MULTICS TECHNICAL BULLETIN

MTB-198
ADDENDUM

To: Distribution

From: Steve Webber

Subject: New Hardcore Primitives

Date: May 22, 1975

Attached is an addendum to MTB-198. Please incorporate this as the first two pages of text.

To: Distribution

From: Steve Webber

Subject: New Hardcore Primitives

Date: May 1975

Questions

The attached MTB proposes some new primitives and interfaces for the Multics system. There are many unresolved issues, and, although solutions are often proposed, there is often little agreement that what is proposed is correct. The following list gives some of the more interesting unresolved issues. The reader is urged to keep this list in mind while reading the MTB.

1. Do we want to use varying strings as extensively as proposed?

2. Is the proposed new storage allocation technique (of using a pointer to a region instead of a PL/I area) better?

3. Is the new <par_segno, ename> interface needed for segments where <seg_ptr> interfaces are also provided? Indeed, would it be better to add a third class of interfaces that take <ep> (pointer to directory entry)?

4. Is the new include file scheme -- with version numbers -- appropriate? Do we want to support include files for system supported structures? Should include files give calling sequences for the interfaces?

5. Should MSFs be supported in ring 0? If so, how extensively? By how many interfaces?

6. Should star processing be removed from the hardcore?

7. Should the fs_move primitives be removed from the hardcore?

8. Should we retain the "SysDaemon" special casing in the ACL interfaces?

9. Should create_ and set_ allow manipulation of ACLs and names?

----

10. Do we want a primitive to return default values for set_ and create_?

11. Should create_ be more primitive and do much less -- for example not allow success if segment is already there?

12. Should a mechanism be provided to copy an entire directory out of the hardcore for user-ring perusal?

13. Should switch parameters be used or should alternate entry points be used?

14. Should we pass structures or pointers to structures?

15. How should status about items within a structure be handled? Do we want status codes returned in the structure?

16. How should links be interpreted (ASCII or binary)?

17. How should we specify keywords such as "working_dir" in search rules structures?

18. Should partial information be returned if there is not enough room for all information or if some nonfatal error occured?

19. Should we bother doing new primitives at all?

To: Distribution

From: Steve Webber

Subject: New Hardcore Primitives

Date: May 1975

Purpose

This memo proposes a new set of subroutines that would define a new interface to ring 0 and directory control, in particular. Some of the functions currently implemented in ring 0 (and invoked through the gate hcs_) will be removed from ring 0. The prime functions of interest being removed from ring 0 are 1) reference name management and 2) pathname management. Since these functions will be implemented outside of ring 0, it will be necessary to remove hcs_ itself from ring 0 so that the target of the hcs_ entries can use the new user-ring primitives for reference name and pathname management.

The need for the new interfaces arises because, for efficiency, reliability, cleanliness, and secureability we are changing the supervisor so that reference names and pathname management are removed from ring 0. With the requisite new set of interfaces, it behooves us to make other cleansing changes to the user interface both for efficiency and consistency. This memo proposes a set of primitive interfaces to directory control in light of this new structure. The programs that hcs_ used to invoke will now reside largely outside of ring 0. These programs will not be writearounds to the new primitives but rather compatible interfaces to new primitives when necessary. (Most of the directory control "primitives" of today are not that primitive, but rather invoke more primitive functions to perform their tasks.) It is this most primitive set of functions which is being proposed -- most other functions will be removed from ring 0.

Since hcs_ will no longer reside in ring 0, replacement gates for all the necessary ring 0 functions outside of directory control must also be provided. These are outlined as well.

This memo is divided into four sections. The first section describes overall goals of any set of interfaces that, I hope, will generally be agreed to. The second section describes a set of rules and conventions that I would propose as a means of satisfying the goals in section 1. The third section describes the actual proposed hardcore primitives which use the rules and

---

guidelines of section 2.

The issue of user-ring subroutines that must interface with the primitives proposed here should not be forgotten. In particular, it may well be that the inner-level primitives are the same ones offered at the user level. When this is appropriate, these primitives should be made available as user level subroutines from the start. Other user level subroutines (not in hcs_) can and should be designed in parallel with what is being proposed here. Section 4 gives some possible user-ring primitives.

Sections 2, 3 and 4 are completely open for debate and it is hoped that we can resolve the major issues in the very near future. We, of course, do not want to barge into the very important area of system primitive design and come up with anything thrown together in a haphazard way. We do, however, want to get on with this task as other work will inevitably depend on it.

Section 1. Design of System Primitives

There are several overall goals to be satisfied when designing a system primitive. There are further requirements when the primitives are to be user interfaces. The following requirements are generally applicable to any set of interfaces in the system:

1. The primitives must be efficient to use.

2. The use of the primitive and its name should be consistent with other primitives in the system, or at least the set of primitives of which it is a part.

3. The primitive set should be complete. There should be primitives to handle all of the normal needs of the caller.

4. The primitives should be extensible, where possible, to allow for future changes in the system.

5. The primitives should be as compatible as seems reasonable to what we have today.

6. The primitives should be easy to use if called from a ring different from the one they execute in.

7. The primitives should be easy to move from one ring to another if appropriate.

Section 2. Proposed Solutions to Requirements

1. The primitives must be efficient to use. There are many ways to design primitives which will make them more efficient to use. The problem is coming up with a useful set that are not too limited or too general. The following are proposed:

   A. Primitives should have as few arguments as is reasonable to minimize argument list preparation.

   B. Primitives should be designed, where possible, so that no descriptors are required in the argument lists. This means that character strings should be passed as char (N) varying, where N is constant; it also means that arrays should not be passed if they are variable in length -- instead a pointer to an array with bound should be passed.

   Some primitives today receive switches to indicate alternate options to take. Such switches should be embedded in an already existent structure, or avoided, if possible, by providing a different primitive.

2. Primitive sets should be consistent. This will require conventions such as:

   A. A status code is always the last argument.

   B. The names on "seg" and "file" primitives should be consistent and it should be possible to guess what the primitive name is and how it's called.

   C. The system functions should be consistently named ("initiate" should only be used with reference names, etc.).

   D. There should not be many primitives that do nearly the same thing. The primitives should be mutually exclusive where appropriate so that a user will know unambiguously which primitive is the one he should be calling.

3. The primitive set should be complete. This means that all necessary functions should be handled. It also means that if a "ptr" entry exists for a particular function and a "file" entry is meaningful, it too should exist. The primitives should be symmetric by providing all of the functions in a consistent, obvious way.

4. The primitives should be extensible. They should be designed so that new primitives can be added in an

obvious way and that existing primitives can be modified to provide new features when necessary. The proposed way to handle this problem is to use version numbers in structures. The structures used will be contained in include files which also include a variable giving the version number of the structure. Structures which are changed should be appended to rather than reordered if possible. All structures passed to the primitive set should have the version number in the first word of the structure.

The use of the version number would be as follows: the first version of a structure would be included in the include file for the structure (which should be available to users and mentioned, by name, in the user documentation). The include file should also include a declaration of the form:

```
declare STR_version_N fixed bin static init (N);
```

where STR is specific to the given structure and N is the version number. Users of the primitive should use the standard include file and should have a statement of the form:

```
STR.version = STR_version_N;
```

This would cause changes in a structure to be caught at recompilation time. Old versions of a structure should be made available in an include file named:

```
STR_old_N_.incl.pl1
```

where STR and N are as above. The user could change his %include statement to the old version if he did not want to recode for the new version (right away).

An error_table_ code should be returned saying a primitive does not support a particular version when that version becomes unsupported (if ever).

The include files provided by the system should probably end in an underscore.

5. The primitives should be functionally similar to what we have today, if possible. An example is the makeunknown/initiate functions of today. Both of these functions are provided by the hcs_$initiate primitive. Note that the initiate function will no longer be implemented in ring 0 and thus the low-level primitive could not provide both these functions.

6. Primitives should be easy to call from another ring. If a primitive is a gate, it must take special action with respect to its arguments. In particular (input) pointer valued arguments must be copied in a way that preserves the validation level. Further all other (input) arguments must be copied to guarantee valid verification of the arguments. This means that, due to the current way the compiler works, it would be inconvenient to pass pointers embedded in structures as structures are not copied in a way permitting hardware validation. A program must not read an input argument more than once or an output argument at all. Output arguments should be stored into exactly once.

   Returning values into a user-supplied area should also be avoided, both because it is more likely to cause crawlouts and because it prevents users from using their own area management routines. It is proposed that when large amounts of variable-length data is to be returned, the user provide a pointer to a region of storage into which the data is copied. The first word of this region should be filled in with the size of the region, in words, before making the call. Relative pointers and indices can then be used to locate selected data items. (A new set of user-ring primitives to manage temporary buffer segments is mentioned in section 4.)

7. It should be easy to move a primitive from one ring to another. Since many new primitives will arise in the near future and since the ring in which they reside depends on the hardcore system at the time, it should be easy to move primitives about with ease. To solve this problem, two requirements must be met. First, all primitives must follow the coding conventions for gates. Second, each primitive (or set of primitives if it is clear that they will forever be an integral set) should have a seperate (reference) name. This means that there will not be a replacement for hcs_ but rather many such. It does not mean that we will need multiple gate segments. Names will be moved to and from the new hardcore gate segment as primitives are moved into and out of the supervisor.

Section 3. The New Primitives

The new primitives fall into the following logical sets:

1. reference name primitives
2. pathname primitives
3. address space manager primitives
4. storage system primitives
5. linker primitives

6. interprocess communication primitives
7. general hardcore utility primitives
8. interprocess signalling primitives
9. hardcore I/O primitives

The primitives described below are intended to form a complete set for all classes except for the Storage System primitives. The storage system primitives, by far the largest set, is only partially specified below. The major omissions are intentional due primarily to the lack of any backup primitives.

In the following descriptions, underlined parameters are output. Items in structures marked with a star (*) are input.

Reference Name Primitives

There are eight proposed primitives for the reference name manager. These are all entries in the procedure ref_name_ which will eventually be a user-ring primitive. These primitives are used by the following hcs_ entries (but may be called directly):

fs_get_call_name
fs_get_ref_name
fs_get_seg_ptr
initiate
initiate_count
make_ptr
make_seg
terminate_noname
terminate_file
terminate_name
terminate_seg

The following is a list of arguments used in the reference name primitives:

1. segno — fixed bin (15) — is the segment number of the segment of interest

2. rname — char (32) var — is the reference name of interest

3. struct_ptr — ptr — is a pointer to the following structure:

```
dcl 1 rni aligned based,
*     2 names_allocated fixed bin,
      2 names_returned fixed bin,
      2 refnames (refer (rni.names_returned))
                    char(32) var;
```

4. entry_ptr — ptr — is a pointer to a (PL/I, e.g.) entrypoint;

5. code fixed bin (35) is a returned status code.

Entry: ref_name_$add (segno, rname, code)

This primitive associates the given reference name with the specified segment number.

Entry: ref_name_$delete (rname, segno, code)

This primitive removes the given reference name from the list of those for the segment to which it is associated.

Entry: ref_name_$delete_segno (segno, code)
ref_name_$delete_refnames (rname, code)

This primitive removes all reference names associated with a given segment.

Entry: ref_name_$get_segno (rname, segno, code)

This primitive returns the segment number of the segment to which a particular reference name is associated.

Entry: ref_name_$get_refnames (segno, struc_ptr, code)
ref_name_$get_refnames_rn (rname, struc_ptr, code)

This primitive returns a user-specified number of reference names associated with the given segment. The parameter struc_ptr points to the rnl structure specified above.

Entry: ref_name_$get_entry_ptr (rname, entry_ptr, code)

This primitive returns a pointer to the entry rname$rname if such an entry exists, i.e., if an entrypoint with the same name as rname exists in the segment whose reference name is rname a pointer to this entry is returned.

The ref_name_ primitives will originally reside in ring 0 and ref_name_ will be a gate with ring brackets (0,0,7). When the reference name manager is removed from ring 0, ref_name_ will no longer be a gate. It will have ring brackets of (1,7,7).

Pathname Management Primitives

There are two pathname management primitives which will remain after the full conversion to the reference name manager. Since pathnames will no longer be recognized in ring 0, eventually, the functions of find_ and hcs_$fs_get_path_name will be provided in the user ring.

The following are arguments for the pathname management primitives:

1. dir_path char (*) var is a directory pathname,

2. dir_segno fixed bin (15) is the segment number of a directory,

3. segno fixed bin (15) is the segment number of a segment (possibly a directory),

4. path char (*) var is a full pathname generated by the primitive,

5. code fixed bin (35) is a status code.

The primitives are:

Entry: find_dir_segno_ (dir_path, dir_segno, code)

This primitive parses the given directory pathname and returns the segment number of the directory specified.

Entry: get_pathname_ (segno, path, code)

This entry generates a pathname for the given segment by concatenating the primary entry names of all superior directories (separated by ">"s).

Note that the pathname manager may use an internal associative memory to avoid a recursion that logically proceeds to the root. Use of such an associative memory perpetuates the current system bug causing strange behavior if directories are renamed.

An outer level primitive that converts a (relative) pathname of a segment into a pointer will also be provided. It, however, is one level removed from these primitives and is not discussed until section 4 of this MTB.

Primitives to Make Segments Known and Unknown

The following set of primitives will always reside in ring 0. They manage the binding of segment numbers to objects in the hierarchy and are the interface to what is commonly called the address space manager.

These primitives are used by the following hcs_ entries:

initiate
initiate_count
make_seg
makeunknown
terminate_file
terminate_name
terminate_noname
terminate_seg

Some of the functions provided by the above hcs_ entries are handled by the reference name management primitives.

The following are arguments used in this set of primitives:

1.	par_segno	fixed bin (15)	is a directory segment number,
2.	ename	char (32) var	is an entryname in a directory,
3.	segno	fixed bin (15)	is a segment number,
4.	struc_ptr	ptr	is a pointer to a structure of (mainly) returned information,
5.	flags	bit (36)	is a string of flags used to control what unbind_segno_ does. The first bit is the "reserve" bit; the second bit is the "force" bit; others are reserved for future expansion.
6.	code	fixed bin (35)	is a status code.

The pointer struc_ptr above points to a user-supplied structure in which information is returned. This structure is declared as follows:

```
dcl 1 bsi aligned based,
 *    2 version fixed bin,
(*)   2 segno fixed bin(15),
 *    2 control,
        3 reserve bit (1) unal,
        3 dirsw bit (1) unal,
        3 no_write bit (1) unal,
        3 mbz bit (33) unal,
      2 bit_count fixed bin (24),
      2 mode like mode,
      2 flags,
        3 seg_already_known bit (1) unal,
        3 may_or_may_not_be_there bit (1) unal,
        3 rest bit (34) unal,
      2 linkname char (168) var;

dcl 1 mode aligned based,
      2 read bit (1) unal,
      2 execute bit (1) unal,
      2 write bit (1) unal,
      2 mbz bit (33) unal;
```

The primitives are:

Entry: bind_segno_ (par_segno, ename, struc_ptr, code)

This primitive binds a segment number to the segment whose name is ename in the directory whose segment number is par_segno unless ename specifies a link. If ename specifies a link, linkname is filled in and a status code is returned. If the segment is already known, that number is returned but the seg_already_known flag is set.

If the entry being made known is a directory, the may_or_may_not_be_there flag is set if it has not yet been established that the user can know about the directory.

Whenever a call to bind_segno_ is made, a usage count for the calling ring is incremented in the KST entry for the segment. This usage count mechanism allows users of bind_segno_ to have a clean, efficient interface analogous to the null reference name interface of today but without the overhead of any name management. Indeed, any programs that merely want a pointer to a segment and have no need for any reference name functions would work smoothly. The bind_segno_ and unbind_segno_ interfaces therefore replace the hcs_$initiate[_count] and hcs_$terminate_noname interfaces in a large number of cases.

Entry: priv_bind_segno_ (par_segno, ename, struc_ptr, code)

This primitive works similarly to bind_segno_ except that special action is taken in ring 0 to allow the segment being made known to be referenced without full regard to the access isolation mechanism's controls.

Entry: unbind_segno_ (segno, flags, code)

This primitive decrements the usage count of the specified segment for the calling ring. If all usage counts are 0, the segment is made unknown. Similarly, if the "force" bit is ON (bit 2 of flags), and all usage counts are 0 in all inner rings, the segment is made unknown.

If the segment is made unknown then its segment number is returned to the free pool of segment numbers unless the "reserve" bit (bit 1 of flags) is ON.

Storage System Primitives

The following set of primitives are intended to represent the external interface to directory control. These primitives are more primitive than the hcs_ entries of today and are also used within ring 0. There are several major changes being proposed including:

1. No allocations in user areas will be done,

2. No star processing will be done,

3. No pathnames will be accepted, and

4. Status "flags" are used instead of some "codes".

The primitives necessary for backup and the reloader are not mentioned here. These primitives will be designed later when a better understanding of the requirements of backup is available. It should be noted, however, that backup will have its own primitives and that the normal user primitives will therefore not be constrained by some little used or otherwise unnecessary feature of backup. It should also be noted that the user-ring hcs_ entries will all be supported and that backup can continue to use these until new backup primitives are available.

Most of the storage system primitives use structures to communicate both input and output information. Some of these structures are too detailed or cumbersome for a clear user interface and so, where appropriate, many user-ring primitives are provided to interface to the actual hardcore primitives (the most obvious set of user-ring primitives will be hcs_).

The following is a list of arguments common to many of the storage system interfaces:

1. par_segno fixed bin (15) is a directory segment number,

2.	ename	char (32) var	is an entry name in a directory,
3.	struc_ptr	ptr	points to an input/output structure,
4.	seg_ptr	ptr	is a pointer to a segment.
5.	names_ptr	ptr	is a pointer to a structure for returning names.
6.	acl_ptr	ptr	is a pointer to an ACL structure.
7.	linkname	char (168) var	is a link name managed by ring 0.
8.	newname	char (32) var	is a name being added to an entry.
9.	delname	char (32) var	is a name being deleted from an entry.
10.	oldname	char (32) var	is a name currently on an entry that is to be deleted.
11.	dir_segno	fixed bin (15)	is a directory segment number.
N.	code	fixed bin (35)	is a returned status code.

Since many of the primitives return information in a structure, some of the structures also include status codes and status flags associated with specific items within the structure. This intermixing of returned values and returned status is a compromise in style and is proposed for lack of a better method known to me. In general, when a "code" field in a structure is returned with a nonzero error_table_ value, the "code" argument will be set to error_table_$partially_successful (or some such name). If a code argument of zero is returned, all structure code fields will be zero, but some status "flags" may be set. See the individual primitive descriptions for more details.

One more item worth mentioning about the storage system primitives is that it is intended that directory operations and file operations ("file" refers to single segment or multisegment "files") be completely independent. There are separate primitives for manipulating these at many levels in the storage system. The intent is to give users appropriate warning if they appear to be performing an operation on the wrong kind of entity.

The new storage system will require certain changes to the user-visible interface to directory control. Some of these changes are required for new capabilities while others are intended to encourage users to "ask the right questions" because of efficiency considerations. An example of new features is the possibility of a very fast fs_move function in certain cases. When and if this is available, new primitives will be proposed. An example of asking the wrong question arises when segment status is requested. The new storage system is potentially more expensive when returning complete status especially if the status for all segments in a directory is requested. For this reason, the default for "list" etc., should be changed to work in the most efficient way.

The storage system primitives are divided into the following classes:

1. creating primitives
2. naming primitives
3. deleting primitives
4. status primitives
5. set primitives (attribute changing)
6. acl primitives
7. quota primitives
8. truncation primitives
9. utility primitives

Primitives for Creating Segments, etc.

The following primitives create an entry in a directory. A major functional change from what is available today is that a user may specify all the reasonable attributes to be applied to a segment. This is because a user with append permission on a directory who does not have modify permission as well, must be able to say everything about the branch being created in the create_ call. (Another independent proposal would require awarding append permission on a directory only if modify permission is awarded.) A second functional change is the addition of the concept of multisegment files into the storage system. This is done by providing a mechanism for converting segments to and from multisegment files and by enforcing some consistency on MSF's in directory control. (For example, making an MSF known should return a (warning) status and creating and deleting MSF components should be special cased. For details of the MSF implementation proposals see the actual descriptions of the primitives below.)

Entry: create_ (par_segno, ename, struc_ptr, code)

This primitive creates a vanilla flavored, single segment file. The parameter struc_ptr points to the following structure:

```
dcl 1 cri aligned based,
*     2 version fixed bin,
*     2 options,
        3 change_attributes_if_exists bit(1) unal,
        3 dont_use_inacl bit (1) unal,
        3 truncate bit (1) unal,
        3 mbz bit (33) unal,
*     2 mode like mode,
*     2 set_array like set_array,
*     2 set_info like set_info;

dcl 1 set_array aligned based,
      2 bit_count bit (1) unal,
      2 ring_brackets bit (1) unal,
      2 entry_bound bit (1) unal,
      2 access_class bit (1) unal,
      2 max_length bit (1) unal,
      2 mbz bit (31) unal;

dcl 1 set_info aligned based,
      2 switches,
        3 safety bit (1) unal,
        3 copy_on_write bit (1) unal,
        3 entry_bound bit (1) unal,
        3 multiple_class bit (1) unal,
        3 mbz bit (32) unal,
      2 bit_count fixed bin (24),
      2 ring_brackets(3) fixed bin (3),
      2 ring_bracket_code fixed bin (35),
      2 entry_bound fixed bin (14),
      2 max_length fixed bin (35),
      2 access_class bit(72);
```

This primitive creates the named segment and sets the various attributes as specified. If the segment exists and "change_attributes_if_exists" is ON, the primitive works like the set_ primitive and merely updates the attributes. Similarly, if the segment exists and "truncate" is ON, the segment will be truncated. The "dont_use_inacl" switch instructs the primitive not to use the initial ACL. The primitive will, in any case, place an ACL entry on the segment consisting of the mode specified (for the calling process's user ID).

The "set_array" field instructs the primitive to use the specified item from the input structure. If a bit is OFF in the set_array field, the corresponding attribute is set by default. For example, the ring brackets would be set to (v,v,v) if cri.set_array.ring_brackets were OFF.

Entry: create_$dir (par_segno, ename, struc_ptr, code)

This primitive creates a directory branch in the specified directory. The input pointer struc_ptr points to the following structure:

```
dcl 1 cdi aligned based,
*     2 version fixed bin,
*     2 options,
        3 change_attributes_if_exists bit (1) unal,
        3 dont_use_inacl bit (1) unal,
        3 mbz bit (34) unal,
*     2 mode like dirmode,
*     2 set_array,
        3 mbz bit (1) unal,
        3 ring_brackets bit (1) unal,
        3 mbz bit (34) unal,
*     2 quota fixed bin,
      2 quota_code fixed bin (35),
*     2 access_class bit (72),
      2 access_class_code fixed bin (35),
*     2 ring_brackets (2) fixed bin (3),
      2 ring_brackets_code fixed bin (35);

dcl 1 dirmode aligned based,
      2 status bit (1) unal,
      2 modify bit (1) unal,
      2 append bit (1) unal,
      2 mbz bit (33) unal;
```

An ACL of "mode" for the calling process's user ID is set.

Entry: create_$link (par_segno, ename, linkname, code)

This primitive creates a link entry in the directory specified. Since links are not interpreted in ring 0, linkname is allowed to be any binary data of up to 168*9 bits in length.

Entry: create_$msf (par_segno, ename, struc_ptr, code)

This primitive creates an MSF entry in the directory specified. Like create_, it is not a fatal error if the entry already exists. However, if the entry is initially a single segment file it is converted to an MSF with the following mappings:

1. The ACL on the MSF component is set to the specified mode; the ACL on the MSF directory is copied from the ACL of the parent directory,

2. The max_length for the MSF, if not specified by the user, is set to sys_info_$max_msf_size, and

3. The segment initial ACL for the MSF directory is set to the segment initial ACL from the MSF's parent directory. (The directory initial ACL is set null.)

Entry: create_$msf_component (par_segno, ename, comp_no, code)

This primitive will create the comp_no'th component of an MSF. Several components may be created in order to ensure the consistency (contiguity) of the MSF. All attributes on the segment created are set to those of the other components of the MSF (guarenteed consistent by the other primitives of directory control). The parameter comp_no is an integer which must be greater than the current number of components in the MSF and less than the maximum number of allowed components.

An MSF component can be created by any user with write permission on the MSF as long as the max_length of the MSF is not exceeded. (Any user with modify permission on the MSF directory can change the max_length.)

Primitives for Manipulating Names

When a segment (dir, link, MSF) is initially created, a single name is associated with it. This name is the primary name and will remain the primary name until it is removed no matter how many other names are subsequently added or deleted. The following primitives are used to change names. To find all names, see the status_ primitives.

Entry: names_$add (par_segno, ename, newname, code)
names_$add_ptr (seg_ptr, newname, code)

This primitive adds the name newname to the list of names associated with the given segment (dir, link, MSF).

Entry: names_$delete (par_segno, ename, delname, code)
names_$delete_ptr (seg_ptr, delname, code)

This primitive removes the given name from the given segment (dir, link, MSF). If delname is the last name on the entry it is not removed and an error code is returned. If delname is the primary name, a new primary name is chosen by the primitive.

Entry: names_$change (par_segno, ename, oldname, newname, code)
names_$change_ptr (seg_ptr, oldname, newname, code)

This primitive replaces the name oldname with the name newname. If oldname is the primary name, the primary name is changed to newname.

Primitives for Deleting Segments, etc.

The following primitives are used to delete entries from directories. All require modify permission on the parent directory.

Entry: del_ (par_segno, ename, code)
del_$ptr (seg_ptr, code)

This primitive deletes the specified segment.

Entry: del_$dir (par_segno, ename, code)
del_$dir_ptr (dir_segno, code)

This primitive deletes the specified directory. If the directory has any branches in it, it is not deleted and an error is returned.

Entry: del_$link (par_segno, ename, code)

This primitive deletes the specified link from the specified directory.

Entry: del_$msf (par_segno, ename, code)

This primitive deletes the MSF named ename from the directory whose segment number is par_segno.

Primitives for Returning Status

There are two major changes to the status primitives. First, the star convention is not recognized. Therefore listing programs must be returned an entire directory's contents. Second, because of new storage system considerations, there will be just two forms of returned status, brief and long. Basically, brief status contains information about the segment independent of its size or use.

Recall that this MTB does not propose replacements for the backup primitives. Hence, the status primitives described below should not be expected to be acceptable for backup use.

One last point to mention is that the primitives below are the hardcore primitives. Additional, user-ring primitives (such as those in hcs_) will augment the hardcore primitives to make a more usable set.

The status primitives are divided into two classes, those which return what are called "directory" attributes and those which are called "segment" attributes. To use the primitives which return directory attributes, status permission on the containing directory is required; for segment attributes all that is required is nonnull access on the segment. (This means that to get status of a link, status permission is required on the containing directory -- since links do not have ACLs.)

<u>Entry</u>: status_ (par_segno, ename, struc_ptr, names_ptr, <u>code</u>)
status_$ptr (seg_ptr, struc_ptr, names_ptr, <u>code</u>)

This primitive returns selected status about a segment (dir, link, MSF) useful to normal users. It requires status permission on the containing directory. Either struc_ptr or names_ptr may be null. If either is, the associated information is not returned. If struc_ptr is nonnull it points to the following structure. The items starting with "dtu" on to the end of the structure are all set to 0.

```
dcl 1 sti aligned based,
*     2 version fixed bin,
*     2 size_allocated fixed bin,
*     2 control,
        3 primary_name_only bit (1) unal,
        3 mbz bit (35) unal,
      2 data like status_info;

dcl 1 status_info aligned based,
      2 type fixed bin,
      2 nnames fixed bin,
      2 ptr_to_first_name ptr unal,
      2 (dtem, dted) bit (36),
      2 uid bit (36),
      2 author char (32) var
      2 effmode like mode,
      2 bit_count_msf_ind fixed bin (24),
      2 switches,
        3 safety bit (1) unal,
        3 copy_on_write bit (1) unal,
        3 entry_bound bit (1) unal,
        3 multiple_class bit (1) unal,
        3 mbz bit (32) unal,
      2 ring_brackets (3) fixed bin (3),
      2 bit_count_author char (32) var,
      2 entry_bound fixed bin (14),
      2 access_class bit (72),
      2 device_name char (32) var,
      2 ex_effmode bit (36),

      2 (dtu, dtm, dtd) bit (36),
      2 records fixed bin (9),
      2 cur_length fixed bin (35),
      2 max_length fixed bin (35),
      2 mbz (13) fixed bin (35);
```

The type element of the status_info structure is interpreted as follows:

```
0           link
1           segment
2           directory
3           MSF
```

This primitive is used by first filling in the starred items and then calling ring 0. If the entry of interest is a link, type is set to 0 and struc_ptr will be assumed to point to the following structure:

```
dcl 1 lki aligned based,
*     2 version fixed bin,
*     2 size_allocated fixed bin,
*     2 control,
        3 primary_name_only bit (1) unal,
        3 mbz bit (35) unal,
      2 type fixed bin,
      2 nnames fixed bin,
      2 ptr_to_first_name ptr unal,
      2 (dtem, dtd) bit (36),
      2 uid bit (36),
      2 author char (32) var,
      2 linkname char (168) var,
      2 mbz (4) fixed bin (35);
```

If the entry of interest is a directory, type is set to 2 and struc_ptr will be assumed to point to the same structure as a nondirectory segment. However, certain items are not defined.

The pointer names_ptr, if nonnull, should point to the following structure:

```
dcl 1 names_str aligned based,
*     2 size_allocated fixed bin,
      2 names (1) char (32) varying;
```

The variable "ptr_to_first_name" in the status structure points to one of the names in the above array. As before, the starred items should be filled in before the call. The variable size_allocated is in words. If the flag "primary_name_only" is set ON in the control array, only the primary name will be returned. In any case, "ptr_to_first_name" will always point to the primary name.

Entry: status_$long (par_segno, ename, struc_ptr, names_ptr, code)
status_$long_ptr (seg_ptr, struc_ptr, names_ptr, code)

This primitive is the same as status_ except the last items (from "dtu" onward) are also returned. With the new storage system, this primitive is potentially more expensive than the status_ primitive.

<u>Entry</u>: status_$all (par_segno, ename, struc_ptr, names_ptr, <u>code</u>)
status_$all_ptr (dir_segno, struc_ptr, names_ptr, <u>code</u>)

This primitive is called to return status information about all entries of a directory. (Any star reduction of the information is done after this call -- in the user ring.) When this entry is called names_ptr is generally not null and points to the same structure as for the status_ call. The struc_ptr parameter may not be null and must point to the following structure:

```
dcl 1 all_stl aligned based,
*     2 version fixed bin,
*     2 size_allocated fixed bin,
*     2 control,
        3 primary_name_only bit (1) unal,
        3 totals_only bit (1) unal,
        3 not_this_type (0:3) bit (1) unal,
        3 mbz bit (30) unal,
      2 num_entries fixed bin,
      2 num_this_type (0:3) fixed bin,
      2 num_names fixed bin,
      2 num_names_this_type (0:3) fixed bin,
      2 data (1) like status_info;
```

(Due to the potentially large amount of storage needed to dump large directories it will quite often be useful to acquire a temporary segment for the returned information.)

If "totals_only" is set ON, only the number of segments, directories, etc., will be returned. (In this case names_ptr may be null). The items "num_entries" and "num_this_type" are always returned. If "not_this_type (i)" is ON, information about the specified entries is not returned.

<u>Entry</u>: status_$all_long (par_segno, ename, struc_ptr, names_ptr, <u>code</u>)
status_$all_long_ptr (dir_segno, struc_ptr, names_ptr, <u>code</u>)

This primitive works as the status_$all primitive except that the last items in the status_info structure also also returned. As above this may be more expensive with the new storage system.

<u>Entry</u> seg_status_ (par_segno, ename, struc_ptr, <u>code</u>)
seg_status_$ptr (seg_ptr, struc_ptr, <u>code</u>)

This primitive is called to return segment attributes of a segment and therefore requires nonnull access on the segment. The parameter struc_ptr points to the following structure:

```
dcl 1 segsti aligned based,
*     2 version fixed bin,
      2 type fixed bin,
      2 effmode like mode,
      2 bit_count fixed bin (24),
      2 entry_bound fixed bin (14),
      2 records fixed bin,
      2 cur_length fixed bin (35),
      2 max_length fixed bin (35);
```

The specified entry must be a segment or a multisegment file.

The above entries are used by the following hcs_ entries:

```
hcs_$fs_get_brackets
hcs_$fs_get_mode,
hcs_$get_author,
hcs_$get_bc_author
hcs_$get_dir_ring_brackets
hcs_$get_max_length
hcs_$get_max_length_seg
hcs_$get_ring_brackets
hcs_$get_safety_sw
hcs_$get_safety_sw_seg
hcs_$star_
hcs_$star_list_
hcs_$status
hcs_$status_
hcs_$status_long
hcs_$status_minf
hcs_$status_mins
```

Primitives to Change Attributes

The following set of primitives is used for changing attributes of an entry. As with the status primitives, the distinction is made between directory attributes (requiring status permission on the containing directory) and segment attributes (requiring write access on the segment).

Entry: set_ (par_segno, ename, struc_ptr, code)
set_$ptr (seg_ptr, struc_ptr, code)

This primitive requires modify permission on the containing directory. The parameter struc_ptr points to the following structure:

```
dcl 1 seti aligned based,
*     2 version fixed bin,
*     2 control like set_array,
*     2 info like set_info;
```

where set_info and set_control are specified in the description of the create_ primitives.

<u>Entry</u>: seg_set_ (par_segno, ename, struc_ptr, <u>code</u>)
seg_set_$ptr (seg_ptr, struc_ptr, <u>code</u>)

This primitive will change segment attributes on a segment and hence does not require as much access as the set_ primitive. (It requires write permission with respect to the segment.) The parameter struc_ptr points to the following structure:

```
dcl 1 sseti aligned based,
*     2 version fixed bin,
*     2 control like set_array,
*     2 bit_count fixed bin (24),
*     2 entry_bound fixed bin (14);
```

<u>Primitives for Manipulating ACLs</u>

This set of primitives is used to add, delete, replace and list ACLs on segments or directories. The distinction is made between directories and segments because, although the structures are quite similar today, we should not get trapped by this. (MSFs are treated as segments.) There are four classes of primitives each of which has an entrypoint for adding, deleting, listing and replacing ACL entries. The four primitive classes are found in acl_, dir_acl_, inacl_ and dir_inacl_.

<u>Entry</u>: acl_$add (par_segno, ename, acl_ptr, <u>code</u>)
acl_$add_ptr (seg_ptr, acl_ptr, <u>code</u>)

This primitive adds the specified ACLs to the specified entry. If a userid is encountered which is already on the ACL for the entry the ACL entry is replaced. In this, and in all of the ACL manipulating entries, the parameter acl_ptr must point to the following structure:

```
dcl 1 acli aligned based,
*     2 version fixed bin,
*     2 n_acls_allocated fixed bin,
(*)   2 count fixed bin,
(*)   2 acla (1),
        3 userid,
          4 personid char (22) unal,
          4 projectid char (9) unal,
          4 tag char (1) unal,
        3 mode like mode,        /* or like dirmode */
        3 exmode bit (36),
        3 code fixed bin (35);
```

Entry: acl_$delete (par_segno, ename, acl_ptr, code)
acl_$delete_ptr (seg_ptr, acl_ptr, code)

This primitive deletes any ACL entries from the specified branch that exactly match one of the userid fields in the input ACL structure. If a specified userid is not on the ACL of the branch the associated code is set and the code parameter is also set.

Entry: acl_$list (par_segno, ename, acl_ptr, code)
acl_$list_ptr (seg_ptr, acl_ptr, code)

This primitive will return ACL information about the specified branch. All ACLs are to be listed and acli.count is set to the number listed. If there is not enough space allocated to list all of the ACL entries, as many as can be returned are and an error code is returned.

Entry: acl_$replace (par_segno, ename, acl_ptr, code)
acl_$replace_ptr (seg_ptr, acl_ptr, code)

This primitive will replace the entire ACL by the ACL specified in the input ACL structure.

The following ACL primitives work analogously and are listed here for completeness:

dir_acl_$add (dir_segno, acl, code)
dir_acl_$delete (dir_segno, acl, code)
dir_acl_$list (dir_segno, acl, code)
dir_acl_$replace (dir_segno, acl, code)
inacl_$add (dir_segno, acl_ptr, ring, code)
inacl_$delete (dir_segno, acl_ptr, ring, code)
inacl_$list (dir_segno, acl_ptr, ring, code)
inacl_$replace (dir_segno, acl_ptr, ring, code)
dir_inacl_$add (dir_segno, acl_ptr, ring, code)
dir_inacl_$delete (dir_segno, acl_ptr, ring, code)
dir_inacl_$list (dir_segno, acl_ptr, ring, code)
dir_inacl_$replace (dir_segno, acl_ptr, ring, code)

Primitives for Manipulating Quota

There are three primitives currently being proposed for manipulating quota. (The newly proposed directory record quota of the new storage system is not covered here.) The primitive to move quota from a directory to its parent or vice versa is:

Entry: quota_$move (par_segno, ename, quota, code)
quota_$move_ptr (dir_segno, quota, code)

This primitive will accept a positive or negative value for quota. If quota is positive, that many records of quota are moved from the parent of the directory ename to ename itself. If quota is negative, the absolute value of quota records are moved from the directory ename to its parent.

Entry: quota_$get (par_segno, ename, struc_ptr, code)
quota_$get_ptr (dir_segno, struc_ptr, code)

This primitive returns quota information about the directory ename. The parameter struc_ptr points to the following structure:

```
dcl 1 qi aligned based,
*     2 version fixed bin,
      2 quota fixed bin,
      2 used fixed bin,
      2 time_record_product fixed bin (71),
      2 time_updated fixed bin (71),
      2 inferior_quotas fixed bin,
      2 terminal_quota bit (1);
```

Entry: hpquota_$set (par_segno, ename, quota, code)
hpquota_$set_ptr (dir_segno, quota, code)

This primitive is privileged to system administrators and provides a means of specifying the quota for a directory without moving it from the parent directory. It is the only means (other than backup primitives not mentioned here) of "generating" quota.

Primitives for Truncating Segments

The following set of primitives are used for truncating segments and MSF's. They require only write permission on the associated segment.

Entry: truncate_ (par_segno, ename, offset, code)
truncate_$ptr (seg_ptr, offset, code)

This primitive truncates the single-segment file specified. The parameter offset specifies the first word truncated.

Entry: truncate_$msf (par_segno, ename, offset, code)

This primitive truncates the specified MSF. As many MSF components as are necessary are deleted in order to bring the MSF down to the specified size. The first component, however, is not deleted.

Primitives of General Utility to the Storage System

The following primitives do not fall into a well defined group and are listed here to complete the list of Storage System primitives.

Entry: user_effmode_ (par_segno, ename, userid, mode, code)
user_effmode_$ptr (seg_ptr, userid, mode, code)

Entry: level_$get (level)

Entry: level_$set (level)

Primitives Used by the Linker

The following set of primitives will be used by the linker. They will initially be available (only) in ring 0, but will be moved to the user ring when name space management and the linker itself are. The primitives below are used by the following hcs_ entries (many of which are not used and are obsolete):

hcs_$assign_linkage
hcs_$fs_search_get_wdir
hcs_$fs_search_set_wdir
hcs_$get_count_linkage
hcs_$get_defname_
hcs_$get_linkage
hcs_$get_lp
hcs_$get_rel_segment
hcs_$get_search_rules
hcs_$get_seg_count
hcs_$get_segment
hcs_$high_low_seg_count
hcs_$initiate_search_rules
hcs_$link_force
hcs_$make_ptr
hcs_$rest_of_datmk_
hcs_$set_lp
hcs_$unsnap_service

The following declarations apply to parameters used by the linker primitives:

storage_ptr	ptr	a pointer to a region of storage allocated for the user
size	fixed bin(18)	the size, in words, of storage to be allocated
working_dir	char(*)varying	is a character string representation of the current working directory. When used as an output quantity it is an absolute pathname; when used as an input quantity it may be a relative pathname.
wdir_segno	fixed bin(15)	is the segment number of the current working directory.
softcore_segno	fixed bin(15)	is the segment number of the first softcore segment.
first_user_segno	fixed bin(15)	is the segment number of the first user-ring segment beyond the softcore segments.
last_valid_segno	fixed bin(15)	is the last valid segment number available to the process.
stack_segno	fixed bin(15)	is the segment number of the (standard) stack segment for the calling ring.
last_used_segno	fixed bin(15)	is the largest segment number used by the process.

<u>Entry</u>: assign_storage_ (<u>storage_ptr</u>, size, <u>code</u>)

This primitive allocates size words of storage (on an even word boundary) in a process's combined linkage segment. A new segment will be created if there is not enough room left in the current segment (or region).

<u>Entry</u>: unassign_storage_ (storage_ptr, size, <u>code</u>)

This primitive returns the given storage to the ring's free pool of storage. (Initially, this function will have no effect.)

<u>Entry</u>: wdir_$get (<u>working_dir</u>, <u>code</u>)

This primitive returns the character string representation of the working directory for the current ring.

Entry: wdir_$get_ptr (wdir_segno, code)

This primitive returns the segment number of the current ring's working directory.

Entry: wdir_$set (working_dir, code)

This primitive sets the working directory for the current ring given a (relative) pathname.

Entry: wdir_$set_ptr (wdir_segno, code)

This primitive sets the working directory for the current ring given the segment number of the directory.

Entry: segno_limits_ (softcore_segno, first_user_segno, last_valid_segno, last_used_segno)

This primitive returns values of useful segment number ranges.

Entry: get_stack_segno_ (stack_segno)

This primitive returns the segment number of the (first) stack segment (created by the supervisor) for the calling ring. This and segno_limits_ are the only two linker primitives that will remain in the supervisor.

Entry: search_rules_$get (struc_ptr, code)

This primitive returns the character string forms for the search rules in effect for the current ring. The parameter struc_ptr points to the following structure:

```
dcl 1 search_rules aligned based,
*     2 count_allocated fixed bin,
      2 count_returned fixed bin,
      2 rules (1 refer (search_rules.count_returned))
              char (168) varying;
```

Entry: search_rules_$get_ptr (struc_ptr, code)

This primitive returns the directory segment numbers for the directories in the current search rules. The following artificial segment number mappings (of today) apply to keywords:

Keyword	Segment Number
initiated_segments	1
referencing_dir	2
working_dir	3

The parameter struc_ptr points to the following structure:

```
dcl 1 search_ptrs aligned based,
*     2 count_allocated fixed bin,
      2 count_returned fixed bin,
      2 rules (1 refer (search_ptrs.count_returned))
              fixed bin (15);
```

Entry: search_rules_$set (struc_ptr, code)

This primitive sets the search rules for the current ring given (relative) pathnames and keywords in an ordered array. The parameter struc_ptr points to the same structure used in the search_rules_$get primitive.

Entry: search_rules_$set_ptr (struc_ptr, code)

This primitive sets the search rules for the current ring given segment numbers (real and artificial) of the directories to search. The parameter struc_ptr points to the same structure as used in the search_rules_$get_ptr primitive.

Primitives for Interprocess Communication

The following set of primitives will be identical to the current IPC primitives in function. The entries are currently in hcs_ (which will become a user-ring program) and hence the new gate below is provided. The hcs_ entries and the new primitives map as follows:

hcs_$assign_channel	hcipc_$assign_channel
hcs_$block	hcipc_$block
hcs_$delete_channel	hcipc_$delete_channel
hcs_$fblock	hcipc_$fblock
hcs_$ipc_init	hcipc_$ipc_init
hcs_$read_events	hcipc_$read_events
hcs_$sfblock	hcipc_$sfblock
hcs_$wakeup	hcipc_$wakeup

Primitives of General Utility

The following primitives execute in ring 0 and hence a new gate to ring 0 must be provided for them. The following direct mapping (renaming, etc.) will be used:

hcs_$cpu_time_and_paging_	cpu_time_and_paging_
hcs_$get_alarm_timer	alarm_timer_$get_alarm_timer
hcs_$get_page_trace	hcu_$get_page_trace
hcs_$get_process_usage	get_process_usage_
hcs_$get_usage_values	OBSOLETE
hcs_$pre_page_info	OBSOLETE
hcs_$proc_info	hcu_$proc_info
hcs_$reset_working_set	OBSOLETE
hcs_$set_alarm	OBSOLETE
hcs_$set_alarm_timer	alarm_timer_$set_alarm_timer
hcs_$set_cpu_timer	cpu_timer_$set_cpu_timer
hcs_$set_pl1_machine_mode	OBSOLETE
hcs_$set_timer	OBSOLETE
hcs_$stop_process	hcu_$stop_process
hcs_$total_cpu_time_	total_cpu_time_
hcs_$trace_marker	hcu_$trace_marker
hcs_$try_to_unlock_lock	hcu_$try_to_unlock_lock
hcs_$usage_values	OBSOLETE
hcs_$virtual_cpu_time_	virtual_cpu_time_

The program hcu_ (for hardcore utility) will be a hardcore gate for calling primitives in the supervisor which do not easily fall into another category. Only primitives that are intrinsically hardcore in nature should be placed in this gate. It is the replacement for hcs_.

The OBSOLETE interfaces will no longer be supported in ring 0, but rather by user-ring writearounds (in hcs_).

Primitives for Interprocess Signalling

The following primitives replace the hcs_ primitives for IPS management. They are currently identical to the hcs_ entries in function.

hcs_$get_ips_mask	ips_$get_ips_mask
hcs_$mask_ips	ips_$mask_ips
hcs_$reset_ips_mask	ips_$reset_ips_mask
hcs_$set_automatic_ips_mask	ips_$set_automatic_ips_mask
hcs_$set_ips_mask	ips_$set_ips_mask
hcs_$unmask_ips	ips_$unmask_ips

Primitives for Performing I/O

The following primitives will be moved from hcs_ to the indicated gate:

hcs_$ioam_list	ioam_$ioam_list
hcs_$ioam_release	ioam_$ioam_release
hcs_$ioam_status	ioam_$ioam_status

The above three primitives will become obsolete when the full RCP management becomes available.

hcs_$tty_abort	tty_gate_$tty_abort
hcs_$tty_attach	tty_gate_$tty_attach
hcs_$tty_detach	tty_gate_$tty_detach
hcs_$tty_detach_new_proc	tty_gate_$tty_detach_new_proc
hcs_$tty_event	tty_gate_$tty_event
hcs_$tty_index	tty_gate_$tty_index
hcs_$tty_order	tty_gate_$tty_order
hcs_$tty_read	tty_gate_$tty_read
hcs_$tty_state	tty_gate_$tty_state
hcs_$tty_write	tty_gate_$tty_write

Primitives no longer supported by Hardcore

The following primitives are obsolete and will not be replaced when hcs_ is removed from ring 0:

hcs_$del_dir_tree
hcs_$fs_move_file
hcs_$fs_move_seg
hcs_$get_link_target
hcs_$star_
hcs_$star_list_

All of these primitives will be supported to some degree in the user ring.

Summary of New Hardcore Interfaces

Of the more than 150 hcs_ entries currently available, many will be replaced by new hardcore gates while others will be moved to the user ring. Initially, the following names should be added to hcs_ (they will eventually be moved to a new hardcore gate segment):

hcipc_
cpu_time_and_paging_
alarm_timer_
hcu_
get_process_usage_

cpu_timer_
total_cpu_time_
virtual_cpu_time_
ips_
ioam_
tty_gate_

When the full conversion is complete, there will be about 100 hardcore interfaces and another 30 in the user ring that replace the functions of hcs_.

A new hardcore gate should be added soon with the following names on it:

```
*  ref_name_
*  get_pathname_
*  find_dir_segno_
   bind_segno_
   unbind_segno_
   create_
   names_
   del_
   status_
   seg_status_
   set_
   seg_set_
   acl_
   dir_acl_
   inacl_
   dir_inacl_
   quota_
   truncate_
   user_effmode_
   level_
*  assign_storage_
*  unassign_storage_
*  wdir_
   segno_limits_
   get_stack_segno_
*  search_rules_
```

The starred items will eventually be removed from ring 0 (by renaming, etc.).

Section 4. User-level Subroutines

This section proposes a few user-level subroutines to be used by system commands and subroutines as well as by general user-written programs. There have been many interesting interfaces proposed over the years but only a few are mentioned here. One purpose for proposing any new subroutines is to show how the new storage system primitives might be used. Another reason is to try to provide subroutines that might be useful in any reprogramming being done.

Command Utility Subroutines

The following subroutines would be used by many commands. Their input is intentially suited for commands which are passed varying character strings by the command processor.

The following is a list of arguments used by these subroutines:

rel_path	char (*) varying	is a varying character string which is typically a command line argument.
par_segno	fixed bin (15)	is the segment number of a containing directory.
ename	char (32) varying	is an entryname in a directory.
struc_ptr	ptr	points to a structure containing input and output information.
suffix	char (16) varying	is a command name or other identifying name to be used when generating the name to be used for a temporary segment.
seg_ptr	fixed bin (15)	is a pointer to a segment.
sptr2	ptr	points to a structure used by the star_ subroutine (described below).
star_names	(*) char (32) var	is an array of star names to be used by the star_ subroutine.

The subroutines are:

Entry: expand_arg_ (rel_path, par_segno, ename, code)

This subrouutine converts the input relative (absolute) pathname into a directory segment number and entry name. It is analogous to expand_path_ which converts a relative pathname into an absolute pathname.

<u>Entry</u>: command_util_$open (rel_path, struc_ptr, <u>code</u>)

This general purpose subroutine performs many functions. Flags in the input/output structure control the actions taken and the amount of information returned, The parameter struc_ptr points to the following structure:

```
      dcl 1 cui aligned based,
      *     2 version fixed bin,
      *     2 control,
              3 dont_chase bit (1) unal,
              3 dont_get_seg_ptr bit (1) unal,
              3 create bit (1) unal,
              3 new_uid bit (1) unal,
              3 truncate bit (1) unal,
              3 delete bit (1) unal,
              3 set_mode bit (1) unal,
              3 want_suffix bit (1) unal,
              3 set_bc bit (1) unal,
              3 set_cc bit (1) unal,
              3 mbz bit (27) unal,
      *     2 mode like mode,
            2 status,
              3 seg_known bit (1) unal,
              3 link bit (1) unal,
              3 no_read bit (1) unal,
              3 no_execute bit (1) unal,
              3 no_write bit (1) unal,
              3 mbz bit (31) unal,
            2 seg_ptr ptr,
            2 bit_count fixed bin (24),
            2 char_count fixed bin (21),
            2 par_segno fixed bin (15),
            2 ename char (32) var,
            2 suffix char (12) var;
```

where:

1. dont_chase — if ON and rel_path indicates a link, don't chase the link; if OFF, chase the link and return information about the ultimate target.

2. dont_get_seg_ptr — if ON a pointer to the indicated segment is not returned (the segment is not made known). Information about the segment, however, is returned. If OFF, the segment is made known and a pointer to it returned.

3. create — if ON and the segment is not found, it is created; if OFF, the segment is not

created. If the flag new_uid is also ON (along with create) a new unique ID will be assigned to the segment thereby effectively deleting the old and recreating it.

4. truncate — if ON the segment is truncated. This action will be taken only if a pointer to the segment is asked for. This control bit applies for both the open and close entrypoints.

5. delete — is used by the close entrypoint. If ON, the segment is deleted after being made unknown.

6. set_mode — is used by the close entrypoint. It specifies that the value of cul.mode is to be placed in the calling process's ACL entry for the segment.

7. want_suffix — if ON indicates that the caller wants suffix processing to be performed.

8. set_bc, set_cc — are used by the close entrypoint to set the bit count.

9. mode — is the desired mode for the segment. An ACL entry for the calling process with this mode is placed on the ACL for the segment. (The mode can also be set again by the close entrypoint as mentioned above.)

   If the access on the segment is not initially at least the desired access, an attempt is made to change the ACL. If this fails, the mode status bits (no_read, etc.) are set.

10. status — is a structure of returned status information of probable interest to the caller but not deemed fatal enough to warrent a nonzero code return value.

11. seg_known — is set ON if the segment being made known was already known.

12. link — is set ON if a link was chased.

13. no_read, etc. — are set ON if the desired access could not be given to the caller.

14. seg_ptr — is returned by the open entrypoint and is set to point to the specified segment. The close entrypoint uses this variable to know which segment to make unknown.

15. bit_count — is returned by the open entrypoint. If desired, it can be input to the close entrypoint as indicated by the set_bc flag.

16. char_count — is returned by the open entrypoint. If desired, it can be input to the close entrypoint as indicated by the set_cc flag.

Only one of set_bc and set_cc should be ON.

17. par_segno — is returned by the open entrypoint. It represents the segment number of the containing directory.

18. ename — is returned by the open entrypoint. Both par_segno and ename will be set to the target segment if a link is chased. If suffix processing is performed, ename will contain the appropriate suffix.

19. suffix — is the desired suffix to be used when suffix processing is called for.

The structure pointed to by struc_ptr above is typically shared by the open and close entrypoints and serves as a storage buffer for information of interest to both entrypoints. Suffix processing consists of making sure the specified suffix exists on the name of the segment passed to the hardcore interfaces. If adding the specified suffix will make the entryname too long, a status is returned.

Entry: command_util_$close (struc_ptr, code)

This entrypoint is used to "clean up" after use of a segment. The segment can be truncated, deleted, etc. under control of the flags in the cui structure as mentioned above.

Note that the command_util_ entrypoints remove nearly all name management tasks from the user programs. The segment is made known and automatically made unknown (if appropriate) by these calls. No reference name operations are performed at all.

Entry: get_temp_seg_ (suffix, seg_ptr, code)

This subroutine returns a pointer to a zero length buffer (temporary) segment in the process directory. An ACL entry of REW (for the calling process) is placed on the ACL of the segment. The name of the segment is a unique name generated using unique_chars_ and also containing the input suffix. A pointer to the segment is returned in seg_ptr.

Entry: release_temp_seg_ (seg_ptr, code)

This subroutine truncates the specified segment, makes it unknown, but does not delete it from the process directory. The segment is placed in a pool of free buffer segments for later use by callers of get_temp_seg_. (These subsequent calls change the name and make it known again.)

Entry: star_ (struc_ptr, names_ptr, sptr2, star_names, code)

This subroutine scans the names specified by struc_ptr and names_ptr (as returned by status_$all or status_$all_long) and creates an array of indices into the original status structure of matching entries.

```
dcl 1 smi aligned based,
*     2 size_allocated fixed bin,
      2 num_matching_entries fixed bin,
      2 data (1 refer (smi.num_matching_entries)),
        3 strx fixed bin,
        3 star_name_index fixed bin;
```

where:

1. strx — is an index into the status array pointed to by struc_ptr thereby indicating the entry which the star name matched.

2. star_name_index — specifies which star name of the star_names array was matched.