries are added; if "1"b, then only the user-supplied segment_acl will replace the existing ACL. (Input)

6) code — is a standard status code. (Output)

Notes

If acl_count is zero then the existing ACL will be deleted and only the action indicated by no_sysdaemon_sw will be performed (if any). In the case when acl_count is greater than zero, processing of the segment_acl entries is performed top to bottom, allowing later entries to overwrite previous ones if the access_name parts are identical.

If the segment is a gate (see the MPM Subsystem Writers' Guide section, Intraprocess Access Control (Rings)) and if the validation level is greater than Ring 1, and only access names that contain the same project as the user, and "SysDaemon" and "sys_control" projects will be allowed. If the replacement ACL is in error then no processing will be performed and the code error_table_$invalid_project_for_gate will be returned.

 (END)

Name: hcs_$replace_dir_acl

This subroutine replaces an entire Access Control List (ACL) for a directory with a user-provided ACL, and can optionally add an entry for *.SysDaemon.* with mode sma to the new ACL. The dir_acl structure described in hcs_$add_dir_acl_entries is used by this subroutine.

Usage

```
declare hcs_$replace_dir_acl entry (char(*), char(*), ptr,
        fixed bin, bit(1), fixed bin(35));

call hcs_$replace_dir_acl (dirname, ename, acl_ptr,
     acl_count, no_sysdaemon_sw, code);
```

1) dirname — Is the path name of the directory superior to the one in question. .(Input)

2) ename — is the entry name of the directory in question. (Input)

3) acl_ptr — points to a user-supplied dir_acl structure that is to replace the current ACL. (Input)

4) acl_count — is the number of entries in the dir_acl structure. (Input)

5) no_sysdaemon_sw — if "0"b, then a *.SysDaemon.* sma entry will be put on the directory's ACL after the existing ACL has been deleted and before the user-supplied dir_acl entries are added; if "1"b, then only the user-supplied dir_acl will replace the existing ACL. (Input)

6) code — is a standard status code. (Output)

Note

If acl_count is zero then the existing ACL will be deleted and only the action indicated by no_sysdaemon_sw will be performed (if any). In the case when acl_count is greater than zero, processing of the dir_acl entries is performed top to bottom, allowing later entries to overwrite previous ones if the access_name parts are identical.

 (END)

Name: hcs_$replace_dir_inacl

This subroutine replaces an entire Initial Access Control List (Initial ACL) for new directories within a specified directory with a user-provided Initial ACL, and can optionally add an entry for *.SysDaemon.* with mode sma to the new Initial ACL. The dir_acl structure described in hcs_$add_dir_inacl_entries is used by this subroutine.

Usage

```
declare hcs_$replace_dir_inacl entry (char(*), char(*), ptr,
        fixed bin, bit(1) aligned, fixed bin,
        fixed bin(35));

call hcs_$replace_dir_inacl (dirname, ename, acl_ptr,
     acl_count, no_sysdaemon_sw, ring, code);
```

1) dirname — is the path name of the directory superior to the one in question. (Input)

2) ename — is the entry name of the directory in question. (Input)

3) acl_ptr — points to a user-supplied dir_acl structure that is to replace the current initial ACL. (Input)

4) acl_count — is the number of entries in the dir_acl structure. (Input)

5) no_sysdaemon_sw — if "0"b, then a *.SysDaemon.* sma entry will be put on the Initial ACL after the existing Initial ACL has been deleted and before the user-supplied dir_acl entries are added; if "1"b, then only the user-supplied dir_acl will replace the existing Initial ACL. (Input)

6) ring — is the ring number of the Initial ACL. (Input)

7) code — Is a standard status code. (Output)

Note

If acl_count is zero then the existing initial ACL will be deleted and only the action indicated by no_sysdaemon_sw will be performed (if any). In the case when acl_count is greater than zero, processing of the dir_acl entries is performed top to bottom, allowing later entries to overwrite previous ones if the access_name parts are identical.

 (END)

Subroutine Call
3/1/73

Name: hcs_$replace_inacl

This subroutine replaces an entire Initial Access Control List (Initial ACL) for new segments within a specified directory with a user-provided Initial ACL, and can optionally add an entry for *.SysDaemon.* with mode rw to the new Initial ACL. The segment_acl structure described in hcs_$add_inacl_entries is used by this subroutine.

Usage

```
declare hcs_$replace_inacl entry (char(*), char(*), ptr,
        fixed bin, bit(1), fixed bin, fixed bin(35));

call hcs_$replace_inacl (dirname, ename, acl_ptr, acl_count,
     no_sysdaemon_sw, ring, code);
```

1) dirname — is the superior directory portion of the path name of the directory in question. (Input)

2) ename — is the entry name portion of the path name of the directory in question. (Input)

3) acl_ptr — points to the user supplied segment_acl structure that is to replace the current Initial ACL. (Input)

4) acl_count — is the number of entries in the segment_acl structure. (Input)

5) no_sysdaemon_sw — if "0"b, then a *.SysDaemon.* rw entry will be put on the Initial ACL after the existing Initial ACL has been deleted and before the user-supplied segment_acl entries are added; if "1"b, then only the user-supplied segment_acl will replace the existing Initial ACL. (Input)

6) ring — is the ring number of the Initial ACL. (Input)

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

Subroutine Call
3/1/73

Name: hcs_$replace_inacl

This subroutine replaces an entire Initial Access Control List (Initial ACL) for new segments within a specified directory with a user-provided Initial ACL, and can optionally add an entry for *.SysDaemon.* with mode rw to the new Initial ACL. The segment_acl structure described in hcs_$add_inacl_entries is used by this subroutine.

Usage

```
declare hcs_$replace_inacl entry (char(*), char(*), ptr,
        fixed bin, bit(1), fixed bin, fixed bin(35));

call hcs_$replace_inacl (dirname, ename, acl_ptr, acl_count,
     no_sysdaemon_sw, ring, code);
```

1) dirname — is the superior directory portion of the path name of the directory in question. (Input)

2) ename — is the entry name portion of the path name of the directory in question. (Input)

3) acl_ptr — points to the user supplied segment_acl structure that is to replace the current Initial ACL. (Input)

4) acl_count — is the number of entries in the segment_acl structure. (Input)

5) no_sysdaemon_sw — if "0"b, then a *.SysDaemon.* rw entry will be put on the Initial ACL after the existing Initial ACL has been deleted and before the user-supplied segment_acl entries are added; if "1"b, then only the user-supplied segment_acl will replace the existing Initial ACL. (Input)

6) ring — is the ring number of the Initial ACL. (Input)

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

Note

If acl_count is zero then the existing initial ACL will be deleted and only the action indicated by no_sysdaemon_sw will be performed (if any). In the case when acl_count is greater than zero, processing of the segment_acl entries is performed top to bottom, allowing later entries to overwrite previous ones if the access_name parts are identical.

 (END)

Subroutine Call
Development System
05/10/71

Name: hcs_$reset_working_set

This entry is called to turn off the used bits of all pages in the page-trace list for the current process. This is equivalent to truncating the pre-page list and starting the gathering of pre-page statistics with the next page fault.

Usage

```
declare hcs_$reset_working_set entry;

call hcs_$reset_working_set;
```

There are no arguments.

(END)

Name: hcs_$set_bc

This subroutine sets the bit count of a segment in the storage system, given a path name. It also sets that segment's bit count author to be the user who called it.

Usage

```
declare hcs_$set_bc entry (char(*), char(*),
        fixed bin(24), fixed bin(35));

call hcs_$set_bc (dirname, ename, bit_count, code);
```

1) dirname is the directory name of the segment whose bit count is to be changed. (Input)

2) ename is the entry name of the segment whose bit count is to be changed. (Input)

3) bit_count is the new bit count of the segment. (Input)

4) code is a standard storage system status code. (Output)

Notes

The user must have write permission with respect to the segment, but does not need write permission with respect to the parent directory.

The subroutine hcs_$set_bc_seg performs the same function, when a pointer to the segment is provided rather than a path name.

 (END)

Name: hcs_$set_bc_seg

This subroutine sets the bit count of a segment in the storage system, given the pointer to the segment. It also sets that segment's bit count author to be the user who called it.

Usage

```
declare hcs_$set_bc_seg entry (ptr, fixed bin(24),
        fixed bin(35));

call hcs_$set_bc_seg (segptr, bitcount, code);
```

1) segptr is a pointer to the segment whose bit count is to be changed. (Input)

2) bitcount is the new bit count of the segment. (Input)

3) code is a standard storage system status code. (Output)

Note

The user must have write permission with respect to the segment, but does not need write permission with respect to the parent directory.

The subroutine hcs_$set_bc performs the same function, when provided with a path name of a segment rather than the pointer.

 (END)

hcs_$set_alarm_timer

Name: set_alarm_timer

This procedure is called to perform the various alarm clock functions. (real time clock)

Entry: set_alarm_timer$set_alarm_timer

This entry is used to set the time (absolute time) or time increment (relative time) in the alarm clock for the calling process.

Usage

```
declare set_alarm_timer entry (fixed bin(71),
        fixed bin, fixed bin(71));

call set_alarm_timer (time, time_sw, channel);
```

1) time — Is the real time, in microseconds, at which the process will receive an alarm clock wakeup. The time may be relative to the current time (as read from the calendar clock by clock_) or it may be an absolute calendar clock time when the wakeup is to take place. (Input)

2) sw — Is a switch indicating whether the first argument is to be considered absolute or relative. The following correspondence is used:

```
sw = 1  relative time;
sw = 2  absolute time.  (Input)
```

3) channel — Is the event channel over which to send the wakeup. If the channel is zero, the condition "alrm" will be signalled; otherwise an Interprocess Communication (IPC) wakeup will occur. (Input)

This procedure is an internal interface to the procedure timer_manager_

hcs_$set_automatic_ips_mask

```
dcl ips_$set_automatic_ips_mask ext entry (bit (36) aligned, bit (36) a
call ips_$set_automatic_ips_mask(mask, oldmask);
```

1) mask is the new value to be used for pds$auto_mas

2) oldmask is the old value of pds$auto_mask (Output).

pds$auto_mask is an array of 36 bits, one for each possible IPS signal. If a bit = "1"b for some interrupt, say I_1, then when I_1 occurs all other interrupts will be masked.

(see hcs_$get_ips_mask

hcs_$set_copysw

```
/* SET$COPYSW changes the setting of the copy switch in the branch effecti
   "entry" in the directory with path name "dirname" to "copy" if caller ha
   write permit in the directory. */

copysw:   entry (a_dirname, a_ename, a_copy, a_code);
```

dcl hcs_$copy_sw entry (char(*), char(*), fixed bin(1), fixed bin(35));

1) a_dirname — is the pathname of the directory containing the entry whose copy switch is to be set

2) a_ename — is the entry whose copy switch is to be set

3) a_copy — value of copy switch (0 or 1)

4) a_code — is a status code.

hcs_$set_cpu_timer

Entry: pxss$set_cpu_timer

This entry is used to set the CPU timer. It may be called from outside ring 0 by any process.

Usage

```
declare pxss$set_cpu_timer entry (fixed bin(71), fixed bin,
        fixed bin(71));

call pxss$set_cpu_timer (time, sw, channel);
```

1) time — is the CPU time, in microseconds, that the process can run before receiving the CPU wake-up. The time may be relative to the current CPU time used by the process or it may be an absolute time, i.e., the wake-up will occur when the process' CPU time exceeds this value. (Input)

2) sw — is a switch indicating whether the first argument is to be considered absolute or relative. The following correspondence is used:

   sw = 1 relative time;
   sw = 2 absolute time. (Input)

3) channel — is the event channel over which to signal the wake-up when and if it occurs. If channel is zero, an interprocess signal (IPS) will occur; otherwise an interprocess communication (IPC) wake-up will occur. (Input)

This procedure is an internal interface to the procedure timer_manager_

Name: hcs_$set_dir_ring_brackets

This subroutine, given the path name of the superior directory and the name of the directory, will set that directory's ring brackets.

Usage

```
declare hcs_$set_dir_ring_brackets entry (char*), char(*),
        (2) fixed bin(3), fixed bin(35));

call hcs_$set_dir_ring_brackets (dirname, ename, drb, code);
```

1) dirname. is the path name of the superior directory. (Input)

2) ename is the entry name of the directory in question. (Input)

3) drb is a 2-element array specifying the ring brackets of the directory. (Input)

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

Notes

The user must have modify permission in the superior directory and the validation level must be less than or equal to both the present value of the first ring bracket and the new value of the first ring bracket that the user wishes set.

Ring brackets and validation levels are discussed in the MPM Subsystem Writers' Guide section, Intraprocess Access Control (Rings).

 (END)

hcs_$set_ips_mask

Entry: ips_$set_ips_mask

This entry is used to replace the entire IPS mask with a new value.

Usage

```
declare ips_$set_ips_mask entry (bit(36) aligned, bit(36)
        aligned);

call ips_$set_ips_mask (mask, oldmask);
```

1) mask — this word replaces the old IPS mask. (Input)

2) oldmask — is the old IPS mask. (Output)

(See hcs_$get_ips_mask)

Subroutine Call
3/30/73

Name: hcs_$set_max_length

This subroutine sets the max length of a segment, given a directory name and an entry name. The max length is the length beyond which the segment may not grow.

Usage

```
declare hcs_$set_max_length entry (char(*), char(*),
        fixed bin(18), fixed bin(35));

call hcs_$set_max_length (dirname, ename, max_length, code);
```

1) dirname is the directory name of the segment whose max length is to be changed. (Input)

2) ename is the entry name of the segment whose max length is to be changed. (Input)

3) max_length is the new value in words for the max length of the segment. (Input)

4) code is a standard storage system status code. (See Notes below.) (Output)

Notes

A directory may not have its max length changed.

Modify permission with respect to the directory containing the segment is required.

Eventually, the max length of a segment will be accurate to units of 16 words, and if max_length is not a multiple of 16 words, it will be set to the next multiple of 16 words. However, currently the max length of a segment should be set in units of 1024 words, due to hardware restrictions.

If an attempt is made to set the max length of a segment greater than the system maximum, sys_info$max_seg_size, code will be set to error_table_$argerr.

If an attempt is made to set the max. length of a segment greater than its current length, code will be set to error_table_$invalid_max_length.

The subroutine hcs_$set_max_length_seg may be used when the pointer to the segment is given, rather than a path name.

 (END)

hcs_$sfblock

Entry: fast_hc_ipc$ipc_sf_block

This entry goes blocked until a wake-up is received on the specified channel or an Interprocess Signal (IPS) wake-up is received.

Usage

```
declare fast_hc_ipc$ipc_sf_block entry (fixed bin(71),
        fixed bin(35));

call fast_hc_ipc$ipc_sf_block (channel, code);
```

1) channel	is the identifier of the special channel in which to go blocked. (Input)
2) code	is a standard status code, the code error_table_$ips_has_occurred indicates that this procedure returned because of an IPS wake-up. (Output)

This entry is intended for processes that want very fast calls to block. It will wait on only a single (special) channel, and will not return until a wakeup occurs on that channel, or an ips interrupt has occured. One should not use hcs_$sfblock and ipc_$block on the same special channel because ipc_ may be holding a wakeup on the channel that hcs_$sfblock will not see.

Name: hcs_$star_

This subroutine is the star convention handler for the storage system. (See The Star Convention in the MPM Reference Guide section, Constructing and Interpreting Names.) It is called with a directory name, and an entry name containing components which may be * or **. The directory is searched for all entries which match the given entry name. Information about these entries is returned in a structure. If the entry name is **, information on all entries in the directory is returned.

Status permission is required with respect to the directory to be searched.

The main entry returns the storage system type and all names which match the given entry name. (See hcs_$star_list_ below to obtain more information about each entry.)

Usage

```
declare hcs_$star_ entry (char(*), char(*), fixed bin(2),
        ptr, fixed bin, ptr, ptr, fixed bin(35));

call hcs_$star_ (dirname, star_name, select_sw,
     areap, ecount, eptr, nptr, code);
```

1) dirname — is the path name of the directory to be searched. (Input)

2) star_name — is the entry name which may contain asterisks. (Input)

3) select_sw — = 1 if information is to be returned about link entries only;

= 2 if information is to be returned about segment entries only;

= 3 if information is to be returned about all entries. (Input)

4) areap — is a pointer to the area in which information is to be returned. If the pointer is null, ecount is set to the total number of selected entries. See Notes immediately below. (Input)

5) ecount — is a count of the number of entries which match the entry name. (Output)

6) eptr — is a pointer to the allocated structure in which information on each entry is returned. (Output)

7) nptr — is a pointer to the allocated array of all the entry names in this directory which match star_name. See Notes immediately below. (Output)

8) code — is a standard storage system status code. See Status Codes below. (Output)

Notes

Even if areap is null, ecount is set to the total number of entries in the directory which match star_name. The setting of select_sw determines whether ecount is the total number of link entries, the total number of segment entries or the total number of all entries.

If areap is not null, the following structure is allocated in the user-supplied area:

```
declare 1 entries (ecount) aligned based (eptr),
        (2 type bit(2),
         2 nnames bit(16),
         2 nindex bit(18)) unaligned;
```

1) type — specifies the storage system type of entry:

0 ("00"b) = link,
1 ("01"b) = nondirectory segment,
2 ("10"b) = directory segment.

2) nnames — specifies the number of names for this entry which match star_name.

3) nindex — specifies the offset in the array of names (pointed to by nptr) for the first name returned for this entry.

All of the names which are returned for any one entry are stored consecutively in an array of all the names, allocated in the user-specified area. The first name for any one entry begins at the offset nindex in the array.

```
declare names (total_names) char(32) aligned based (nptr);
```

where total_names is the total number of names returned.

It should be noted that the user must provide an area large enough for this subroutine to store the requested information.

Entry: hcs_$star_list_

This entry returns more information about the selected entries.

Usage

```
declare hcs_$star_list_ entry (char(*), char(*),
        fixed bin(3), ptr, fixed bin, fixed bin, ptr, ptr,
        fixed bin(35));

call hcs_$star_list_ (dirname, star_name, select_sw,
     areap, seg_count, link_count, eptr, nptr, code);
```

1) dirname is as above. (Input)

2) star_name is as above. (Input)

3) select_sw =1 if information is to be returned about link entries only;

=2 if information is to be returned about segment entries only;

=3 if information is to be returned about all entries;

=5 if information is to be returned about link entries only, including the path name associated with each link entry;

=7 if information is to be returned about all entries, including the path name associcated with each link entry. (Input)

4) areap is a pointer to the area in which information is to be returned. If the pointer is null, seg_count and link_count are set to the total number of selected entries. See Notes immediately below. (Input)

5) seg_count is a count of the number of segments and directories which match the entry name. (Output)

6) link_count is a count of the number of links which match the entry name. (Output)

7) eptr is as above. (Output)

8) nptr — is a pointer to the allocated array in which selected entry names and path names associated with link entries are stored. (Output)

9) code — is as above. (Output)

Notes

Even if areap is null, seg_count is set to the total number of segments and directories which match star_name, if information on segments is requested. If information on links is requested, link_count is the total number of links which match star_name.

The following structure is allocated in the user-supplied area, if areap is not null:

```
declare entries (count) bit(144) aligned based (eptr);
```

where count = seg_count + link_count.

For each unit of the array, one of two structures will be found. Which structure should be used may be determined by the type item which is located at the base of each structure. It should be noted that the first three items in each structure are identical to the structure returned by hcs_$star_.

The following structure is used if the entry is a segment or a directory:

```
declare 1 branches aligned based (eptr),
        (2 type bit(2),
         2 nname bit(16),
         2 nindex bit(18),
         2 dtm bit(36),
         2 dtu bit(36),
         2 mode bit(5),
         2 pad bit(13),
         2 records bit(18)) unaligned;
```

1) type — is as above.

2) nname — is as above.

3) nindex — is as above.

4) dtm — is the date and time the segment or directory was last modified.

5) dtu — is the date and time the segment or directory was last used.

6) mode is the current user's access to the segment or directory. See the MPM write-up of hcs_$append_branch for a description of modes.

7) pad is unused space in this structure.

8) records is the number of 1024-word records of secondary storage which have been assigned to the segment or directory.

The following structure is used if the entry is a link:

```
declare 1 links aligned based (eptr),
        (2 type bit(2),
         2 nname bit(16),
         2 nindex bit(18),
         2 dtm bit(36),
         2 dtd bit(36),
         2 pathname_len bit(18),
         2 pathname_index bit(18)) unaligned;
```

1) type is as above.

2) nname is as above.

3) nindex is as above.

4) dtm is the date and time the link entry was last modified.

5) dtd is the date and time the link entry was last dumped.

6) pathname_len is the number of significant characters in the pathname assocated with the link.

7) pathname_index is the offset in the array of names for the link pathname. See below.

If the path name associated with each link entry was requested, the path name will be placed in the names array and will occupy six units of this array. The offset of the first unit is specified by pathname_index in the links array. The length of the path name is given by pathname_len in the links array.

Status Codes

If no match with star_name was found in the directory, code will be returned as error_table_$nomatch.

If star_name contained illegal syntax with respect to the star convention, code will be returned as error_table_$badstar.

If the user did not provide enough space in the area to return all the requested information, code will be returned as error_table_$notalloc. In this case the total number of entries (for hcs_$star_) or the total number of segments and the total number of links (for hcs_$star_list_) will be returned, to provide an estimate of the space required.

 (END)

Name: hcs_$status_

This subroutine consists of a number of hardcore, user-callable, storage system entry points which return various items of information about a specified hierarchy entry.

The main entry point (hcs_$status_) returns the most often needed information about a specified entry. (See hcs_$status_long below.)

Usage

```
declare hcs_$status_ entry (char(*), char(*), fixed bin(1),
        ptr, ptr, fixed bin(35));

call hcs_$status_ (dirname, entry, chase, eptr, nareap,
     code);
```

1) dirname — is the directory portion of the path name of the entry in question. (Input)

2) entry — is the entry name portion of the path name of the entry in question. (Input)

3) chase — =0: if the entry is a link, return link information;
=1: if the entry is a link, return information about the entry to which it points. (Input)

4) eptr — is a pointer to the structure in which information is returned. See Notes immediately below. (Input)

5) nareap — is a pointer to the area in which names are returned. If the pointer is null, no names are returned. See Notes immediately below. (Input)

6) code — is a storage system status code. See Access Requirements below. (Output)

Notes

The argument eptr points to the following structure if the entry is a segment or directory:

```
declare 1 branch based (eptr) aligned,
        (2 type bit(2),
         2 nnames bit(16),
         2 nrp bit(18),
         2 dtm bit(36),
         2 dtu bit(36),
         2 mode bit(5),
         2 pad1 bit(13),
         2 records bit(18)) unaligned;
```

1) type specifies the type of entry:

```
0 ("00"b) = link;
1 ("01"b) = segment;
2 ("10"b) = directory.
```

2) nnames specifies the number of-names for this entry.

3) nrp is a relative pointer (relative to the base of the segment containing the user-specified free storage area) to an array of names.

4) dtm contains the date and time the segment was last modified.

5) dtu contains the date and time the segment was last used.

6) mode contains the mode of the segment with respect to the current user. See the MPM write-up of hcs_$append_branch for a description of modes.

7) pad1 is unused space in this structure.

8) records contains the number of 1024-word records of secondary storage which has been assigned to the segment.

The argument eptr points to the following structure if the entry is a link:

```
declare 1 link based (eptr) aligned,
        (2 type bit(2),
         2 nnames bit(16),
         2 nrp bit(18),
         2 dtem bit(36),
         2 dtd bit(36),
         2 pnl bit(18),
         2 pnrp bit(18)) unaligned;
```

1) type — as above.

2) nnames — as above.

3) nrp — as above.

4) dtem — contains the date and time the link was last modified.

5) dtd — contains the date and time the link was last dumped.

6) pnl — specified the length in characters of the link path name.

7) pnrp — is a relative pointer (relative to the base of the segment containing the user-specified free storage area) to the link path name.

Note that the user must provide the storage space required by the above structures. The status entry point merely fills them in.

If nareap is not null, entry names are returned in the following structure allocated in the user-specified area.

```
declare names (nnames) char(32) aligned based (np);
```

where np = ptr (nareap, eptr->entry.nrp).

The first name in this array is defined as the primary name of the entry.

Link path names are returned in the following structure allocated in the user-specified area.

```
declare pathname char(pnl) aligned based (lp);
```

where lp = ptr (nareap, eptr->link.nrp);

Note that the user allocates the area and it must be large enough to accommodate a reasonable number of names.

Access Requirements

The user must have status permission on the parent directory to obtain complete information.

If the user lacks status permission but does have non-null access to a segment, the following per-segment attributes may be returned: type, effective access, bit count, records and current length. In this instance if the entry point hcs_$status_ or hcs_$status_long is called, the status code error_table_$no_s_permission is returned to indicate that incomplete information has been returned.

Entry: hcs_$status_minf

This subroutine returns the bit count and entry type given a directory and entry name. The access required to use this subroutine is status permission on the directory or non-null access to the entry.

Usage

```
declare hcs_$status_minf entry (char(*), char(*), fixed
        bin(1), fixed bin(2), fixed bin(24), fixed bin(35));

call hcs_$status_minf (dirname, entry, chase, type, bitcnt,
     code);
```

1) dirname is as above. (Input)

2) entry is as above. (Input)

3) chase is as above. (Input)

4) type specifies the type of entry:

   0 = link;
   1 = segment;
   2 = directory. (Output)

5) bitcnt is the bit count. (Output)

6) code is as above. (Output)

Entry: hcs_$status_mins

This subroutine returns the bit count and entry type given a pointer to the segment. The access required to use this subroutine is status permission on the directory or non-null access on the segment.

Usage

```
declare hcs_$status_mins entry (ptr, fixed bin(2),
        fixed bin(24), fixed bin(35));
```

```
call hcs_$status_mins (segptr, type, bitcnt, code);
```

1) segptr is a pointer to the segment about which information is desired. (Input)

2) type is as above. (Output)

3) bitcnt is as above. (Output)

4) code is as above. (Output)

Entry: hcs_$status_long

This subroutine returns most user-accessible information about a specified entry. The access required to use this subroutine is the same as that required by hcs_$status_ and described in Access Requirements above.

Usage

```
declare hcs_$status_long entry (char(*), char(*),
        fixed bin(1), ptr, ptr, fixed bin(35));

call hcs_$status_long (dirname, entry, chase, eptr, nareap,
     code);
```

Arguments are as above.

Notes

The argument eptr points to the same structure as before if the entry is a link. It points to the following structure if the entry is a segment or directory:

```
declare 1 branch based (eptr) aligned,
        (2 type bit(2),
         2 nnames bit(16),
         2 nrp bit(18),
         2 dtm bit(36),
         2 dtu bit(36),
         2 mode bit(5),
         2 pad1 bit(13),
         2 records bit(18),
         2 dtd bit(36),
         2 dtem bit(36),
         2 pad2 bit(36),
         2 curlen bit(12),
         2 bitcnt bit(24),
```

```
2 did bit(4),
2 pad3 bit(4),
2 copysw bit(9),
2 pad4 bit(9),
2 rbs (0:2) bit(6),
2 uid but(36)) unaligned;
```

1-8) are as described above in the structure for segments and directories returned by hcs_$status_.

9) dtd is the date and time the segment was last dumped.

10) dtem is the date and time the branch was last modified.

11) pad2 is unused space in this structure.

12) curlen is the current length of the segment in units of 1024-word records.

13) bitcnt is the bit count associated with the segment.

14) did specifies the secondary storage device (if any) on which the segment currently resides.

15) pad3 is unused space in this structure.

16) copysw contains the setting of the segment copy switch.

17) pad4 is unused space in this structure.

18) rbs contains the ring brackets of the segment.

19) uid is the segment unique identifier.

 (END)

hcs_ $ stop_process

Name: stop_process

This procedure is used to initiate the stopping of a process.

Usage

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

call stop_process (process_id);
```

1) process_id is the id of the process to be stopped. (Input)

It is called by the logout and new_proc commands after they have signalled the initializer process to destroy the process.

hcs_$tdcm_attach

Name: tdcm

This module is the magnetic tape Device Control Module (DCM) for all magnetic tape I/O. It provides a physical interface to the tape controller by which all tape controller commands may be issued. Drive multiplexing and dynamic buffer allocation are handled by this module.

Entry: tdcm$tdcm_attach

This entry is called to attach a free drive for tape operations. The drive number is allocated from a list of those free.

Usage

```
declare tdcm$tdcm_attach entry (ptr, fixed bin);

call tdcm$tdcm_attach (tsegp, code);
```

1) tsegp — is a pointer to a data base in the User Ring containing per drive information. (See the tseg write-up in the SPS.) (Input)

2) code — is an error code and is nonzero if no free drive can be found. (Output)

hcs_$tdcm_detach

Entry: tdcm$tdcm_detach

This entry is called to detach a drive after all operations have been completed. This call will not result in the drive being unloaded.

Usage

```
declare tdcm$tdcm_detach entry (ptr, fixed bin);

call tdcm$tdcm_detach (tsegp, code);
```

1,2) as above.

hcs_$tdcm_iocall

Entry: tdcm$tdcm_iocall

This is the main working entry by which I/O and control operations are performed. By setting proper variables in the tseg data base, an array of controller commands or a sequence of read/write operations can be initiated. (See the tseg write-up in the SPS.)

Usage

```
declare tdcm$tdcm_iocall entry (ptr, fixed bin);

call tdcm$tdcm_iocall (tsegp, code);
```

1) tsegp — is a pointer to the tseg data base. (Input)

2) code — is an error code:

2 = drive number out-of-bounds;
3 = drive not attached;
4 = drive not owned by this process;

5 = buffer size exceeded;
6 = buffer count is too large. (Output)

hcs_$tdcm_message

Entry: tdcm$tdcm_message

This entry results in a mount request message to be typed on the operator's console.

Usage

```
declare tdcm$tdcm_message entry (ptr, char(*), fixed bin(1),
        fixed bin);

call tdcm$tdcm_message (tsegp, reelid, ring, code);
```

1) tesgp — as above. (Input)

2) reelid — is a character string representation of the reel name or number. (Input)

3) ring — if nonzero, it indicates a ring is to be used. (Input)

4) code — as above. (Output)

hcs_$tdcm_reset_signal
hcs_$tdcm_set_signal

Entry: tdcm$tdcm_set_signal, tdcm$tdcm_reset_signal

These entries are used to set and reset a switch which controls the handling of special interrupts from the tape controller. When this switch is set, all special interrupts result in a signal to the attached process for this drive. Otherwise, all special interrupts are ignored.

Usage

```
declare tdcm$tdcm_set_signal_entry (ptr, fixed bin);

call tdcm$tdcm_set_signal (tsegp, code);

declare tdcm$tdcm_reset_signal entry (ptr, fixed bin);

call tdcm$tdcm_reset_signal (tsegp, code);
```

1,2) as above.

Name: hcs_$terminate_file

This subroutine, given the path name of a segment in the current process, removes all the reference names of that segment and then removes the segment from the address space of the process. For a discussion of reference names, see the MPM Reference Guide section, Contructing and Interpreting Names.

Usage

```
declare hcs_$terminate_file entry (char(*), char(*),
        fixed bin(1), fixed bin(35));

call hcs_$terminate_file (dir_name, entry_name, rsw, code);
```

1) dir_name is the directory portion of the path name of the segment in question. (Input)

2) entry_name is the entry name portion of the path name of the segment in question. (Input)

3) rsw is the reserved segment switch. If equal to 1, the segment number should be saved in the reserved segment list; if equal to 0, the segment number should not be saved. (Input)

4) code is a standard storage system status code. (Output)

Notes

The subroutine hcs_$terminate_seg performs the same operation given a pointer to a segment instead of a path name; hcs_$terminate_name and hcs_$terminate_noname terminate a single reference name.

The subroutine term_ performs the same operation as hcs_$terminate_file.

In fact, only those reference names are removed for which the ring level associated with the name is greater than or equal to the validation level of the process. If the user needs to concern himself with rings, he should refer to the MPM Subsystem Writers' Guide section, Intraprocess Access Control (Rings).

 (END)

Subroutine Call
2/20/73

Name: hcs_$terminate_name

This subroutine terminates one reference name from a segment. If it is the only reference name for that segment, the segment is removed from the address space of the process. For a discussion of reference names see the MPM Reference Guide section, Constructing and Interpreting Names.

Usage

```
declare hcs_$terminate_name entry (char(*), fixed bin(35));

call hcs_$terminate_name (ref_name, code);
```

1) ref_name is the reference name to be terminated. (Input)

2) code is a standard storage system status code. (Output)

Note

The subroutine hcs_$terminate_noname terminates a null reference name from a specified segment; hcs_$terminate_file and hcs_$terminate_seg completely terminate a segment given its path name or segment number, respectively.

The subroutine term_$single_refname performs the same operation as hcs_$terminate_name.

 (END)

Name: hcs_$terminate_noname

This subroutine terminates a null reference name from the specified segment. If this is the segment's only reference name, the segment is removed from the address space of the process. This entry is used to clean up after initiating a segment by a null name; see also the MPM write-up for hcs_$initiate. For a discussion of reference names, see the MPM Reference Guide section, Constructing and Interpreting Names.

Usage

```
declare hcs_$terminate_noname entry (ptr, fixed bin(35));

call hcs_$terminate_noname (segptr, code);
```

1) segptr is a pointer to the segment in question. (Input)

2) code is a standard storage system status code. (Output)

Note

The subroutine hcs_$terminate_name terminates a specified non-null reference name; hcs_$terminate_file and hcs_$terminate_seg completely terminate a segment given its path name or segment number, respectively.

 (END)

Subroutine Call
2/20/73

Name: hcs_$terminate_seg

This subroutine, given a pointer to a segment in the current process, removes all the reference names of that segment and then removes the segment from the address space of the process. For a discussion of reference names, see the MPM Reference Guide section, Constructing and Interpreting Names.

Usage

```
declare hcs_$terminate_seg entry (ptr, fixed bin(1),
        fixed bin(35));

call hcs_$terminate_seg (segptr, rsw, code);
```

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

2) rsw is the reserved segment switch. If equal to 1, the segment number should be saved in the reserved segment list; if equal to 0, the segment number should not be saved. (Input)

3) code is a standard storage system status code. (Output)

Notes

The subroutine hcs_$terminate_file performs the same operation given the path name of a segment instead of a pointer; hcs_$terminate_name and hcs_$terminate_noname terminate a single reference name.

The subroutine term_$segptr performs the same operation as hcs_$terminate_seg.

In fact, only those reference names are removed for which the ring level associated with the name is greater than or equal to the validation level of the process. If the user needs to concern himself with rings, he should refer to the MPM Subsystem Writer's Guide section, Intraprocess Access Control (Rings).

 (END)

Name: hcs_$truncate_file

This subroutine, given a pathname, truncates a segment to a specified length. If the segment is already shorter than the specified length, no truncation is done. The effect of truncating a segment is to store zeros in the words beyond the specified length.

Usage

```
declare hcs_$truncate_file entry (char(*), char(*),
        fixed bin, fixed bin(35));

call hcs_$truncate_file (dirname, ename, length, code);
```

1) dirname is the directory portion of the path name of the segment in question. (Input)

2) ename is the entry portion of the path name of the segment in question. (Input)

3) length is the new length (decimal) of the segment in words. (Input)

4) code is a standard storage system error code. (Output)

Notes

The subroutine hcs_$truncate_seg performs the same function when given a pointer to the segment instead of the path name. See also the restrictions discussed in that write-up under Notes.

 (END)

Name: hcs_$truncate_seg

This subroutine, given a pointer, truncates a segment to a specified length. If the segment is already shorter than the specified length, no truncation is done. The effect of truncating a segment is to store zeros in the words beyond the specified length.

Usage

```
declare hcs_$truncate_seg entry (ptr, fixed bin,
        fixed bin(35));

call hcs_$truncate_seg (segptr, length, code);
```

1) segptr is a pointer to the segment to be truncated. Only the segment number portion of the pointer is used. (Input)

2) length is the new length (decimal) of the segment in words. (Input)

3) code is a standard storage system status code. (Output)

Notes

The write attribute is required with respect to the segment.

A directory may not be truncated.

The implementation is such that pages will be thrown away starting from the next page after the word number length and the remainder of the last page will be zeroed.

The subroutine hcs_$truncate_file performs the same function when given the pathname of the segment instead of the pointer.

 (END)

hcs_$try_to_unlock_lock

Name: t[illegible]_unlock_lock

This entry checks that the indicated lock is locked by an existing process. If the process that locked the lock no longer exists, the lock will be reset to the lock ID of calling process. It is assumed that users of this entry obey the convention of locking locks with the system provided lock ID.

Usage

```
declare try_to_unlock_lock entry (ptr, fixed bin);

call try_to_unlock_lock (lock_ptr, state);
```

1) lock_ptr — is a pointer to the lock whose validity is in question. (Input)

2) state — = 1 if the lock is locked by an existing process;
= 2 if the lock is not locked;
= 3 if the lock was invalidly locked and has been reset. (Output)

hcs_$tty_abort

Entry: tty_index$tty_abort

This routine is used to reset read-ahead and/or write-behind (i.e., it flushes all I/O still in the buffers in tty_buf).

Usage

```
declare tty_index$tty_abort entry (fixed bin, fixed bin,
        fixed bin, fixed bin);

call tty_index$tty_abort (tw_index, reset_switch, state,
     code);
```

1) tw_index — identifies the typewriter channel. (Input)

2) reset_switch — may be as follows:

   1= reset only read-ahead;

   2= reset only write-behind; else, reset both. (Input)

3-4) are as above. (Output) (See hcs_$tty_index)

hcs_$tty_detach

Entry: tty_index$tty_detach

This procedure is called to detach a terminal. Input and output can be allowed to finish or can be terminated.

Usage

```
declare tty_index$tty_detach entry (fixed bin, fixed bin,
        fixed bin, fixed bin);

call tty_index$tty_detach (tw_index, dflag, state, code);
```

1) tw_index — identifies the typewriter channel. (Input)

2) dflag
- = 0 detach, finish read-ahead and write-behind, do not hangup but rather let the typewriter lie dormant. (Only the process currently using this typewriter can make this call.) The next process to invoke tty_index will be attached to this typewriter.
- = 1 detach and hangup after completing any output. (Only the answering service can make this call.) (Input)

3-4) are as above. (Output) (See hcs_$tty-index)

hcs_$tty_detach_new_proc

Entry: tty_index$new_proc

This entry is also used to detach a terminal. It does not hangup the terminal and makes it available to a named process.

Usage

```
declare tty_index$new_proc entry (fixed bin, bit(36)
        aligned, fixed bin, fixed bin);

call tty_index$new_proc (tw_index, nproc, state, code);
```

1) tw_index — identifies the typewriter channel. (Input)

2) nproc — is the ID of the process to which this typewriter is now available. (Input)

3-4) — are as above. (Output) (See hcs_$tty_index)

hcs_$tty_event

Entry: tty_index$tty_event

The tty_event call is used by the user ring portion of the typewriter DIM to inform the ring 0 portion of the event channel over which all further interprocess communication (i.e., wake-up signals) should occur.

Usage

```
declare tty_index$tty_event entry (fixed bin, fixed bin(71),
        fixed bin, fixed bin);

call tty_index$tty_event (tw_index, event, state, code);
```

1) tw_index — identifies the typewriter channel. (Input)

2) event — is the event channel over which subsequent interprocess communication is to occur. (Input)

3-4) are as above. (Output) (See hcs_$tty_index)

hcs_$tty_index

Entry: tty_index

The tty_index procedure is called through hcs_ by the User Ring portion of the typewriter Device Interface Module (DIM) during its processing of the attach call. This entry is used to assign a specific typewriter channel to the current process and to return a numeric typewriter index for use in identifying the channel in subsequent calls to the ring 0 typewriter DIM.

Usage

```
declare tty_index entry (char(*), fixed bin, fixed bin,
        fixed bin);

call tty_index (name, tw_index, state, code);
```

1) name — Is the symbolic name of the typewriter channel to be assigned. (Input)

2) tw_index — Is the numeric typewriter index to be used to identify the typewriter channel in all subsequent calls. (Output)

3) state — Is the current status of the typewriter channel and may take on any of the following numeric values:

   1 = channel inactive;
   2 = channel active, waiting for dialup;
   5 = typewriter is dialed up. (Output)

4) code — Is a status code and may take on one of the following values:

   0 = call processed normally;
   ¬0 = error detected. (Output)

hcs_$tty_order

Entry: tty_index$tty_order

This procedure is invoked to change the status, line length, or code conversion modes. It is also used to enable or disable the quit feature and it can be used to return certain information about a terminal.

Usage

```
declare tty_index$tty_order entry (fixed bin, char(*), ptr,
        fixed bin, fixed bin);

call tty_index$tty_order (tw_index, order, argptr, state,
     code);
```

1) tw_index — identifies the typewriter channel. (Input)

2) order — is the request made. (Input)

3) argptr — is an argument for use by requests. (Input)

4-5) — are as above. (Output) (See hcs_$tty_index)

Notes

The orders (see Usage immediately above) are:

hangup — hangs up the telephone connection.

info — returns the structure:

```
declare 1 info based (argptr) aligned,
          2 id char(4),
          2 pad char(8),
          2 type fixed bin;
```

1) id Is the terminal answer back code.

2) pad is padding.

3) type Is the kind of terminal.

1 = IBM 1050;
2 = IBM 2741 (with MIT modifications);
3 = Teletype Model 37;
4 = Terminet 300;
5 = ARDS;
6 = IBM 2741 (standard);
7 = Teletype Model 33 or 35.
8 = Teletype Model 38.

line_length sets the maximum line length to the integer pointed to by argptr. (Obsolete. Use modes.)

listen causes calls to this data set to be answered.

makebusy prevents calls to this data set from being answered.

modes changes various typewriter modes and then returns the previous settings. The following structure is used:

```
declare 1 modes based (argptr) aligned,
          2 len fixed bin (init(length(modes.str))),
          2 str char(128);
```

1) len Is the length of str. 128 characters is probably an adequate length.

2) str Is a string of key words representing modes separated by commas. If a key word is preceded by a not sign (¬), then that mode is turned off; otherwise, the mode is turned on. Modes not in str are not changed.

The key words and functions are:

can - does canonical form conversion.

hcs_$tty_order (3)

crecho	echos a carriage return upon receipt of a line-feed from a terminal. (This mode is applicable for device types 3, 4, 7, and 8 only.)
edited	does output conversion in edited mode.
erkl	does erase and kill processing.
esc	does input escape conversion.
llnn	sets line length to nn (or nnn).
rawi	does not code-convert input characters.
rawo	does not code-convert output characters.
red	allows red and black shift characters to be sent to terminal.
tabs	inserts tabs. If off, the tabs will be replaced with blanks.
default	is a combination of modes used to set the standard modes without the caller knowing what all the modes are. Currently, default is equivalent to can, esc, erkl, ^rawi, ^rawo.

The previous mode settings are returned in a form that may be used to restore the modes to their original settings. For example, a quit handler would:

```
str = "default";

call tty_order (twx, "modes", addr (modes), state, err);
```

The original modes are then restored by:

```
call tty_order (twx, "modes", addr (modes), state, err);
```

printer_off	turns off the printer.
printer_on	turns on the printer.
quit_disable	does not inform traffic control of quits. (Obsolete)
quit_enable	informs traffic control of quits. (Obsolete)

set_table	enters edit mode if the character string pointed to by argptr is XXeX, where X is any character. Otherwise, normal mode is in effect. (Obsolete. Use modes.)
start	sends wake-up to the current process.

Status codes for all tty DIM entries are:

1) error_table_$dec_int_assnd	the symbolic name is incorrect in some way.
2) error_table_$invalid_device	the symbolic name is incorrect in some way.
3) error_table_$io_no_permission	an attempt was made to access a TTY belonging to another process.
4) error_table_$undefined_order_request	an illegal order request was made.

hcs_$tty_read

Name: tty_read

Entry: $tty_read

The tty_read entry is used to code-convert and transfer to the caller's workspace data which has already been read-ahead from the user's typewriter. If a complete line has not as yet been read from the typewriter, control is immediately returned to the caller after insuring the typewriter channel is in read-ahead mode.

Usage

```
declare tty_read entry (fixed bin, ptr, fixed bin, fixed
        bin, fixed bin, fixed bin, fixed bin);

call tty_read (tw_index, workspace, offset, nelem, nelemt,
     state, code);
```

1) tw_index — identifies the typewriter channel. (Input)

2) workspace — is a pointer to the beginning of the caller's workspace into which read-ahead data is to be transferred. (Input)

3) offset — is the offset in characters from the start of the user's workspace and indicates where the data transfer should begin. (Input)

4) nelem — is the number of characters which may be transmitted to the user's workspace. Note that a new line character will terminate the data transfer unless the number of characters in the line exceeds nelem. In this case, the remaining characters will be transmitted in subsequent calls. To insure reading a line in proper canonical form, nelem must be large enough to contain the entire line as typed. (Input)

5) nelemt — is the number of characters actually transmitted by the read call. (Output)

6) state — is the status of the typewriter channel as described under tty_index. (Output)

7) code — is the status code as described under tty_index. (Output)

hcs_$tty_state

Entry: tty_index$tty_state

This entry returns the state of a typewriter channel.

Usage

```
declare tty_index$tty_state entry (fixed bin, fixed bin,
        fixed bin);

call tty_index$tty_state (tw_index, state, code);
```

1) tw_index identifies the typewriter channel. (Input)

2-3) are as above. (Output) (See hcs_$tty_index)

hcs_$tty_write

Name: tty_write

Entry: $tty_write

The tty_write entry is used to transfer data from the caller's workspace to the user's typewriter. The ASCII data in the workspace is converted to the proper device codes on the way to the typewriter. If all the data cannot be written at once, nelemt will be returned less than nelem. When this occurs, the user ring caller must block the process and, upon wakeup, reissue the call as outlined in the example below.

Usage

```
declare tty_write entry (fixed bin, ptr, fixed bin, fixed
        bin, fixed bin, fixed bin, fixed bin);

call tty_write (tw_index, workspace, offset, nelem, nelemt,
     state, code);
```

1) tw_index identifies the typewriter channel. (Input)

2) workspace is a pointer to the beginning of the caller's workspace from which data is to be transferred. (Input)

3) offset is the offset in characters from the start of the user's workspace and indicates where the data transfer should begin. (Input)

4) nelem is the number of characters which should be transferred from the user's workspace. (Input)

5) nelemt is the number of characters actually transmitted by the write call. (Output)

6) state is the status of the typewriter channel as described under tty_index. (Output)

7) code is the status code as described under tty_index. (Output)

If all the data was not written (i.e., nelem ≠ nelemt), the calling routine should first block and then try to write out the rest of the data. Also, there are cases when tty_write wants to force a return out of ring 0 to allow a possible quit interrupt

or timer runout. In these cases, it returns a code of 3000005. The caller should update offset, nelem and recall. A proper sequence of instructions is:

```
loop:  call hcs_$tty_write (--);
       if nelem ¬= nelemt then do;
            if code = 0
            then call ipc$block (--); /* go to sleep */
            else if code ¬= 3000005 then
            go to error;
            offset = offset + nelemt;
            nelem = nelem - nelemt;
            go to loop;
       end;
```

hcs_$unmask_ips

Entry: ips_$unmask

This entry is called to unmask specified IPS interrupts.

Usage

```
declare ips_$unmask_ips entry (bit(36) aligned, bit(36)
        aligned);

call ips_$unmask_ips (mask, oldmask);
```

1) mask — for each bit ON in this word, the corresponding IPS interrupt is unmasked. (Input)

2) oldmask — is the old IPS mask. (Output)

(See hcs_$get_ips_mask)