Mica Working Design Document
Image Activation

Revision 1.2

8–February–1988

Issued by:

Lou Perazzoli

digital™

TABLE OF CONTENTS

EXAMPLES

FIGURES

Revision History

Date	Revision Number	Summary of Changes
30-May-1986	0.0	Initial entry. (Lou Perazzoli)
9-Sep-1986	0.1	Changed to reflect new improved map_file directive. (Lou Perazzoli)
25-Sep-1986	0.2	Incorporated review comments and changed name to image mapper. (Lou Perazzoli)
04-Nov-1987	1.0	Eliminated moldy P.TBD concepts such as environments, map_file directive, etc. and changed to reflect current linker and memory management designs. Add sections on autoloader and synchronizations. Added section on shareable image loading and initializations. (Lou Perazzoli)
25-Nov-1987	1.1	Minor edits to conform with object language and linker chapters. (Lou Perazzoli)
01-Feb-1988	1.2	Add section on thread local storage and modify autoloader design. (Lou Perazzoli)

CHAPTER 1

IMAGE ACTIVATION

1.1 Overview

The linker produces an executable image as the end product of program development. During process creation, the thread creating the process specifies the image to be executed by the new process. After the creation of the process and the initial process thread, the image file is mapped into the newly created address space. This mapping occurs in the context of the initial process thread.

Image mapping involves several steps that prepare the image for execution. The image activator opens the image file, thereby establishing a channel to obtain the necessary information to map the file. If the image does not already have an associated segment object, the image activator creates a segment object for the image, building prototype PTEs for the image file. The image activator maps the image into the user's address space, resolves certain address references, and establishes the debugger and traceback handlers.

1.1.1 Goals/Requirements

The Mica image activator has the following goals:

- All images are automatically and transparently shared among all users.
- Optimal performance is achieved by issuing a minimal number of disk reads to initially map the image and delaying most fixups by delaying the loading of shareable images.

1.1.2 Functional Description

1.1.2.1 Image Initialization

No special code exists in Mica to read images into memory for initial execution. Instead, the paging mechanism is used to "page" an image into memory. The image activator configures the process page tables to reflect all pages in the image file.

Mica performs the following steps to support image activation:

1. Opens the image file

   The image activator issues a read-only share open service on the image file. This service returns a channel ID to the file.

2. Creates a section

   The image activator calls the *exec$create_section* system service. The caller specifies the channel ID from the previous system service call, and a mapping type of *e$k_image_map*. This service returns a *section_id*.

3. Maps the section

   The image activator calls the *exec$map_section* system service, specifying the *section_id* returned from the *exec$create_section* system service. This service returns the starting and ending addresses that delimit the mapped image in the virtual address space.

4. Performs fixups

   The starting address identifies where the image's image header begins. The image activator examines the image header, and performs the necessary image fixup operations on the image.

5. Handles message sections

   If any message sections are present in the image, as indicated by the image header, the image activator calls the routine to add these message sections to the process. The nature and functions of this routine are described in Chapter 3, Status Codes and Messages.

6. Maps shareable images marked "activate immediately"

   The image activator examines the image header, and maps any shareable images which are marked "activate immediately". The image activator performs the external fixups for those shareable images once they are mapped. Note that this is a recursive call to the image activator.

7. Calls initialization procedures

   Once the image activator maps and fixes up the "activate immediately" images, it examines the image header, and calls any initialization procedures at their specified entry points. These initialization procedures provide the functionality of the LIB$INITIALIZE routine in VMS. The image activator does not guarantee the order between images of initialization procedure calls, but it does guarantee each procedure is called only once before the user executes any code within that shareable image.

8. Invokes image

   After the image activator has invoked all initialization procedures, it calls the image at its transfer address.

1.1.2.2 Image Exit

When an image returns to the image activator, the *exec$thread_exit* system service is issued, which begins thread termination. The *exec$thread_exit* system service simply calls each exit handler that has been declared by the thread, and then invokes the *exec$delete_thread* system service. If the process has other threads executing, those threads continue to execute normally. Image exit occurs when the last thread in the process exits and the mapping objects and section objects for the image are deleted during object container rundown.

1.1.2.3 Autoload Procedure

The autoload procedure operates similarly to the "activate immediately" method described above. The autoload procedure loads shareable images and resolves the external references when the reference is encountered, rather than at initial image activation time. This reduces the overhead of initial image activation, and maps shareable images only when they are actually required.

1.1.2.4 Installation of Images

The Install Utility serves two purposes. It provides for the installation of a shareable image:

- Within the shareable image space
- With the WRITE attribute

The Install Utility creates a section object for the image, which causes a segment object to be built. The segment object has a "system channel" to the image which implies that the image is effectively installed "opened".

1.1.2.5 Images Within Shareable Image Space

When a shareable image is installed in the shareable image space by use of the /BASE qualifier, the Install Utility opens the image file, and creates a segment causing prototype PTEs to be built. The segment object for the shareable image contains the base address for the image within the shareable address space. When the shareable image is loaded, it is mapped at the specified address. If the image cannot be mapped at the specified address due to addressing conflicts, an error is returned, and the shareable image is not mapped.

When the shareable image is installed no fixups, internal or external, are performed. Note, however, that since no external fixups are performed, any referenced shareable images are treated just like referenced external images. This allows later versions of shareable images to be installed at different base addresses while the system is running, and the latest image is properly loaded.

Images installed in shareable image space may reference other images though use of the autoload capability or the activate immediately capability. These referenced images do not need to reside in shareable image space.

1.1.3 Issues to be Resolved

- Exact detail of message section addition. This is dependent on the design of message sections and the definition of the routines.

1.2 Image Activation

1.2.1 Thread Startup

Once the process and the initial thread have been created with a minimal address context (user stack, process control region, thread control region, last chance condition handler), the initial thread startup routine is invoked in user mode. The initial thread startup routine exists in the system portion of the address space. Its sole purpose is to map the shareable image, *mica$fm_share*, that performs the actual user-mode thread startup.

A section exists for the *mica$fm_share* shareable image in a known system container. To map the image an object name translation is performed on the image name in the system container. The resulting section ID is mapped and its base address in shareable image space.

The *mica$fm_share* shareable image contains the image activator, the image fixup, the global autoloader, and the support routines to start the initial image.

After mapping the *mica$fm_share* shareable image, the initial thread mapping procedure examines the image header to locate the transfer vector offset for the *imact$initialize* procedure. The image activator then calls the *imact$initialize* procedure, passing the base of the system service vector page as an argument. The base of the system service vector page is required to allow various system services to be called before the system service shareable image has been loaded.

The *imact$initialize* procedure maps the section for the system services, and opens the specified image file as execute only.

If the open service on the image file succeeds, the *exec$create_section* service is called to create the structures necessary to share the file among multiple address spaces.

The *exec$create_section* service performs an object name lookup to determine if the specified image file currently has an associated segment object. Note that all segment objects exist in the same system container. If the segment indicates that the file is mapped as a data file rather than an image file, the *exec$create_section* service returns an error indicating that the file is mapped in an incompatible state. If the segment object specifying image mapping currently exists for the image file, a section object is created which refers to the segment object. If the segment object exists, the image activator continues the process of image activation by mapping the section (See Section 1.2.3). Otherwise, memory management software must create the segment (See Section 1.2.2).

1.2.2 Segment Does Not Exist for Image File

If a segment object does not currently exist for the image, the *exec$create_section* system service creates and initializes a segment object for the image. A system channel is created to the specified file, and the pointer to the channel object is stored in the segment object.

The segment object header contains, among other items, the system channel pointer of the file, the number of pages in the segment, the required size of the user stack, and the mapping type (in this case, *mm$c_image_map*).

1.2.2.1 Image Header

The first sixteen virtual blocks of the image are read into memory and examined. If the image is less than 16 blocks, an error status is returned in the IOSB, but the read still completes, delivering all the blocks in the image. The image header (which is at least one block) contains an item describing the number of 64-Kbyte pages required to map this image. The number of 64-Kbyte pages determines the allocation required for the segment object.

1.2.2.1.1 Image Section Descriptors

The analysis of the image header continues by examining the image section descriptors (ISD). The image section descriptors describe the layout for creating the prototype PTEs for the image. Each image section is aligned by the linker on a virtual disk block number (VBN). An image section descriptor starts on a 64-Kbyte virtual boundary. The image section is either demand zero, or it contains the number of VBNs in the section, and the starting VBN number of the section. The ISD also contains the page protection for the section. See Chapter 29, Linker for more details.

The image activator determines the format and the protection attributes of the prototype PTE from the information in the ISD. Example 1–1 describes how the protection is set.

Example 1–1: Prototype PTE Protection Attributes

Flags in ISD			Settings in Prototype PTE					
READ	WRITE	EXECUTE	READ	WRITE	FOR	FOW	FOE	COM
0	0	0	0	0	0	1	0	1
0	0	1	1	0	1	1	0	1
0	1	0	Invalid in PRISM					
0	1	1	Invalid in PRISM					
1	0	0	1	0	0	1	1	1
1	0	1	1	0	0	1	0	1
1	1	0	1	1	0	1	1	*
1	1	1	Invalid in an image file					

* Set if MODIMG$IMAGE_SECTION_DESCRIPTOR.COPY_ON_MODIFY is true.

The *copy_on_modify* flag is set on all nonwritable pages. This allows the protection on a nonwritable page to be changed to writable, causing the materialization of a private page before the actual write. Also, fault on write is enabled on all pages even if those pages are not writable. This allows the protection to be changed to writable without having to also enable fault on write. For ISDs which are read/write and not copy on modify, fault on write is enabled to maintain the modified state of the page.

When a writable image section descriptor is encountered without *copy_on_modify* set, the file system is queried to see if the user has opened the file for write access. If the file has not been opened for write access, the *exec$create_section* service fails indicating the file was not opened for write access in the status value. If the file was opened for write access, the *exec$create_section* proceeds. Only a shareable image may have image sections without *copy_on_modify* set.

Once the stack descriptor has been found, the size of the stack is recorded in the segment object.

Once the segment object has been initialized, the section object is created in the process-private container that refers to the segment object.

At this time, the section object and segment object for the file have been created. All local image sections have been processed and the prototype PTEs have been created for the segment. It is interesting to notice that at this time all prototype PTEs either point to a subsection, are no access, or are demand zero.

1.2.3 Mapping the Image into Virtual Address Space

The *exec$map_section* service is issued to map the section into the user's address space. This causes the creation of a mapping object in the process-private object container. The mapping object contains a reference pointer to the section object and the virtual address limits where the section was mapped.

The image activator attempts to map images at their based address to eliminate internal fixups.

If the initial user stack is less than the stack size specified in the segment object, the *exec$expand_stack* service is called to produce a stack of the proper size.

1.2.3.2 Image Fixup

An image section contains the fixup information necessary to resolve any internal image addresses. Once the *exec$map_section* code completes, the image fixup is done in user mode by examining the fixup information, and adding the difference between the real and the base address of the image to each fixup location.

If necessary, the protection of the page is changed to writable in order to allow fixup information to be written to the page. By examining the fixup information, an argument list for the Set Protection service is created which sets the protection of all necessary pages, that is, pages which reside in nonwritable program sections, to write enable. The fixups are then calculated and written to the appropriate locations. The protection on the pages is changed back to write disable.

1.2.3.3 Thread Local Storage Fixups

The shareable image *mica$fm_share* contains the current sum of all thread local storage regions in each loaded image. When an image is loaded via the activate immediate mechanism or the autoloader, the image being loaded is given the current sum, and the sum is updated by adding the number of thread local storage regions found in the image being loaded to the current sum.

The accessing and modification of the sum is performed under synchronization to prevent multiple threads from storing or updating the sum simultaneously.

1.2.3.3 Activate Immediately Shareable Images

After completing internal fixups, the image activator examines the image header to locate any "activate immediately" shareable images. If found, the image activator issues a call to an object service to determine if a mapping object exists with the name of the image desired. If the image has been mapped previously, the virtual addresses of where it is mapped may be obtained from the mapping object. The virtual address of the shareable image is used to perform the external fixups. Note that the linker determines whether a shareable image is autoloaded or "activate immediately".

If an "activate immediately" image is not currently mapped, the shareable image synchronization primitives described in a later section are used to ensure that only one copy of the shareable image is mapped, fixed up, and initialized.

As each shareable image is mapped and fixups performed as necessary, any initialization procedures for that shareable image are invoked. As "activate immediately" image operations are performed, external fixups are performed in the invoking image.

1.2.3.4 Debugger

The debugger is invoked as an activate immediately shareable image which has an initialization procedure. The debugger's initialization procedure checks to see if the debugger should be invoked and if so does whatever operations are necessary to create a debugging environment.

1.2.4 Loading of Shareable Images

The following pseudo-code describes the steps to ensure that two threads do not attempt to load the same shareable image at the same time.

```
! General shareable image synchronization.
disable ASTs
status = exec$create_event (name = shareable_image_name)
IF status == collision THEN
    status = exec$wait_any (collided event_id)

    ! Shareable image is loaded and initialized when wait
    ! completes. Note wait could return an invalid ID error.

END IF

status = exec$translate_object_name (
        object_id = returned_mapping_id,
        object_type = mapping_object,
        object_name = shareable_image_name
        )

IF status == name_does_not_exist THEN

    ! Image has not been loaded. Load it.

    open_file (shareable_image)
    create section (shareable_image_channel)
    map section (shareable_image_section_id)
    fixup shareable image
    activate immediates
    call initialization procedures

ELSEIF status == success THEN

    ! Shareable image is loaded and initialized.

    exec$get_mapping_info (
        object_id = returned_mapping_id,
        item = base_address
        )
ELSE
    restore AST state
    ! Unexpected error - raise condition.
END IF

store the base_address
status = exec$set_event (event_id)          !ignore any errors
status = exec$delete_object_id (event_id)   !ignore any errors
restore AST state
```

1.2.5 Autoloading of Shareable Images

Automatic loading of shareable images is the act of loading a shareable image only when a procedure within that shareable image is invoked. This allows the overhead of loading the shareable image and performing fixups and initialization routines to be deferred until the shareable image is actually required.

When the first call to a procedure within a shareable image is made, that image is automatically loaded. All other calls to procedures within that shareable image from the image making the call are fixed to directly call the loaded shareable image. Note that this involves changing the protection of read-only memory to read-write, and then changing it back again.

1.2.5.1 Linkage Pair

A *linkage pair* consists of two longwords. The first contains a pointer to the invocation descriptor for the procedure and the second contains the address of the code for the procedure. The following instructions are generated to call the procedure:

```
LDQ     routine(Rx),R10  ;load R10 with invoc. desc and r11 with code address
JSR     R11,(R11)        ;call the procedure
```

Note that routines which access the linkage pair must always be accessed with a LDQ instruction. This prevent synchronization problems when the linkage pair is being modified.

For procedures which reside in automatically loaded shareable images the first longword of the linkage pair contains the address of the autoload vector and the second longword contains the address of the transfer code. The linkage pair and autoload vector reside in the linkage section, which is read-only.

1.2.5.2 Autoload Vector

The autoload vector is the data structure that maintains the information about the automatic linkage to a routine in another shareable image. The autoload vector consists of 5 longwords, and is quadword aligned. The first longword is the address of the transfer code, the second longword is the address of the entry descriptor for the transfer code, the third longword contains the address of the autoload vector, the fourth longword contains the address of the autoloader code, and the fifth longword contains the address of the image descriptor for this shareable image. The third and fourth longwords are changed after the image is loaded to contain the address of the routine's real invocation descriptor and entry point. This allows subsequent calls to the routine using the autoload vector to work (which might happen in an optimizing compiler).

For each procedure called within a autoloaded shareable image a autoload vector exists. The autoload vector is created by the linker in the image's header, which is read-only memory. When the desired image has been loaded and all fixups and initializations performed, the autoload vector is modified. The address of the autoload vector field and the pointer to the local autoload code are changed to be the real routines linkage pair using a STQ instruction.

1.2.5.3 Transfer code

The transfer code consists of the following 3 instructions:

```
LDQ     8(R10),R4       ;load address of autoloader code
OR      R4,R0,R10       ;load R10 with address of autoload vector
JSR     R0,(R5)         ;jump to the local autoloader
```

Since the autoload vector itself is fixed up during automatic image loading, the transfer code on subsequent calls transfers the caller directly to the called routine. The transfer code is generated by the linker in a special psect that allows execution.

1.2.6 Image Descriptor

The image descriptor consists of the image name, the self-relative base address for the current image, and an array of longwords consisting relative pointers to each linkage pair which references procedure in the shareable image described by this image descriptor. Note that the image descriptor is created by the linker in the image header in read-only memory.

Figure 1–1: Structures Before Autoload Has Occurred

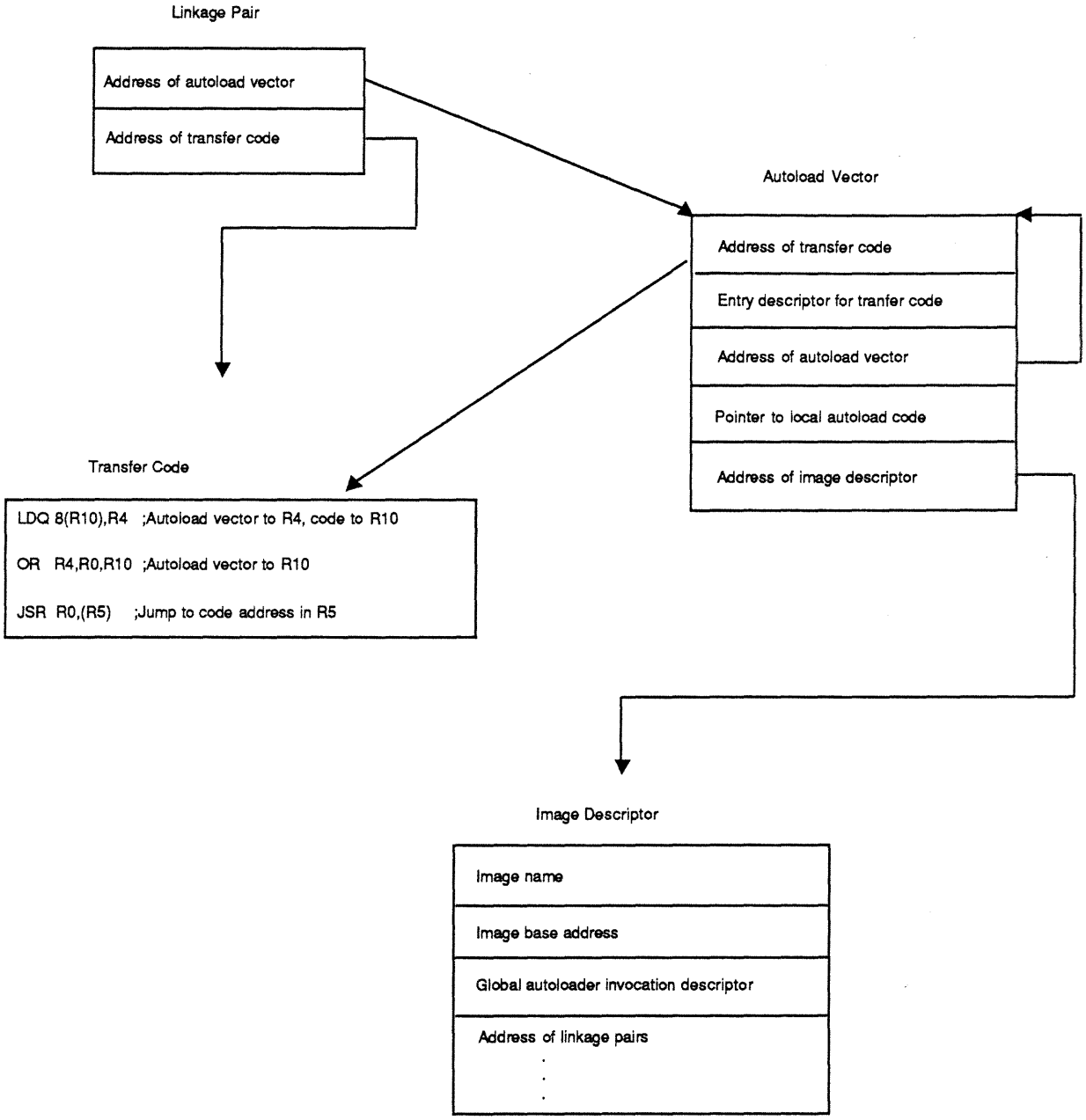
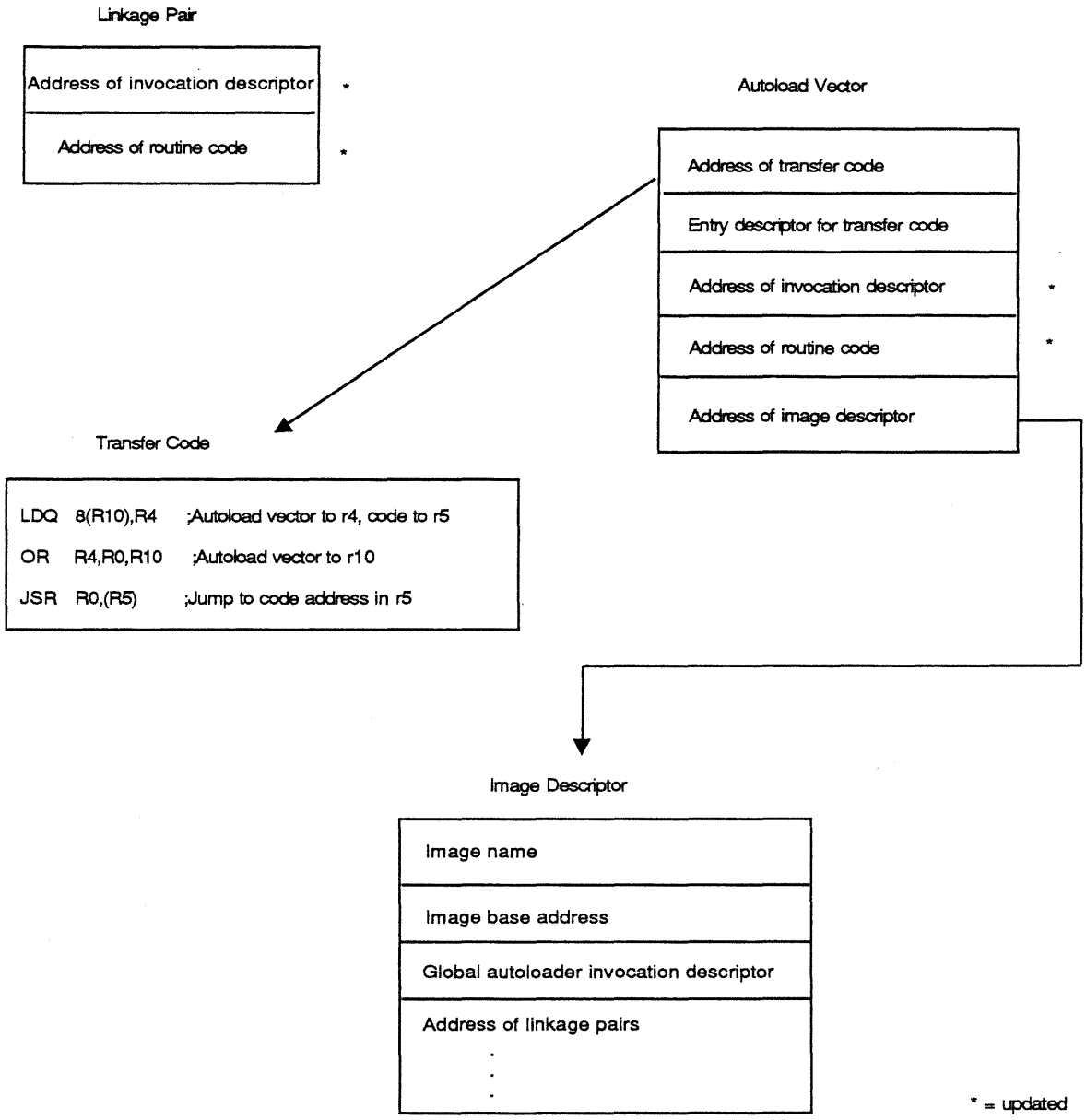


Figure 1–2: Structures After Autoload Has Occurred



1.2.6.1 Autoloader

The autoloader consists of two parts, a local portion generated by the linker and a global portion which resides in the shareable image *mica$fm_share*.

The local portion does the following:

- Checks to ensure that R10 contains the address of the autoload vector. This is accomplished by comparing R10 to 8(R10). If they are unequal, then the image is already loaded and may be called by executing the transfer code instructions.
- Saves all scratch registers except R4 and R5.

- Loads R10 with invocation descriptor, and loads R11 with the code address of the global autoload routine.
- Calls the global autoload routine.
- Upon return, restores the saved registers and executes the transfer code sequence to call the newly loaded routine.

The global autoloader does the following:

- Disables ASTs.
- Performs synchronization to ensure that multiple threads are not attempting to load the same shareable image simultaneously. See Section 1.2.4 for details.
- Checks to ensure that the autoload has not already occurred. This is done by comparing R10 with 8(R10). If they are not equal, 8(R10) contains the invocation descriptor address and 12(R10) contains the code address.
- Maps in the shareable image if it is not already mapped.
- Performs fixups on the shareable image.
- Invokes the shareable image's initialization procedures.
- Performs synchronization to ensure that multiple threads are not attempting to update the shareable image's linkage pair area simultaneously.
- Sets the protection of the linkage pair areas to read/write and updating all elements represented in the image descriptor's array of longword pairs. As linkage pairs are updated, the autoload vector which the invocation descriptor refers to is also updated.
- Restores AST state.
- Returns to the local autoload procedure.

After the autoloader has executed, the linkage pair contains the actual address of the invocation descriptor and the code, and the autoload vector contains the actual address of the invocation descriptor in the third longword and the actual address of the code in the fourth longword.

\This section will track any changes in the Prism calling standard.\

1.2.7 Autoloading System Services

At system initialization some pages of system space are allocated as kernel entry pages (user read, fault on execute, kernel entry point fields are set). The starting address of these pages is stored in the global variable *e$system_services_base*.

Also, at system initialization, the *exec$create_section* service is called to create a section for the system services shareable image. This image is mapped into shareable image space, thus creating a segment for the subsequent references. In order to fixup the vectors, the address found in *e$system_services_base* is added to each offset. Thus, when a JSR is issued to a system service, the destination address of the JSR is within the system service vector page.

1.2.8 Image Startup

Once the fixup operation has completed and all "activate immediately" shareable images have been loaded and initialized, the thread startup procedure, still running in user mode, locates the transfer address to be called.

The transfer address is called with the standard argument list. This transfer address is the entry point of the user image.

\ Exact details of the argument list are TBD. \

1.2.9 Merged Image Activation

The image activator supports merged image activation to allow the emulation of the *lib$find_image_symbol* routine. Note that VMS compatible routines for *$imgact* or *imgfix* is not provided since they are not documented in the VMS system services manual.

1.2.10 Installation of Images

The Install Utility serves two purposes. It allows:

- Installation of a shareable image within the shareable image space
- INSTALL /WRITE functionality

All images which are installed have a segment object. Since the segment object contains a channel to the specified image file, the image is effectively installed "opened".

1.2.10.1 Images Within Shareable Image Space

When a shareable image is installed in the shareable image space by use of the /BASE qualifier, the image file is opened and prototype PTEs are built. The segment object for the shareable image contains the base address for the image within the shareable address space. When the shareable image is mapped, it is mapped at the BASE address specified. If the image cannot be mapped at the specified address space, due to addressing conflicts, an error is returned and the shareable image is not mapped.

When the shareable image is installed no fixups are performed. Internal fixups are not performed because of the complex nature in forcing the fixups back to the prototype PTEs. However, by linking the shareable image as based and installing the shareable image at its linked based address, no internal fixups are required.

External fixups are not performed to allow later versions of referenced shareable images to be installed (at different base addresses) while the system is running, and the latest image is autoloaded.

It is not possible for the same address space to have two different versions of the same shareable image loaded. This problem is avoided because the synchronization rules followed when shareable images are loaded.

1.2.11 Image Mapping into System Space

During system initialization and normal system operations shareable images need to be loaded into system space. Such shareable images could be function processors, object service routines or user loadable system services, or in the case of system initialization, components of the executive.

The image activator is subsetted to create a system image loader which provides this functionality. The system image loader loads and binds shareable images with the executive.

\There also needs to be a executive service to allow system management to invoke the system loader.\

Mica Working Design Document Naming Standards and Pillar Coding Conventions

Revision 0.6

13–January–1988

Issued by:
Kris K. Barker

digital™

TABLE OF CONTENTS

Revision History

Date	Revision Number	Author	Summary of Changes
4-DEC-1986	0.1	Benn Schreiber	Original.
16-JAN-1987	0.2	Benn Schreiber	Incorporate review comments.
1-MAR-1987	0.3	Benn Schreiber	Comments from public review.
18-SEP-1987	0.4	Kris Barker	Reorganize chapter and add coding conventions.
21-OCT-1987	0.5	Kris Barker	Incorporate changes from architect review.
12-NOV-1987	0.6	Kris Barker	Incorporate changes following further review and notes file comments.

CHAPTER 1

NAMING STANDARDS AND PILLAR CODING CONVENTIONS

1.1 Introduction

This chapter defines standards for the naming of data types, logical names, module names, and so on throughout the Mica operating system. It also provides preferred conventions for all Mica programs written in Pillar.

1.2 Naming Standards

Naming standards are used for all names accessible from user-mode programs throughout Mica. Such names are commonly referred to as public names.

1.2.1 Goals

Naming standards are important for several reasons:

- To present a consistent, easy-to-remember name space to users and developers
- To ensure that system software uses consistent naming to aid future developers in maintaining and extending the software
- To ensure that customer-written software is not invalidated by future releases of DIGITAL products that add new symbols
- To facilitate straightforward usage within Pillar; the names are similarly usable in all other DIGITAL-supported languages

1.2.2 Scope

This section covers the public naming standards for:

Facility names
The software facility name based on the product or component name.

Module names
The names assigned to program source modules.

Procedure names
The names of system services, system routines, kernel routines, and run-time library routines, and the names of the arguments to those procedures.

Files and directories
The format for naming files that constitute the system software.

Program data and type names

Including:

- Types—Pillar named types including records and record fields
- Global variables—global symbols known to the linker
- Constants—compile-time named constants including:
  - Item codes
  - Function codes
  - I/O parameter record codes
  - Other named constants
- Message names—symbols that define unique message values

Logical names

System or group logical names used to alter, define, or control a facility.

Compile-time facility macros and procedures

Macros and command procedures used during the compilation process.

These are discussed in the following sections. The standards in this section cover all public software interfaces for layered products, as well as bundled Mica software.

1.2.3 General Naming Standards

Names should not be short acronyms. Use full English word(s) whenever possible. For instance, a parameter representing the desired access mode should be named *access_mode* rather than *acmode*.

If the name consists of more than one word, the words must be separated with the underscore ("_") character.

\Throughout this document multiword names in naming examples are hyphenated. This is done to improve readability and to point out exactly where underscores are required. For example,

```
facility$C_name-of-constant
```

is an example of a constant name; "name-of-constant" might be something such as *user_buffer*.\

The exception to this standard occurs when a name is too long, that is, longer than the maximum allowed symbol length. In this case, the engineer must use good judgment and derive an acceptable name that is easily remembered. While the maximum symbol length on Mica is TBD, this standard recommends limiting symbols to 31 characters, especially for code that may be ported to VAX/VMS.

All DIGITAL-supplied public symbols that can be referenced by users and where the scope of the symbols overlaps with the user name space, are prefixed with "facility$" where:

- "facility"—the facility to which this symbol belongs
- "$"—indicates a DIGITAL reserved name

Users must not use the currency sign in their definitions. This ensures separation of name spaces and prevents naming collisions in future releases. See Section 1.2.5.1 for more information.

When something is named in several different places throughout the system, it must have the same name. For instance, all services that accept an event object ID as an argument should name the argument *event_id*.

1.2.3.1 Case Sensitivity

Unlike the VAX/VMS object language, the Mica object language is case sensitive. To accommodate common coding practices in case-sensitive languages such as C, case-insensitive compilers output symbols completely in lowercase. This eliminates the need for generating both lower and uppercase versions of global symbols. The names given to system services, RTL routines, item codes, objects and object containers, message names, and so forth should, therefore, be completely in lowercase.

\The sample names presented in this chapter do not always follow this lowercase guideline exactly. This is for readability only. The rules for presenting sample names in this chapter are:

- Generic portions (that is, portions of the name that are determined by the engineer based on where and how the name is used) are in lowercase.
- Specific portions (that is, portions of the name that must be exactly as specified in the example) are in uppercase. In actual code, these portions would be written in lowercase.

For example, when an engineer creates a constant name that has been presented in this chapter as:

```
facility$C_name-of-constant
```

- The generic "facility" is replaced by the facility name (in lowercase).
- The specific "$C_" is written as "$c_".
- The generic "name-of-constant" is replaced by a descriptive name for the constant (in lowercase).

The actual name would be something such as *linker$c_maximum_symbols*.\

1.2.3.2 What is a Facility?

A facility is a collection of code and data which operate together to perform a function or set of functions. For the purposes of the Mica naming scheme, each utility or layered product is typically considered to be a facility.

For Mica, most of the executive is considered a single facility. However, separate facilities are defined for code that appears to provide executive functionality, but in reality resides elsewhere (remote procedure call support, for example). Exceptions to this include support that is viewed as part of the executive on VAX/VMS.

1.2.4 Facility Names

Good judgment must be used when defining facility names. In general, facility names should be the full name of the facility. For instance, the Pillar compiler should use the facility name *pillar*, the linker should use *linker*. Use of the facility name itself is preferred over use of the verb describing the function performed by the facility (for example, *linker* rather than *link*).

Facility names must be carefully chosen so that messages issued from the various facilities can be easily identified without requiring extensive prior knowledge of the software or a need to feed facility names through an alphabet-soup-to-English translation program.

There are facilities to be ported from VAX/VMS that have three-letter acronyms, such as the various components of the Run-Time Library. It is preferred that these facilities maintain their acronyms to maximize compatibility and to minimize confusion for both developers and users who migrate from VAX/VMS to Mica.

If the facility name is eight characters or less, the facility name must be used as is. If the facility name exceeds eight characters, an acronym must be chosen that is sensible and easy to remember. (For example, *perform* might be used as the facility name for the PERFORMANCE facility.)

The facility name has no direct relation to the method of software distribution used (bundled versus layered). Facilities that are bundled with the Mica operating system have their own facility names. For instance, *debugger* is the facility name for the debugger.

Facility names and facility codes must be registered. A list of registered facility names and facility codes is presented in the Type, Record, and Name Appendix of this document.

\For the remainder of this chapter, the term "facility prefix" is used to indicate the facility name followed by a currency ("$") sign.\

1.2.5 Module Names

The name given to a particular source module depends on its type. There are three types of source modules in Pillar: definition modules, implementation modules, and combination modules. Rules for naming these modules are presented in the following sections.

1.2.5.1 Definition Modules

Definition modules contain only value, type, variable, and procedure definitions. Two types of definition modules are:

- "Internal" or "Private" definition modules

  These are modules containing definitions used internally within Mica. Since these are not seen by customers, granularity of declarations within a facility or the executive is that deemed most appropriate to Mica development.

  Private definition module names are of the form:

  ```
  facility$module-name_DEF
  ```

- "External" or "Public" definition modules

  These are modules containing definitions visible to customers. Typically, the public definition module for a particular facility is a collection of selected portions of private definitions modules used by that facility. Only one such public definition module is allowed per facility. Most facilities will not even have a public definition module.

  Public definition module names are of the form:

  ```
  facility$DEFINITION
  ```

All exported procedures, types, variables, and constants (defined in both private and public definition modules) must have names beginning with "facility$". Furthermore, all non-exported procedures, types, variables, and constants (defined in implementation modules) may not have names beginning with "facility$".

1.2.5.2 Implementation Modules

Implementation modules contain only code. Implementation module names are of the form:

```
facility$module-name
```

1.2.5.3 Combination Modules

Combination modules are modules that contain both data definition and implementation components. They are named as specified in Section 1.2.5.2. Note, however, that use of combination modules is discouraged. Developers should use separate definition and implementation modules instead.

1.2.6 File Names and File Types

All file names are prefixed with "facility$" to identify the facility to which a file belongs. Typically, source file names are taken directly from the name of the module which is implemented within the file.

The names of all files supplied with the Mica operating system that are facility-independent are prefixed with *mica$*. This includes the operating system images, system support images, and utilities typically identified with the operating system rather than as their own facility. Facilities such as TPU, the debugger, and the Run-Time Library, although supplied with the operating system, are typically identified as their own facility, and therefore use *tpu$*, *debugger$*, and so on, as the facility prefix.

\The *mica$* prefix is what was specified in the previous version of this chapter. If we believe that the name Mica will disappear in the actual product, this prefix should probably be changed.\

Long file names use underscores ("_") to separate words within the file name. \The hyphen as a separator was rejected because it would cause an inconsistency between file names and procedure names.\

File types must be registered. The method for registering file types is TBD (see Section 1.4). A list of registered file types is included in an appendix of this document.

The three-character file type limit that was once imposed on VAX/VMS file type naming does not exist on Mica or VAX/VMS V4.0 and following. When defining new file types, there is no reason to be limited to three characters.

1.2.7 Procedures

Public procedures provided by DIGITAL for Mica are of the form:

```
facility$entry-name
```

In general, non-public procedures are not visible. This is because the bulk of the system is implemented in Pillar which allows the use of module-qualified symbols for intermodule communication. However, there are some facilities coded in BLISS or other languages that do not support the concept of module-qualified symbols. In such languages, non-public procedures that must be declared as global for intermodule communication have names of the form:

```
facility$$entry-name
```

\SIL does not support module-qualified symbols. However, a mechanism has been added to SIL to permit prefixing all exported names with a specified string. This is accomplished with the LINKAGE OPTIONS LOCAL PREFIX statement. When SIL programs are converted to Pillar, this statement will be removed, and the symbols and references to these symbols will be through module-qualified symbols.\

1.2.7.4 System Routines

System routines are those routines that are:

- Provided by DIGITAL
- Run in user mode
- Required to have a documented, supported public interface
- Not officially part of the Mica RTL provided by SDT
- Viewed by users as having "system" functionality

Examples of system routines are Get Active Thread Count, Formatted ASCII Output, and Get Cycle Count.

The facility prefix for system routine names is *exec$*.

1.2.7.5 System Services and Executive Routines

System services run in kernel mode in the Mica executive. The facility prefix for user-visible Mica system service names is *exec$*.

Executive routines also run in kernel mode but do not have user-visible interfaces. General purpose executive routines have the facility prefix *e$*. Other executive routines which provide non-general-purpose functionality have facility names which reflect that particular area of the executive. Such routines are generally callable only if certain conditions have been met, such as acquisition of one or more mutexes, or executing in a particular module such as a device driver. The actual facility names for these executive routines are presented elsewhere in this document.

1.2.7.6 Kernel Routines

Kernel routines may only be called by the Mica operating system. Kernel routines are not visible to user programs. The facility prefix for kernel routine names is *k$*.

1.2.7.7 Procedure Arguments

Procedure arguments must have names that describe the argument's purpose. Do not indicate anything about data type or passing mechanism in an argument name.

1.2.8 Types

The basic format for type names is:

```
facility$name-of-type
```

- "facility"—the facility to which this type belongs
- "$"—indicates a DIGITAL reserved name
- "name-of-type"—descriptive name of type

1.2.8.8 Enumerated Type Element Names

Enumerated type names are as described in Section 1.2.8. The names of the elements of an enumerated type are as described in Section 1.2.10 for naming constants.

In cases where naming conflicts require further qualification of enumerated type element names, the "name-of-element" portion may include a portion of the enumerated type name itself.

1.2.8.9 Data Structure Types

Data structure names consist of two parts: the name of the structure and the name of the field within the structure. Data structure names follow the standard described in Section 1.2.8. Data structure field names are not required to follow any specific naming standards. They should be as descriptive as is reasonable. Field names should not include any indication of size or alignment within the record; size and alignment information is specified in the structure definition itself.

\Previous versions of this chapter called for field names which included the facility name. It was felt that this was necessary for software written in C. This understanding has since changed; current C language products require fully qualified structure references requiring field names to be unique only within a given structure.\

1.2.9 Global Variables

Global variables are those data locations known to the linker as global symbols. In general, global variables are typically not provided in public program interfaces. However, in those cases where public global variables must be defined, the names are specified as:

```
facility$name-of-variable
```

- "facility"—the facility to which this type belongs
- "$"—indicates a DIGITAL reserved name
- "name-of-variable"—descriptive name of global variable

1.2.10 Constants

Named constants have names of the following form:

```
facility$C_name
```

- "facility"—the facility to which this type belongs
- "$"—indicates a DIGITAL reserved name
- "C_"—Mica-specific portion indicating the use of the constant. All constants including item code names, function codes, I/O parameter record codes, and so on have the "C_" to indicate that they are constants.

  Note that this creates a problem for system services that accept an item list as input. Normally, such services would use *exec* as the facility portion of the name. A collision will occur if two different parts of the executive choose the same name for different valued item codes. For these system routines and services, the facility name may be specified as the service name or an acronym of the service name.
- "name"—descriptive name of constant

Due to internationalization requirements, string constants used for display purposes (either on a terminal or in a listing) must not exist within programs. String constants must be implemented via the message facility.

\The previous paragraph deals with the content of string constants rather than their names. It is felt, however, that this rule is important and should be stated here.\

1.2.11 Messages

Message names are of the form:

```
facility$_status-name
```

The "status-name" string is derived by using the first two or three words of the English message text.

Engineers must use good judgment when selecting message names, as these names are used constantly by application programmers. Choose names that are reasonable and easily remembered.

Status codes returned by the Mica executive are of the form:

```
EXEC$_status-name
```

1.2.12 Logical Names

Logical names are of the form:

```
facility$name
```

The "name" string should consist of one or more English (this does not present an internationalization problem) underscore-separated words describing the purpose of the logical name.

Although the Mica executive facility prefix is *exec$*, logical names defined by the operating system use the facility prefix *sys$*. This is done for compatibility and familiarity with VAX/VMS.

1.2.13 Objects and Object Containers

There is no standard for naming objects (the optional ASCII string associated with an object). In most cases, objects will not be named.

System object container names are of the form:

```
facility$name_OBJECT_CONTAINER
```

- "facility"—the facility which creates and uses this container
- "$"—indicates a DIGITAL reserved name
- "name_"—describes the use of the container (for example, *process_* could be used to indicate that the object container contains process IDs)
- "OBJECT_CONTAINER"—indicates that this is an object container

The facility prefix for object containers created by the operating system is *exec$*.

1.2.14 Compile-time Facility Macros and Procedures

TBD.

1.3 Pillar Coding Conventions

All Mica programs written in Pillar follow certain coding conventions. In the following sections, all guidelines apply to both the Pillar and SIL languages.

1.3.1 Goals

Writing code which follows these conventions has the following benefits:

- More readable code—Standardized coding makes code much easier to read and understand.
- More easily maintained code—Standardized coding makes code easier to modify and maintain.
- More consistent code to writers for inclusion in documentation—It is highly desirable to eliminate the need to alter code for inclusion in documentation.

Making decisions about "religious" issues such as coding style is never easy. The following conventions were developed based on the response to a questionnaire and discussions with people in both the Pillar compiler and Mica OS groups.

1.3.2 Indentation

Each level of indentation is 4 spaces. For multiple levels, spaces are preferable to tabs as spaces make level adjustments easier. Statement format (that is, what the actual indentation is for each type of statement) is described in Section 1.3.13.

1.3.3 Capitalization

All Pillar language keywords are in uppercase. Built-in types and procedures should not be in uppercase. A list of keywords may be found in Chapter 2 of the current "Obsolete SIL Reference Manual".

All identifiers must be lowercase. Uppercasing any identifiers defeats the purpose of uppercasing keywords.

\Pillar is an example of a case-insensitive language. Therefore, as described in Section 1.2.3, Pillar exports symbols in lowercase as required by the naming standard.\

1.3.4 Line Length

The maximum source line length is 112 characters.

\112 characters was chosen as the maximum source line length for the following reasons:

- The naming standards described in Section 1.2.3 require the use of descriptive names for data types, global variable names, procedures, and so on. Traditional 80 column source forces many statements to be broken up over several lines. A line length of 112 columns allows long names to be used in a single line.
- A source line length of 112 columns allows listing files to fit within 132 columns.
- 112-character source lines allow source displays using a default full-sized font on workstations. A 132-column font, which is less readable, is not required).\

1.3.5 Multistatement Lines and Multiline Statements

Each source line should contain one statement or part of a statement. No lines should contain multiple statements. There are no exceptions to this rule.

1.3.6 Comments

These conventions describe formats for four different uses of comments.

1.3.6.1 Module Level Comments

Module level comments document the purpose of the module, contain the DIGITAL copyright notice, document the module's author, revision history, and so on. Module comments are in the following form:

```
MODULE module_name;

!*********************************************************
!*                                                       *
!*           DIGITAL Copyright                           *
!*                                                       *
!*********************************************************

!++
!
! Facility:
!
!   Name of facility
!
! Abstract:
!
!   A paragraph that describes the basic functionality provided by
!   the module.
!
! Author:
!
!   Author's name
!
! Date:
!
!   Original date
!
! Revision History:
!
!   Vx.x-yy    Date    EDIT#   Modifier's Name
!       Description of modification
!
!--
```

- Module copyright format is describe in Section 1.3.9.
- Vx.x-yy is the software revision level
- EDIT# is the modifier's edit number (for example KKB047)
- Dates are expressed as DD-MMM-YYYY

To avoid excessively long revision histories in the module level comment block format shown above, revisions are removed on each major software release. For example, when version 2.0 is released, all revision comments pertaining to all 1.n versions will be removed. This process is especially important at release 1.0. At that time, the entire pre-release revision history will be removed.

\The numeric portion of the edit number is a running count of edits made by the engineer over the life of the project or work at DIGITAL.\

\The file COMPILER$:[WORK.COPYRIGHT]MODULE.HEADER contains the module level comment format described above.\

1.3.6.2 Procedure Level Comments

Procedure level comments describe the function performed by the procedure, and list and describe procedure inputs and outputs. Procedure comments are formatted as follows:

```
PROCEDURE procedure_name (
            .
            .
            .
            ) RETURNS return_type;

!++
!
! Routine description:
!
!   Description of function of procedure.
!                   .
!                   .
!                   .
!
! Arguments:
!
!   arg1 - This argument supplies some value.
!   argument2 - This argument supplies another value.
!   arg3 - This argument returns some value.
!   argument4 - This argument supplies some value and returns
!               another value.
!
! Return value:
!
!   The procedure returns some value.
!
!--
```

Notice that the argument descriptions are listed in the order of the procedure declaration and that the words "supplies" and "returns" are used to indicate which are inputs, outputs, or both. This alternative was chosen over the previous "Inputs" and "Outputs" grouping because:

- It is easier to read and maintain since grouping by inputs and outputs frequently is in a different order than the parameters are ordered in the declaration.
- There is no problem with determining where to describe an argument that is both an input and an output.

Also, the hyphens ("-") separating the argument names and their descriptions are not aligned (see Section 1.3.7).

1.3.6.3 Block Comments

Block comments are used to describe the function performed by a section of code. They appear prior to the code section and are indented to the same level as the code which is being documented. Block comments are in the following form:

```
pillar statement;
      .
      .
      .
```

```
!
! This is a block comment describing a section of
! Pillar code which follows it.  Notice that the
! actual text portion of the comment is proceeded
! and followed by a blank line and an empty comment
! line.  Block comments should always be expressed in
! complete sentences.
!
pillar statement;
      .
      .
      .
```

1.3.6.4 Line Comments

Line comments describe a single line of code. Line comments are only allowed in the declaration sections of modules and procedures; they are not allowed in procedure code. Comments in procedure code should be in the block form described in Section 1.3.6.3. Line comments should be aligned vertically within a given section of code. For example, the line comments used to describe the fields in records should all line up within the TYPE declaration section as in:

```
TYPE

    sample : RECORD                                  ! Sample record type
        code : integer;                              ! Record code
        data : array [1..max_length] of real;        ! Data portion
        next_record : sample_pointer;                ! Pointer to next
    END RECORD;

    sample_pointer : POINTER sample;                 ! Pointer to sample record
```

\This example is a non-exported type declaration.\

1.3.7 "Whitespace"

"Whitespace" (in this context) is a term used to describe spacing between Pillar tokens. In general, whitespace is good. For example:

```
a = b + c;
```

is preferable to

```
a=b+c;
```

and:

```
IF xyz <> abc THEN
```

is preferable to

```
IF xyz<>abc THEN
```

For declarations, initializers, and procedure parameters, the guideline for whitespace around the colon (":") and equal sign ("=") characters is that both characters have a space on either side.

For example:

```
VALUE

    value_name = some_value;

VARIABLE

    variable_name : variable_type = variable_initializer;

PROCEDURE foo (
            IN arg : arg_type;
            );
```

Other than appropriate indentation, do not align colons or equal signs in declarations and statements or hyphens in procedure argument definitions. This makes code more difficult to maintain. For example:

```
VALUE

    a_name = 10;
    another_name = 20;
    yet_another_name = 30;
    a_final_name = 40;
```

is preferred to:

```
VALUE

    a_name           = 10;
    another_name     = 20;
    yet_another_name = 30;
    a_final_name     = 40;
```

Procedure invocations and multiline assignment statements are places where it makes sense to attempt to line up code to improve readability. For example:

```
the_resulting_value = one_term_with_a_very_long_name +
                      another_term_with_a_very_long_name;
```

or

```
proc_result = proc_name(
                argument_1 = first_argument,
                arg2 = second_argument
                );
```

1.3.8 Module Format

The general format for a Pillar module is:

```
MODULE module_name;

! Module-level comments

Interface section

Implementation section

Module linkage options

END module_name;
```

- Module-level comments are described in Section 1.3.6.1.
- Interface section:

```
IMPORT
    import_module COMPONENTS component1, component2;
    another_import_module COMPONENTS other1, other2;

VALUE, TYPE, VARIABLE, BIND, PROCEDURE -- exported declarations
```

- Implementation section:

```
IMPLEMENT
    implement_name COMPONENTS comp1, comp2;
    other_impl_name COMPONENTS all*;

IMPORT (as above - these imports are not available to the interface section)

VALUE, TYPE, VARIABLE, BIND, procedure bodies -- non-exported declarations
```

1.3.9 Copyright Formats

For internal sources, the copyright format is:

```
!******************************************************************************
!*                                                                            *
!*  (C) DIGITAL EQUIPMENT CORPORATION 19xx                                    *
!*                                                                            *
!*      This is an unpublished work which was created in the indicated        *
!*      year, which contains confidential and secret information,  and        *
!*      which is protected under the copyright laws.  The existence of        *
!*      the copyright notice is not to be construed as an admission or        *
!*      presumption that publication has occurred. Reverse engineering        *
!*      and unauthorized copying is strictly prohibited.   All  rights        *
!*      reserved.                                                             *
!*                                                                            *
!******************************************************************************
```

\The above copyright statement is available in:

COMPILER$:[WORK.COPYRIGHT]UNPUBLISHED.COPYRIGHT\

For distributable sources, the copyright format is TBD.

\Current policy is to use the internal format for all sources until sources are ready to ship. At that time (or before if the format is defined), all distributable sources will their have copyrights updated.\

1.3.10 Procedure Format

The general format for a Pillar procedure is:

```
PROCEDURE procedure_name...

!++
! Procedure-level comments
!--

VALUE, TYPE, VARIABLE, BIND Declarations

BEGIN
    statement-sequence ...

SUBPROCEDURES

    PROCEDURE sub_procedure_name...

    !++
    ! Procedure-level comments
    !
    ! Notice that subprocedures are indented one level.  If a sub-
    ! procedure itself contains a SUBPROCEDURES section, those
    ! subprocedures are indented one more level and so on.
    !--

    VALUE, TYPE, VARIABLE, BIND Declarations

    BEGIN
        statement-sequence ...
    END sub_procedure_name;

END procedure_name;
```

Procedure-level comments are described in Section 1.3.6.2.

1.3.11 Condition Handler Format

\Coding conventions for condition handlers will be added pending complete definition of Pillar's condition handling syntax.\

1.3.12 Order of Declarations

Declarations are normally grouped together by the type of declaration (such as VALUE, TYPE, and so on). In large modules and procedures, however, declarations may be grouped by function. Within each functional grouping, declarations are grouped together by type. Declarations should appear in the following order:

- VALUE
- TYPE
- VARIABLE
- BIND

1.3.13 Statement Format

The following sections describe preferred formats for several Pillar statements.

1.3.13.1 IF/THEN/ELSE

IF/THEN/ELSE statements are formatted as follows:

```
IF condition THEN
    statement-sequence ...
ELSEIF condition THEN
    statement-sequence ...
ELSE
    statement-sequence ...
END IF;
```

1.3.13.2 LOOP

The various forms of the LOOP statement are formatted as follows:

```
LOOP
    statement-sequence ...
END LOOP;

FOR name ... LOOP
    statement-sequence ...
END LOOP name;

WHILE clause LOOP
    statement-sequence ...
END LOOP;
```

For more complicated loops, use one of these formats:

```
FOR name ... BY ... DOWN TO ... WHILE ... LOOP
    statement-sequence ...
END LOOP name;
```

or:

```
FOR name ...
    BY ...
    DOWN TO ...
    WHILE ... LOOP
    statement-sequence ...
END LOOP name;
```

if all of the loop control does not fit on one line.

1.3.13.3 CASE

CASE statements are formatted as follows:

```
CASE expression
    WHEN set-of-values THEN
        statement-sequence ...
    WHEN set-of-values THEN
        statement-sequence ...
    .
    .
    .
    WHEN OTHERS THEN
        statement-sequence ...
END CASE;
```

1.3.13.4 Blocks (WITH Statement)

Code blocks (defined by the WITH statement) are formatted as follows:

```
WITH

VALUE, TYPE, VARIABLE, BIND declarations

BEGIN
    statement-sequence ...
END;
```

1.3.13.5 VALUE, TYPE, VARIABLE, BIND Declarations

VALUE, TYPE (except record and enumerated types), VARIABLE, and BIND declarations are formatted as follows:

```
VALUE

    first_value = some_value;
    second_value = some_value;

TYPE

    some_type : a_type_declaration;
    another_type : ARRAY [1..first_value] OF integer;

VARIABLE

    variable1 : integer = 10;
    variable_two : POINTER another_type;
    third_variable : boolean;

BIND

    name = variable_name;
```

Notice that a blank line preceeds and succeeds the declaration keyword and the declarations are indented one level from the declaration keyword.

1.3.13.5.1 RECORD Types

RECORD type declarations are examples of declarations which typically span multiple lines. They are formatted as follows:

```
TYPE

    name : RECORD
        CAPTURE ...
        field-list
        LAYOUT
            layout-list
        END LAYOUT;
    END RECORD;
```

The field list is:

```
first_field : field_type;
second_field : field_type;
third_field : field_type;
        .
        .
        .
```

Within records, unions and variants are formatted as follows:

```
UNION CASE ...
    WHEN set-of-values THEN
        field-list
    WHEN set-of-values THEN
        field-list
END UNION;

VARIANTS CASE ...
    WHEN set-of-values THEN
        field-list
    WHEN set-of-values THEN
        field-list
END VARIANTS;
```

1.3.13.5.2 Enumerated Types

Another type declaration which can span multiple lines is that of an enumerated type. Short enumerated type declarations may be written on a single line. For longer declarations where multiple lines are required, the following format is used:

```
TYPE

    enumerated_type_name : (
        enumerated_name_1,
        enumerated_name_2,
                .
                .
                .
        enumerated_name_n
        );
```

1.3.13.6 Procedure Declarations

Procedure declarations are formatted as follows:

- External declarations:

```
PROCEDURE

    procedure_name1 (
        IN first_param : some_type;
        .
        .
        .
        ) RETURNS return_type;
        EXTERNAL;

    procedure_name2 (
        .
        .
        .
        ) RETURNS return_type;
        EXTERNAL;
```

or

```
PROCEDURE procedure_name1 (
            IN first_param : some_type;
                .
                .
                .
            ) RETURNS return_type;
            EXTERNAL;

PROCEDURE procedure_name2 (
                .
                .
                .
            ) RETURNS return_type;
            EXTERNAL;
```

\The second form is required to use the Pillar procedure expansion support provided as an extension to TPU.\

- Normal declarations:

```
PROCEDURE procedure_name (
            IN first_parameter : some_type;
            OUT second_parameter : another_type;
            BIND third_parameter : another_type;
            IN OUT fourth_parameter : another_type;
            ) RETURNS return-type;
```

Section 1.3.10 describes the complete procedure format.

Note:

- Placing parentheses ("(" and ")") on lines which do not contain parameters makes parameter reordering easier.
- The semicolon (";") following the last parameter is optional in Pillar; it should be included to make parameter reordering easier.
- For procedure declarations, the keyword PROCEDURE is just like other declaration keywords (for example TYPE, VALUE, and so on) in that multiple procedure declarations may be made following it. \However, as noted above, this format should be avoided if the TPU Pillar procedure expansion support package is being used.\

- For procedure implementations, the procedure arguments are indented to the closest tab stop following the procedure name (under the 3rd character of the procedure name).

1.3.13.7 Procedure Invocation

The following format is used to invoke a procedure:

- Keyworded parameters:

```
procedure_name(
    first_parameter = parameter1,
    second_parameter = parameter2,
    third_parameter = parameter3
    );
```

- Positional parameters:

```
procedure_name(arg1, arg2, arg3);
```

The use of keywords to specify the arguments in a procedure call is preferred, but not required. Use of keywords when invoking externally declared procedures is strongly recommended. Code examples used in documentation must not use positional arguments in function calls.

\Additional information regarding use of the KEYWORD parameter option TBS.\

1.3.14 Message and Condition Declarations

\Coding conventions declaring messages and conditions will be added pending complete definition of Pillar's message and condition declaration syntax and use.\

1.3.15 Miscellaneous

The following is a list of several other conventions which do not fall under any of the previous groupings.

- Use of pointer dereference character ("^") in record field references—Pillar does not require that pointers to records be explicitly dereferenced when the fields of those records are being accessed. It is felt, however, that use of the dereference character provides more information about the record, especially when multiple levels of dereferencing are required. The prefered convention is to explicitly dereference all pointers. The following code fragments illustrates use of explicit pointer dereferencing.

```
TYPE

    sample_record : RECORD
        data : integer;
        flag : boolean;
        record_pointer : sample_record_pointer;
    END RECORD;
    sample_record_pointer : POINTER sample_record;

VARIABLE

    first_record, second_record : sample_record_pointer;

BEGIN

    !
    ! Allocate the records.
    !

    ALLOCATE first_record LOCAL;
    ALLOCATE second_record LOCAL;
```

```
!
! Set the data and flag values in the first record.  Pointer
! dereferencing here is not required but is preferred.
!

first_record^.data = 1;
first_record^.flag = false;

!
! Set the second record equal to the first record.  Since the
! entire record is being accessed, pointer dereferencing is
! required.
!

second_record^ = first_record^;

!
! Set the records to point to each other.  Pointer dereferencing
! is not required to access the "record_pointer" field but is
! preferred; the entire record is not dereferenced as it is needed
! as a pointer.
!

first_record^.record_pointer = second_record;
second_record^.record_pointer = first_record;
```

\The ALLOCATE statement used above is not available in SIL.\

- Others TBD.

1.4 OPEN ISSUES

The following issues are yet to be resolved.

- Compile-time facility procedure and macro names.
- Use of "MICA$" as the file name prefix for system files.
- Condition handler format.
- Message and condition declarations.

Mica Working Design Document
Process Structure

Revision 0.6
29–OCT–1987

Issued by:
Mark Lucovsky

digital™

TABLE OF CONTENTS

INDEX

EXAMPLES

FIGURES

TABLES

Revision History

Date	Revision Number	Author	Summary of Changes
10–Jun–86	0.0	Tom Miller	Initial entry
29–Jun–86	0.1	Tom Miller	Incorporating review comments
27–Aug–86	0.2	Tom Miller	Multiple environment support
06–Apr–87	0.3	Tom Miller	Rewrite for second WDD
27–AUG–1987	x.1	Mark Lucovsky	First Draft for third WDD
04–SEP–1987	x.2	Mark Lucovsky	Incorporate comments from first draft. Most notable change was the addition of IO accounting, process and thread exit status, and section on thread /process startup/rundown summary
08–OCT–1987	x.3	Mark Lucovsky	Incorporate comments from Second draft. Most notable change was the section on system threads, and the thread parameter passing scheme
09–OCT–1987	0.4	Mark Lucovsky	Added exec$create_user( ) and description of thread_ record
16–OCT–1987	0.5	Mark Lucovsky	Proofreading corections, moved security profile from process object to the thread object, added cancel io by thread support to the thread object.
29–OCT–1987	0.6	Mark Lucovsky	Added access restrictions to user object, revised hierarchy collapse description

CHAPTER 1

PROCESS STRUCTURE

1.1 Introduction

This chapter describes the external interfaces and data structures of the Mica process structure, the architecture of which is based on the User, Job, Process, Thread (UJPT) hierarchy. This chapter also describes the UJPT implementation in terms of its algorithms and dependencies on other portions of the Mica system (e.g. the kernel and object architecture).

1.2 Goals/Requirements

The goal of the UJPT architecture is to provide a vehicle for controlling multiple threads of execution in a single address space. The architecture provides facilities for resource usage control, security profile management, address space and image management, and object container directory services.

1.3 UJPT Hierarchy

The UJPT architecture consists of a hierarchy of objects. The objects provide a logical grouping of functionality and control.

1.3.1 The User Object

The User object appears at the highest level of the UJPT hierarchy. Its primary function is to provide a focal point for acquiring security profiles and resource quotas/limits for its underlying objects.

The User object is implemented as a system level object in the "USER$OBJECT_CONTAINER" object container.

1.3.1.1 Object Structure

Each user of the Mica system is assigned a unique username, a security profile, and a set of resource limits or quotas. The Mica system keeps track of this information in a system-wide authorization file. If the user has at least one active job, the information is also kept in his user object. As we shall see later in this chapter, information from the user object is propagated down the UJPT hierarchy on an as-needed basis.

NOTE

The intent of the Mica executive is to remain independent of the system-wide authorization file. Therefore, all Mica user attributes are stored in the user object. In addition, the Mica executive places no restrictions on the source of information stored in the user object. It does, however, place a Digital-reserved identifier in the ACL for the user object OTD which limits who can create user objects.

The user object is split into a user object body and a user control block. The user object body contains the information necessary to support the UJPT hierarchy. The user control block contains the vital information of the user object. Example 1–1 illustrates the data structures used to represent the user object.

Example 1–1: User Object Structure

```
e$t_user_object_body: RECORD
    u_obj_id: e$t_object_id;                        ! Object ID of the user object
    u_user_flags: e$t_user_flags;                   ! User object flags
    u_job_queue_mutex: k$dispatcher_object(mutex)   ! Mutex for job management
    u_job_count: integer;                           ! Number of Jobs owned by the user
    u_job_queue_hd: e$t_linked_list;                ! List head of job objects
    u_ucb: e$t_user_control_block;                  ! User Control Block
END RECORD;

e$t_user_control_block: RECORD
    ucb_username: string(e$c_max_user_name);              ! User Name
    ucb_security_profile: e$t_security_profile;           ! User Security Profile
    ucb_quotas: e$t_quotas;                               ! Resource usage control information
    ucb_thread_priority: k$combined_priority;             ! Default thread priority
    ucb_access_restrictions: e$t_access_restrictions;     ! Access Restrictions
    ucb_user_allocation_list: e$t_allocation_list;        ! objects allocated to the user object
END RECORD;
```

1.3.1.1.1 Security Profile

The security profile maintained in the user object contains the list of identifiers assigned to the Mica user. The identifier list gives access rights to the user object as described in Chapter 11, Security and Privileges.

1.3.1.1.2 Resource Control

The goals of the Mica system resource control and quota architecture are:

- Prevent a single user from abusing the system by over running system resources.
- Be simple, predictable and easy to understand.
- Provide repeatable consistent behavior.

The Mica system achieves these goals through data structures maintained in the user object and through policies implemented in the object architecture, memory management system, and the kernel. Example 1–2 illustrates the resource-control data structures maintained in the user object.

Example 1–2: Resource Control Structures

```
!
! User Object Resource Control
!
e$t_quotas: RECORD
    q_usage_and_limits: e$t_quota_usage_and_limits;  ! Currently Used Quotas and Quota Limits
    q_per_job_limits: e$t_quota_limits;              ! Per Job Limits
    q_per_process_limits: e$t_quota_limits;          ! Per Process Limits
END RECORD;

!
! Quota Limits
!
e$t_quota_limits: RECORD
    ql_deductable_limits: e$t_quota_deductable_limits;       ! Deductable Resource Limits
    ql_nondeductable_limits: e$t_quota_nondeductable_limits; ! Non-Deductable Resource Limits
END RECORD;

!
! Quota Usage and Limits
!
e$t_quota_usage_and_limits: RECORD
    qual_mutex: k$dispatcher_object(mutex);            ! Used for block quota allocations
    qual_limits: e$t_quota_limits;                     ! Resource limits for this object
    qual_usage: e$t_quota_usage;                       ! Resources used by this object
END RECORD;

!
! Deductable Limits
!
e$t_quota_deductable_limits: RECORD
    qdl_paging_file_limit: e$t_resource_counter;     ! Max blocks of paging file usable by object
    qdl_paged_pool_limit: e$t_resource_counter;      ! Max number bytes paged pool usable by object
    qdl_non_paged_pool_limit: e$t_resource_counter;  ! Max number bytes non paged pool usable by object
    qdl_cpu_time_limit: e$t_time_value;              ! Max cpu time used by object
END RECORD;

!
! Non Deductable Limits
!
e$t_quota_nondeductable_limits: RECORD
    qnl_working_set_limit: e$t_resource_counter;     ! Max pages in working set
    qnl_working_set_extent: e$t_resource_counter;    ! Largest Possible Working Set
END RECORD;

!
! Quota Usage
!
e$t_quota_usage: RECORD
    qu_paging_file_in_use: e$t_resource_counter;     ! Number blocks of paging file in use by object
    qu_paged_pool_in_use: e$t_resource_counter;      ! Number bytes paged pool in use by object
    qu_non_paged_pool_in_use: e$t_resource_counter;  ! Number bytes non paged pool in use by object
    qu_working_set_in_use: e$t_resource_counter;     ! Pages in working set for this object
    qu_cpu_time_used: e$t_time_value;                ! Cpu time used by object
END RECORD;
```

During user-object creation, the *ucb_quotas* field of the user control block is initialized. The values are obtained from the *user_record* parameter to the *exec$create_user()* system service.

Once established, the *ucb_quotas* field of the user control block becomes the focal point for resource allocation limitation. The Mica system organizes resource limits as deductable and non-deductable resources. All operations on *e$t_quota_limits* are performed in terms of the attributes of deductable and non-deductable resource limits.

1.3.1.1.2.1 Deductable Resource Limits

Deductable resource limits are charged to the next highest object in the UJPT hierarchy at object creation time. An example of this property can be seen in the creation of a process object. Assume a job object had 100 units of paged pool available in *qdl_paged_pool_limit*, and the user object specified that the per process limit for *qdl_paged_pool_limit* was 50 units. After the process object was created, the job object would be charged with 50 units of paged pool in *qu_paged_pool_in_use*. The process object would have 50 units of paged pool available in *qdl_paged_pool_limit*, and would be charged with 0 units of paged pool in *qu_paged_pool_in_use*.

1.3.1.1.2.2 Non-Deductable Resource Limits

Non-deductable resource limits are limits enforced by policies of the Mica system, but are not charged for against the higher level objects pool of available resources. For example, assume that in the creation of a process the user object specified a working set limit of 50 units. As a consequence, all job objects and process objects would contain the 50 units of resource in their *qnl_working_set_limit* fields.

1.3.1.1.3 Access Restrictions

The user object maintains the current system access restrictions for the Mica user that it represents. The access restrictions are not enforced by the UJPT architecture. External processes may inspect the access restrictions in the current set of user objects and determine what type of enforcement actions are necessary. Example 1–3 illustrates the data structures used to maintain the access restrictions placed in the user object.

Example 1–3: Access Restriction Data Structures

```
!
! Access Restrictions
!
e$t_access_restrictions: RECORD
    ar_restriction_vector: ARRAY[e$t_job_class] OF e$t_class_access_restrictions;
    ar_expiration_date: e$t_date;       ! The last day that user can access the system
END RECORD;

!
! Per Job Class Access Restrictions
!
e$t_class_access_restrictions: RECORD
    car_prime_days: e$t_day_set;         ! The prime days user can access system
    car_non_prime_days: e$t_day_set;     ! The non-prime days user can access system
    car_prime_hours: e$t_hour_set;       ! The hours on prime days user can access system
    car_non_prime_hours: e$t_hour_set;   ! The hours on non prime days user can access system
END RECORD;
```

1.3.1.2 Functional Interface

The Mica executive provides entry points capable of creating and deleting user objects, and setting and extracting various attributes of a User object.

1.3.1.2.1 User Creation

Creating a user object also causes a UJPT hierarchy to be created. The system service *exec$create_user()* creates a user object, job object, process object, and thread object. If there is a name collision between the new user object and an existing user object for the same user, then the new user object is discarded, and the job, process, and thread objects are attached to the existing user object. Example 1–4 illustrates the interface to *exec$create_user()*.

Example 1–4: User Object Creation System Interface

```
PROCEDURE exec$create_user (
            OUT object_id: exec$t_object_id;
            IN  container: exec$t_object_id = DEFAULT;
            IN  name: exec$t_object_name = DEFAULT;
            IN  acl: exec$t_acl = DEFAULT;

            IN user_record: exec$t_user_record;
            IN user_allocation_list: exec$t_allocation_list = DEFAULT;

            IN job_record: exec$t_job_record = DEFAULT;
            IN job_initial_container: exec$t_object_id = DEFAULT;
            IN job_allocation_list: exec$t_allocation_list = DEFAULT;

            IN process_record: exec$t_process_record;
            IN process_public_container: exec$t_object_id = DEFAULT;
            IN process_private_container: exec$t_object_id = DEFAULT;
            IN process_allocation_list: exec$t_allocation_list = DEFAULT;

            IN thread_record: exec$t_thread_record = DEFAULT;
            IN thread_allocation_list: exec$t_allocation_list = DEFAULT;
            IN thread_data_block: quadword_data(*) CONFORM OPTIONAL;
            IN thread_immediate_parameter1: exec$t_thread_parameter = DEFAULT;
            IN thread_immediate_parameter2: exec$t_thread_parameter = DEFAULT;
            IN thread_status: exec$t_object_id = DEFAULT;
            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Create a user, job, process, and thread object as specified by the parameters.
!   If the user object collides with an existing user object, then use the existing
!   user object
!
! Arguments:
!
!   object_id                   Object ID of the resulting user object
!   container                   Object container for user object (ignored)
!   name                        Name of user object
!   acl                         ACL to place on user object
!   user_record                 Attributes of new user (from authorization file ?)
!   user_allocation_list        Objects to be allocated to the user object. If not present then
!                               no objects are allocated to the user
!   job_record                  Attributes of the job being created. If not present, then
!                               values are obtained from current user object
!   job_initial_container       Job level object container to be transfered into the job
!                               level container directory for this job. If not present then
!                               container directory comes up empty
!   job_allocation_list         Objects to be allocated to the job object. If not present then
!                               no objects are allocated to the job
!   process_record              Attributes of the process being created
!   process_public_container    Process level public container to be transfered into the process
!                               level container directory for the process. If not present then
!                               container comes up empty.
!   process_private_container   Process level private container to be transfered into the process
!                               level container directory for the process. If not present then
!                               container comes up empty.
!   process_allocation_list     Objects to be allocated to the process object. If not present then
!                               no objects are allocated to the process
!   thread_record               Attributes of the thread being created
!   thread_allocation_list      Objects to be allocated to the thread object. If not present then
!                               no objects are allocated to the thread
!   thread_data_block           Arbitrary data block passed to initial thread. Pointer in TCR, if
!                               pointer is NIL, then no data block was passed
!   thread_immediate_parameter1 Immediate parameter passed to thread through TCR
!   thread_immediate_parameter2 Immediate parameter passed to thread through TCR
!   thread_status               Exit status object to be bound to the initial thread. If not present
!                               then the thread is created without an exit status object
!
! Return value:
!
!   TBS
!
!--
```

From the interface to *exec$create_user()*, it is clear that the *user_record* can have an impact on the structure of the user being created. Example 1–5 illustrates the layout of the *user_record*.

Example 1–5: User Record Structure

```
    !
    ! The User Record
    !
    exec$t_user_record: RECORD
        !
        ! User Fields
        !
        ! The User fields are only used to initialize a user object if no user
        ! object exists. The intent is for the contents of these fields come from
        ! the system wide authorization file
        !
        user_username: string(e$c_max_user_name);          ! User Name
        user_security_profile: e$t_security_profile;       ! User Security Profile from Authorization File
        user_per_user_limits: e$t_quota_limits;            ! Per User Resource Limits
        user_per_job_limits: e$t_quota_limits;             ! Per Job Resource Limits
        user_per_process_limits: e$t_quota_limits;         ! Per Process Resource Limits
        user_thread_priority: k$combined_priority;         ! Default thread priority
        user_access_restrictions: e$t_access_restrictions  ! Users Access Restrictions
    END RECORD;
```

1.3.1.2.2 Get/Set User Information

The *exec$get_user_information* and *exec$set_user_information* system services provide a mechanism to obtain and to modify attributes of the specified user object. Example 1–6 illustrates the interfaces to the user object get/set system services.

Example 1–6: Get/Set User Information System Interface

```
PROCEDURE exec$get_user_information (
            IN user_object_id: exec$t_object_id = DEFAULT;
            IN user_get_items: exec$t_item_list;
            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Return information about the user object to the caller. The
!   information returned is item list driven
!
! Arguments:
!
!   user_object_id      if present, the object ID of user object that is to be inspected
!                       otherwise, the user object of the calling thread is assumed
!   user_get_items      item list identifying user object information to be extracted
!
! Return value:
!
!   TBS
!
!--
```

Example 1–6 Cont'd. on next page

Example 1–6 (Cont.): Get/Set User Information System Interface

```
PROCEDURE exec$set_user_information (
            IN user_object_id: exec$t_object_id = DEFAULT;
            IN user_get_items: exec$t_item_list;
            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Modify information in the user object. The
!   information to be modified is item list driven
!
! Arguments:
!
!   user_object_id      if present, the object ID of user object that is to be modified
!                       otherwise, the user object of the calling thread is assumed
!   user_get_items      item list identifying user object information to be modified
!
! Return value:
!
!   TBS
!
!--
```

Only certain pieces of the user object may be inspected or modified. Table 1–1 illustrates the possible item codes and the information read or written when using the item code.

Table 1–1: Get/Set User Information Item Codes

Item Code	Set Action	Get Action
e$i_job_count	error	return u_job_count
e$i_job_ids	error	return object ID's of jobs owned by user
e$i_username	error	return username of user
e$i_security_profile	replace ucb_security_profile	return ucb_security_profile
e$i_quotas	error	return ucb_quotas
e$i_user_limits	replace qual_limits	return qual_limits
e$i_job_limits	replace q_per_job_limits	return q_per_job_limits
e$i_process_limits	replace q_per_process_limits	return q_per_process_limits
e$i_thread_priority	replace ucb_thread_priority	return ucb_thread_priority
e$i_access_restrictions	replace ucb_access_restrictions	return ucb_access_restrictions
e$i_allocation_list	error	return ucb_user_allocation_list

1.3.1.2.3 User Deletion

The *exec$force_exit_user()* system service provides a mechanism for removing an active Mica user from the system. The service effectively causes an entire UJPT hierarchy to be removed, including all jobs, processes, and threads that are directly beneath the user object. Example 1–7 illustrates the interface used to remove a user object from the Mica system.

Example 1–7: User Object Deletion System Interface

```
PROCEDURE exec$force_exit_user (
            IN user_object_id: exec$t_object_id = DEFAULT;
            IN exit_status: exec$exit_status;
            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Causes the UJPT hierarchy whose user object head is user_object_id to
!   be removed from the Mica system
!
! Arguments:
!
!   user_object_id      the user object to be removed. If not specified,
!                       then the current user is assumed
!   exit_status         the reason that the user is force-exiting
!
! Return value:
!
!   TBS
!
!--
```

1.3.2 The Job Object

The job object appears at the second level of the UJPT hierarchy. Its sole function is to provide a set of resource limits for a collection of processes running as a job. The job object also provides a job level container directory.

The job object is implemented as a system level object in the "JOB$OBJECT_CONTAINER" object container.

1.3.2.1 Object Structure

Each job in the Mica system represents a set of active processes and is responsible for controlling the resources used by those processes.

The job object is split into a job object body and a job control block. The job object body contains the information necessary to maintain its position in a UJPT hierarchy. The job control block contains the information necessary to provide resource management for the job's processes. Example 1–8 illustrates the job object.

Example 1–8: Job Object Structure

```
    !
    ! Job Object Body
    !
    e$t_job_object_body: RECORD
        j_obj_id: e$t_object_id;                            ! Object ID of The job object
        j_user_pointer: POINTER e$t_user_object_body;       ! Referenced Pointer to owning User
        j_job_flags: e$t_job_flags;                         ! Job Flags
        j_job_queue: e$t_linked_list;                       ! List of users jobs
        j_process_queue_mutex: k$dispatcher_object(mutex);  ! Mutex for process management
        j_process_count: integer;                           ! Number Of processes of the job
        j_process_queue_hd: e$t_linked_list;                ! List head of jobs processes
        j_jcb: e$t_job_control_block;                       ! Job Control Block
    END RECORD;
```

Example 1–8 Cont'd. on next page

Example 1–8 (Cont.): Job Object Structure

```
!
! Job Control Block
!
e$t_job_control_block: RECORD
    jcb_job_class: e$t_job_class;                          ! The jobs class
    jcb_usage_and_limits: e$t_quota_usage_and_limits;      ! Current resources used/resource limits
    jcb_job_condir_mutex: k$dispatcher_object(mutex);      ! Job Level Condir mutex
    jcb_job_condir_id: e$t_object_id;                      ! Job Level Container directory ID
                                                           !   visible in jobs context
    jcb_job_alt_condir_id: e$t_object_id;                  ! Job Level Container directory ID
                                                           !   visible in an arbitrary context
    jcb_job_condir_pointer: POINTER e$t_object_header;     ! Pointer to Job Level Condir
    jcb_job_allocation_list: e$t_allocation_list;          ! Objects allocated to the job objects
END RECORD;
```

1.3.2.1.1 Resource Control

The job object maintains resource usage information for itself, in addition to providing a pool of resources to its processes on an as-needed basis. During job-object creation, the *jcb_usage_and_limits.qual_limits* field of the job control-block is set to the value of *q_per_job_limits* from the user control block. The *jcb_usage_and_limits.qual_usage* field of the job control-block is then set to *zero()*, and the *q_usage_and_limits.qual_usage* field of the user control block is incremented by *q_per_job_limits* to reflect the resources allocated to the job. Once this resource shuffling operation has completed, the value of *jcb_usage_and_limits.qual_limits* represents the amount of system resources available to the job object and to all its process.

While the above resource allocation scheme is the normal case, during job creation a parameter specifying the per-job limits for the job can be specified, altering the algorithm. This value simply overrides the value from *q_per_job_limits* in the above example and applies to the newly created job.

1.3.2.2 Functional Interface

The Mica executive provides entry points capable of creating and deleting job objects, and setting and extracting various attributes of a job object.

As part of job object creation, all of the necessary support data structures are created, including a job level container directory and associated kernel mutex dispatcher object.

1.3.2.2.1 Job Creation

The system service *exec$create_job()* causes the creation of a job object, a process object, and a thread object. These objects appear beneath the user object of the calling thread. Example 1–9 illustrates the interface to *exec$create_job()*.

Example 1–9: Job Object Creation System Interface

```
PROCEDURE exec$create_job (
            OUT object_id: exec$t_object_id;
            IN  container: exec$t_object_id = DEFAULT;
            IN  name: exec$t_object_name = DEFAULT;
            IN  acl: exec$t_acl = DEFAULT;

            IN job_record: exec$t_job_record = DEFAULT;
            IN job_initial_container: exec$t_object_id = DEFAULT;
            IN job_allocation_list: exec$t_allocation_list = DEFAULT;

            IN process_record: exec$t_process_record;
            IN process_public_container: exec$t_object_id = DEFAULT;
            IN process_private_container: exec$t_object_id = DEFAULT;
            IN process_allocation_list: exec$t_allocation_list = DEFAULT;

            IN thread_record: exec$t_thread_record = DEFAULT;
            IN thread_allocation_list: exec$t_allocation_list = DEFAULT;
            IN thread_data_block: quadword_data(*) CONFORM OPTIONAL;
            IN thread_immediate_parameter1: exec$t_thread_parameter = DEFAULT;
            IN thread_immediate_parameter2: exec$t_thread_parameter = DEFAULT;
            IN thread_status: exec$t_object_id = DEFAULT;

            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Create a job, process, and thread object as specified by the parameters.
!   If no user object exists, then also create a user object.
!
! Arguments:
!
!   object_id                   Object ID of the resulting job object
!   container                   Object container for job object (ignored)
!   name                        Name of job object
!   acl                         ACL to place on job object
!   job_record                  Attributes of the job being created. If not present, then
!                               values are obtained from current user object
!   job_initial_container       Job level object container to be transfered into the job
!                               level container directory for this job. If not present then
!                               container directory comes up empty
!   job_allocation_list         Objects to be allocated to the job object. If not present then
!                               no objects are allocated to the job
!   process_record              Attributes of the process being created
!   process_public_container    Process level public container to be transfered into the process
!                               level container directory for the process. If not present then
!                               container comes up empty.
!   process_private_container   Process level private container to be transfered into the process
!                               level container directory for the process. If not present then
!                               container comes up empty.
!   process_allocation_list     Objects to be allocated to the process object. If not present then
!                               no objects are allocated to the process
!   thread_record               Attributes of the thread being created
!   thread_allocation_list      Objects to be allocated to the thread object. If not present then
!                               no objects are allocated to the thread
!   thread_data_block           Arbitrary data block passed to initial thread. Pointer in TCR, if
!                               pointer is NIL, then no data block was passed
!   thread_immediate_parameter1 Immediate parameter passed to thread through TCR
!   thread_immediate_parameter2 Immediate parameter passed to thread through TCR
!   thread_status               Exit status object to be bound to the initial thread. If not present
!                               then the thread is created without an exit status object
!
! Return value:
!
!   TBS
!
!--
```

From the interface to *exec$create_job()*, it is clear that the *job_record* can have an impact on the structure of the job being created. Example 1–10 illustrates the layout of the *job_record*.

Example 1–10: Job Record Structure

```
    !
    ! The Job Record
    !
    exec$t_job_record: RECORD
        !
        ! Job Fields
        !
        job_class: e$t_job_class;              ! The class of the job being created (i.e. network,batch...)
        !
        ! Per Job Resource limits. This value is used as the
        ! qual_limits value for the job object, and is deducted
        ! from the qual_usage field of the owning user object.
        ! A value of zero() in any one of fields means to use the
        ! corresponding value of the q_per_job_limit from the
        ! user structure
        !
        job_per_job_limits: e$t_quota_limits;
    END RECORD;
```

1.3.2.2.2 Job Deletion

The *exec$force_exit_job()* system service provides a mechanism for removing job objects from the system. The removal of a job has the following system-wide effects:

- All processes beneath the job are removed from the system.
- The amount of resources available to the job (*qual_limits–qual_usage*) is returned to the job's user object by decrementing *qual_usage* in the user object.
- If the job object is the last job owned by its user object, then the user object is removed from the system.

Example 1–11 illustrates the interface to *exec$force_exit_job()*.

Example 1–11: Job Object Deletion System Interface

```
PROCEDURE exec$force_exit_job (
            IN job_object_id: exec$t_object_id = DEFAULT;
            IN exit_status: exec$t_exit_status;
            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Causes the job object specified by job_object_id to
!   be removed from the Mica system
!
! Arguments:
!
!   job_object_id       the job object to be removed. If not specified,
!                       then the current job is assumed
!   exit_status         the reason that the job is force-exiting
!
! Return value:
!
!   TBS
!
!--
```

1.3.2.2.3 Get/Set Job Information

The *exec$get_job_information()* and *exec$set_job_information()* system services provide a mechanism to obtain and to modify attributes of the specified job object. Example 1–12 illustrates the interfaces to the job object get/set system services.

Example 1–12: Get/Set Job Information System Interface

```
PROCEDURE exec$get_job_information (
            IN job_object_id: exec$t_object_id = DEFAULT;
            IN job_get_items: exec$t_item_list;
            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Return information about the job object to the caller. The
!   information returned is item list driven
!
! Arguments:
!
!   job_object_id       if present, the object ID of job object that is to be inspected
!                       otherwise, the job object of the calling thread is assumed
!   job_get_items       item list identifying job object information to be extracted
!
! Return value:
!
!   TBS
!
!--

PROCEDURE exec$set_job_information (
            IN job_object_id: exec$t_object_id = DEFAULT;
            IN job_get_items: exec$t_item_list;
            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Modify information in the job object. The
!   information to be modified is item list driven
!
! Arguments:
!
!   job_object_id       if present, the object ID of job object that is to be modified
!                       otherwise, the job object of the calling thread is assumed
!   job_get_items       item list identifying job object information to be modified
!
! Return value:
!
!   TBS
!
!--
```

Only certain pieces of the job object may be inspected or modified. Table 1–2 illustrates the possible item codes and the information read or written when using the item code.

Table 1–2: Get/Set Job Information Item Codes

Item Code	Set Action	Get Action
e$i_user_id	error	return object ID of jobs user object
e$i_process_count	error	return j_process_count
e$i_process_ids	error	return object ID's of processes owned by job
e$i_usage_and_limits	error	return jcb_usage_and_limits
e$i_job_limits	replace qual_limits	return qual_limits
e$i_job_condir_id	error	return jcb_job_condir_id
e$i_allocation_list	error	return jcb_job_allocation_list
e$i_job_class	error	return jcb_job_class

1.3.3 The Process Object

The Process object appears at the third level of the UJPT hierarchy. Its primary function is to provide address space support and program image support for a set of execution threads, and to manage the set of process-level objects. The process object is the target of all accounting information. The process object can also act as a focal point for control operations.

There can be multiple processes in a job. Processes created as a result of job creation are *top level* processes. Once established, a process may cause the creation of other processes. These new processes are *sub–processes*, or *child processes*. Their creating processes are referred to as *parent processes*.

The Process object is implemented as a system level object in the "PROCESS$OBJECT_CONTAINER" object container.

1.3.3.1 Object Structure

Each process in the Mica system represents a set of execution threads and in some cases a set of sub-processes. The process object is responsible for managing the address spaces of its execution threads and for controlling the resource allocation limits of its execution threads.

The process object is split into a process object body and a process control block. The process object body contains the information necessary to maintain its position in the UJPT hierarchy, a task which includes coordinating its sub-process objects. The process control block contains the information necessary to manage the address space, to control the resource usage, and to pool accounting information of all of its execution threads. Example 1–13 illustrates the process object.

Example 1–13: Process Object Structure

```
!
! Process Object Body
!
e$t_process_object_body: RECORD
    p_obj_id: e$t_object_id;                                    ! Obje9ct ID of process object
    p_job_pointer: POINTER e$t_job;                             ! Referenced pointer to owning job
    p_parent_process: POINTER e$t_process;                      ! Referenced pointer to owning process, or NIL
    p_process_flags: e$t_process_flags;                         ! Process Flags
    p_process_queue: e$t_linked_list;                           ! List of jobs processes
    p_sub_process_queue: e$t_linked_list;                       ! List of parents sub-processes
    p_thread_queue_mutex: k$dispatcher_object(mutex);           ! Mutex for thread management
    p_thread_count: integer;                                    ! Number of threads of the process
    p_thread_queue_hd: e$t_linked_list;                         ! List head of processes threads
    p_sub_process_queue_mutex: k$dispatcher_object(mutex);      ! Mutex for sub-process management
    p_sub_process_count: integer;                               ! Number of sub-processes of the process
    p_sub_process_queue_hd: e$t_linked_list;                    ! List head of processes sub-processes
    p_pcb: e$t_process_control_block;                           ! Process Control Block
END RECORD;

!
! Process Control Block
!
e$t_process_control_block: RECORD
    pcb_usage_and_limits: e$t_quota_usage_and_limits;           ! Current resources used/resource limits
    pcb_process_condir_id: e$t_object_id;                       ! Process Level Container directory ID
                                                                !   visible in an processes context
    pcb_process_alt_condir_id: e$t_object_id;                   ! Process  Level Container directory ID
                                                                !   visible in an arbitrary context
    pcb_accounting: e$t_accounting_summary;                     ! Process accounting summary
    pcb_pcr_base: POINTER e$t_process_control_region;           ! User Readable Process Control Region
    pcb_process_control_pte: mm$pte;                            ! Prototype PTE for seg 1 page table page
    pcb_ptbr: POINTER k$page_table;                             ! Pointer to page table
    pcb_kernel_process_block: k$process;                        ! Kernel Process Block
    pcb_exit_status_id: e$t_object_id;                          ! Exit Status Object ID for process
    pcb_exit_status_ptr: POINTER e$t_exit_status_body;          ! Exit Status for Process
    pcb_process_allocation_list: e$t_allocation_list;           ! objects allocated to the process object
    !
    ! Object Architecture Defined Container Directory Vector
    !
    pcb_condir_mutex: ARRAY [e$t_level_type] OF POINTER k$dispatcher_object(mutex);
    pcb_condir_address: ARRAY [e$t_level_type] OF POINTER e$t_object_header;
END RECORD;
```

Example 1–13 Cont'd. on next page

Example 1–13 (Cont.): Process Object Structure

```
!
! Process Control Region
!
! The process control region appears in the processes address space as user read only/ system
! read write.
!
e$t_process_control_region: RECORD
    pcr_image_name: string(e$c_max_image_name);          ! process image name
    pcr_number_running_threads: e$t_resource_counter;    ! number of running threads for this process
    pcr_object_id: e$t_object_id;                        ! process object id - duplicate of p_obj_id
    pcr_exit_handlers: e$t_exit_handlers;                ! Process exit Handlers
    pcr_exec_dispatch_table: e$t_dispatch_table;         ! Executive routines dispatch table
END RECORD;
```

1.3.3.1.1 Resource Control

The process object maintains resource usage information for all of its threads. Unlike the job object, the process object's *qual_usage* values represent resources actively in use by its threads. Each time one of the process objects threads consume paged pool, the *qu_paged_pool_in_use* field is incremented by the amount of pool actually used. This action is called *pooling* the resource usage from the thread level to the process level.

During process object creation, the *pcb_usage_and_limits.qual_limits* field of the process control block is set to the value of *q_per_process_limits* from the user control block. The *pcb_usage_and_limits.qual_usage* field of the process control block is then set to *zero()*, and the *q_usage_and_limits.qual_usage* field of the job control block is incremented by *q_per_process_limits* to reflect the resources allocated to the process. Once this resource shuffling operation has completed, the value of *pcb_usage_and_limits.qual_limits* represents the amount of system resources available to the process object which can be consumed by all its thread objects.

While the above resource allocation scheme is the normal case, during process creation a parameter specifying the per-process limits for the process can be specified, altering the algorithm. This value simply overrides the value from *q_per_process_limits* in the above example and applies to the newly created process.

1.3.3.1.2 Process Accounting

The process object maintains accounting information for all of its threads. Process accounting information is pooled from the thread level to the process level. Example 1–14 illustrates the types of information accounted for at the process level in the Mica system.

NOTE

Process accounting information is recorded with interlocked instructions, such that the information is always maintained in an up-to-date state.

Example 1–14: Process Accounting Structure

```
!
! Process Accounting Summary
!
! The final accounting record contains this information in TLV format
! in addition to fields identifying the process, image name, user ...
!
e$t_accounting_summary: RECORD
    acct_cpu_cycles: e$t_counter;                    ! Number of cycles used by the process
    acct_total_page_faults: e$t_counter;             ! Total number of page faults
    acct_hard_page_faults: e$t_counter;              ! Number of page faults for non resident pages
    acct_soft_page_faults: e$t_counter;              ! Number of page faults fixed from reclaim list
    acct_dzro_page_faults: e$t_counter;              ! Number of demand zero page faults
    acct_com_page_faults: e$t_counter;               ! Number of copy on modify page faults
    acct_peak_virtual_memory: e$t_counter;           ! Peak virtual memory size
    acct_peak_working_set_size: e$t_counter;         ! Peak working set size
    acct_start_time: e$t_time_value;                 ! Start time of process
    acct_end_time: e$t_time_value;                   ! End time of process
    acct_page_file_usage: e$t_counter;               ! Peak page file usage
    acct_paged_pool_usage: e$t_counter;              ! Peak paged pool usage
    acct_non_paged_pool_usage: e$t_counter;          ! Peak non paged pool usage
    !
    ! IO Accounting
    ! Request IO's are counted once.
    ! Each FPU that passes on an IRP (execute_io's) must also record the transfer
    ! by incrementing the counter for its class of FPU
    !
    acct_request_io_count: e$t_counter;              ! Number of request_io's
    acct_execute_io_count: ARRAY[e$fpu_class]        ! Number of execute_io's per fpu class
                           OF e$t_counter;
END RECORD;
```

1.3.3.2 Functional Interface

The Mica executive provides entry points capable of creating and deleting process objects, setting and extracting various attributes of a Process object, and performing control operations on all threads of a process. Control operations are Suspend/Resume Process, Hibernate/Wake Process, and Signal Process.

As part of process-object creation, all of the necessary support data structures are created, including a read only process control region (PCR), and a process-level object-container directory. The PCR is part of the process's user-mode read-only address space. The Mica executive places information in the PCR so that the process can read it without entering the system.

1.3.3.2.1 Process Creation

The *exec$create_process()* system service extends an existing UJPT hierarchy by causing the creation of a process object and a thread object. The newly created process object becomes a sub-process of the process above the calling thread. Example 1–15 illustrates the interface to *exec$create_process()*.

Example 1–15: Process Object Creation System Interface

```
PROCEDURE exec$create_process (
            OUT object_id: exec$t_object_id;
            IN  container: exec$t_object_id = DEFAULT;
            IN  name: exec$t_object_name = DEFAULT;
            IN  acl: exec$t_acl = DEFAULT;

            IN process_record: exec$t_process_record;
            IN process_public_container: exec$t_object_id = DEFAULT;
            IN process_private_container: exec$t_object_id = DEFAULT;
            IN process_allocation_list: exec$t_allocation_list = DEFAULT;

            IN thread_record: exec$t_thread_record = DEFAULT;
            IN thread_allocation_list: exec$t_allocation_list = DEFAULT;
            IN thread_data_block: quadword_data(*) CONFORM OPTIONAL;
            IN thread_immediate_parameter1: exec$t_thread_parameter = DEFAULT;
            IN thread_immediate_parameter2: exec$t_thread_parameter = DEFAULT;
            IN thread_status: exec$t_object_id = DEFAULT;

            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Create a Process and thread object as specified by the parameters.
!
! Arguments:
!
!   object_id                   Object ID of the resulting process object
!   container                   Object container for process object (ignored)
!   name                        Name of process object
!   acl                         ACL to place on process object
!   process_record              Attributes of the process being created
!   process_public_container    Process level public container to be transfered into the process
!                               level container directory for the process. If not present then
!                               container comes up empty.
!   process_private_container   Process level private container to be transfered into the process
!                               level container directory for the process. If not present then
!                               container comes up empty.
!   process_allocation_list     Objects to be allocated to the process object. If not present then
!                               no objects are allocated to the process
!   thread_record               Attributes of the thread being created
!   thread_allocation_list      Objects to be allocated to the thread object. If not present then
!                               no objects are allocated to the thread
!   thread_data_block           Arbitrary data block passed to initial thread. Pointer in TCR, if
!                               pointer is NIL, then no data block was passed
!   thread_immediate_parameter1 Immediate parameter passed to thread through TCR
!   thread_immediate_parameter2 Immediate parameter passed to thread through TCR
!   thread_status               Exit status object to be bound to the initial thread. If not present
!                               then the thread is created without an exit status object
!   process_status              TBS
!
! Return value:
!
!   TBS
!
!--
```

From the interface to *exec$create_process()*, it is clear that the *process_record* has an impact on the structure of the process being created. Example 1–16 illustrates the layout of the *process_record*.

Example 1–16: Process Record Structure

```
    !
    !The Process Record
    !
    exec$t_process_record: RECORD
        process_status_object: e$t_object_id;        ! Object ID of processes status object
        process_image_name: string(e$c_max_image_name); ! Image name for process being created
        !
        ! Per Process Resource limits. This value is used as the
        ! qual_limits value for the process object, and is deducted
        ! from the qual_usage field of the owning job object.
        ! A value of zero() in any one of fields means to use the
        ! corresponding value of the q_per_process_limit from the
        ! user structure
        !
        process_per_process_limits: e$t_quota_limits; ! Resource limits for this process
    END RECORD;
```

1.3.3.2.2 Process Deletion

The *exec$force_exit_process()* system service provides a mechanism for removing process objects from the system. The removal of a process has the following system-wide effects:

- All threads of the process are removed from the system.
- The amount of resources available to the process (*qual_limits–qual_usage*) is returned to the processes job object by decrementing *qual_usage* in the job object.
- If the process object is the last process owned by its job object, then the job object is removed from the system.

Example 1–17 illustrates the interface to *exec$force_exit_process()*.

Example 1–17: Process Object Deletion System Interface

```
PROCEDURE exec$force_exit_process (
            IN process_object_id: exec$t_object_id = DEFAULT;
            IN exit_status: exec$t_exit_status;
            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Causes the Process object specified by process_object_id to
!   be removed from the Mica system
!
! Arguments:
!
!   process_object_id   the process object to be removed. If not specified,
!                       then the current process is assumed
!   exit_status         the reason that the process is force-exiting
!
! Return value:
!
!   TBS
!
!--
```

1.3.3.2.3 Get/Set Process Information

The *exec$get_process_information()* and *exec$set_process_information()* system services provide a mechanism to obtain and modify attributes of the specified process object. Example 1–18 illustrates the interfaces to the process object get/set system services.

Example 1–18: Get/Set Process Information System Interface

```
PROCEDURE exec$get_process_information (
            IN process_object_id: exec$t_object_id = DEFAULT;
            IN process_get_items: exec$t_item_list;
            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Return information about the process object to the caller. The
!   information returned is item list driven
!
! Arguments:
!
!   process_object_id   if present, the object ID of process object that is to be inspected
!                       otherwise, the process object of the calling thread is assumed
!   process_get_items   item list identifying process object information to be extracted
!
! Return value:
!
!   TBS
!
!--

PROCEDURE exec$set_process_information (
            IN process_object_id: exec$t_object_id = DEFAULT;
            IN process_get_items: exec$t_item_list;
            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Modify information in the process object. The
!   information to be modified is item list driven
!
! Arguments:
!
!   process_object_id   if present, the object ID of process object that is to be modified
!                       otherwise, the process object of the calling thread is assumed
!   process_get_items   item list identifying process object information to be modified
!
! Return value:
!
!   TBS
!
!--
```

Only certain pieces of the process object may be inspected or modified. Table 1–3 illustrates the possible item codes and the information read or written by using the item code.

Table 1–3: Get/Set Process Information Item Codes

Item Code	Set Action	Get Action
e$i_job_id	error	return object ID of processes job object
e$i_parent_id	error	return object ID of processes parent process object
e$i_sub_process_count	error	return p_sub_process_count
e$i_sub_process_ids	error	return object ID's of sub_processes owned by process
e$i_thread_count	error	return p_thread_count
e$i_thread_ids	error	return object ID's of threads owned by process
e$i_usage_and_limits	error	return pcb_usage_and_limits
e$i_process_limits	replace qual_limits	return qual_limits
e$i_process_condir_id	error	return pcb_process_condir_id
e$i_accounting	error	return pcb_accounting
e$i_pcr_base	error	return pcb_pcr_base
e$i_allocation_list	error	return pcb_process_allocation_list

1.3.3.2.4 Process Control Operations

Two process control operations exist in the Mica system to coordinate the execution of all threads of a process. The first provides a primitive which can alter the execution flow of another process by causing a condition to be raised in the target process. The second provides primitives to *block* and *unblock* the execution of the target process. In this latter technique, there are two classes of control operations. One class allows user-mode activity within the process to continue via user-mode AST routines, while the other class disables user-mode activity.

1.3.3.2.4.1 Process Signaling

The *exec$signal_process()* system service provides a mechanism to alter the execution flow of all threads of the process by causing a *condition* to be raised in the threads context.

NOTE

Process signalling is implemented through user-mode ASTs; therefore, if ASTs are disabled then so are signals.

Example 1–19 illustrates the interface to *exec$signal_process()*.

Example 1–19: Signal Process System Interface

```
PROCEDURE exec$signal_process (
            IN object_id: exec$t_object_id;
            IN condition_value: exec$t_condition_value;
            IN argument: longword CONFORM = DEFAULT;
            ) RETURNS status;
            EXTERNAL;

!++
!
! Routine description:
!
!   Cause a condition of type condition_value to be raised in all threads owned by the process
!   specified by object_id. The condition handler is passed argument.
!
! Arguments:
!
!   object_id           the object_id of the process to be signaled
!   condition_value     A descriptor for the condition to be raised in all threads
!                       of the target process
!   argument            If present, the value that is passed to the condition handler
!
! Return value:
!
!   TBS
!
!--
```

1.3.3.2.4.2 Process Hibernate/Wake

The *exec$hibernate_process()* and *exec$wake_process()* provide a mechanism to block and unblock the execution flow of all threads within the target process. The block is implemented by causing all threads within the target process to issue a wait on the auto-clearing hibernate-event object within the thread control block. During the block, the only user-mode activity that is allowed is execution within user-mode AST routines; kernel-mode ASTs remain enabled. The unblock of the process is implemented by setting the auto-clearing hibernate event object within the thread control block of all threads of the target process. Example 1–20 illustrates the interfaces to *exec$hibernate_process()* and *exec$wake_process()*.

Example 1–20: Hibernate/Wake Process System Interface

```
!
! Hibernate Process
!
PROCEDURE exec$hibernate_process (
            IN object_id: exec$t_object_id;
            ) RETURNS status;
            EXTERNAL;

!++
!
! Routine description:
!
!   Cause all threads owned by the process specified by object_id to issue a wait on the
!   auto-clearing hibernate event object in their TCB. User mode AST's remain enabled
!
! Arguments:
!
!   object_id       object ID of target process
!
! Return value:
!
!   TBS
!
!--

!
! Wake Process
!
PROCEDURE exec$wake_process (
            IN object_id: exec$t_object_id;
            ) RETURNS status;
            EXTERNAL;
```

Example 1–20 Cont'd. on next page

Example 1–20 (Cont.): Hibernate/Wake Process System Interface

```
!++
!
! Routine description:
!
!   Cause all threads owned by the process specified by object_id to have their waits on the
!   auto-clearing hibernate event object in their TCB to be satisfied by setting the event.
!
! Arguments:
!
!   object_id       object ID of target process
!
! Return value:
!
!   TBS
!
!--
```

1.3.3.2.4.3 Process Suspend/Resume

The *exec$suspend_process()* and *exec$resume_process()* provide a mechanism to block and unblock the execution flow of all threads within the target process. The block is implemented by causing all threads within the target process to issue a wait on the auto-clearing suspend event object within the thread control block. During the block, no user-mode activity is possible; only kernel-mode normal and special AST routines may be executed. The unblock of the process is implemented by setting the auto-clearing suspend event object within the thread control block of all threads of the target process. Example 1–21 illustrates the interfaces to *exec$suspend_process()* and *exec$resume_process()*.

Example 1–21: Suspend/Resume Process System Interface

```
!
! Suspend Process
!
PROCEDURE exec$suspend_process (
            IN object_id: exec$t_object_id;
            ) RETURNS status;
            EXTERNAL;

!++
!
! Routine description:
!
!   Cause all threads owned by the process specified by object_id to issue a wait on the
!   auto-clearing suspend event object in their TCB. User mode AST's are disabled.
!
! Arguments:
!
!   object_id       object ID of target process
!
! Return value:
!
!   TBS
!
!--

!
! Resume Process
!
PROCEDURE exec$resume_process (
            IN object_id: exec$t_object_id;
            ) RETURNS status;
            EXTERNAL;
```

Example 1–21 Cont'd. on next page

Example 1–21 (Cont.): Suspend/Resume Process System Interface

```
!++
!
! Routine description:
!
!   Cause all threads owned by the process specified by object_id to have their waits on the
!   auto-clearing suspend event object in their TCB to be satisfied by setting the event.
!
! Arguments:
!
!   object_id       object ID of target process
!
! Return value:
!
!   TBS
!
!--
```

1.3.4 The Thread Object

The thread object appears at the lowest level of the UJPT hierarchy. Its primary function is to provide a thread of execution.

In addition, the thread object has the following functions:

- It is the schedulable entity in the Mica system.
- It maintains the processor state as it executes the program steps of an image.
- It is the consumer of resources. All accounting and resource limitation data structures reside in the thread's process object, with the thread's activity pooled to the process level.
- It can act as a focal point for synchronization.

The thread object is implemented as a system level object in the "THREAD$OBJECT_CONTAINER" object container.

1.3.4.1 Object Structure

The thread object maintains the state of the processor as it moves through the program steps of the program image mapped into its processes address space.

The thread object is split into a thread object body and a thread control block. The thread object body contains information necessary to maintain the thread's position within the UJPT hierarchy. The thread control block contains the information necessary to move the execution thread through the steps of the program image. Example 1–22 illustrates the thread object.

Example 1–22: Thread Object Structure

```
!
! Thread Object Body
!
e$t_thread: RECORD
    t_obj_id: e$t_object_id;                                ! Object ID of thread object
    t_process_pointer: POINTER e$t_process;                 ! Referenced pointer to owning process
    t_thread_flags: e$t_thread_flags;                       ! Thread Flags
    t_thread_queue: e$t_linked_list;                        ! List of processes threads
    t_tcb: e$t_thread_control_block;                        ! Thread Control Block
END RECORD;

!
! Thread Control Block
!
e$t_thread_control_block: RECORD
    tcb_previous_mode: e$t_processor_status;                ! saved processor status
    tcb_thread_context: e$t_thread_context;                 ! Processor State of Thread
    tcb_kernel_thread_block: k$dispatcher_object(thread);   ! Kernel Thread Block
    tcb_hibernate_event: k$dispatcher_object(event);        ! auto-clearing hibernate event
    tcb_suspend_event: k$dispatcher_object(event);          ! auto-clearing suspend event
    tcb_pcb_pointer: POINTER e$t_process_control_block;     ! Pointer to PCB
    tcb_tcr_base: POINTER e$t_thread_control_region;        ! Pointer to TCR
    tcb_exit_status_id: e$t_object_id;                      ! Exit Status Object ID for Thread
    tcb_exit_status_ptr: POINTER e$t_exit_status_body;      ! Exit Status for Thread
    tcb_exit_status_value: e$t_exit_status;                 ! Exit Status
    tcb_security_profile: e$t_security_profile;             ! The threads security profile
    tcb_thread_allocation_list: e$t_allocation_list;        ! Objects allocated to the thread object
    !
    ! Memory Management Events
    !
    tcb_initial_page_event: k$dispatcher_object(event);     ! Memory Management
    tcb_secondary_page_event: k$dispatcher_object(event);   ! Memory Management
    tcb_current_page_event: integer;                        ! Memory Management
    !
    ! I/O
    !
    tcb_io_synchronization_event: k$dispatcher_object(event);   ! I/O synchronization event
    tcb_irp_list_head: e$t_linked_list;                         ! I/O Request Packet List Head
    tcb_cancel_io: boolean;                                     ! Cancel io by thread in progress
    tcb_cancel_event: k$dispatcher_object(event);               ! Cancel io synchronization

END RECORD;

!
! Thread Context
!
e$t_thread_context: RECORD
    tc_priviledged_context_block: k$hwpcb;                              ! Hardware Privileged Context Block
    tc_general_purpose_registers: POINTER e$t_general_purpose_registers;! Scalar Register Set
    tc_vector_registers: POINTER e$t_vector_registers;                  ! Vector Register Set
END RECORD;

!
! Thread Control Region
!
! The thread control region appears in the processes address space as user read only/ system
! read write
!
e$t_thread_control_region: RECORD
    tcr_object_id: e$t_object_id;                           ! Object ID of this thread
    tcr_pcr_pointer: POINTER e$t_process_control_region;    ! Pointer to process control region
    tcr_start_address: e$t_thread_entry_point;              ! initial start address of thread
    tcr_initial_sp: e$t_scalar_register;                    ! Initial Value of Stack Pointer
    tcr_stack_limit: e$t_scalar_register;                   ! Primary Stack Limit
    tcr_stack_base: e$t_scalar_register;                    ! Primary Stack Base
    tcr_condition_initial_sp: e$t_scalar_register;          ! Initial Value of Condition Stack Ptr
    tcr_condition_stack_limit: e$t_scalar_register;         ! Condition Stack Limit
    tcr_condition_stack_base: e$t_scalar_register;          ! Condition Stack Base
    tcr_exit_handlers: e$t_exit_handlers;                   ! Thread exit handlers
    tcr_vectored_handlers: e$t_vectored_handlers;           ! Entry descriptors for vectored
                                                            ! condition handlers
    !
    ! Initial Thread Parameters
    !
    tcr_block_data: POINTER anytype;                        ! Initial thread data block or NIL
    tcr_block_data_length: integer;                         ! Byte length of data block rounded to quadword
    tcr_parameter1: e$t_thread_parameter;                   ! Immediate parameter / or zero()
```

Example 1–22 Cont'd. on next page

Example 1–22 (Cont.): Thread Object Structure

```
        tcr_parameter2: e$t_thread_parameter;                    ! Immediate parameter / or zero()
    END RECORD;

    !
    ! Immediate Parameter
    !
    e$t_thread_parameter: e$t_register;      ! Same size as a machine register

    !
    ! Thread Entry Point
    !
    e$t_thread_entry_point: PROCEDURE();
```

1.3.4.2 Functional Interface

The Mica executive provides entry points capable of creating, deleting, and controlling thread objects, in addition to setting and extracting various attributes of a thread object.

Thread object control services are Suspend/Resume thread, Hibernate/Wake thread, and Signal thread.

As part of Thread object creation, all of the necessary support data structures are created including the read-only thread control region (TCR), the read/write thread environment block (TEB), and user and kernel stacks. The TCR is part of the process's user-mode read-only address space. The Mica executive places information in the TCR so that the thread can read it without entering the system. The TEB is part of the user-mode thread architecture. The MICA executive initializes the TEB to point to the TCR.

1.3.4.2.1 Thread Creation

The *exec$create_thread()* system service extends an existing UJPT hierarchy by causing the creation of a thread object. The newly created thread object begins execution within the address space of its process at a start address passed to the system interface. Example 1–23 illustrates the interface to *exec$create_thread()*.

Example 1–23: Thread Object Creation System Interface

```
PROCEDURE exec$create_thread (
            OUT object_id: exec$t_object_id;
            IN  container: exec$t_object_id = DEFAULT;
            IN  name: exec$t_object_name = DEFAULT;
            IN  acl: exec$t_acl = DEFAULT;

            IN thread_procedure: exec$t_thread_entry_point;
            IN thread_record: exec$t_thread_record = DEFAULT;
            IN thread_allocation_list: exec$t_allocation_list = DEFAULT;
            IN thread_data_block: quadword_data(*) CONFORM OPTIONAL;
            IN thread_immediate_parameter1: exec$t_thread_parameter = DEFAULT;
            IN thread_immediate_parameter2: exec$t_thread_parameter = DEFAULT;
            IN thread_status: exec$t_object_id = DEFAULT;
            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Create a thread object as specified by the parameters.
!
! Arguments:
!
!   object_id                   Object ID of the resulting process object
!   container                   Object container for thread object (ignored)
!   name                        Name of thread object
!   acl                         ACL to place on thread object
!   thread_record               Attributes of the thread being created
!   thread_allocation_list      Objects to be allocated to the thread object. If not present then
!                               no objects are allocated to the thread
!   thread_data_block           Arbitrary data block passed to initial thread. Pointer in TCR, if
!                               pointer is NIL, then no data block was passed
```

Example 1–23 Cont'd. on next page

Example 1–23 (Cont.): Thread Object Creation System Interface

```
!   thread_immediate_parameter1 Immediate parameter passed to thread through TCR
!   thread_immediate_parameter2 Immediate parameter passed to thread through TCR
!   thread_procedure            pointer to thread entry point entry descriptor
!   thread_status               Exit status object to be bound to the thread. If not present
!                               then the thread is created without an exit status object
!
! Return value:
!
!   TBS
!
!--
```

From the interface to *exec$create_thread()*, it is clear that the *thread_record* can have an impact on the structure of the thread being created. Example 1–24 illustrates the layout of the *thread_record*.

Example 1–24: Thread Record Structure

```
    !
    ! The thread record
    !
    e$type_thread_record: RECORD
        thread_stack_size: integer;                ! If 0 then system wide default
        thread_priority: k$combined_priority;      ! initial thread priority if all 0 then default
        thread_affinity: k$affinity;               ! processor affinity If all 0 then all processors
    END RECORD;
```

1.3.4.2.2 Thread Deletion

Thread deletion is the action which causes the removal of a thread object. The Mica system provides two mechanisms for deleting thread objects. The first mechanism, *simple exit*, will in some cases not cause the thread object to be removed; however, it is the normal path for thread exit when a thread wants to exit. The second mechanism, *forced exit*, will cause the thread object to be removed unconditionally. The forced exit path occurs when any thread wants the specified thread to exit.

The deletion of a thread object causes the thread's exit handlers to execute. In the simple exit case, exit handlers may run indefinitely, possibly never completing; thus, thread object may not occur. In the forced exit case, the thread's exit handlers are executed with a CPU time limit. If a time limit is exceeded, the next handler is executed. This technique guarantees that all exit handlers will be invoked and that afterwards thread object deletion will proceed. The *exec$exit_thread()* system interface provides the simple exit functionality. The *exec$force_exit_thread()* system service provides the forced-exit functionality.

When the last thread of a process is deleted, the process object is removed from the system.

Example 1–25 illustrates the interfaces to *exec$exit_thread()*, and *exec$force_exit_thread()*.

Example 1–25: Thread Object Deletion System Interfaces

```
!
! Thread Exit System Service
!
PROCEDURE exec$exit_thread (
            IN exit_status: exec$t_exit_status;
            );

!++
!
! Routine description:
!
!   Cause the deletion of the calling thread object. Place
!   thread_status in the threads tcb at tcb_exit_status_value
!
! Arguments:
!
!   thread_status       the exit status of the thread
!
! Return value:
!
!   none
!
!--

!
! Thread Force Exit System Service
!
PROCEDURE exec$force_exit_thread (
            IN object_id: exec$t_object_id = DEFAULT;
            IN exit_status: exec$t_exit_status;
            ) RETURNS status;

!++
!
! Routine description:
!
!   Cause the deletion of the thread object specified by object_id
!
! Arguments:
!
!   object_id       the object ID of the thread object being deleted. If not specified,
!                   then the calling thread is assumed
!   exit_status     the reason that the thread is force-exiting
!
! Return value:
!
!   TBS
!
!--
```

1.3.4.2.3 Get/Set Thread Information

The *exec$get_thread_information()* and *exec$set_thread_information()* system services provide a mechanism to obtain and modify attributes of the specified thread object. Example 1–26 illustrates the interfaces to the thread object get/set system services.

Example 1–26: Get/Set Thread Information System Interface

```
PROCEDURE exec$get_thread_information (
            IN thread_object_id: exec$t_object_id = DEFAULT;
            IN thread_get_items: exec$t_item_list;
            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Return information about the thread object to the caller. The
!   information returned is item list driven
!
! Arguments:
!
!   thread_object_id   if present, the object id of thread object that is to be inspected
!                      otherwise, the calling thread is assumed
!   thread_get_items   item list identifying thread object information to be extracted
!
! Return value:
!
!   TBS
!
!--

PROCEDURE exec$set_thread_information (
            IN thread_object_id: exec$t_object_id = DEFAULT;
            IN thread_get_items: exec$t_item_list;
            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Modify information in the thread object. The
!   information to be modified is item list driven
!
! Arguments:
!
!   thread_object_id   if present, the object ID of thread object that is to be modified
!                      otherwise, the calling thread is assumed
!   thread_get_items   item list identifying thread object information to be modified
!
! Return value:
!
!   TBS
!
!--
```

Only certain pieces of the thread object may be inspected or modified. Table 1–4 illustrates the possible item codes and the information read or written by using the item code.

Table 1–4: Get/Set Thread Information Item Codes

Item Code	Set Action	Get Action
e$i_process_id	error	return object ID of threads process object
e$i_tcr_base	error	return tcb_tcr_base
e$i_tcr_start_address	set tcr_start_address	error
e$i_allocation_list	error	return tcb_thread_allocation_list

1.3.4.2.4 Thread Control Operations

Two thread control operations exist in the Mica system to coordinate the execution of threads. The first provides a primitive which can alter the execution flow of another thread by causing a condition to be raised in the target thread. The second provides primitives to block and unblock the execution of the target thread. In this latter technique, there are two classes of control operations. One class allows user-mode activity within the thread to continue via user-mode AST routines, while the other class disables user-mode activity.

1.3.4.2.4.1 Thread Signaling

The *exec$signal_thread()* system service provides a mechanism to alter the execution flow of a thread by causing a condition to be raised in the context of the target thread.

NOTE

The thread signalling mechanism is implemented through user-mode ASTs; therefore, if ASTs are disabled, then so are signals.

Example 1–27 illustrates the interface to *exec$signal_thread()*.

Example 1–27: Signal Thread System Interface

```
PROCEDURE exec$signal_thread (
            IN object_id: exec$t_object_id;
            IN condition_value: exec$t_condition_value;
            IN argument: longword CONFORM = DEFAULT;
            ) RETURNS status;
            EXTERNAL;

!++
!
! Routine description:
!
!   Cause a condition of type condition_value to be raised in the thread
!   specified by object_id. The condition handler is passed argument.
!
! Arguments:
!
!   object_id           the object_id of the thread to be signaled
!   condition_value     A descriptor for the condition to be raised in the target thread
!   argument            If present, the value that is passed to the condition handler
!
! Return value:
!
!   TBS
!
!--
```

1.3.4.2.4.2 Thread Hibernate/Wake

The *exec$hibernate_thread()* and *exec$wake_thread()* provide a mechanism to block and unblock the execution flow of a thread. The block is implemented by causing the thread to issue a wait on the auto-clearing hibernate event object within the thread control block. During the block, the only user-mode activity that is allowed is execution within user-mode AST routines. The unblock of the thread is implemented by setting the auto-clearing hibernate event object within the thread control block. Example 1–28 illustrates the interfaces to *exec$hibernate_thread()* and *exec$wake_thread()*.

Example 1–28: Hibernate/Wake Thread System Interface

```
!
! Hibernate Thread
!
PROCEDURE exec$hibernate_thread (
            IN object_id: exec$t_object_id;
            ) RETURNS status;
            EXTERNAL;

!++
!
! Routine description:
!
!   Cause the thread specified by object_id to issue a wait on the
!   auto-clearing hibernate event object in the TCB. User mode AST's remain enabled
!
! Arguments:
!
!   object_id       object ID of target thread
!
! Return value:
!
!   TBS
!
!--

!
! Wake Thread
!
PROCEDURE exec$wake_thread (
            IN object_id: exec$t_object_id;
            ) RETURNS status;
            EXTERNAL;

!++
!
! Routine description:
!
!   Cause the thread specified by object_id to have the wait on the
!   auto-clearing hibernate event object in the TCB to be satisfied by setting the event.
!
! Arguments:
!
!   object_id       object ID of target thread
!
! Return value:
!
!   TBS
!
!--
```

1.3.4.2.4.3 Thread Suspend/Resume

The *exec$suspend_thread()* and *exec$resume_thread()* provide a mechanism to block and unblock the execution flow of the target thread. The block is implemented by causing the thread to issue a wait on the auto-clearing suspend-event object within the thread control block. During the block, no user-mode activity is possible. Only kernel-mode normal and special AST routines may be executed. The unblock of the thread is implemented by setting the auto-clearing suspend-event object within the thread control block. Example 1–29 illustrates the interfaces to *exec$suspend_thread()* and *exec$resume_thread()*.

Example 1–29: Suspend/Resume Thread System Interface

```
!
! Suspend Thread
!
PROCEDURE exec$suspend_thread (
            IN object_id: exec$t_object_id;
            ) RETURNS status;
            EXTERNAL;

!++
!
! Routine description:
!
!   Cause the thread specified by object_id to issue a wait on the
!   auto-clearing suspend event object in the TCB. User mode AST's are disabled.
!
! Arguments:
!
!   object_id       object ID of target thread
!
! Return value:
!
!   TBS
!
!--

!
! Resume Thread
!
PROCEDURE exec$resume_thread (
            IN object_id: exec$t_object_id;
            ) RETURNS status;
            EXTERNAL;

!++
!
! Routine description:
!
!   Cause the thread specified by object_id to have the wait on the
!   auto-clearing suspend event object in the TCB to be satisfied by setting the event.
!
! Arguments:
!
!   object_id       object ID of target thread
!
! Return value:
!
!   TBS
!
!--
```

1.3.4.2.4.4 Hibernate and Suspend Comparison

Both the *exec$hibernate_thread()* system service, and the *exec$suspend_thread()* system service block the execution of the specified thread. The difference between these two types of blocked states is the ability of the blocked thread to receive and execute in the context of user-mode ASTs. Threads that are blocked due to the *exec$hibernate_thread()* system service are able to receive and execute in the context of user-mode ASTs; threads that are blocked due to the *exec$suspend_thread()* system service are not.

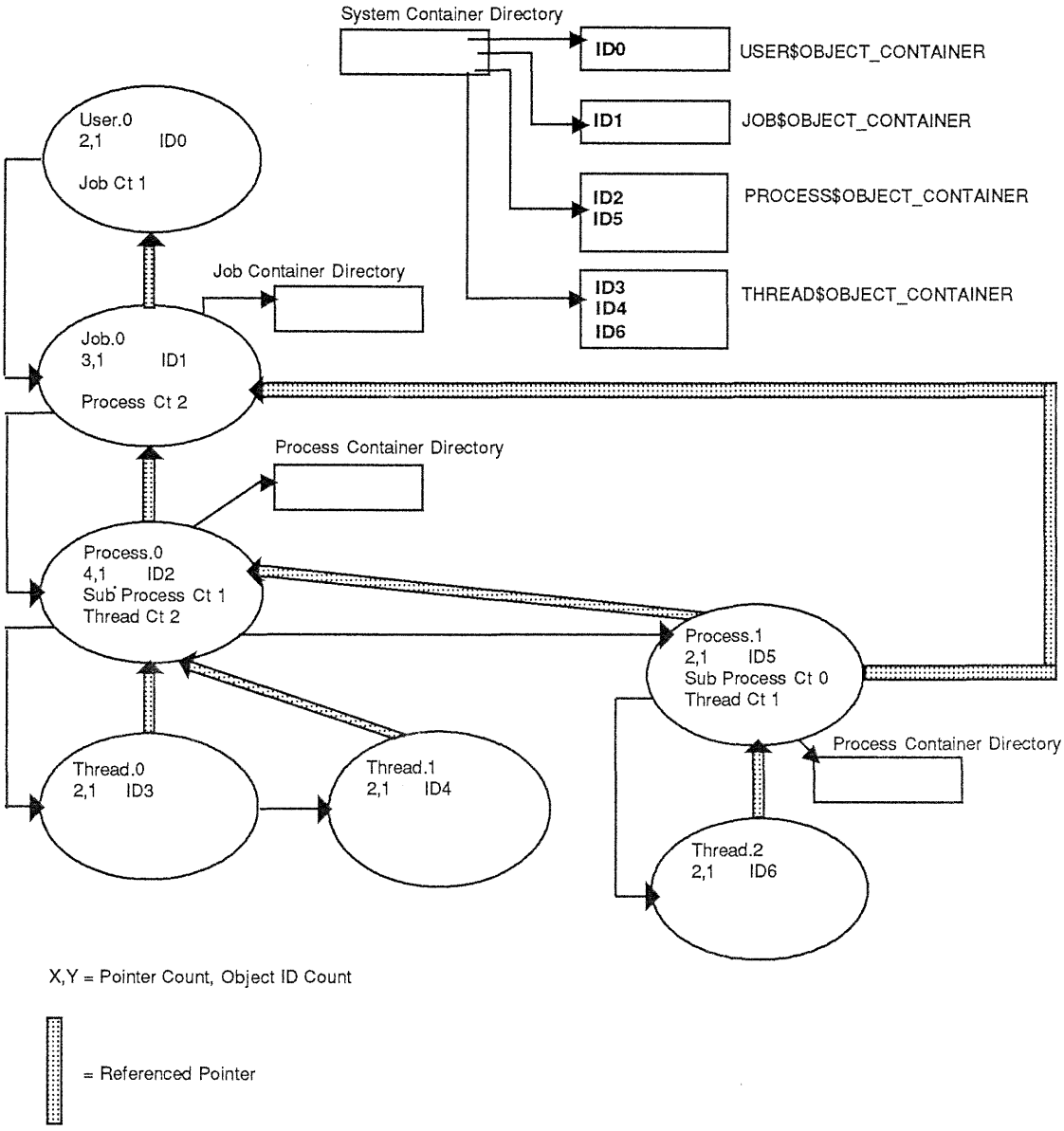
1.4 UJPT Object Linkages

The UJPT hierarchy is bound together through the existence of object IDs and referenced pointers. The following section describes the implementation of the object linkages, the steps of hierarchy creation, and the actions which lead to the collapse of a UJPT hierarchy. This section does not describe process or thread creation in terms of address space creation or the intricate details of kernel, memory management, or object architecture interactions.

1.4.1 Linkage Structure

The UJPT object linkage structure requires that objects in lower levels of the hierarchy point to the object immediately above them using a referenced pointer. The reference pointer guarantees the existence of the higher-level object for the life of the lower-level object. Figure 1–1 illustrates a complex UJPT hierarchy consisting of a user object, a job object, and a process object consisting of two immediate threads and a sub-process object with a single thread.

Figure 1–1: Complex UJPT Hierarchy



ZS-24347-87

1.4.2 Hierarchy Creation

The creation of a UJPT hierarchy is triggered by the *exec$create_user()* system service. At this time, a hierarchy is either created or extended, depending on the existence of a user object representing the Mica user specified in the *user_record.user_username* field of the *user_record* parameter.

The following steps occur during the creation of a UJPT hierarchy.

1. Determine if a user object exists for *user_record.user_username*. If the object exists, then obtain a referenced pointer to the user object. Otherwise, create the user object in the system container directory, initialize the user object with the information from the *user_record* and then obtain a referenced pointer to the user object.
2. Create the job object in the system container directory and obtain a referenced pointer to the job object. Initialize the job object according to the following tasks.
   - Set *j_obj_id* equal to the object ID of the job being created.
   - Set *j_user_pointer* to the referenced pointer of the proper user object.
   - Link the job object to the user object's *u_job_queue_hd*, and initialize the *j_process...* fields of the job object.
   - Create the job-level container directory, and populate it with the *job_initial_container* parameter.
3. Create the process object in the system container directory and obtain a referenced pointer to the process object. Initialize the process object according to the following tasks.
   - Set *p_obj_id* equal to the object ID of the process being created.
   - Set *p_job_pointer* to the referenced pointer of the proper job object.
   - Link the process object to the job object's *j_process_queue_hd*.
   - Initialize the *p_thread...* fields and *p_sub_process...* fields of the job object.
   - Create the process level container directory, and populate it with the *process_public_container* parameter and the *process_private_container* parameter.
4. Create the thread object in the system container directory.
5. Obtain a referenced pointer to the thread object.
6. Initialize the thread object such that *t_obj_id* contains the object ID of the thread, and *t_process_pointer* contains the referenced pointer to the proper process object.
7. Link the thread object to the process object's p_thread_queue_hd.

1.4.3 Hierarchy Collapse/Deletion

The collapse of a UJPT hierarchy can be triggered by force-exiting any component of a hierarchy. The ultimate collapse is always the result of a thread's exit, whether it be a forced exit or a voluntary exit.

The forced exit of a component in the UJPT hierarchy eventually causes all threads beneath that object to exit. The following actions occur during a thread exit.

- If the exiting thread is the last thread in its process, then cause the process to exit by removing its object ID.
- If the exiting process has any sub-processes, then cause its sub-processes to exit by force-exiting them.
- If the exiting process is the last process in its job, then cause the job to exit by removing its object ID.
- If the exiting job is the last job in its user, then cause the user to exit by removing its object ID.

1.4.3.1 Force-Exit Routines

Each component of the hierarchy provides a force-exit interface as part of its primitive object service routines. The basic action performed in these routines is the forced exit of the object's sub-objects.

1.4.3.1.1 User-Object Force-Exit Routine

The user-object force-exit routine is responsible for causing the forced exit of all of its job objects. This is implemented by setting its force-exit in-progress flag, and looping over the linked list of its job objects headed by *u_job_queue_hd* and a force-exit of that job via *e$force_exit_job()*.

1.4.3.1.2 Job-Object Force-Exit Routine

The job object force-exit routine is responsible for causing the forced exit of all of its process objects. This is implemented by setting its force-exit in progress flag, looping over the linked list of its process objects headed by *j_process_queue_hd*, and causing a forced exit of that process via *e$force_exit_process()*.

1.4.3.1.3 Process-Object Force-Exit Routine

The process object force-exit routine is responsible for causing the removal of all of its thread objects and sub-processes represented as process objects. This is implemented by setting its force-exit in progress flag, and looping over the linked list of its thread objects headed by *p_thread_queue_hd* and causing a force-exit of that thread via *e$force_exit_thread()*. Then the routine loops over the linked list of sub-processes headed by *p_sub_process_queue_hd* and causes a forced exit of that process via *e$force_exit_process()*.

1.4.3.1.4 Thread Object Force Exit Routine

The routine occurs in two phases. The first phase is to cleanly enter the exiting thread's context to begin the thread exit. The second phase is to complete the exit of the thread by calling *exec$exit_thread()*, an action which starts the second phase of hierarchy collapse and finally brings the "exiting" thread out of the system. Before starting the forced-exit processing, the force-exit in-progress flag is set in the thread object.

During a thread forced-exit, there is a moment when control is returned to the original caller of *exec$exit_thread()* even though the thread to be exited is still part of the system. The exit is considered complete with respect to the caller after the system has delivered an AST to the exiting thread that will cause the thread itself to exit. The exit is complete with respect to the exiting thread once the thread has issued its call to *k$terminate_thread()*;

1.4.3.1.4.1 Thread Context Entry

To force-exit a thread, that thread's context must be entered in a controlled manner in a "trusted" user-mode routine. This is achieved by delivering a user-mode AST to the thread. The target procedure of the AST is a routine that is part of the Mica executive but is executed in user mode. The AST target procedure is the function *e$in_context_force_exit()*. The purpose of this function is to bring the thread into a "clean" state so that it can complete its exit. The following steps occur in *e$in_context_force_exit()*:

- The thread issues an *e$unwind()* specifying an exit unwind.
- Once the unwind has completed, the thread issues a call to *exec$exit_thread()*.

1.4.3.1.4.2 Thread Exit

The second phase of thread exit processing begins at the entry point *exec$exit_thread()*. The purpose of this function is to execute all of the exit handlers for the thread and, when completed, to bring the thread object out of the system. The following steps occur in *exec$exit_thread()*:

- Dequeue the first exit handler from the thread control region.
- If the thread is in the force-exit in progress state, then establish a CPU time quota for the thread.

NOTE

The thread-exit CPU time quota is on accumulated user-mode CPU time. It is not an elapsed time limit.

- If the CPU time quota expires, then deliver a user-mode AST to the thread. The target procedure of the AST is executive code that runs in "trusted" user-mode at the *e$exit_handler_quota_expire()*. This entry point causes the termination of the current exit handler and begins the next by calling *exec$exit_thread()*.
- Vector to the exit handler in user-mode.
- If no more exit handlers for the thread exist, then remove the object ID of the thread by calling *e$remove_object_id()*, passing it the object ID of the thread stored in *t_obj_id* in the thread object body. This action begins the second phase of hierarchy collapse by causing the execution of the affected object's remove routines. If there are more exit handlers, then repeat the above steps.
- After completion of *e$remove_object_id()*, the thread removes itself from the system by calling *k$terminate_thread()*. This action begins the third phase of hierarchy collapse by causing the execution of the affected object's delete routines.

1.4.3.2 Object Remove Routines

The object remove routines are called when the *objhdr$object_id_count* within the object header decrements to zero. This occurs during the second phase of hierarchy collapse as a result of a call to *e$remove_object_id()* for the "exiting" object. Object remove routines are always executed in the context of the object being removed.

NOTE

In order to ensure the above context restrictions, objects within the UJPT hierarchy may not have alias object IDs, and their ACLs are such that only the function *exec$exit_thread()* is capable of removing their object IDs.

Assuming the UJPT hierarchy from Figure 1–1, the following legal contexts exist to execute the remove routines for the hierarchy.

- Thread.0 will execute its remove routine in the context of thread.0.
- Thread.1 will execute its remove routine in the context of thread.1.
- Thread.2 will execute its remove routine in the context of thread.2.
- Process.0 could execute its remove routine in either the context of thread.0, or thread.1. The context would be determined by the context of the last thread to begin the second phase of exit.
- Process.1 will execute its remove routine in the context of thread.2.
- Job.0 will execute its remove routine in the context that was used to execute process.0's remove routine.
- User.0 will execute its remove routine in the context that was used to execute job.0's remove routine.

1.4.3.2.1 User-Object Remove Routine

The user-object remove routine performs no actions related to hierarchy collapse.

1.4.3.2.2 Job Object Remove Routine

The job-object remove routine is responsible for breaking the link between itself and its user object. If the job object is the last object of its user object then it must guarantee the removal of the user object. This occurs as follows:

- The job object is de-linked from the *u_job_queue_hd* in the user object pointed to by *j_user_pointer*.
- If the *u_job_count* field is decremented to zero by this action, then the user object is removed by calling *e$remove_object_id()* specifying the object ID of the user object (*u_obj_id*) stored in the user object body.

1.4.3.2.3 Process Object Remove Routine

The process object remove routine is responsible for breaking the link between itself and its job object, and if the process is a sub-process, it must break the link between itself and its *parent process* i.e. the process above it. Two different paths are followed during the process remove routine. The following occurs in the remove routine for a process without a parent.

- The process object is de-linked from the *j_process_queue_hd* in the job object pointed to by *p_job_pointer*.
- If the *j_process_count* field is decremented to zero by this action, then the job object is removed by calling *e$remove_object_id()* specifying the object ID of the job object (*j_obj_id*) stored in the job object body.

The remove routine for a sub-process i.e. a process with a parent simply de-links itself from the *p_sub_process_queue_hd* in the process object pointed to by *p_parent_pointer*.

1.4.3.2.4 Thread Object Remove Routine

The thread object remove routine is responsible for breaking the link between itself, and its process object. If the thread object is the last object of its process object then it must guarantee the removal of the process object. This occurs as follows:

- The thread object is de-linked from the *p_thread_queue_hd* in the process object pointed to by *t_process_pointer*.
- If the *p_thread_count* field is decremented to zero by this action, then the process object is removed by calling *e$remove_object_id()* specifying the object ID of the process object (*p_obj_id*) stored in the process object body.

1.4.3.3 Object Delete Routines

The object delete routines are called as a result of the *objhdr$pointer_count* field decrementing to zero. This occurs during the third phase of hierarchy collapse as a result of the call to *k$terminate_thread()* in *exec$exit_thread()*.

The function of *k$terminate_thread()* is to remove the thread from the system. This is accomplished by queuing a pointer to the thread object to a queue served by a system thread running *e$terminate_thread()*. This thread is responsible for dereferencing the thread object which begins the third phase of hierarchy collapse.

Object delete routines always execute in the context of the system thread running *e$terminate_thread()*.

NOTE

At the time that *k$terminate_thread()* is called, the thread object's *objhdr$pointer_count* is 1, and the *objhdr$object_id_count* is 0.

1.4.3.3.1 User-Object Delete Routine

The user-object delete routine performs no actions related to hierarchy collapse.

1.4.3.3.2 Job-Object Delete Routine

The job-object delete routine simply dereferences its user object by calling *e$dereference_object()* passing it the referenced pointer to the user object stored in *j_user_pointer*.

1.4.3.3.3 Process-Object Delete Routine

The process-object delete routine performs the following actions:

- If the process has a parent process, its parent process object is dereferenced by calling *e$dereference_object()*, passing it the referenced pointer to the parent process object stored in *p_parent_pointer*.
- The job object is dereferenced by calling *e$dereference_object()*, passing it the referenced pointer to the job object stored in *p_job_pointer*.

1.4.3.3.4 Thread-Object Delete Routine

The thread-object delete routine simply dereferences its process object by calling *e$dereference_object()*, passing it the referenced pointer to the process object stored in *t_process_pointer*.

1.5 Address Space and Execution Threads

Execution threads exist within a context which includes an address space and processor state. The creation and deletion of execution threads involves heavy interactions with the Mica kernel and memory management subsystems. This section describes execution thread creation and deletion in terms of its interactions with the Mica kernel, executive, and memory management subsystems. Interactions with the object architecture are not discussed.

1.5.1 Creation

The creation of an execution thread has two distinct paths.

The first path occurs when an execution thread is being created, an action which requires the creation of both an address space and a processor state. This path is a result of an *exec$create_user()*, an *exec$create_job()*, or *exec$create_process()* system service. This path is known as *initial thread creation*.

The second path occurs when an execution thread is being created within an existing address space. The only context that needs to be established is the processor state. This path occurs as a result of an *exec$create_thread()* system service and is known as *subsequent thread creation*.

1.5.1.1 Initial Thread Creation

During initial thread creation, the following actions occur.

- An address space must be created and initialized.
- A transition to the new thread's partial context must occur.
- Both thread- and process-control region address space must be created and initialized.
- The program image for the new process must be mapped into the process address space.
- The thread must begin execution at the program image starting address

1.5.1.1.1 Address Space Creation

The creation of a Mica address space occurs as a result of a call to *e$create_process_address_space()*. Example 1–30 illustrates the interface to this function.

Example 1–30: Address Space Creation

```
PROCEDURE e$create_process_address_space (
            IN process_control_pte : POINTER mm$pte;
            OUT ptbr : integer;                              !page table base register
            OUT kernel_stack_pointer : POINTER anytype;
            );
            EXTERNAL;
!++
!
! Routine description:
!
!   This routine creates the foundation of a process address space.
!   Pages are allocated for the segment 1 page table, the segment 2
!   page table for the control region, the kernel stack and the
!   working set list.
!
!   NO ADDRESSES WITHIN THE ADDRESS SPACE ARE VALID, THIS INCLUDES THE
!   KERNEL STACK POINTER WHICH IS RETURNED.
!
!   Once an address space foundation has been created, k$initialize_thread
!   and k$ready_thread are invoked to create the initial thread running
!   within this new address space.
!
! Arguments:
!
!   IN  process_control_pte - pointer to the process_control_pte in the process
!                             control block.  Upon return the prototype PTE
!                             referred to by process_control_pte will contain
!                             the prototype PTE for the segment 1 page table
!                             page.  The PFN database PTP element will contain
!                             this address (process_control_pte) so it must
!                             be in non paged system space.
!
!   OUT ptbr - the value to be used for the page table base register
!
!   OUT kernel_stack_pointer - the value to be used for the kernel stack pointer
!
! Return value:
!
!   none.
!
!--
```

The created address space is only valid in the context of the new thread. The next phase of address space creation occurs in the context of the new thread.

1.5.1.1.2 Execution Thread Creation

Once the address space for the initial thread is created, the thread must be started in its context. This occurs by calls to the the kernel interfaces *k$initialize_thread()*, and *k$ready_thread()*. After the completion of *k$ready_thread()*, the new thread is eligible to run in its own context, and the calling thread considers the thread creation complete.

The new thread begins execution at *e$initial_thread_startup()*. Example 1–31 illustrates the entry point for all initial threads.

Example 1–31: Initial Thread Entry Point

```
PROCEDURE e$initial_thread_startup ();
            EXTERNAL;
!++
!
! Routine description:
!
!   The entry point for all initial threads. This routine is responsible for completing an
!   execution thread which involves
!
!       o Initializing the threads address space
!       o creating and initializing the control region memory pool
!       o initializing the pcr and tcr
!       o mapping the program image into the new address space
!       o starting the thread at the image entry point
!
! Arguments:
!
!   none
!
! Return value:
!
!   none
!
!--
```

1.5.1.1.2.1 Address Space Initialization

The first action performed by *e$initial_thread_startup()* is the initialization of the process address space. This action makes it possible for the thread to begin taking page faults within its address space. Address space initialization is accomplished by calling *e$initialize_address_space()*. Example 1–32 illustrates the interface to *e$initialize_address_space()*.

Example 1–32: Address Space Initialization

```
PROCEDURE e$initialize_address_space (
            IN working_set_extent: e$t_resource_counter;
            IN working_set_quota: e$t_resource_counter;
            );
            EXTERNAL;
!++
!
! Routine description:
!
!   This routine initializes an address space which was previously
!   created by e$create_process_address_space.
!
!   It must now be running in the non paged portion of the exec
!   with the newly created address space mapped.  No page faults
!   may be taken until this routine has been invoked.
!
!   This routine will create the working set list, mark the control
!   region, kernel stack, and working set list as locked in the
!   working set.
!
!   The arguments are derived from the process control block
!   qnl_working_set_limit and qnl_working_set_extent fields of
!   pcb_usage_and_limits structure.
!
! Arguments:
!
!   IN working_set_extent - maximum size of the working set.
!   IN working_set_quota - current size of the working set.
```

Example 1–32 Cont'd. on next page

Example 1–32 (Cont.): Address Space Initialization

```
!
! Return value:
!
!   none - it had better work.
!
!--
```

1.5.1.1.2.2 Control Region Initialization

Once the process address space has been initialized, the control region memory pool must be initialized. The control region is at a fixed virtual address within the process's address space and is user read-only, kernel read/write. The standard Mica pool header for pool type *e$k_pool_control* is initialized and fed by calling *e$initialize_control_region()*.

Once the control region pool has been created, a process control region and thread control region are allocated from the control region pool. The control regions are then initialized by copying dummy control regions allocated from non-paged pool to the real control regions. Finally, the thread control region is linked to its thread control block, and the process control region is linked to the process control block and the thread control region.

1.5.1.1.2.3 Program Image Mapping

The program image to be executed must be mapped into the newly created process address space. This occurs by transitioning into user-mode at the entry point *e$program_image_startup()*.

The function of *e$program_image_startup()* is to map the program image and cause it to begin execution at the image start address. To map the image, the function *exec$map_image()* is called passing it the image name stored in its process control region. Once mapped, the thread startup address stored in the thread control region is set using *exec$set_thread_information()*. The image is then called. The initial thread parameters may be found in the thread control region.

1.5.1.2 Subsequent Thread Creation

During subsequent thread creation the following must occur.

- Creation of a kernel mode and user mode stack for the thread.
- Creation and initialization of the thread control region.
- Transition to the new thread's context at the proper start address.

1.5.1.2.1 Thread Stack Creation

The creation of a kernel and user mode stack for the new thread occurs as a result of calling *e$create_thread_stacks()*.

1.5.1.2.2 Control Region Initialization

A thread control region is allocated for the new thread from the control region pool of the calling thread's process. The thread control region is then initialized with the values obtained from the *exec$create_thread()* parameters. The thread control region is then linked to the thread control block and is set to point to the proper process control region.

1.5.1.2.3 Transition to new Thread

The final steps in subsequent thread creation require that the thread be started in its context. This is achieved by making calls to *k$initialize_thread()*, and *k$ready_thread()*. After the completion of the call to *k$ready_thread()*, the new thread is eligible to be run in its own context, and the calling thread assumes that the thread creation has completed.

The new thread begins execution at *e$subsequent_thread_startup()*. This entry point simply forces a transition to user-mode at the address specified by the thread control blocks *tcr_start_address_field*.

1.5.2 Deletion

Address space and execution thread deletion happen as part of the process object and thread object delete routines.

1.5.2.1 Execution Thread Deletion

Execution thread deletion happens in two phases. The first phase is executed within the context of the terminating thread and is responsible for thread resource cleanup. The second phase occurs outside the context of the calling thread and is responsible for the deletion of the kernel stack of the terminating thread.

NOTE

The context restrictions are enforced by the lack of alias object IDs on components of the UJPT hierarchy, and through restrictions on the removal of objects within the hierarchy.

1.5.2.1.1 In-Context Thread Deletion

In-context thread deletion involves returning to the system all resources owned by the thread. This may include AST control blocks, IO request packets, and other outstanding system resources. All mutexes owned by the thread must be dealt with, and the thread control region must be returned to the control region pool of its process. These actions occur as part of the thread object's remove routine.

The second phase of execution thread deletion is then started by calling the kernel primitive *k$terminate_thread()*.

1.5.2.1.2 Out of Context Thread Deletion

The call to *k$terminate_thread()* is responsible for queuing a terminate-thread descriptor on a queue served by the system thread responsible for out-of-context thread deletion. The server causes the thread object's delete routine to be executed by dereferencing the pointer to the thread object.

The thread object delete routine deletes the kernel stack of the terminating thread by calling *e$delete_thread_stack()*.

At the end of out-of-context thread deletion, all data structures that represent the thread are returned to the system. This includes the entire thread object and thread control block.

NOTE

The thread control region is deallocated during in context thread deletion because it must refer to the thread's process address space.

1.5.2.2 Address Space Deletion

If the terminating execution thread is the last thread of its process, then the address space of the process must also be deleted. This occurs in the process delete routine.

NOTE

Address space, as used above, means address-space management data structures such as page tables, working set lists, and the last thread's kernel stack.

The user-mode address space is deleted mostly as a result of removing the process level container directory, since user-mode address space is represented as section objects.

The process delete routine calls *e$delete_process_address_space()*, specifying the page table base register value from process object body.

1.6 Exit Status

The exit status mechanism in the Mica system supports the ability to obtain the exit status from a process and, in some cases, from an individual thread within a process.

The exit status mechanism is coordinated through the exit status object.

1.6.1 Object Structure

The exit status object contains information describing the termination state of the object it is bound to. Example 1–33 illustrates the layout of the exit status object.

Example 1–33: Exit Status Object Structure

```
!
! Exit Status Object Body
!
e$t_exit_status_body: RECORD
    es_exit_status_summary: e$t_exit_status_summary;     ! Exit Status Summary
    es_exit_status_event: k$dispatcher_object(event);      ! Signaled on status summary valid
END RECORD;

!
! Exit Status Summary
!
e$t_exit_status_summary: RECORD
    status_valid: boolean;                                  ! True if status summary valid
    status_bound_object_type: e$t_status_object_types;   ! Process or Thread
    status_bound_object_id: e$t_object_id;               ! Object ID of object reporting status
    status_value: e$t_exit_status;                       ! Exit Status
END RECORD;
```

1.6.2 Functional Interface

The Mica executive provides interfaces to create and obtain information from exit status objects.

1.6.2.1 Exit Status Object Creation

Exit status objects are created by the *exec$create_exit_status()* system service. Exit status objects are created in a "invalid" state and are not bound to either a process or a thread object. The object binding occurs during thread and process object creation. The "validation" of exit status objects occurs during process and thread deletion. Example 1–34 illustrates the interface to *exec$create_exit_status()*.

Example 1–34: Exit Status Object Creation System Interface

```
PROCEDURE exec$create_exit_status (
            OUT object_id: exec$t_object_id;
            IN  container: exec$t_object_id = DEFAULT;
            IN  name: exec$t_object_name = DEFAULT;
            IN  acl: exec$t_acl = DEFAULT;
            ) RETURNS status;
            EXTERNAL;

!++
!
! Routine description:
!
!   Create an invalid exit status object
!
! Arguments:
!
!   object_id           The object ID of the created exit status object
!   container           Object container for exit status object
!   name                Name of exit status object
!   acl                 ACL to place on exit status object
!
! Return value:
!
!   TBS
!
!--
```

1.6.2.2 Get Exit Status Information

The *exec$get_exit_status_information()* system service provides a mechanism for obtaining the information stored in an exit status object. Example 1–35 illustrates the interface to *exec$get_exit_status_information()*.

Example 1–35: Get Exit Status Information System Interface

```
PROCEDURE exec$get_exit_status_information (
            IN object_id: exec$t_object_id;
            OUT status_summary: e$t_status_summary;
            ) RETURNS status;
            EXTERNAL;

!++
!
! Routine description:
!
!   Return the es_exit_status_summary field from the exit status object
!   specified by object_id
!
! Arguments:
!
!   object_id       object ID of exit status object
!   status_summary  es_exit_status_summary field from specified exit status object
!
! Return value:
!
!   TBS
!
!--
```

1.6.3 Usage

Exit status objects are used to report the exit status of exiting processes and exiting threads.

Each thread in the Mica system may optionally be bound to an exit status object. The binding occurs during the creation of the thread.

Each process in the Mica system is bound to an exit status object. The binding occurs during the creation of the process.

Exit status objects are "invalid" at object creation time and remain invalid until the object that they are bound to is removed from the system.

1.6.3.1 Thread Exit Status Object Usage

If the *thread_status* parameter is specified during the direct or indirect creation of a thread, then the thread is bound to the specified exit status object. The exit status object is made valid during the object remove routine for an exiting thread. This occurs as follows:

- Set the *tcb_exit_status_value* field to the value stored in the thread control block *tcb_exit_status_value* field.
- Set to true the *status_valid* field in the existing status object bound to the exiting thread.
- Set the *status_value* field to the value stored in the thread control block *tcb_exit_status_value* field.
- Set to true the *status_valid* field in the exit status object bound to the exiting threads process.
- Signal the *es_exit_status_event* in the exit status object bound to the exiting thread.

If the exiting thread is not bound to an exit status object, then the following occurs.

- Set the *status_value* field to the value stored in the thread control block *tcb_exit_status_value* field.
- Set to true the *status_valid* field in the exit status object bound to the exiting threads process.

1.6.3.2 Process Exit Status Object Usage

Each process in the Mica system is bound to an exit status object. During the object remove routine for a process object, the process exit status object is signaled by setting the *es_exit_status_event* in the exit status object bound to the exiting process. The *status_valid* and *status_value* fields were previously set during the individual thread exits for all of the processes threads.

1.7 Process/Thread Startup/Rundown Summary

This section is an attempt to summarize the steps that occur during the creation, execution, and termination of a thread in the Mica system. A very simple hierarchy will be studied in this description. The hierarchy consists of user.0, job.0, process.0, and thread.0 from Figure 1–1.

1.7.1 Startup Summary

The sample hierarchy is created as a result of the *job controller* calling *exec$create_user()*. The following steps occur as a result of this call.

1. A user object named user.0 is created. The user control block is initialized from the *user_record* parameter. The user object body is initialized to contain an empty job list and a job count of zero.
2. A job object named job.0 is created. The job control block is initialized by allocating *q_per_job_limits* quota from user.0, and assigning it to *jcb_usage_and_limits*. A job level container directory is created and optionally populated based on the existence of the *job_initial_container* parameter. The job object body is initialized to contain an empty process list and a process count of zero. The *j_user_pointer* is set to be a referenced pointer to user.0, and job.0 is linked to user.0's job list. User.0's job count is incremented to 1.

3. A process object named process.0 is created. The process control block is initialized by allocating *q_per_process_limits* quota from user.0 and assigning it to *pcb_usage_and_limits*. A security profile for the process is obtained from user.0. The accounting structure in the process control block is initialized to *zero()*. A process level container directory is created and optionally populated based on the existence of the *process_public_container* and *process_private_container* parameters. The *pcb_condir_address* and *pcb_condir_mutex* vectors are initialized. The process object body is initialized to contain an empty thread and sub-process list. The thread count and sub-process count is set to zero. The *p_job_pointer* is set to be a referenced pointer to job.0, and process.0 is linked to job.0's process list. Job.0's process count is incremented to 1.

4. A thread object named thread.0 is created. The thread control block is initialized by clearing all events and setting the *tcb_irp_list_head* to empty. The *tcb_pcb_pointer* field is initialized to point to process.0's process control block. If specified in the *thread_status* parameter, the exit status object for the thread is referenced and stored in *tcb_exit_status_ptr*. The *tcb_exit_status_value* is cleared. The thread object body is initialized by setting the *t_process_pointer* to be a referenced pointer to process.0, and thread.0 is linked to process.0's thread list. Process.0's thread count is incremented to 1.

5. An address space is created for process.0 by calling *e$create_process_address_space()*. This call initializes portions of the *tcb_thread_context*, and the *pcb_ptbr*.

6. The kernel context for the thread is initialized by calling *k$initialize_thread()*.

7. The thread is made eligible to run in kernel mode at the *e$initial_thread_startup()* entry point by calling *k$ready_thread()*. At this point, the original caller of *exec$create_user()* is returned to with a "successful" user creation. Failures in thread startup after this point occur in the context of the created thread and are treated as an abnormal termination status of the thread.

8. The first action performed by the thread at *e$initial_thread_startup()* is a call to *e$initialize_address_space()*.

9. Once the address space has been initialized, the thread initializes the control region by calling *e$initialize_control_region()*. The control region appears as a user-mode read-only, kernel-mode read/write portion of process.0's address space. A *buddy system* memory pool is created and initialized in the control region as a result of calling *e$initialize_control_region()*.

10. The process control region is allocated by calling *e$pool_allocate()* specifying a pool type of *e$k_pool_control_region*. The *pcb_pcr_base* field of process.0's process control block is set to point to the allocated pcr, and the pcr is initialized by portions of the *initial_thread_parameters* parameter, and the object ID of process.0.

11. The thread control region is allocated by calling *e$pool_allocate()* specifying a pool type of *e$k_pool_control_region*. The *tcb_tcr_base* field of thread.0's thread control block is set to point to the allocated tcr, and the tcr is initialized by portions of the *initial_thread_parameters* parameter, the object ID of thread.0, the address of process.0's pcr, and various attributes of the thread specific address space.

12. The program image specified by the *process_record* field of the *initial_thread_parameters* parameter is mapped into process.0's address space by transitioning into user-mode at *e$program_image_startup()*.

13. Once at *e$program_image_startup()*, the thread issues a call to *exec$map_image()* and then sets the thread start address in the thread control region to the value returned by *exec$map_image()* by calling *exec$set_thread_information()*.

14. The thread entry point stored in *tcr_start_address* is "called" and is passed the thread parameters stored in *itp_thread_parameter_list*.

1.7.1.1 Additional Thread Startup Summary

This section describes the startup procedures for subsequent threads of a process. Assuming the hierarchy of the previous section, the following occurs when thread.0 makes a call to *exec$create_thread()* creating thread.1.

1. A thread object named thread.1 is created. The thread control block is initialized by clearing all events and setting the *tcb_irp_list_head* to empty. The *tcb_pcb_pointer* field is initialized to point to process.0's process control block. If specified in the *thread_status* parameter, the exit status object for the thread is referenced and stored in *tcb_exit_status_ptr*. The *tcb_exit_status_value* is cleared. The thread object body is initialized by setting the *t_process_pointer* to be a referenced pointer to process.0, and thread.1 is linked to process.0's thread list. Process.0's thread count is incremented to 2.
2. A partial address space is created for thread.1 calling *e$create_thread_stacks()*. This call initializes portions of the *tcb_thread_context*.
3. The thread control region is allocated by calling *e$pool_allocate()* specifying a pool type of *e$k_pool_control_region*. The *tcb_tcr_base* field of thread.1's thread control block is set to point to the allocated tcr, and the TCR is initialized by portions of the *initial_thread_parameters* parameter, the object ID of thread.1, the address of process.0's pcr, and various attributes of the thread specific address space.
4. The thread start address in thread.1's TCR is initialized to the value specified in the *thread_procedure* parameter.
5. The kernel context for the thread is initialized by calling *k$initialize_thread()*.
6. The thread is made eligible to run in kernel mode at the *e$subsequent_thread_startup()* entry point by calling *k$ready_thread()*. At this point, the caller of *exec$create_thread()* is returned to with an "successful" thread creation. Failures in thread startup after this point occur in the context of the created thread and are treated as an abnormal termination status of the thread.
7. Once at *e$subsequent_thread_startup()*, the thread entry point stored in *tcr_start_address* is "called" and is passed the thread parameters stored in *thread_parameter_list*.

1.7.2 Rundown Summary

At some point in the threads lifetime it will either voluntarily exit by calling *exec$exit_thread()* or be forcibly exited by calling *exec$force_exit_thread()* on itself or having some other thread issue an *exec$force_exit_thread()* specifying that thread.

For the following rundown example, it is assumed that thread.99 issues an *exec$force_exit_thread()* specifying the object ID of thread.0. The hierarchy consists of user.0, job.0, process.0, and thread.0.

1. The Mica executive is entered at *e$force_exit_thread()*. The force-exit in progress flag is set in the thread object body of thread.0. The purpose of this flag is to prevent the creation of new exit handlers for the thread and to prohibit the thread from creating new threads, processes, and jobs.
2. The next step is to cause thread.0 to begin execution in "trusted" user-mode at the *e$in_context_force_exit()* executive entry point. At this point the force-exit of thread.0 is complete with respect to thread.99. The following steps are then taken to force thread.0 into taking an active role in its exit. This occurs as follows:
   - An elapsed timer is set to expire in a TBD period. If the timer expires, all of these steps are repeated, in addition to enabling user mode ASTs, setting the ast queue flush flag in the thread object body, and a call to *k$flush_ast_queue()* is issued.
   - A user-mode AST is queued to thread.0. The target procedure of the AST is *e$in_context_force_exit()*.
3. Once at *e$in_context_force_exit()*, thread.0 unwinds its stack by calling *e$unwind()*.

4. Once the stack has been unwound and all unwind handlers have been executed, thread.0 executes a call to *exec$exit_thread()*.

5. The code at *exec$exit_thread()* assigns the parameter *exit_status* to the *tcb_exit_status_value* field of thread.0's thread control block. If thread.0 was created with an exit status object, the *exit_status* is also assigned to *tcb_exit_status_ptr^.es_exit_status_sumary.status_value*. The value *exit_status* is then assigned to *pcb_exit_status_ptr^.es_exit_status_sumary.status_value* in process.0's exit status object. The force-exit in-progress flag for thread.0 is examined. Since the flag was set, the elapsed timer set up in *e$force_exit_thread()* is dismissed.

NOTE

If the timer in the example above had expired, that would indicate that the user-mode AST was not delivered, or that there was an exceptional delay in making progress through the stack unwind. In any case, timer expiration causes a retry which will eventually be successful.

6. Thread.0 is then allowed to execute each one of its exit handlers. Since the thread is being force-exited, its exit handlers are assigned a small CPU time quota. When the quota expires, a user-mode AST is delivered to the thread that causes it to execute an *exec$exit_thread()*. The method for delivering the user-mode AST is similar to the technique used to cause the thread to execute at *e$in_context_force_exit()*, only the AST procedure target is *e$exit_handler_expire()*. The function of *e$exit_handler_expire()* is to simply call *exec$exit_thread()*.

7. Thread.0 issues a call to *e$remove_object_id()* specifying its object id (*t_obj_id*. This action causes thread.0's object remove routine to be called.

8. Thread.0's object remove routine is entered. It performs the following steps.

   - All outstanding resources that require cleanup by the thread are processed. This includes the dismissal of all outstanding I/O by callin *e$cancel_io_by_thread()*, the dismissal of outstanding ASTs, and ...(TBS).
   - The thread control region is returned to the control region pool of process.0 by calling *e$pool_deallocate()*.
   - The thread object is de-linked from the *p_thread_queue_hd* of process.0.
   - Since the above step causes the *p_thread_count* field to decrement to zero, the process.0 object is removed by calling *e$remove_object_id()* specifying the object ID of process.0.
   - If thread.0 was created with an exit status object, then the *es_exit_status_event* in the object is "set". The exit status object is then dereferenced by calling *e$dereference_object()*.

9. The object remove routine for process.0 is entered as a result of thread.0's object remove routine being entered. The following occurs during process.0's object remove routine.

   - The job level container directory whose address is stored in *pcb_condir_array* is dereferenced.
   - The process level container directory is removed from the system by calling *e$remove_object_id()*, and specifying *pcb_process_condir_id*
   - The process control region is returned to its control region pool by calling *e$pool_deallocate()*.
   - The process object is de-linked from the *j_process_queue_hd* of job.0.
   - Since the above step causes the *j_process_count* field to decrement to zero, the job.0 object is removed by calling *e$remove_object_id()* specifying the object ID of job.0.
   - The *es_exit_status_event* in process.0's exit status object is "set". The exit status object is then dereferenced by calling *e$dereference_object()*.

10. The object remove routine for job.0 is entered as a result of process.0's object remove routine being entered. Th=e following occurs during job.0's object remove routine.
    - The job level container directory is removed from the system by calling *e$remove_object_id()*, and specifying *jcb_job_condir_id*.
    - The job object is de-linked from the *u_job_queue_hd* of user.0.
    - Since the above step causes the *u_job_count* field to decrement to zero, the user9.0 object is removed by calling *e$remove_object_id()* specifying the object ID of user.0.
11. The object remove routine for user.0 is entered as a result of job.0's object remove routine being entered. The routine performs no significant actions
12. The original call in *exec$exit_thread()* which removed the object ID of thread.0 returns. The next step is a call to *k$terminate_thread()*. The purpose of *k$terminate_thread()* is to remove the specified thread (thread.0) from execution within the Mica system. Once all of the kernel related activities are complete, a pointer to thread.0 is queued to a special system thread known as the *thread eater*. The thread eater executes the loop at *e$terminate_thread()*.
13. The function of *e$terminate_thread()* is to dequeue the thread's arriving on its queue, and to dereference the thread objects. When the thread eater processes thread.0, it calls *e$dereference_object()* specifying thread.0. The delete routine for thread.0 is entered. It is important to note that the delete routine for thread.0 is entered in the context of the thread eater.
14. The delete routine for thread.0 is entered. It performs the following actions.
    - Thread level accounting information is rolled up to the thread's process.
    - The thread specific address space (user-mode, and kernel-mode stacks) of thread.0 are returned to the address space of process.0 by calling *e$delete_thread_stacks()*.
    - The referenced pointer to process.0 is dereferenced. This causes the delete routine for process.0 to be executed.
15. The delete routine for process.0 is entered. It performs the following actions.
    - An accounting record is written to the TBD message function processor. The information for the accounting record is obtained from the *pcb_accounting* field from process.0's process control block.
    - All resources accounted for in process.0's *pcb_usage_and_limits* are returned to job.0's *jcb_usage_and_limits* using the rules of deductable and non-deductable resource arithmetic.
    - The address space of process.0 is returned to the system by calling *e$delete_process_address_space()*.
    - The referenced pointer to job.0 is dereferenced. This causes the delete routine for job.0 to be executed.
16. The delete routine for job.0 is entered. It performs the following actions.
    - All resources accounted for in job.0's *jcb_usage_and_limits* are returned to user.0's *ucb_quotas.q_usage_and_limits* using the rules of deductable and non-deductable resource arithmetic.
    - The referenced pointer to user.0 is dereferenced. This causes the delete routine for user.0 to be executed.
17. The delete routine for user.0 is entered. It performs no significant actions.
18. Once the call frame has returned from the original call to *e$dereference_object()* issued by the thread eater on thread.0, the UJPT hierarchy consisting of user.0, job, process.0, and thread.0 is removed from the system, and the thread eater goes back to its queue of threads to be processed.

1.8 System Threads

This section describes the interface for creating system threads. It also describes the differences between system threads and normal threads, and the special restrictions placed on system threads.

1.8.1 System Thread Creation

The *e$create_system_thread()* executive interface creates a system thread. The system thread executes within the UJPT hierarchy of the system. The address space of the system thread is that of the initial system process. Example 1–36 illustrates the interface to *e$create_system_thread()*.

Example 1–36: System Thread Creation Executive Interface

```
PROCEDURE exec$create_system_thread (
            OUT object_id: e$t_object_id;
            IN  container: e$t_object_id = DEFAULT;
            IN  name: e$t_object_name = DEFAULT;
            IN  acl: e$t_acl = DEFAULT;

            IN thread_procedure: e$t_thread_entry_point;
            IN thread_record: e$t_thread_record = DEFAULT;
            IN thread_allocation_list: e$t_allocation_list = DEFAULT;
            IN thread_immediate_parameter1: e$t_thread_parameter = DEFAULT;
            IN thread_immediate_parameter2: e$t_thread_parameter = DEFAULT;
            IN thread_status: e$t_object_id = DEFAULT;
            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Create a System thread object as specified by the parameters.
!
! Arguments:
!
!   object_id                   Object ID of the resulting process object
!   container                   Object container for thread object (ignored)
!   name                        Name of thread object
!   acl                         ACL to place on thread object
!   thread_record               Attributes of the thread being created
!   thread_allocation_list      Objects to be allocated to the thread object. If not present then
!                               no objects are allocated to the thread
!   thread_immediate_parameter1 Immediate parameter passed to thread through TCR
!   thread_immediate_parameter2 Immediate parameter passed to thread through TCR
!   thread_procedure            pointer to thread entry point entry descriptor
!   thread_status               Exit status object to be bound to the thread. If not present
!                               then the thread is created without an exit status object
!
! Return value:
!
!   TBS
!
!--
```

1.8.2 System Thread Restrictions

The important differences between system threads and normal threads are as follows:

- System threads may not execute in user-mode.
- System threads are incapable of processing or executing in the context of user-mode ASTs. Algorithms such as the one in *exec$signal_thread()* that employ user-mode ASTs either understand system threads and modify their algorithms or don't support the functions on system threads.
- The thread control region for system threads exists in paged pool.
- There is no thread environment block for system threads.
- System threads execute within the address space of the system.
- There are no provisions for passing block data to a system thread through the *tcr_block_data* field in a system thread's TCR.

- The *exec$force_exit_thread()* system service is not supported for system threads.

Mica Working Design Document
Record Management Services

Revision 0.3

7–January–1988

Issued by:

Sumanta Chatterjee

digital™

TABLE OF CONTENTS

Revision History

Date	Revision Number	Author	Summary of Changes
10-Dec-1987	.1	S. Chatterjee	Initial Draft
04-Jan-1988	.2	S. Chatterjee	1. Major restructure of interface parameters 2. Incorporated comments from the primary reviewers
06-Jan-1988	.3	S. Chatterjee	1. Minor editing changes

CHAPTER 1

RECORD MANAGEMENT SERVICES

1.1 Introduction

Mica RMS is a set of generalized library routines that assist user programs in processing and managing files and their contents. The interfaces provided by Mica RMS routines are used uniformly to access files within the defined client-server environment. This document describes the framework for Mica RMS implementation in the following sections:

- This introduction briefly discusses the design philosophy, lists the goals of the project, states the functions provided, and lists the VMS RMS functions omitted from Mica RMS. The section concludes with a short discussion on Mica file service support that is used by RMS.
- The second section defines the functions that are available on each supported device.
- The third section describes the Mica RMS programming interfaces.
- The fourth section outlines the overall request flow. Algorithms used in implementing a few select Mica RMS functions are also included in this section.
- Appendix A outlines a preliminary plan for testing RMS software.

1.1.1 Design Philosophy

Mica RMS, together with the applications interface architecture (AIA), provides the highest level user interface in the Mica system. The purpose of Mica RMS is to provide a convenient interface to process and manage files and their contents. Much of the RMS file processing capabilities are inherited from the unlerlying infrastructure of the Mica I/O subsystem. However, record-level management is provided only through RMS. VMS RMS replicates many of the functions that are available through the I/O subsystem primarily as a user convenience. In Mica, the I/O architecture provides a straightforward interface to user-mode processes, thereby eliminating the requirement of replicating many of the functions at the RMS level. However, user convenience is not forgotton. Thus, for example, RMS provides ways for users to create or delete files.

Mica RMS services operate in user mode. Many of the design decisions reflect this. A few results of operating in user mode are listed below:

- RMS is not notified if the user program exits abnormally. As a consequence, the user buffers allocated by RMS are flushed by exit handlers.
- RMS procedures are directly callable from the user program, without requiring a context switch.
- The data structures maintained by RMS for its users can be corrupted by an erring user program.
- RMS runs in the user's process context, within the user's address space. RMS allocates buffers for the user by calling a system function. Thus, much of the information maintained by VMS RMS in the process I/O (PIO) segment are no longer required.

Software is a piece of text that specifies computations. A piece of software that provides nontrivial functions is best constructed in component pieces that interact with each other by way of well defined interfaces. The philosophy is to have as general an interface as possible. This guiding principle is used in designing not only the external interfaces, but also in interactions between various internal procedures and modules.

1.1.2 Goals

Mica RMS is designed to meet several goals:

- Ease of use—This goal is reflected by the user interface design. Mica RMS services are accessed through procedure calls. Each service procedure has a few (less than a dozen) parameters, many of which are optional and default to often-used values. The parameters appearing in the interface are the commonly-used file attributes, the required buffer pointers, and the outputs from the service. For infrequently used input options, the services provide an input parameter, which is an item list. One advantage of using an item list is that the options can be enhanced without affecting the user interface.
- Fast response time—The data retrieval services are designed to minimize run-time decision making.
- Device independence—Mica RMS, like VMS RMS, offers device-independent file handling.
- Modularity—The RMS implementation supports easy addition of enhancements. easily added. For example, supporting a new device type or file organization can be done in a fairly straightforward manner. Implementation avoids exception code as much as possible.

1.1.3 RMS Functionality

Mica RMS provides user programs with the capability to do the following:

- Parse and wildcard file names.
- Specify multiple file organizations (sequential, indexed or relative); at FRS, only sequential files are supported.
- Specify multiple record formats (fixed, variable, VFC, stream, streamCR, streamLF, and undefined).
- Specify multiple ways to access records (delete, get, put, update, and truncate).
- Specify multiple ways to share files and enforce access control to files (shared delete, get, put, update, nil and user-provided interlocking). At FRS, the available support allows multiple processes to read share a single disk file. Also, a file may be shared between a single writer and mutiple readers. See Section 1.2.1.
- Specify multiple device types for record access. At FRS, RMS supports I/Os to disk devices only. Paths are also provided for conducting I/O to terminal devices connected to the client systems. See Section 1.2.3.
- Specify ways to lock and unlock records. At FRS, there is no support available for record locking.

1.1.4 Functions Not Available

This section lists VAX/VMS RMS functions that are either not available at FRS, or have been permanently excluded from Mica RMS. A few of the VAX/VMS RMS functions are excluded permanently as these functions are easily available through the Mica File system. A few other functions are excluded permanently as they are available as system services.

The following lists the functions that are not available at FRS, but are planned for future releases:

- File organization—Indexed and relative files
- File access—Shared write access to disk files
- Record locking—Ways to lock and unlock records
- Transaction logs—Journal file I/O operations

The rest of this section lists the VAX/VMS RMS functions that are not planned to be included as Mica RMS functions.

The following VAX/VMS RMS functions are excluded permanently from Mica RMS:

- Asynchronous I/O operations (Mica RMS supports synchronous I/Os only)
- Direct record access to mailboxes or message devices
- Remote file access and task-to-task communication by way of DECnet
- Implicit file spooling
- DECK and EOD checking
- Multiple record streams
- File disposition option submit command file on execution of RMS$CLOSE
- Set date and time for file creation, expiration, revision or backup

The I/O subsystem functions not replicated in Mica RMS are:

- $ENTER
- $EXTEND
- $NXTVOL
- $REMOVE
- $RENAME
- $SPACE

ODS2-3 defines the following six date and time values which are maintained as file attributes:

1. Creation date and time
2. Expiration date and time
3. Backup date and time
4. Revision date and time
5. Read date and time
6. Header write date and time

VMS RMS allows its users to set any of the first four date and time values at file creation time. The Mica file system automatically sets the creation date and time, and the Mica RMS interface does not provide an explicit way to set any of these times except the expiration date and time. The user may examine the date and time values through the Display service.

By default, Mica file system sets and maintains all the date and time values, except the expiration date and time. However, a user may call *exec$request_io* with the function code *io$c_dfile_write_attributes* to set the date and time values.

A user may influence how items 4 and 5, revision date and time, and read date and time are maintained. By default, these items are updated in memory and written out at file close time. The user can choose to force a disk update, at substaintial performance penalty, on every read or write.

The following system services are not available through Mica RMS:

- SYS$RMSRUNDOWN
- SYS$SETDDIR
- SYS$SETDFPROT
- $WAIT

The undocumented VAX/VMS RMS function $MODFY is not available in Mica RMS.

1.1.5 Interface to Mica File Sytem

The Mica I/O architecture provides a set of services through which user-mode processes access functions provided by the I/O subsystem. The Mica I/O architecture defines function processors through which specific I/O requests are satisfied. RMS accesses disk resident files through function processors belonging to the disk file function processor (DFFP) class. The specific function processor used depends upon the volume on which the file resides. For example, FILES-11 function processor provides access to local Mica volumes, and the distributed file service (DFS) function processor provides access to nonlocal volumes. However, every function processor of the DFFP interface class provides the same user interface. RMS accesses the DFFP class function processors uniformly.

A fully specified file name is of the form:

```
volume_name:[directory_specification]file_name.type;version
```

To separate the type and the version fields, either ";" or "." may be used. A function processor accepts I/O requests to one of its volumes through a function processor unit (FPU) that represents the volume. RMS accesses the I/O subsystem by using the following steps:

1. In order to access the I/O subsystem services, the FPU object ID is required. RMS obtains the FPU object ID by calling *exec$translate_object_name*, with the volume name as an input parameter.
2. RMS calls *exec$create_channel* to establish an I/O channel to the FPU. A channel is deleted by calling *exec$delete_object_id* and specifying the channel's object ID as the input parameter to the call.
3. RMS calls *exec$get_fpu_information* to determine that the channel is assigned to an FPU that belongs to the supported interface class. This call also provides volume-specific information (for example device characteristics).
4. Functions provided by DFFP are obtained by calling *exec$request_io*. A caller specifies a DFFP function code while calling *exec$request_io* to access a DFFP function. The following DFFP functions are used:
   - Create a file (*io$c_dfile_create*)
   - Specify and read file attributes (*io$c_dfile_write_attributes, io$c_dfile_read_attributes*)

- Allocate storage and deallocate storage (*io$c_dfile_allocate_storage, io$c_dfile_deallocate_storage*)
- Access and deaccess files (*ioc_dfile_access, ioc_dfile_deaccess*)
- Transfer data (*ioc_dfile_read, ioc_dfile_write*)
- Search for a file (*io$c_dfile_search_dir*)
- Read one or all the entries from a given directory (*io$c_dfile_read_dir_entries*)
- Enter or remove a directory entry (*io$c_dfile_modify_dir_entries*)
- Delete a file by the file ID (*io$c_dfile_delete_by_fid*)

1.2 Devices Supported

Mica RMS provides device independent file access. Device type is not a file attribute. The mounted device (the volume) on which the file resides or on which the file is to be created, is derived explicitly (user specifies it) or implicitly from the user's environment. The I/O subsystem provides accesses to the devices by way of function processors. Ideally, higher layer software like RMS is not required to possess knowledge of device characteristics. However, to prevent certain operations, for example, creating an indexed file on a magnetic tape device, some knowledge of major device characteristics is required. Other than that, if the function processors provide uniform interface, many of the device characteritics are transparent to RMS.

Although RMS is designed to support a variety of devices, its functions are geared towards mass storage devices, especially random access devices. The following paragraphs describe the range of RMS functions available on the supported device type.

1.2.1 Disk Devices

The functions that are available on disk devices are discussed in this section. The functions are classified as:

- File creation time options that define file characteristics
- File creation time options to specify file names
- File creation time options that specify file allocation and position control
- File access and file sharing criteria
- Reliability options in I/O operations
- Run-time options to specify file disposition
- Run-time record retrieval options
- Run-time record insertion options

Each of these items is discussed in the following sections.

1.2.1.1 File Characteristics

Files created on disk devices can have the following characteristics:

- File organization—Disk file organization can be sequential, indexed or relative. At FRS, only sequential files can be created.
- Record format—The record format can be fixed length, variable length, variable length with fixed length control, stream or undefined. The default is variable length.
- Record attributes—All the record options specified by *rms$record_attributes* are applicable (see the description of *rms$create* for record structure definition, Section 1.3.1.2.3). For example, the user specifies that the records may span block boundaries by setting the *rms$blk* bit in *rms$record_attribute*. The user may set *rms$record_options.max_rec_size* to specify the maximum record size.
- Date information—This is defined in *rms$display*. Date information provides date and time values for file backup, file creation, file expiration, last accessed, last header write and the file revision. Date and time values are set and maintained by the Mica file system. See Chapter 20, Disk File System Function Processors, for the rules used to set and maintain date and time values. Through RMS, the user can set the expiration date and time of the file at file creation time.
- File protection—Mica files are protected by way of access control lists. The mechanism and the interface are TBS.

- Index file characteristics—These are TBS.

1.2.1.2 Filename Creation

At disk file creation time, the following options may be used.

- Create if nonexistent—Creates the file if the file of the same name does not exit in the specified directory. If the file exists, then the file is opened.
- Maximize version—Creates a disk file with a specified version number or a version number one greater than a file of the same name in the specified directory.
- Supersede version—Supersedes the file of the same file name, file type, and version number.
- Temporary marked for delete—The file is created, without any directory entry. The file is automatically deleted when the file is closed.
- Temporary—The file is created without any directory entry. The file is retained after being closed. However, the file can only be reopened if the file ID is supplied.

1.2.1.3 File Allocation

At the time a disk file is created, the file space allocation amount, default extension amount and placement control can be specified by the record *rms$create_in_alloc_options*. See Section 1.3.1.4.1. If the allocation option is not used, RMS sets the default extension area to be equivalent to the track size of the device. Thus, initially the user creates a file with zero allocated size. At the time of the first output, an area equivalent to the default extension is allocated automatically for the user.

1.2.1.4 File Sharing

The user specifies the way a disk file is to be accessed and the way the file is to be shared with other users at file open or file creation time. The file access and share rules are set on calls to *rms$create* or *rms$open*, through the input parameter *access_request*. For more information, see Section 1.3.1.3. In the initial version, Mica RMS does not accomodate multiple writers to the same file. However, a single writer may share the file with multiple readers. If a file is accessed for write, then by default, the file is opened for exclusive use, which prohibits sharing. If, however, the writer wants to allow readers, then the *rms$c_shrget* and *rms$c_upi* need to be set.

If a file is accessed for read only, the default sharing is *rms$c_shrget*. If the reader wants to allow a writer, then the *rms$c_shrput* bit and the *rms$c_upi* bit need to be set.

A request for access to a file is policed by the file system and not directly by Mica RMS. Whether an access to a file is allowed or not, is determined by the most current sharing value. For example, a file currently has 3 readers (A, B, and C) and one writer (X). At this point if another writer (Y) tries to open the file for write access, the open fails. If, however, X closes the file and then Y tries the open again, the open succeeds.

A file shared between a writer and multiple readers requires that the buffers written by the writer are periodically flushed out. It is proposed that Mica RMS forces a flush operation after 'n' (say 100) buffers are written. This ensures that the file attribute end-of-file VBN is updated for the reader's benefit. Independently, the user may call the Flush service to force an update.

At FRS, Mica RMS does not provide record locking facilities.

1.2.1.5 Reliability Options

Reliability options are set at disk file creation or open time. Through the input option item *rms$file_characteristics*, the user specifies *rms$c_read_check* and *rms$c_write_check* options to ensure that the data transfers from or to the disk volumes are to be checked by a read-compare operation. Reliability checks effectively double the amount of disk I/Os performed.

The user may set *rms$c_force_write_thru_dates* to force update the last read date and time, and the last write date and time on the file header, on every I/O.

1.2.1.6 Runtime File Disposition Options

Mica RMS provides read-ahead and write-behind buffer management for all sequential files. Mica RMS provides only synchronous I/O operations to its users. Through the input option item *rms$run_time_access*, the user can specify the following file disposition options at the time the is file is created or the file is opened.

- Truncate end-of-file—Indicates that the unused space is to be deallocated at the time the file is closed.
- Delete on close—Indicates that the file is to be created and a directory entry is made. The file is deleted automatically when closed.

1.2.1.7 Record Retrieval Options

Records on sequential disk files are accessed sequentially or randomly (by record's file address). Records on sequential files with fixed record formats can be accessed randomly by the relative record position.

By default, locate mode is used for data retrieval operations. The user may optionally set move mode for record retrieval. See Section 1.3.5.

If the record format is variable length with fixed control (VFC), the user can specify the length of the fixed control portion.

1.2.1.8 Record Insertion Options

For sequential files records are usually inserted at the end of the file. The records to be inserted cannot be larger than the maximum record size (*max_record_size*) as defined in the record *rms$record_definition*. See Section 1.3.1.2. A record can also be inserted randomly by key in a sequential file with fixed length records. To insert randomly, the record access mode *rms$c_update* must be selected.

The *truncate_on_put* option allows new records to be inserted in a sequential file, in locations other than at the end of the file. After the record is inserted, the file is truncated immediately after the inserted record. The end-of-file marker is updated to the new location. To perform this operation, the user needs to select *rms$c_truncate* record access mode.

1.2.2 Magnetic Tape Devices

At FRS, I/Os to magnetic tape devices through RMS are not available.

1.2.3 Terminal Devices

At FRS, terminal devices are not directly connected to the Glacier (compute server) or the Cheyenne (data base server) systems. The terminals are connected to client systems and are considered to be part of the client environment. At FRS, Mica RMS accesses terminals through client context server (see Chapter 55, VMS Compute Server Support).

To read from a terminal device, the user must specify *move_mode* and an input buffer to receive the data. Data read from the client site is copied into the user input buffer. Read ahead and write behind are automatically turned off for I/Os to terminals.

At FRS, Mica RMS supports a minimal set of terminal options. None of the read verify functions of terminal drivers are available at FRS. If a file is used for terminal I/O, the file and record attributes that may be specified are:

- File organization is sequential only.
- Record access mode is sequential only.
- Terminal options in Get and Put services are listed below. The terminal options are not interpreted by Mica RMS, and they are forwarded to the client site. Enforcement of the options are carried out by the client site terminal driver.
    - Cancel CTRL/O—Guarantees that terminal output is not discarded if the operator presses CTRL/O.
    - Uppercase—Changes characters to uppercase on a read from a terminal.
    - Prompt option—The contents of the prompt buffer are to be used as a prompt for reading data from a terminal.
    - Purge type ahead—Eliminates any information that may be in the type-ahead buffer on a read from a terminal.
    - Read no echo—Input data is not echoed on the terminal.
    - Read no filter—Indicates CTRL/U, CTRL/R and DELETE are not to be considered as control commands on terminal input.
    - Timeout—Specifies the maximum number of seconds to wait between characters being typed.

1.2.4 Mailboxes

Mailboxes are not supported at FRS.

1.3 RMS Programming Interface

The following sections define various Mica RMS services. The services are presented in the following order:

- File creation and other basic services:
  - *rms$create*
  - *rms$open*
  - *rms$close*
  - *rms$get**
  - *rms$put**
- Filename parse and search services:
  - *rms$parse*
  - *rms$search*
- Other services:
  - *rms$display*
  - *rms$erase*
  - *rms$flush*
  - *rms$free* (not available at FRS)
  - *rms$release* (not available at FRS)
  - *rms$rewind*
  - *rms$truncate*
  - *rms$update*

Mica RMS services are provided by a set of user-mode run-time library procedures. The procedures are designed with the two goals of ease of use and flexibility. The RMS user specifies various file attributes to suit the requirements of a particular application. There are two categories of file attributes: the ones that are used for file level functions (such as Create and Open) and the ones that are used for record level functions (such as Get and Put). The attibutes are specified to the Mica RMS services by parameters. The attributes appear either as explict parameters, or as options in item lists.

Typically, the file attributes that appear explicitly are:

- The required information for the service
- The frequently specified attributes to the service (Usually, these parameters, also have associated defaults. Thus, if an attribute default has a suitable value, the user need not explicitly specify the parameter.)
- The values that are always returned by the service on successful completion

One set of file attributes is used to statisfy very specific user requirements (placement control of files, for example). These attributes are expressed as items of item lists. Item lists, as input options and output options, appear explicitly as parameters. For each service, if necessary, there is a valid list of input items and output items. Input items are those attributes that the user specfies to RMS, so that the file or record management is done appropriately. If an attribute appears both as an explicit parameter, and as an item, RMS uses the value specified in the item. Generally, the design avoids multiple ways to specify the same file attributes. Output items are those attributes that RMS returns

to the user. The user has complete flexibility in using the input items and the output items from the set of available options. Items and item lists are defined below:

```
!+
!RMS wide definition of an item and item list
!-

positive_integer : INTEGER[0..];

rms$item: RECORD
    code: LONGWORD;
    buffer_length: positive_integer;
    buffer_ptr : POINTER anytype;
    return_length_ptr: POINTER positive_integer;
    LAYOUT
        code;
        buffer_length;
        buffer_ptr;
        return_length_ptr;
    END LAYOUT;
END RECORD;

rms$item_list(n:positive_integer): RECORD
    CAPTURE n;
    rms$items: ARRAY [1..n] OF rms$item;
    LAYOUT
        n;
        fill_1:filler(longword,*);
        rms$items;
    END LAYOUT;
END RECORD;
```

Mica RMS does not provide support for multistreaming. However, RMS returns the next record pointer after every successful data retrieval operation (*rms$get**). The user maintains multiple record positions with multiple next record pointers. As explicit multiple streams are not provided, the VMS RMS Connect service is no longer necessary.

Mica RMS provides buffer management for its users. The user is not required to specify the number of blocks or the number of buffers. Read-ahead and write-behind functions are provided.

For each RMS service, the error conditions and the status values are required to be specified. Error conditions and status values for Mica RMS are not yet specified.

1.3.1 Create Service

The Create service (*rms$create*) creates files according to the attributes specified in the parameters. If a parameter is not specified, its default value is used. This service implicitly calls the Open (*rms$open*) service. The *rms$create* service does not replicate the Display (*rms$display*) service functions. However, for user convenience, the service returns, if completed successfully, some information about the opened file.

The *rms$create* service returns the file identity, as maintained by the file system. The *file_id* field is a part of the *quick_file_ref_out* parameter. The *rms$create* service also returns a file handle, which is the file reference maintained by RMS.

The caller checks for successful completion condition returned by the implicit *rms$open* service by examining the value returned in the *status*.

```
PROCEDURE rms$create(
            IN file_name : string(*) OPTIONAL;
            IN default_file_string : string(*) OPTIONAL;
            IN quick_file_ref_in :  rms$file_ref_identifier OPTIONAL;
            IN record_definition : rms$record_definition OPTIONAL;
            IN access_request : rms$file_access_share OPTIONAL;
            IN create_input_options : POINTER rms$item_list = NIL;
            OUT file_information : rms$standard_file_info OPTIONAL;
            OUT output_file_specification : rms$file_reference OPTIONAL;
            OUT quick_file_ref_out :  rms$file_ref_identifier OPTIONAL:
            OUT file_handle : rms$file_handle;
            ) RETURNS status;
            EXTERNAL;
```

1.3.1.1 File Identification

The caller specifies the file to be created and/or opened by either the *file_name* parameter or by the *quick_file_ref_in* parameter. The file name string cannot contain wildcard characters or a node name.

The parameter *file_name* is an instance of a valid file name. A file name is a string of characters from which the primary file specification is derived. The caller provides the default file specifications through the *default_file_string* parameter. Mica RMS uses the information contained in the *file_name* parameter and, if necessary, the *default_file_string* parameter to construct a full file specification.

The optional input parameter *quick_file_ref_in* is a record that contains the file ID and the volume object ID, the two necessary and sufficient information to identify and locate the file, without requiring any further file-name processing. This information is returned as output (*quick_file_ref_out*) by several RMS services. If this information is available, (for example, from an earlier call to the Parse and Search services), the caller returns the *quick_file_ref_out* to *rms$create* through the input parameter *quick_file_ref_in*. The *quick_file_ref_out* is an output of the *rms$parse*, *rms$search*, *rms$open* as well as *rms$create* service.

If *quick_file_ref_in* is present, and the caller has requested to create the file only if the file is nonexistant, then *rms$create* first tries to access the file by the file ID. RMS uses the volume object ID to open a channel to the FPU.

The *quick_file_ref* parameter is a pointer to a record with the following structure:

```
TYPE
    rms$file_ref_identifier: RECORD
        volume_object_id : exec$object_id;
        file_id : rms$f11_file_id;
    END RECORD;

    !+
    ! file$f11_file_id is defined by the Mica file system.
    ! It is reproduced below for easy reference.
    !-
    rms$f11_file_id : file$f11_file_id;

    file$f11_file_id : RECORD
        f11_fid_num : integer [0..65535] SIZE(word); ! file number 16 low bits
        f11_fid_seq : integer [0..65535] SIZE(word); ! sequence number
        f11_fid_rvn : integer [0..255] SIZE(byte);   ! relative volume number
        f11_fid_nmx : integer [0..255] SIZE(byte); ! file number 8 high bits
        LAYOUT
            f11_fid_num,
            f11_fid_seq,
            f11_fid_rvn,
            f11_fid_nmx;
        END LAYOUT;
    END RECORD;
```

1.3.1.2 Record Definition

This record structure is used to specify the following:

- File organization
- Record format
- Record attributes
- Maximum record size
- Longest record length

The record structure for record defintion is shown below:

```
TYPE
    rms$record_definition: RECORD
        file_organization : rms$file_organization;
        record_format : rms$file_record_format;
        record_attribute : rms$record_attribute;
        max_record_size : integer[0..32767];
        vfc_control_head_size : integer[0..255];
        longest_record_length : integer[0..32767];
        version_number : integer [1..65535] SIZE(word);
    END RECORD;
```

Each field of *rms$record_definition* is discussed in the following sections.

1.3.1.2.1 File Organization

The default file organization is sequential. The initial version of Mica RMS supports sequential organization only.

```
    !+
    !The following values are used to define file organizations
    !-
VALUE
    rms$c_sequential = 0;
    rms$c_relative = 1;
    rms$c_indexed = 2;

TYPE
    rms$file_organization : integer[0..2] SIZE(BYTE);
```

1.3.1.2.2 Record Format

The default record format is variable length records.

```
    !+
    !The following values define the file record format
    !-
VALUE
    rms$c_undefined = 0;     ! undefined
    rms$c_fixed = 1;         ! fixed length
    rms$c_variable = 2;      ! variable length
    rms$c_vfc = 3;           ! variable fixed control
    rms$c_stream = 4;        ! stream
    rms$c_stream_lf = 5;     ! lfstream (seq files ONLY)
    rms$c_stream_cr = 6;     ! cr stream (seq files ONLY)

TYPE
    rms$file_record_format : integer [0..6] SIZE (byte);
```

If the VFC format is chosen, the user specifies the fixed control area size by the field *vfc_control_head_size*.

1.3.1.2.3 Record Attributes

The valid input record attrubtes to *rms$create* are:

- BLK—records are not permitted to cross block boundaries
- CR —preced each record with a LF, and follow with CR
- FTN—FORTRAN carriage control character
- PRN—print file format

```
    !+
    !A record attribute type is defined below:
    !+
TYPE
    rms$record_attribute_names : (
        rms$c_ftn,      ! FORTRAN carriage control
        rms$c_cr,       ! preced each rec with CR, follow with LF
        rms$c_prn,      ! print file format
        rms$c_blk       ! records do not cross block boundaries
        );

    rms$record_attribute : SET rms$record_attribute_names[..] SIZE (BYTE);
```

Only *rms$c_blk* option can be paired with another option. The options *rmsc_ftn, rmsc_cr, rms$c_prn* cannot be used in any combination. The default value of this field is *rms$c_cr*.

1.3.1.2.4 Maximum Record Size

This integer value represents, in bytes, the size of all records in a file with fixed length records, the maximum size of variable length records, the maximum size of the data area for variable with fixed-control records.

1.3.1.2.5 VFC Control Head Size

This field is used to specify the length of the fixed-control area of a file with VFC record format. The default value is 2 bytes.

1.3.1.2.6 Longest Record Length

RMS returns through this field the numeric value of the longest record in the file. The field is used only if the record format is not fixed length.

1.3.1.3 Access Request

This record specifies the desired way the caller wishes to access the file and the way the caller wishes to share the file with other users. This is an optional input parameter to the *rms$create*. At file creation time, the default value of file accessing is put access, and the default value for file sharing is allowing shared read. Block I/O operations are considered as data retrieval operations, rather than access modes. Hence, block I/O options are removed from the set of file access options. The access request record structure is defined below:

```
TYPE

    rms$file_access_control : RECORD
        access : rms$file_access;
        share : rms$file_share;
    END RECORD;
```

```
rms$file_access_names : (
    rms$c_delete,    ! request delete access
    rms$c_get,       ! request read access
    rms$c_put,       ! request write access
    rms$c_update,    ! request update access
    rms$c_truncate   ! request truncate access
    );

rms$file_access : SET rms$file_access_names[..] SIZE (BYTE);

rms$file_share_names :(
    rms$c_shrdel,   ! allow delete access
    rms$c_shrget,   ! allow get access
    rms$c_shrput,   ! allow put access
    rms$c_shrupd,   ! allow update access
    rms$c_shrnil,   ! prohibit sharing
    rms$c_upi       ! user provided interlocking (allows
    );              ! a single writer and multiple readers to seq files)

 rms$file_share : SET rms$file_share_names[..] SIZE (BYTE);
```

1.3.1.4 Create Input Options

Input options to the *rms$create* service are passed as items in an item list. The item codes are defined in *rms$create_in_options_item_code*. The valid input options to the *rms$create* are shown below:

```
!+
!This enumerated type is used to define the input options item codes
!for Create service.
!-

rms$create_in_options_item_code :(
    rms$create_allocation_options,
    rms$create_protection_options,
    rms$create_filename_creation,
    rms$create_runtime_access,
    rms$create_file_characteristics,
    rms$create_set_expiration_date, ! set expiration date and time
    );
```

Each option that can be specified at file creation time is discussed in the following sections.

1.3.1.4.1 Allocation Options

Through the allocation options the RMS user can exercise additional control over file or area space allocation on disk devices, to optimize performance. In the following description, the terms *file* and *area* are synonymous for sequential and relative files, as these file organizations are limited to a single area.

If the allocation options are not used, the user file is created as a zero length file. However, at the time the first Put operation is performed, the file is automatically extended. The default extension size is equal to the track size of the device, on which the file resides.

The allocation options are defined by the following record:

```
!+
!The following record describes the valid allocation options
!for input to the Create service.
!-
rms$create_in_file_alloc (number_of_areas: INTEGER[0..254]): RECORD
    CAPTURE number_of_areas;
    default_extention : INTEGER[0..65535];
    block_count : ARRAY [0..number_of_areas] OF integer [1..1073741824];
    area_position : ARRAY [0..number_of_areas] OF file$file_alloc;
END RECORD;
```

```
!+
!The type file$file_alloc are defined by the
!Mica File system.  It is copied below for easy reference.
!-

!+
!These enumerated types and record are used to control the allocation of
!disk blocks.
!-

file$alloc_place: (
    file$c_place_cylinder,      !Allocate on specific cylinder.
    file$c_place_lblock,        !Allocate on specific logical block.
    file$c_place_vbn,           !Allocate on specific virtual block.
    file$c_place_rfi            !Allocate to related file.
    );

file$alloc_align: (
    file$c_align_cylinder,        !Align to cylinder.
    file$c_align_onsector         !Align to sector.
    );

!+
! Provide data for alignment options.
!-
file$place_data (placement: file$alloc_place): RECORD
     CAPTURE placement;
     VARIANTS CASE placement

     WHEN file$c_place_cylinder THEN                 !Data for align to cylinder.
        crvn:  [0..255]  SIZE(byte);                !Relative Volume Number.
        cylinder: integer [0..] SIZE(longword);     !Cylinder number.

     WHEN file$c_place_lblock THEN                   !Data for align to logical block.
        lrvn:  [0..255]  SIZE(byte);                !Relative Volume Number.
        lblock_num: integer [0..] SIZE(longword);   !Logical Block Number.

     WHEN file$c_place_vbn THEN                      !Data for align to virtual block.
        vblock_num: integer [0..] SIZE(longword);    !Virtual Block Number.

     WHEN file$c_place_rfi THEN                      !Data for align to related file.
        rfi_vbn: integer [0..] SIZE(longword);      !VBN of related file to align with.
        rfi_file_id: file$f11_file_id;              !Related file FID.

    END VARIANTS;
END RECORD;

!+
! This record holds all of the allocation information, and is
! the type passes as the allocation argument to the allocate
! and the create function

file$file_alloc: RECORD
    hard : boolean;                     ! error if can't alloc. as specified
    contiguous : boolean;               ! contiguous
    contiguous_best_try : boolean;      ! contiguous best try
    placement : file$alloc_place;       ! placement
    alignment : file$alloc_align;       ! alignment
    location : POINTER file$place_data; ! alignment data
 END RECORD;
```

Use of some of the fields of the record *rms$create_in_file_alloc* are described below:

- Default extension—This represents the quantity, in number of blocks, to be added to the file, when automatic extension is required.
- Block count—This indicates the number of blocks to be allocated for each area. For sequential files only one area is applicable.

- Area Position—Specifies placement control for each allocated area. The position control fields are described below:
  - The field *file$file_alloc.hard* indicates that if the requested alignment cannot be done, then an error is returned. By default, the allocation is performed as near as possible to the requested alignment.
  - The field *file$file_alloc.contiguous* indicates that the initial allocation extension must use contiguous blocks only. The allocation fails if the requested number of contiguous blocks is not available.
  - The *file$file_alloc.contiguous_best_try* indicates that allocation or extension should use contiguous blocks, on a "best effort" basis.
  - The *file$file_alloc.placement* indicates one of the following:
    - *file$c_place_cylinder*—Indicates that allocation is to begin on the specified cylinder.
    - *file$c_place_lblock*—Indicates that allocation is to begin on the specified logical block.
    - *file$c_place_vbn*—Applies to area extension only. This indicates that the area extension should begin as close to the virtual block number as specified in *file$place_data.vblock_num*.
    - *file$c_place_rfi*—Applies to area extension only. This indicates that the area extension is to start as close as possible to the file identified in the field *file$place_data.rfi_file_id*. The extent begins with the virtual block number specified in *file$extent_descriptor.starting_vbn*.
  - The *file$file_alloc.alignment* indicates one of the following:
    - *file$c_align_cylinder*—Align on cylinder boundary
    - *file$c_align_onsector*—Align on sector (track) boundary
- The *file$place_data* provides alignment options.
  - The field *crvn* represents the relative volume number upon which the file is to be allocated. This field corresponds to the VMS RMS field XAB$W_VOL. The field *cylinder* represents the cylinder number on the volume, at which the allocation is to start. This field corresponds to the VMS RMS field XAB$L_LOC, when in the XAB$B_ALN field, XAB$C_CYL option is specified.
  - The field *lrvn* represents the relative volume number upon which the file is to be allocated. This field corresponds to the VMS RMS field XAB$W_VOL. The field *lblock_num* represents the logical block number on the volume, at which the allocation is to start. This field corresponds to the VMS RMS field XAB$L_LOC, when in the XAB$B_ALN field, XAB$C_LBN option is specified.

1.3.1.4.2 File Protection Options

File protection options are used to specify ownership, accessibility and protection of a file. Presently, file protection options are not defined, as th Mica file system file protection mechanism is not yet specified.

1.3.1.4.3 Filename Creation Options

The following file-name options may be set while creating a file. A programmer can choose any or all of the options.

- Create If Nonexistent —Opens an already existing file.
- Maximize Version Number—Indicates that the version number of the file should be the maximum of the explicit version number given in the file specification, or one more than the highest version number for an existing file in the same directory with the same file name and file type. This option enables the user to create a file with a specific version number.
- Supersede—Allows an existing file to be superseded on creation. Existing files can be superseded by a new file of the same name, type and version. The *create_if* and *max_version* options take precedence over *supersede.*
- Temporary—Indicates that a temporary file is to be created and retained but no directory entry is made. After the file is closed, the only way to refer to the file is by way of the file ID (provided through the output record *quick_file_ref_out.*
- Temporary marked for delete—Indicates that a temporary file is to be created. The file is automatically deleted when it is closed.

By default, none of the above options is set. The consequences are:

- If the file is specified without explicit version number then the file is created, even if a file with the same name were present in the directory. In this case, the newly created file gets a higher version number. For example, if a file A.TXT;1 exists, and the user tries to create A.TXT, and the create if nonexistant flag is not set, the file A.TXT;2 is created.
- The version number of the created file is not maximized. For example, if a file A.TXT;2 exists, and the user tries to create a file A.TXT;2 the file creation attempt fails. On the other hand, if the option maximize version number were set, the same file creation attempt succeeds, and the file A.TXT;3 is created for the user.
- The files are not superseded. If a file with the same name, type and version exists, the file creation attempt fails.
- By default permanent files are created. A directory entry is made for the file, and the file is not deleted when it is closed.

The filename creation options are specifed below:

```
!+
!This enumerated type is used to define the options for
!the create options
!-
rms$filename_creation_options: (
    rms$c_create_if,       ! create if non-existent
    rms$c_max_version,     ! maximize version number
    rms$c_supersede,       ! supersede
    rms$c_temporary,       ! temporary file
    rms$c_temp_marked_del ! temporary marked for delete
    );

rms$filename_creation : SET rms$filename_creation_options[..] SIZE(BYTE);
```

1.3.1.4.4 Run-time Access Options

The run-time access options are used to specify file desposition at the time the file is closed. The runtime access options are defined below:

- Truncate end-of-file—Indicates that the unused space allocated to a file is to be deallocated, when the file is closed.
- Delete on close—Indicates that the file will be deleted when closed.

```
rms$run_time_access_options : (
    rms$c_truncate_eof,
    rms$c_delete_on_close,
    );

rms$run_time_access : SET rms$run_time_access_options[..] SIZE(byte);
```

By default, none of the above options is set. Run-time access options are to be exercised explicitly.

1.3.1.4.5 File Characteristics

At file creation time, the following reliability oriented file characteristics can be specified:

- Read check—Specifies that transfers from disk volumes are to be checked by read-compare operations. By default, read check is not performed.
- Write check—Specifies that transfers to disk volumes are to be checked by read-compare operations. By default, write check is not performed.
- Force write through dates—Specifies that the read date and time field, and revision date and time field are updated on disk, every time the a record is read from or written to the file. By default, the fields are updated in memory by the Mica file system and written out only at file close time. Forced write through of date and time causes a severe performance penalty.

```
rms$file_characteristics_options : (
    rms$c_read_check,
    rms$c_write_check,
    rms$c_force_write_thru_dates
    );

rms$file_characteristics : SET rms$file_characteristics_options[..] SIZE(byte);
```

1.3.1.4.6 Set Expiration Date and Time

At file creation time, the expiration date and time field can be set. This date and time field is used only by the file owner. The Display service outputs expiration date and time.

```
rms$expiration_date_time : RECORD
    expiration_date : longword;
    expiration_time : integer;
END RECORD;
```

1.3.1.5 File Information

This is an optional output parameter. If the file is opened due to the filename creation option *rms$c_create_if*, then RMS returns through this record structure some file related information. The record is defined below:

```
rms$standard_file_info : RECORD
    device_characteristics : rms$device_characteristics;
    record_structure : rms$file_record_definition;
END RECORD;
```

The type *rms$file_record_definition* is specifed earlier in section Section 1.3.1.2. The device characteristics are defined below:

```
!+
!This enumerated type define the device characteristics as
!returned by Create and Open
!-
rms$device_char_names : (
    directory_structured,
    file_oriented,
    foreign,
    dev_read_check_enabled,
    dev_write_check_enabled,
    random_access,          !disk
    seq_block_oriented,     !magnetic tape
    terminal,               !terminal
    unknown                 !devices handled indirectly
    );

rms$device_characteristics: SET rms$device_char_names[..] SIZE (WORD);
```

1.3.1.6 Output File Specification

This optional output parameter returns to the user the resultant file specification. If RMS encounters an error while creating the file, the expanded file specification that was used by the Create service is returned via this record. The record is specified below. The record is structured to facilitate users to extract individual fields from the file specification string.

```
rms$file_reference : RECORD
    device_name_offset : integer [0..rms$c_max_length] SIZE(word);
    device_name_length : integer [0..rms$c_max_length] SIZE(word);
    number_of_dir_levels : integer [0..32] SIZE(word);
    dir_name_offset : integer [0..rms$c_max_length] SIZE(word);
    dir_name_length : integer [0..rms$c_max_length] SIZE(word);
    file_name_offset : integer [0..rms$c_max_length] SIZE(word);
    file_name_length : integer [0..rms$c_max_length] SIZE(word);
    extension_offset : integer [0..rms$c_max_length] SIZE(word);
    extension_length : integer [0..rms$c_max_length] SIZE(word);
    version_number : integer [0..rms$c_max_length] SIZE(word);
    file_specification : varying_string(rms$c_max_length);
END RECORD;
```

1.3.1.7 Output Quick File Reference

This optional output parameter provides both the file ID and the volume object ID. The record can be saved and used later to access the file through the file's file ID. See Section 1.3.1.1.

1.3.1.8 File Handle

Mica RMS returns a *file_handle* after a successful *rms$create* operation. The user is required to use *file_handle* as an input argument for all future file and record operations. The *file_handle* is a pointer to a datastructure that is maintained and used only by RMS. The file context datastructure is a hidden type, and it is not visible to the user.

```
!+
!This is a declaration for RMS file handle, which points to a
!data-structure that maintains the file context.
!-

rms$file_handle : POINTER rms$file_context;
```

1.3.2 Open Service

The Open (*rms$open*) service allows an existing file to become available for processing. The procedure is defined below:

```
PROCEDURE rms$open(
            IN file_name : string(*) = "" ;
            IN default_file_string : string(*) = "";
            IN quick_file_ref_in : rms$file_ref_identifier OPTIONAL;
            IN access_control : rms$file_access_control;
            IN open_input_options : POINTER rms$item_list = NIL;
            OUT file_information : rms$standard_file_info OPTIONAL;
            OUT resultant_file : rms$file_reference OPTIONAL;
            OUT quick_file_ref_out :  rms$file_ref_identifier OPTIONAL;
            OUT file_handle : rms$file_handle;
            ) RETURNS status;
```

1.3.2.1 File Identification

The description in Section 1.3.1.1 is also applicable in this context. A file that is to be opened may be identified by any one of the following ways:

- File name string only
- File name string augmented by a default file name string
- File ID and volume object ID

If the file to be opened is identified by the file-name only, then the caller uses the *file_name* parameter. In this case, the caller can optionally specify the default file specification string. A file name string cannot have any embedded wildcard characters or a node name. The file name processing is done by RMS.

Alternatively, the caller uses the *quick_file_ref_in* parameter to specify the file ID and the volume object ID. In this case, no further file name processing is required.

1.3.2.2 Access Request

This record specifies the desired way the caller wishes to access the file and the way the caller wishes to share the file with other users. This is an optional input parameter to the *rms$open*. At file open time, the default value of file accessing is put access, and the default value for file sharing is allowing shared read. See Section 1.3.1.3.

1.3.2.3 Open Input Options

Input options to the *rms$open* service are passed as items of an item list. The item codes are defined in *rms$open_in_options_item_code*. The valid input options to the *rms$open* are shown below:

```
!+
!This enumerated type is used to define the input item codes
!for Open service.
!All items are prefixed "open"
!-
rms$open_in_options_item_code : (
    rms$open_file_characteristics,
    rms$open_runtime_access,
    );
```

At file open time, the file characteristics options that can be specified are described in Section 1.3.1.4.5. The defaults values at file open time are the same as the defaults values at file create time.

At file open time, the run-time access options that can be set are described in Section 1.3.1.4.4. The defaults values at file open time are the same as the defaults values at file create time.

1.3.2.4 File Information

Some file-related information is returned through this optional output parameter. See Section 1.3.1.5.

1.3.2.5 Resultant File

This optional output parameter is used to return the resultant file specification of the opened file. See Section 1.3.1.6.

1.3.2.6 Output Quick File Reference

The file ID and volume object ID are returned through this optional output parameter. See Section 1.3.1.7.

1.3.2.7 File Handle

After a successful open operation, *rms$open* provides a file handle.

1.3.3 Close Service

The Close (*rms$close*) service terminates file processing and closes the file. If the file was created or opened with the option to *delete on close*, or the option is set in the *rms$close* service, the file is deleted as well. If the file is not deleted on close, then buffers that were not yet written are written out. All the buffers allocated for the file are deallocated. The caller can modify the file protection, and ownership of the file by specifying the appropriate options fields.

```
PROCEDURE rms$close(
            IN OUT file_handle : rms$file_handle;
            IN in_options : POINTER rms$item_list = NIL;
            ) RETURNS status;
```

1.3.3.1 File Identification

The input parameter *file_handle* is the file handle that was provided by *rms$open* service. After closing the file, RMS clears the file handle.

1.3.3.2 Input Options

The following input options are valid.

```
!+
! The following items are input to Close.
!-

 rms$close_in_options_item_code :(
     rms$close_disposition_options,
     rms$close_protection_options
     );

 !+
 !the datastructures used by the items for Close service
 !-

 rms$close_desposition : SET rms$run_time_access_options[..] SIZE(byte);
 rms$close_protection : rms$file_protection_options;
```

1.3.3.2.1 Close Disposition Options

The following file disposition option can be used at file close time:

- Truncate end of file—See Section 1.3.1.4.4
- Delete on close—See Section 1.3.1.4.4.

1.3.3.2.2 Close Protection Options

Please see Section 1.3.1.4.2. The file protection option that can be set at file close time is TBS.

1.3.4 Data Retrieval and Output Services

Mica RMS data retrieval and output operations are designed to minimize the number of decision points at run time. There are several retrieval decisions that are based upon static file charateristics. For example, the file organization does not change. The static class of file attributes are available to RMS once the caller opens a file. This technique effectively uses available information, while eliminating further query.

There is another class of file attributes that are generally unpredictable, and occur at the time of data retrieval. For example, record access mode may be changed (from RFA to sequential), or the retrieval operation is switched to a data output operation (switch from Get to Put). The implementation strategy is to consider all the variations and opt for a path of least decision making.

In Mica RMS, data retrieval services are classified according to the type of record access operation. Thus, there are three different access routines based upon sequential, RFA or key access. Further, there can be three different kinds of operations, (Get, Put or Find). As for each of the I/O operations any of the three access modes is permissible, nine possible options are available. Note, Mica RMS offers only synchronous I/O operations.

The nine data retrieval options apply to all three types of file organization (sequential, relative or indexed) and also to the three types of devices (disk, magnetic tape or terminal). Hence, the options rise to eighty-one. Then there are options on record formats (Fixed, VFC, STM, STMCR, STMLF, UDF or Variable). This raises the options to five hundred and sixty seven. However, not all of the options are legal. For example, it does not make sense to do an indexed file operation on a terminal device.

In addition to the record access options described in the preceding paragraphs, a user may wish to use blocks of data or just characters for conducting I/O operations. Support for block level and character level I/O is TBD for Mica RMS.

Much of the above described complexity is transparent to RMS users. For instance, RMS could offer different procedures to its users, based upon the operation and access methods. The addresses of the appropriate procedures are contained in the vector *rms$retrieval_serv_vec*. RMS builds the vector of RMS data retrieval service routines at file open time, based upon the static file attributes. Thus, the user can enter a *rms$get_sequential* access routine, knowing that the device is a disk, and the file is sequentially organized. Within this *rms$get_sequential* access routine there are no tests done to check for the device type, file organization, record format or any other statically known option. Thus, there are many *rms$get_sequential* routines; the one being used depends upon the combination of the file static attributes. The vector of data retrieval and output services may be organized as:

- Get sequential
- Put sequential
- Find sequential
- Get RFA
- Find RFA

- Get key
- Put key
- Find key

The basic difference between the Get and Find services is that the Get service retrieves data, indicates the length of the record, the record itself and the record file address. On the other hand, the Find service locates the specified record and returns the record's file address. Because the Find service is a subset of Get, the interfaces are merged. In the Get service, a Find option is set to indicate the Find service. Thus, the number of vector entries is reduced to five entries.

In Mica RMS, data retrieval defaults to locate mode. That is, users need to explicitly specify move mode. The Get services also returns the next record pointer to the user after every successful Get operation. This facilitates the user to keep multiple record contexts without requiring multiple connects. Further, Mica RMS is not required to keep the user's record contexts.

The address of each data retrieval service applicable to the file is placed in a specific vector slot. Each of the services may thereafter be accessed by referencing the appropriate vector slot.

The procedure types of the data retrieval routines are individually described below. Each of the procedure types has a *ptype* prefix.

1.3.5 Get Sequential

The Get Sequential (*rms$ptype_get_sequential*) interface is used to access records sequentially. All types of file organizations and devices can use this data retrieval mode.

```
TYPE
     rms$ptype_get_sequential: PROCEDURE(
         IN file_handle : rms$file_handle;
         IN record_position : POINTER anytype OPTIONAL;
         IN user_in_buffer_pointer : POINTER anytype CONFORM;
         IN user_in_buffer_length : integer;
         IN move_mode : boolean = FALSE;
         IN in_options : POINTER rms$item_list = NIL;
         OUT current_record_pointer : rms$record_file_address OPTIONAL;
         OUT next_record_position : POINTER anytype OPTIONAL;
         OUT read_data_buffer_pointer : POINTER anytype;
         OUT read_data_length : integer;
         ) RETURNS status;

    rms$record_file_address: RECORD
        UNION CASE *
            WHEN 1 THEN
                record_descriptor: word_data(3);
            WHEN 2 Then
                record_vbn : longword;
                record_offset : integer [0..65535];
        END UNION;
        LAYOUT
            UNION
                OVERLAY
                    record_descriptor ALIGNMENT (WORD);
                OVERLAY
                    record_vbn ALIGNMENT (WORD);
                    record_offset ALIGNMENT (WORD);
            END UNION;
        END LAYOUT;
    END RECORD;
```

The parameters of the procedure *rms$get_sequential* are described in the following sections.

1.3.5.1 File Identification

The caller supplies the *file_handle* which was provided earlier by *rms$open* service.

1.3.5.2 Record Position

This optional input parameter is used to identify the record that needs to be retrieved. The user returns in this field the record that was provided by RMS through the output parameter *next_record_position*. If *record_position* is not provided, then Mica RMS retrieves records in the following steps:

- If the file organization is sequential, then the record stored in the next sequential order, relative to the last record accessed is returned.
- RMS action for indexed and relative files are TBS.

1.3.5.3 User Input Buffer

The user input buffer is specified by two required input parameters: *user_in_buffer_pointer* and *user_in_buffer_length*. RMS moves the record into this user specified buffer if either the user has specified *move_mode* or the record being retrieved, crosses block boundaries (and records crossing block boundaries is permitted).

1.3.5.4 Move Mode

The user can force records to be moved to a user specified input buffer by setting this input parameter to TRUE. By default, Mica RMS uses locate mode.

1.3.5.5 Input Options

Following are valid input options for the Get operation:

```
!+
!This enumerated type is used to define the input options item codes
!for Get service.
!All items are prefixed "get"
!-

rms$get_in_options_item_code :(
    rms$get_find_operation,           ! just find the record
    rms$get_record_header_definition, ! for VFC format
    rms$get_basic_terminal_options,   ! for terminals at client site
    rms$get_record_locking_options,   ! not used presently
    rms$get_key_ref_definition,       ! not used presently
    rms$get_index_file_options        ! not used presently
    );
```

1.3.5.5.1 Find Operation

If this option is chosen, then a find operation is done. This option does not require any buffer space to qualify the option.

1.3.5.5.2 Record Header Definition

This field is used to specify the fixed-control area of a file with VFC record format. The fixed-control area allows the user to include within the record additional data that may have no direct relationship to other contents of the record. For example, the fixed-control area may contain line-sequence number for every record in the file.

1.3.5.5.3 Basic Terminal Options

The basic terminal options for reading data from a client site are:

- Uppercase—Changes characters to uppercase on a read from a terminal
- Prompt option—The contents of the prompt buffer are to be used as a prompt for reading data from a terminal
- Purge type ahead—Eliminates any information that may be in the type-ahead buffer on a read from a terminal
- Read no echo—Input data is not echoed on the terminal
- Read no filter—Indicates CTRL/U, CTRL/R and DELETE are not to be considered as control commands on terminal input
- Timeout—Specifies the number of seconds to wait between characters being typed

The basic terminal options for *rms$get_sequential* and *rms$put_sequential* are set by the following record. Note, if prompt option is set, then *rms$basic_terminal_options.prompt_buffer* contains the prompt character string. The prompt character string is output to the terminal before the *rms$get_sequential* is performed. If the timeout option is set, then *rms$basic_terminal_options.timeout_period* contains the delay time in seconds.

```
!+
!The basic terminal options are set through this record structure
!-
rms$basic_terminal_options : RECORD
    prompt_buffer : longword_data(1);
    timeout_period : integer[0..255] SIZE(byte);
    term_control : rms$set_terminal_control;
END RECORD;

rms$set_terminal_control_names : (
    cancel_control_o,
    upcase_input,
    read_with_prompt,
    purge_type_ahead,
    read_no_echo,
    read_no_filter,
    read_with_timeout
    );
rms$set_terminal_control : SET rms$set_terminal_control_names[..] SIZE(byte);
```

1.3.5.5.4 Key Reference

The key reference contains a key value for an indexed file. The use and structure is TBS.

1.3.5.5.5 Record Locking Options

Record locking options are not defined presently, and are not available at FRS.

1.3.5.5.6 Indexed File Options

Indexd file options are not defined presently, and are not available at FRS.

1.3.5.6 Current Record Pointer

Upon a successful *get_sequential* operation, RMS returns the current record's virtual block number and the offset. This is an optional output parameter.

1.3.5.7 Next Record Position

RMS returns through this optional output parameter information to facilitate retrieval of the next record (next, relative to the current record). The format for this record is not yet specified.

1.3.5.8 Read Data Buffer

In the default locate mode, RMS sets the output parameter *read_data_buffer_pointer* to point to the beginning of the data, retrieved in the call. The output parameter *read_data_length* specifies the length of the retrieved data. If move mode is set, then these fields have the same values as *user_in_buffer_pointer* and *user_in_buffer_length* respectively.

/****** An implementation note **********/

The above is an example of *get_sequential* procedure type. All the sequential read procedures are based upon this procedure type. The convention is to name the procedures based upon the static file attributes. The attributes are delimited by an underscore character. Thus, the procedures are named as: *OP_ACC$ORG_DEV_FMT*. Where OP is the operation (Get or Put); ACC is the record access mode (sequential, RFA or key); ORG is the file organization (sequential, relative or indexed); DEV is the device type (disk, mag tape or terminal); FMT is the record format (fixed length, STM, STMCR, STMLF, UDF, variable length or VFC). Thus, there are procedures that appear as:

```
PROCEDURE get_seq$seq_dsk_vfc(
              file_handle,
              record_position,
              user_in_buffer_pointer,
              user_in_buffer_length,
              move_mode,
              in_options,
              current_record_pointer,
              next_record_position,
              read_data_buffer_pointer,
              read_data_length
              ) OF  TYPE rms$ptype_get_sequential;
              EXTERNAL;
```

The vector that contains the procedure variables is defined as:

```
rms$retrieval_serv_vec: RECORD
    get_sequential    : rms$ptype_get_sequential;
    put_sequential    : rms$ptype_put_sequential;
    get_rfa           : rms$ptype_get_rfa;
    get_key           : rms$ptype_get_key;
    put_key           : rms$ptype_put_key;
END RECORD;
```

Once a file is opened, the data retrieval service vector can be initialized. For example:

```
rms$retrieval_serv_vec.get_sequential = get_seq$seq_dsk_vfc;
```

/****** End implementation note **********/

1.3.6 Get Random by RFA

Get random by RFA (*rms$ptype_get_rfa*) access mode is used to retrieve records by directly specifying the record's address within the file. The service returns the *next_record_position*, which may be used in a subsequent sequential access mode.

```
TYPE
     rms$ptype_get_rfa : PROCEDURE(
         IN file_handle : rms$file_handle;
         IN current_record_pointer : rms$record_file_address;
         IN user_in_buffer_pointer : POINTER anytype CONFORM;
         IN user_in_buffer_length : integer;
         IN move_mode : boolean = FALSE;
         IN in_options : POINTER rms$item_list;
         OUT next_record_position : POINTER anytype OPTIONAL;
         OUT read_data_buffer_pointer : POINTER anytype;
         OUT read_data_length : integer;
         ) RETURNS status;
```

The interface for *rms$ptype_get_rfa* is similar to *rms$ptype_get_sequential*. The major difference being the use of the parameter *current_record_pointer*. In this case, *current_record_pointer* is the required input parameter, which is used to retrieve the record. Locate mode is the default mode of access. Based upon the file's organization, RMS returns a *next_record_position*, which can be used as input for subsequent sequential accesses.

The Input Options, *in_options*, specified for *rms$ptype_get_sequential* are vaild for *rms$ptype_get_rfa*, except for:

- Key reference field and index file options are not applicable
- Terminal options are not applicable in this mode of access

1.3.7 Get Random by Key

Records may be accessed by specifying a "key" value. For sequential files with fixed records, and relative files, a relative record number is specified. This mode of data retrieval is most meaningful for indexed files. For indexed files, the record structure *isam_key* specifies the key definitions to the data retrieval service. However, indexed file operations are not specified for Mica RMS presently. Data retrieval operations on indexed files are not available at FRS.

```
TYPE
     rms$ptype_get_key: PROCEDURE(
         IN file_handle : rms$file_handle;
         IN relative_record_number : integer OPTIONAL;
         IN isam_key : POINTER rms$key_definition OPTIONAL;
         IN user_in_buffer_pointer : POINTER anytype CONFORM;
         IN user_in_buffer_length : integer;
         IN move_mode : boolean = FALSE;
         IN in_options : POINTER rms$item_list = NIL;
         OUT current_record_pointer : rms$record_file_address OPTIONAL;
         OUT next_record_position : POINTER anytype OPTIONAL;
         OUT read_data_buffer_pointer : POINTER anytype;
         OUT read_data_length : integer;
         ) RETURNS status;
```

Most of the parameters of Get Key (*rms$ptype_get_key*) are the same as the parameters and options defined in the *rms$ptype_get_sequential*, and are not repeated here. The differences are noted below:

The interface for *rms$ptype_get_key* provides explicit parameter for defining the relative record number (*relative_record_number*) for sequential or relative files. The optional input parameter *relative_record_number* is used for the random access of sequential or relative files. Sequentially organized files having fixed length records can be retrieved by the *relative_record_number* value, which in this case represents the record number (records are numbered in ascending order, starting with number 1).

For indexed sequential file keys are defined by the record pointed by the *isam_key*. The record structure *rms$key_definition* has not yet been specified. The optional input parameter *isam_key* is valid only for indexed files.

In this mode of access, terminal options are not valid.

1.3.9.4 Input Options

Following are valid input options for *rms$put_sequential*:

- Disposition options—For sequential access, only *truncate_on_put* option can be set.
- Record header definition—This is defined below.
- Basic terminal options—The options are for the terminals connected to the client.

The Put service (both sequential and keyed) input options item list is defined below:

```
!+
!This enumerated type is used to define the input options item codes
!for Put service.
!All items are prefixed "put"
!-

rms$put_seq_item_code :(
    rms$put_disposition,
    rms$put_record_header,          ! for VFC format
    rms$put_basic_terminal,         ! for terminals at client site
    rms$put_record_lock             ! not used presently
    );
```

1.3.9.4.1 Put Disposition Options

Through this input item, the user may specify the following data output time file despositions:

- Truncate on put—This option specifies that in the sequential record output mode, data may be palced anywhere in the file. The file is truncated at the point immediately after the output record. The end-of-file mark is reset to the new position. This option is used only in *rms$ptype_put_sequential* type procedures.
- Update if—This option allows the user to overwrite a record in a sequential file that is being accessed randomly by the relative record number. This option is only used in *rms$ptype_put_key* type procedures.

1.3.9.4.2 Record Header Definition

This field is used to specify the fixed-control area of a file with VFC record format. The fixed-control area allows the user to include within the record additional data that may have no direct relationship to other contents of the record. RMS writes the contents of the specified buffer to the file as the fixed-control area portion of the record.

1.3.9.4.3 Basic Terminal Options

The basic options for writing data to a terminal connected to a client system are:

- Cancel control/O—Guarantees that terminal output is not discarded if the operator presses CTRL/O
- Timeout —Specifies the number of seconds to wait between characters being typed

The above options are selected on the record described in Section 1.3.5.5.3. Note, if the timeout option is selected, the field *rms$basic_terminal_options.timeout_period* is set to indicate the allowed dealy in number of seconds.

1.3.8 Put Services

The Put (*rms$put*) service adds a record to the file. The user provides a buffer and the length indicating the record that is to be added. The records are placed at the end of sequential files. Put operations on relative and indexed files are not defined presently.

1.3.9 Put Sequential

The *rms$put_sequential* record access mode service may be used to insert records for sequential, relative or indexed files.

For sequential files, the *rms$put_sequential* service inserts records at the end of the file. However, records can be inserted in locations other than the end-of-file, *truncate_on_put* is set. When the record is inserted, the file is automatically truncated to a new end-of-file. The new end-of-file is the position immediately after the inserted record. If both, the file disposition option *truncate_on_put* and the file access mode *rms$c_truncate* are not set, then records cannot be inserted at locations other than at the end of the file.

The Put service initializes the internally maintained next record position at the end-of-file. If the position where the record is to be inserted is not specified, RMS inserts the record as defined in the next record position. If the next record position is not the end-of-file (for example, in between the two Put operations, the user has done a random Get, which altered the next record position), then record output operation fails unless *truncate_on_put* and the file access mode *rms$c_truncate* are set.

Record insertion operations on indexed and relative files are not available at FRS.

This generic interface is used to write records. The file organization, the record format, the device type have been resolved prior to the Put operation. The interface to the service is described below.

```
TYPE
     rms$ptype_put_sequential: PROCEDURE (
         IN file_handle : rms$file_handle;
         IN data_out_buffer_pointer : POINTER anytype;
         IN data_out_buffer_length : integer;
         IN record_position : POINTER anytype OPTIONAL;
         IN in_options : POINTER rms$item_list;
         OUT current_record_pointer : rms$record_file_address OPTIONAL;
         OUT next_record_position : POINTER anytype OPTIONAL;
         ) RETURNS status;
```

The parameters of the procedure *rms$put_sequential* are described below.

1.3.9.1 File Identification

The caller supplies the *file_handle* which was provided earlier by *rms$open*.

1.3.9.2 User Output Buffer

User specifies the data output buffer through the input parameters *data_out_buffer_pointer* and *data_out_buffer_length*.

1.3.9.3 Record Position

This optional parameter is used to specify a location where the record is to be inserted. If this field is specified, and the record position is not the same as the end-of-file position, then both, the file disposition option *truncate_on_put*, and the file access mode *rms$c_truncate* must be set. If these are not set, the record ouput operation fails.

1.3.9.5 Current Record Pointer

RMS returns the current record's file address through this optional output parameter.

1.3.9.6 Next Record Position

This optional output parameter provides a position context for the next record.

1.3.10 Put Key

The Put Key (*rms$ptype_put_key*) service is used to insert records randomly by relative record number into sequential files. Operations on indexed and relative files are TBS.

For sequential files records are usually inserted at the end of the file. However, records may be inserted randomly by relative record number on a disk resident sequential file with fixed length record format, if the file disposition option *update_if* is set and the file access mode *rms$c_update* is set.

```
TYPE
     rms$ptype_put_key: PROCEDURE (
          IN file_handle : rms$file_handle;
          IN data_out_buffer_pointer : POINTER anytype;
          IN data_out_buffer_length : integer;
          IN relative_record_number : integer OPTIONAL;
          IN in_options : POINTER rms$item_lists;
          OUT current_record_pointer : rms$record_file_address OPTIONAL;
          OUT next_record_position : POINTER anytype OPTIONAL;
          ) RETURNS status;
```

The input parameters *file_handle*, *data_out_buffer_pointer* and *data_out_buffer_length* are previously described in *rms$put_sequential*. Please see Section 1.3.9.

1.3.10.1 Relative Record Number

This optional input parameter is used if the file organization is sequential or relative. See Section 1.3.7 for details.

1.3.10.2 Input Options

The input options are defined in in *rms$put_sequential*. See Section 1.3.9.4. The following input options are vaild for *rms$ptype_put_key*:

- Disposition options—For random access, only *update_if* option can be set.
- Record header definition

1.3.10.3 Current Record Pointer

RMS returns the current record's file address through this optional output parameter.

1.3.10.4 Next Record Position

This optional output parameter provides a position context for the next record.

1.3.11 Parse Service

The Parse (*rms$parse*) service analyzes a file specification and returns an expanded file specification through the output parameter *expanded_file*. It processes wildcard characters and stores the context for subsequent searches. By default, *rms$parse* also assigns a channel and performs a directory lookup. The Parse service can be used in one of the following modes:

- Syntax check only—This indicates that the file specification is checked for syntax validity without requiring any I/O processing to ensure that the device, directory and the file actually exists.
- Device check—This indicates that after doing the work for syntax check, *rms$parse* checks that the device exists. RMS also returns the device characteristics, and the volume object ID.
- File check—This indicates that after completing device check, RMS checks that the directory and the file exists. RMS returns the device characteristics, and the volume object ID. If there were no wildcards, then the file ID is also returned.

The procedure is defined below:

```
PROCEDURE rms$parse(
            IN file_name : string(*);
            IN default_file_string : string(*) OPTIONAL;
            IN related_files : POINTER rms$related_file_list = NIL;
            IN parse_option : rms$parse_option = rms$c_file_check;
            OUT device_characteristics : rms$device_characteristics OPTIONAL:
            OUT wild_card_ctx : POINTER anytype OPTIONAL;
            OUT expanded_file : POINTER rms$file_reference;
            OUT quick_file_ref_out : POINTER rms$file_ref_identifier OPTIONAL;
            OUT file_name_sts : rms$file_name_status;
            ) RETURNS status;
```

1.3.11.1 File Specification

The file name that is to be parsed is specified by the *file_name* parameter. This is a required input parameter, and is the primary file specification.

If the primary file specification does not contain all the components of a file specification, then defaults are applied to fill the missing components. The default file specification string is specified by the input parameter *default_file_string*. This is not a required input parameter.

If after applying the default file specification, a full file specification is not achieved, then the related file specification string is applied to fill in the missing directory, file name and file type fields. The related file specification is specified by the input parameter *related_files*. The *related_files* is a link list of related file specifications. This is not a required input parameter.

```
TYPE
    rms$related_file_list : RECORD
        related_file_specifications : varying_string(255);
        related_file_list_flink : POINTER rms$related_file_list;
    END RECORD;
```

1.3.11.2 Parse Options

One of the following parse options may be set:

- Syntax check only
- Device check
- File check

The default is to check for the file specification.

```
TYPE
    rms$parse_option : (
        rms$c_syntax_check,
        rms$c_device_check,
        rms$c_file_check
        );
```

1.3.11.3 Device Characteristics

This optional output parameter contains the device characteristics. See Section 1.3.1.5 for the description of the type *rms$device_characteristics*. RMS returns the device characteristics, if the parse option *rms$c_syntax_check* is not set.

1.3.11.4 Wild Card Context

The *wild_card_ctx* parameter points to a storage area which contains wildcard processing information for a subsequent *rms$search* operation.

1.3.11.5 Expanded File Specification

The output of the *rms$parse* service is primarily the *expanded_file* parameter. The record is specified in Section 1.3.1.6.

1.3.11.6 Quick File Reference

This optional output parameter is returned, if the parse option *rms$c_file_check* is set. If there were no wildcards in the file specification, then this record contains the file ID, as well as the volume object ID. This record can be used as an input to *rms$open* to open the file, without requiring filename processing.

1.3.11.7 File Name Status

This output parameter *file_name_sts* indicates status information about the file, as determined by the *rms$parse* service. This parameter is used as input to the *rms$search* service. The record structure that specifies the type *rms$file_name_status*, is not yet specified.

1.3.12 Search Service

The Search (*rms$search*) service scans a directory file specified within the *wildcard_context* area. The *wildcard_context* area has been set up by an earlier *rms$parse* service call. It is assumed that *rms$parse* saves the expanded file name within the wildcard context area. The *rms$search* service looks for entries that matches the file name, type and version number specified in the *wildcard_context*. Matched file entries are returned through the *matched_files* parameter. The *rms$search* service may be used to find a series of file specifications, whose names match a given file specification with wildcard characters. If there are no wildcard characters, then the file specified is matched.

```
PROCEDURE rms$search(
            IN OUT wildcard_context : POINTER anytype;
            OUT file_name_status : rms$file_name_status;
            OUT matched_files : POINTER rms$match_entries;
            ) RETURNS status;
```

1.3.12.1 Wildcard Context

This is a pointer to a context block which was built by *rms$parse*. The contents of the wildcard context block is not yet specified. Among other information, the wildcard context block contains the expanded file string, as well as context information for further search.

The context information for further search specifies the starting point within the specified directory from which to continue returning matched entries. This context is built by *rms$search*, if it were unable to return all the matched entries due to buffer overflow. A buffer overflow implies that the the buffer allocated to receive the matched entries was not adequate. The caller determines that a buffer overflow has occured by examining the output parameter *file_name_status*. In the case, where there is a buffer overflow, the caller simply reinvokes the *rms$search* service with the *wildcard_context*, to receive the balance matched entries. If so desired by the caller, this mechanism can be used to mimic the VMS RMS Search service behavior of returning one matched entry per invocation.

1.3.12.2 File Name Status

This output parameter contains status information about the file that is being matched by the *rms$search* service. The information returned by this output parameter has not yet been specified.

1.3.12.3 Matched Files

The output may contain zero, one or many matches. The entries that match the input file specification are returned in the array pointed by *matched_files*. If the buffer pointed by *matched_files* is allocated by the caller. If the buffer area overflows or no matches are found, *rms$search* service returns a suitable indication. The definition of the buffer that contains the matched files is shown below:

```
rms$match_entries (number_of_entries : INTEGER[1..65535]) : RECORD
    CAPTURE number_of_entries;
    files : ARRAY [1..number_of_entries] OF rms$file;
END RECORD;
```

1.3.13 Display Service

The Display (*rms$display*) service returns various file and record attributes. A file must be open for access by *rms$create* or *rms$open* before *rms$display* can be invoked. The file and record attributes for which information is desired may be specified by the various options listed in the *outputs* item list. RMS returns the file ID and the volume object ID, via the *quick_file_ref_out* parameter.

```
PROCEDURE rms$display(
            IN file_handle : rms$file_handle;
            IN outputs : POINTER rms$item_list;
            OUT quick_file_ref_out : POINTER rms$file_ref_identifier OPTIONAL
            ) RETURNS status;
```

1.3.13.1 File Identification

The file is referenced by the *file_handle* parameter.

1.3.13.2 Output Options

The valid output options are described below. The fields have been defined individually in *rms$create*.

```
!+
! The output items for the Display service is shown below.
!-

rms$display_out_options_item_code :(
    rms$display_allocation_options,
    rms$display_protection_options,
    rms$display_date_time_options,
    rms$display_file_header_definitions,
    rms$display_magtape_options,          !not defined presently
    rms$display_key_definitions           !not defined presently
    );
```

1.3.13.2.1 Allocation Options

The values of the following fields are returned:

- Allocation quantity
- Default extension quantity

The record structure for displaying the allocation quantities is shown below:

```
rms$display_allocation : RECORD
    allocation_quantity : integer[0..] SIZE(longword);
    default_extention : integer[0..65535];
END RECORD;
```

1.3.13.2.2 Protection Options

Please see Section 1.3.1.4.2. The information that is returned by the Display service is TBS.

1.3.13.2.3 Date and Time Options

The following date and time values are returned. The data structure for date and time options is defined below.

- Backup date and time
- Creation date and time
- Expiration date and time
- Revision date and time
- Read date and time
- Header write date and time

```
!+
!This record defines the date and time options.
!This record structure is used as output option item of Display.
!-

rms$date_time_options: RECORD
    revision_number : INTEGER [0..65535] SIZE(WORD);
    filler_1 : INTEGER [0..65535] SIZE(WORD);
    UNION CASE *
        WHEN 1 THEN
            revision_date_time: large_integer;
        WHEN 2 THEN
            revision_date : longword;
            revision_time : integer;
    END UNION;

    UNION CASE *
        WHEN 1 THEN
            creation_date_time: large_integer;
        WHEN 2 THEN
            creation_date : longword;
            creation_time : integer;
    END UNION;

    UNION CASE *
        WHEN 1 THEN
            expiration_date_time : large_integer;
        WHEN 2 THEN
            expiration_date : longword;
            expiration_time : integer;
    END UNION;

    UNION CASE *
        WHEN 1 THEN
            backup_date_time : large_integer;
        WHEN 2 THEN
            backup_date : longword;
            backup_time : integer;
    END UNION;

    UNION CASE *
        WHEN 1 THEN
            read_date_time : large_integer;
        WHEN 2 THEN
            read_date : longword;
            read_time : integer;
    END UNION;
```

```
        UNION CASE *
            WHEN 1 THEN
                header_write_date_time : large_integer;
            WHEN 2 THEN
                header_write_date : longword;
                header_write_time : integer;
        END UNION;

        LAYOUT
            revision_number;
            filler_1;
            UNION
                OVERLAY
                    revision_date_time ALIGNMENT (LONGWORD);
                OVERLAY
                    revision_date;
                    revision_time;
            END UNION;

            UNION
                OVERLAY
                    creation_date_time ALIGNMENT (LONGWORD);
                OVERLAY
                    creation_date;
                    creation_time;
            END UNION;
            UNION
                OVERLAY
                    expiration_date_time ALIGNMENT (LONGWORD);
                OVERLAY
                    expiration_date;
                    expiration_time;
            END UNION;

            UNION
                OVERLAY
                    backup_date_time ALIGNMENT (LONGWORD);
                OVERLAY
                    backup_date;
                    backup_time;
            END UNION;

            UNION
                OVERLAY
                    read_date_time ALIGNMENT (LONGWORD);
                OVERLAY
                    read_date;
                    read_time;
            END UNION;

            UNION
                OVERLAY
                    header_write_date_time ALIGNMENT (LONGWORD);
                OVERLAY
                    header_write_date;
                    header_write_time;
            END UNION;

        END LAYOUT;
    END RECORD;
```

1.3.13.2.4 File Header Characteristics

The file header characteristics are returned by the following record structure.

```
!
! This record defines the file header characteristics
!
rms$file_head_characteristics : RECORD
    UNION CASE *
        WHEN 1 THEN
            file_org_n_rec_format : byte;
            raw_record_attributes : byte_data(1);
            longest_record_length : word;
            raw_highest_virtual_block : byte_data(4);
            raw_end_of_file_block : byte_data(4);
            first_free_byte : word;
            fhc_fill_1 : byte_data(1);
            vfc_header_size : byte;
            max_record_size : word;
            default_extention_qty : word;
            fhc_fill_2 : word_data(1);
            fhc_fill_3 : byte_data(8);
            version_limit : word;
            start_lbn_if_ctg : longword;

        WHEN 2 THEN
            record_attribute_ftn : bit;
            record_attribute_cr : bit;
            record_attribute_prn : bit;
            record_attribute_blk : bit;
            highest_virtual_block_0 : word;
            highest_virtual_block_2 : word;
            end_of_file_block_0 : word;
            end_of_file_block_2 : word;
    END UNION;
    LAYOUT
        UNION
            OVERLAY
                file_org_n_rec_format ALIGNMENT(BYTE) POSITION(bit,0);
                raw_record_attributes ALIGNMENT(BYTE) POSITION(bit,8);
                longest_record_length ALIGNMENT(BYTE) POSITION(bit,16);
                raw_highest_virtual_block ALIGNMENT(BYTE) POSITION(bit,32);
                raw_end_of_file_block ALIGNMENT(BYTE) POSITION(bit,64);
                first_free_byte ALIGNMENT(BYTE) POSITION(bit,96);
                fhc_fill_1 ALIGNMENT(BYTE) POSITION(bit,112);
                vfc_header_size ALIGNMENT(BYTE) POSITION(bit,120);
                max_record_size ALIGNMENT(BYTE) POSITION(bit,128);
                default_extention_qty ALIGNMENT(BYTE) POSITION(bit,144);
                fhc_fill_2 ALIGNMENT(BYTE) POSITION(bit,160);
                fhc_fill_3 ALIGNMENT(BYTE) POSITION(bit,176);
                version_limit ALIGNMENT(BYTE) POSITION(bit,240);
                start_lbn_if_ctg ALIGNMENT(BYTE) POSITION(bit,256);

            OVERLAY
                fhc_record_filler_1 : FILLER(bit,*);
                record_attribute_ftn POSITION(bit,8);
                record_attribute_cr POSITION(bit,9);
                record_attribute_prn POSITION(bit,10);
                record_attribute_blk POSITION(bit,11);
                fhc_record_filler_2 : FILLER(bit,*);
                highest_virtual_block_0 POSITION(bit,32);
                highest_virtual_block_2 POSITION(bit,48);
                end_of_file_block_0 POSITION(bit,64);
                end_of_file_block_2 POSITION(bit,80);
```

```
            END UNION;
        END LAYOUT;
    END RECORD;
```

1.3.13.3 Quick File Reference

This output parameter contains the file ID of the file, and the volume object ID, on which the file resides.

1.3.14 Erase Service

The Erase (*rms$erase*) service deletes a disk file and removes the file's directory entry as specified in the path to the file. The file must be closed before it can be deleted. The *rms$close* service can also delete a file if the delete on close option was set. The *rms$erase* service returns the erased file's fully qualified file name through *erased_file* parameter.

```
PROCEDURE rms$erase(
            IN file_name : STRING(*) OPTIONAL;
            IN default_file_string : string(*) OPTIONAL;
            IN quick_file_ref_in :  rms$file_ref_identifier OPTIONAL;
            OUT erased_file : POINTER rms$file_reference OPTIONAL;
            ) RETURNS status;
```

1.3.14.1 File Specification

The file that is to be erased maybe specified by the *file_name* parameter, if the file ID is unknown to the user. If the *file_name* does not contain all the components of a file specification, then defaults are applied to fill the missing components. The default file specification string is specified by the input parameter *default_file_string*. This is not a required input parameter.

Alternatively, if the file ID is known, the *quick_file_ref_in* parameter may be used. The use of this input parameter elimates the need for filename processing in the *rms$erase* service.

1.3.14.2 Erased File Specification

The erased file's specification is returned by the optional output parameter *erased_file*. See Section 1.3.1.6 for defintion of the record structure.

1.3.15 Flush Service

The Flush (*rms$flush*) service writes out all modified I/O buffers and file attributes associated with the file.

```
PROCEDURE rms$flush(
            IN file_handle    : rms$file_handle;
            ) RETURNS status;
```

1.3.15.1 File Identification

The user provides the *file_handle*, which was provided earlier by *rms$open* service.

1.3.16 Free and Release Services

The Free (*rms$free*) service unlocks all records that were previously locked. The Release (*rms$release*) service unlocks the record specified by the contents of the record's file address. The locking and unlocking functions are not presently supported, both *rms$free* and *rms$release* are unavailable at FRS.

1.3.17 Rewind Service

The Rewind (*rms$rewind*) service sets the context of a record stream to the first record in the file. For sequential and relative files, *rms$rewind* service establishes the next-record position as the first record or the record cell in the file, regardless of the access mode. For indexed files, the next-record position is established at the first record of the current key of reference. The *rms$rewind* service performs an implicit Flush service. This operation cannot be performed on terminal devices as well as those devices that are accessed by way of the client context server.

```
PROCEDURE rms$rewind(
            IN file_handle : rms$file_handle;
            IN key_ref : rms$key_of_reference OPTIONAL;
            OUT next_record_position : POINTER anytype OPTIONAL;
            ) RETURNS status;
```

1.3.17.1 File Identification

The user specifies the file by the *file_handle* parameter.

1.3.17.2 Key Reference

This optional parameter is required for indexed files. The parameter contains a key value.

1.3.17.3 Next Record Position

The reference to the next record position is returned to the caller.

1.3.18 Truncate Service

The Truncate (*rms$truncate*) service applies to sequential files on disks or magnetic tapes only. The service deletes the record indicated as the current record, and all following records. The end-of-file indicator is set at the current record pointer. The *rms$truncate* service may immediately follow a successful *rms$get*, or *rms$update*. The file being truncated must not be accessed for block I/O.

```
PROCEDURE rms$truncate(
            IN file_handle : rms$file_handle;
            ) RETURNS status;
```

1.3.18.1 File Identifier

The user specifies the file by the *file_handle* parameter.

1.3.19 Update Service

The Update (*rms$update*) service modifies an existing record in a file. The record to be updated is to be retrieved by calling *rms$get* (with or without the *find* flag set). The *current_record_pointer*, as provided by the *rms$get* service, may be returned as the *record_position*. If the *record_position* is not supplied, RMS uses the internally maintained record position to update the record. As with the *rms$put* service, the user is required to provide a buffer descriptor holding the record that is to be updated. The user program is required to establish the current-record position before calling this service.
For sequential files, the record length of the update record cannot be different from the record being updated.

```
PROCEDURE rms$update (
            IN file_handle : rms$file_handle;
            IN record_position : POINTER anytype OPTIONAL;
            IN data_out_buffer_pointer : POINTER anytype;
            IN data_out_buffer_length : integer;
            IN in_options : POINTER rms$item_list;
            OUT next_record_position : POINTER anytype OPTIONAL;
            ) RETURNS status;
```

1.3.19.1 File Identification

The user specifies the *file_handle* parameter to identify the file.

1.3.19.2 Record Position

The *record_position* represents the record that is to be updated.

1.3.19.3 User Output Buffer

User specifies the data output buffer through the input parameters *data_out_buffer_pointer* and *data_out_buffer_length*.

1.3.19.4 Input Options

Please see Section 1.3.9.4. The following sections describe the valid input options to the *rms$update* service.

1.3.19.4.1 Record Header Buffer

This buffer contains the descriptor of the record (VFC format only) header buffer.

1.3.19.4.2 Record Locking Options

Record locking options are not available at FRS.

1.3.19.5 Next Record Position

RMS returns through this optional output parameter information to facilitate retrieval of the next record (next, relative to the current record). The format for this record is not yet specified.

1.4 Algorithms for File Management

This section outlines a sample I/O request through Mica RMS. The section also includes the major steps followed by a few of the services specified in the previous section.

1.4.1 Sample I/O Request Flow

1. An application calls *rms$create* to create a file MYFILE.TXT.
2. The *rms$create* service processes the file name and determines that:
   - The volume name is MYVOL
   - The directory in which the file is to be created is BETA. Directory ID of BETA is 4798,11,0
   - The file name is MYFILE.TXT
3. The *rms$create* service calls *exec$translate_object_name* with the volume name string MYVOL as input parameter, to obtain the FPU object ID.
4. The *rms$create* service calls *exec$create_channel* with the FPU object ID as input, to obtain the channel object ID.
5. The *rms$create* service calls *exec$request_io* with input parameters channel ID, IOSB, function code *io$c_dfile_create*, the file name with the complete directory path and the file attribute list, to obtain the file ID of the file created. At this point, the file has no storage allocated to it.
6. The caller has specified storage allocation, therefore, the *rms$create* service calls *exec$request_io* with the function code *io$c_dfile_allocate_storage* to allocate space for MYFILE.TXT.
7. The *rms$create* service calls *exec$request_io* with the function code *io$c_dfile_access* to open the file
8. The *rms$create* service builds a client context, and returns a file handle to the user. A data retrieval vector has also been set up for all I/O operations. The vector entries are (for example):

```
get_seq$seq_dsk_var
put_seq$seq_dsk_var
get_rfa$seq_dsk_var
get_key$seq_dsk_var
put_key$seq_dsk_var
```

9. At this point, I/O operations can be done on the file. The user wants to write a record to the file, and calls rms$put_sequential procedure. Within the RMS procedure, the call is made to *put_seq$seq_dsk_var*. The user data is moved into a buffer area. After several user write operations, the buffer fills up, and is written out.
10. The *put_seq$seq_dsk_var* calls the I/O subsystem procedure *exec$request_io* with the function code *io$c_dfile_write* with the input parameters specifying the I/O channel object ID, IOSB, the VBN at which to start writing the data, the pointer to the data buffer in memory and the length of the buffer. The status is checked to see that the operation is successful.
11. The user closes the file by invoking *rms$close* service. The *rms$close* checks that there are no I/Os outstanding on the file, writes out the dirty buffers, and calls *exec$request_io* with the function code *io$c_dfile_deaccess* to close the file. The *rms$close* deletes the I/O channel by calling *exec$delete_object_id*. The file context area is deallocated, and the pointer to the file context area is initialized to nul.

1.4.2 Create Service

The Create service performs the following significant operations:

1. Allocates space for maintaining the file context.
2. Tests if the file organization is sequential. If not sequential, RMS cannot satisfy the request at FRS, so exits with an appropriate indication.
3. Processes the file name:
   a. If the argument *quick_file_ref_in* is present, then it performs the following steps:
      1. Checks to see if the volume's object ID has been provided (if not, then error exit)
      2. Checks to see that a file ID has been provided (if not, then error exit)
      3. Sets an appropriate status
   b. If the *quick_file_ref_in* argument is not present, the Create service uses the string passed as *file_name*. It calls an internal procedure to processes the file name by applying all name processing rules. Note, node names or wildcards are not allowed. This returns the file name in a record structure which can be used directly as an input parameter to the I/O subsystem. The file-name processing procedure also provides a status. The status may indicate that the file is to be opened by way of client call back support routines. That is, if after file name translation, it is determined through a status value that the access is to be made by way of client context server routines (*clientcs*). In this case, *rms$create* calls *clientcs$rms_open* in step 14. The procedure that processes file names also indicates if search lists are present.
4. If a search list is present, and if the user has set the create if nonexistent option, then the Create service performs the following steps:
   a. Tries to access the file. If successful, the Create service then sets an indicator that this Create call is now going to complete like an Open call. What this means is that the file already exists and RMS treats the call as though the user has called *rms$open*.
   b. Repeats the above steps until there are no more items in the search list (If the file is not found, no problem, just continue with Create).
5. Performs organization-specific checks. Only sequential files are handled at FRS. The basic checks for a sequential file are:
   a. For magnetic tape device (not supported at FRS) the Create service:
      1. Checks to ensure that records cannot cross block boundaries flag is set
      2. Sets the block size
   b. Sets EOF VBN = 1 and first free byte (FFB) = 0
   c. If the record format is fixed, ensures that the records are not longer than one block size
6. In order to request the underlying file system or DFS to create a file, the Create service supplies the following:
   a. A channel ID. To obtain a channel ID, the Create service calls the executive service Create Channel (specifying the volume ID). The volume ID is obtained from the the system service that translated volume name to volume ID. If the volume is not mounted, it causes an error and exits the procedure.
   b. Address of a IOSB block.
   c. Target directory entry (which is in the form of *file_entry*).
   d. A Write Attribute list. Form this list using the user supplied file attributes. If required, it uses defaults.

e. An access control structure.

f. A directory entry control structure.

g. A conditional create flag. The Create service sets this flag if the user has specified *create_if* (see Section 1.3.1.4.3).

7. Calls *exec$request_io* (function code = *io$c_dfile_create*). If this is successful, then it continues.

8. Performs allocation and placement controls from user specification. If the user has not specified area allocation quantity, the Create service uses the track size of the device as the intial allocation quantity. The file area is allocated by calling *exec$request_io* (function code = *io$c_dfile_allocate_storage*). If the call is successful, the file has been created.

9. Returns information back to the user as per user request. Returns the *file_handle*.

10. If the device type is magnetic tape (not supported at FRS), and if rewind on close is requested, rewinds the tape.

11. Saves the options that the user has requested to be performed when the file closes.

12. Sets the suitable data retrieval and output procedures for the file. That is, arm the data retrieval and output vector with the appropriate procedure variables.

13. If *clientcs$open* is called, checks the status return. If successful, returns to the user an appropriate status and the *file_handle*. In this case, arms the data retrieval and output vector with the appropriate get_sequential and put_sequential procedures.

14. The Create service sets RMS status.

1.4.3 Open Service

The Open service performs the following significant operations:

1. Allocates space for maintaining the file context.

2. Processes the file name:

   a. If the argument *quick_file_ref_in* is present, then it performs the following steps:

      1. Checks to see if the volume's object ID has been provided (if not, then error exit)

      2. Checks to see that a file ID has been provided (if not, then error exit)

      3. Sets an appropriate status

   b. If the *quick_file_ref_in* argument is not present, the Open service uses the string passed as *file_name*. It calls an internal procedure to processes the file name by applying all name processing rules. Note, node names or wildcards are not allowed. The procedure returns the file name in a record structure which can be used directly as an input parameter to the I/O subsystem. The file name processing procedure also provides a status. The status may indicate that the file is to be opened by way of client context server routines. That is, if after file name translation, it is determined through a status value that the access is to be made by way of client context server routines (*clientcs*). In this case, *rms$open* calls *clientcs$rms_open*.

3. Opens a channel to the volume.

4. Tries to access the file by calling the *exec$request_io* (function code = *io$dfile_access*). The Open service specifies the access and share constraints as specified by the user. If the I/O call is successful, then continues. Otherwise, the Open service checks to see if there is a search list.

5. If a search list is present, the Open service gets the next list item and tries to access the file. The Open service repeats the process until a file is found, or the search list is exhausted. If the file is not found, the Open service exits unsuccessfully, as the file is not found.
6. If the file is found, then the file attributes are known.
7. The Open service performs organization specific chores. Only sequential files are handled at FRS. For other types of file organizations, the Open service exits with appropriate indication. For sequential files, the Open service takes the following steps:
   a. Sets the end of file at VBN = 1, FFB = 0
   b. Saves the options for Close service in the file context area.
   c. If the device is a magnetic tape (not available at FRS), performs the magnetic tape specific checks and set eof postion.
8. Sets data retrieval and output vector.
9. Returns the output parameters, as requested by the user.
10.
11. The Open service sets RMS status.

1.4.4 Close Service

The Close service performs the following significant operations:

1. Checks to see if the file has any outstanding I/Os in progress (if so, this Close operation fails)
2. Checks if the operation is to be handled by calling *clientcs$close* routine (if required, calls *clientcs$close*)
3. Checks the file desposition on close options
4. If delete-on-close is set, then calls *exec$request_io* (function_code = *io$c_dfile_delete*)
5. Otherwise, the Close service writes out all the dirty buffers
6. Deaccesses the file by calling *exec$request_io* (function_code = io$c_dfile_deaccess)
7. Deassigns the I/O channel
8. Deallocates the file context area
9. Sets a nul value to the *file_handle*
10. The Close service sets RMS status.

1.4.5 Parse Service

The objective of *rms$parse* is to form a fully-qualified file specification, which is returned to the caller by the *expanded_file* record. The string provided in the *file_name* field is the primary file specification. The secondary file specifications are supplied by the *default_file_string* and *related_files*.

1. The input file specification is parsed to its constituent elements.
2. If the file specification contains only a name (without a terminating colon or period). This can be either a logical name or a file name. If it is a logical name then the following must be true:
   a. There must be no other file name elements
   b. The name must translate
3. Assuming the name to be a logical name, the Parse service attempts to translate the logical name, to obtain an equivalence string.
4. If the process has an associated client context (it is a bound process), then translation is first attempted at the client site. This is done through the client context server procedure *clientcs$rms_translate_logical_name*. If the name translates successfully at the client site, no further attempts are made to translate the name at the server site. If, however, the name is not translated at the client site, the name is then translated at the server site. The results of translations obtained from the client site is not used with the results from the server site. If the translation at the client site is successful, the equivalence string is reapplied for translation at the client site, until there are no more translations.
5. If the process is a free running process, then translation is attempted only at the server site. If the translation is successful, the equivalence string is reapplied for translation.
6. If the name cannot be translated then the name is assumed to be a file name. The Parse service processes the file name further by applying defaults and, if necessary, the related file specifications to form a fully specified file.
7. If translation from the client site returns an indication that the file is to be processed by way of the client context server procedures, then the file name parsing is completed. The file is a special file that needs to be handled by client context server procedures.
8. If the file specification has other consitituent parts, then it sequentially checks the following:
   a. If a device name is seen then it is set aside for processing after completing parsing of other consitituent parts. Once remaining elements are parsed, an attempt is made to translate the device name as a logical name. If the translation succeeds, the equivalence string is then parsed, and its elements are merged in or discarded into the original file name string to form a new string. With the new string, the parsing operation is repeated. If the device name did not translate successfully, then it is truely a device name.
   b. If a directory name is seen (a left square or angle bracket is found) then RMS takes the following actions:
      1. Determines the directory format. The format can be any one of the following formats: [group,member] format or the following normal formats: [directory_name] format or [directory_name1.directory_name2...] format or [.directory_name...] format. The Parse service identifies the format.
      2. If the format is a normal format directory name, checks for [ ], [.directory_name] or [-.directory_name]. Presence of any of these implies explicit use of default directory.
      3. If there are leading minus signs, repeatedly applies default directory for each minus. Each minus sign represents one level of directory.
      4. If the directory name is null, applies default directory.

5. If there is a root directory specification, processes the file name using the rules for rooted directory. For example, there cannot be a minus sign, as it is illegal to reference a directory above the rooted directory.

c. If there is a name, checks the name for validity in syntax and length (The Parse service checks for type as well as version).

9. If after parsing the file name string, there are missing elements of a full file specification, defaults are applied until either there are no missing elements or no more defaults to be added. The defaults are applied in the following order:

a. Program defaults—First, the default file name string (if any) is applied, and then, the related file name strings (if any) are used. The default file name string can apply to any of the elements of a full file specification. The parsing and copying is handled in the same manner as for primary file specification with the exception that duplicate fields do not cause error. Duplicates are simply discarded. If a logical name is provided by the default file name string, it is not discarded simply because there is already a device name. The logical name is translated fully and applied for defaults. However, in this case, the translation must not yield duplicate fields. If either the file name or file type remains blank, and a related file is specified, then the related file specification is parsed and the file name and/or the file type is copied in to form a full file specification.

b. System defaults—First, the default device name is applied, which is followed by the default directory name. If the device component of the expanded name is missing, an attempt is made to obtain the default device name by calling *exec$translate_object_name* with the name *sys$default_device*. The equivalence string obtained from this translation is merged into the expanded name string just as done for default file name string. This step must yield a device name. If the directory component of the expanded name is missing, then the default directory name is copied in. The default directory name is obtained from the process public display container by translating *sys$default_directory*.

c. If after applying the system defaults there is no device name, it is an error, and the Parse service exits with appropriate status.

10. At this point the Parse service has an expanded the file name.

11. Using the device name, obtains the device object ID.

12. A channel is assigned and the device characteristics are obtained. If the channel assignment fails, the Parse service exits with error.

13. For each directory encountered, finds its directory ID. In finding the next directory, the following steps are taken:

a. First of all, the base directory is setup. Subsequent subdirectories are appended, in order, to the base directory. To set the base directory:

1. Copies all the leading nonwild tokens. If all tokens were nonwild, the Parse service simply finds the directory ID and returns.

2. If the very first pattern token is wild, the base dierectory is the Master File Directory (MFD).

3. Alternatively, the base directory is the last nonwild name(if any). The Parse service gets the directory ID of the base directory.

b. Sets the minimum number of directory levels that needs to be traversed.

c. Checks if there are any more wildcards left in the pattern string. If not, wildcard processing is done.

d. Gets the directory IDs of all the leading nonwild tokens.

14. Performs the various outputs requested by the user. Sets RMS status.

15. Deassigns the channel and returns.
16. The Parse service sets RMS status.

1.4.5.1 Miscellaneous Notes on File Name Parsing

- A file name string may not contain any node name or node name delimiter. The file name string may not contain any imbedded blanks or lower-case alphabetic characters. Quoted strings are not permitted.
- There may be only one logical/device-name field in a string. The name must be terminated by a ':'.
- At most ten logical name translations are done.
- The default directory string is maintained in the process public display container. The default directory is obtained by translating the name *sys$default_directory*.
- The default device string is maintained in the process public display container. The Parse service tries to assign a channel to the device. If the device is not mounted, an appropriate error code is generated. The default device is obtained by translating *sys$default_device.*
- The default name string should not contain device/logical name.

1.4.6 Search Service

The basic service provided by *rms$search* is that within a given a directory, it looks for entries that match the file name, type and version number, specified in the *wildcard_context*. If there is a wildcard character embedded in the file specification string, then there is a possiblility of finding multiple matches. As the Search service returns all the entries that match, applications are no longer required to call the Search service repeatedly. The matched outputs are placed in a buffer. If all matched items cannot be placed in the buffer, the Search service returns an indication. The application may then make another call to the Search service, to obtain the rest of the items. The Search service performs the following significant steps:

1. Checks if a previous context exists (for example if the Parse service was invoked earlier). If a context is available it proceeds to the next step, otherwise the Search service exits.
2. Checks to see if in the previous context "no more file" condition was encountered. If so, there is nothing more to do.
3. Checks to see if there a wildcard within the input file string. If no wildcards are seen, the Search service gets the file from the input specification. It issues a call to the I/O subsystem with the function code *io$c_dfile_search_dir_tree* to locate the file. This search path is now complete.
4. If there is a wildcard, then the Search service issues a call to the I/O subsystem with function code *io$c_dfile_read_dir_entries* with the match criteria "all". If a previous context has to be passed to the I/O subsystem, the Search service passes it via the input parameter *first_entry* (for details see Chapter 20, Disk File System Function Processors).
5. Using the input file specification, the Search service matches all the entries that were returned by the I/O subsystem. The matched entries are returned via the output parameter *matched_files*.
6. The *wildcard_context* is updated to indicate the state of the search operation. For example, if the buffer for return entries overflows, the next file context is saved in the *wildcard_context*.
7. The Search service sets RMS status.

1.4.7 Data Retriveal Services

The user accesses records from a sequential file, by calling the following generic services:

- *rms$get_sequential*
- *rms$get_rfa*
- *rms$get_key*

At the time a file is opened, along with others, the following file attributes are known:

- The file organization (at FRS, only sequential files are supported.)
- The record format
- The device type (at FRS only disk files are supported. Terminal devices are supported by way of callback services.)

Using the above file attributes, Mica RMS sets up an internal data retrieval vector. The vector is defined by the record *rms$retrieval_serv_vec*. See the implementation note in Section 1.3.5. The individual items of the data retrieval vector are armed with specific procedure variables. The procedure variable used depends upon the file attributes listed above. For example, if a file MYFILE has the file attributes:

```
file name = myfile.txt
file organization = sequential
record format = variable
device on which the file resides = disk
```

Then, Mica RMS loads *rms$retrieval_serv_vec* with the following procedure variables.

```
rms$retrieval_serv_vec.get_sequential = get_seq$seq_dsk_var;
rms$retrieval_serv_vec.put_sequential = put_seq$seq_dsk_var;
rms$retrieval_serv_vec.get_rfa = get_rfa$seq_dsk_var;
rms$retrieval_serv_vec.get_key = get_key$seq_dsk_var;
rms$retrieval_serv_vec.put_key = put_key$seq_dsk_var;
```

The user invokes the RMS interface service *rms$get_sequential*, which is a jacket routine. In *rms$get_sequential* the following call is made:

```
result = rms$retrieval_serv_vec.get_sequential(
            file_handle,
            record_position,
            user_in_buffer_pointer,
            user_in_buffer_length,
            move_mode,
            in_options,
            current_record_pointer,
            next_record_position,
            read_data_buffer_pointer,
            read_data_length
            );
```

Note, the call is being made to *get_seq$seq_dsk_var* procedure, on user's behalf. If, for example, the record format of the file were *fixed*, the procedure called from the jacket routine would have been to *get_seq$seq_dsk_fixed*.

For data retrieval operations on sequentially organized disk files, the following procedures are defined:

- For sequential access:
  - *get_seq$seq_dsk_var*
  - *get_seq$seq_dsk_vfc*
  - *get_seq$seq_dsk_stm*

- *get_seq$seq_dsk_stmcr*
- *get_seq$seq_dsk_stmlf*
- *get_seq$seq_dsk_fixed*
- *get_seq$seq_dsk_udf*

- For access via the record's file access:
  - *get_rfa$seq_dsk_var*
  - *get_rfa$seq_dsk_vfc*
  - *get_rfa$seq_dsk_stm*
  - *get_rfa$seq_dsk_stmcr*
  - *get_rfa$seq_dsk_stmlf*
  - *get_rfa$seq_dsk_fixed*
  - *get_rfa$seq_dsk_udf*
- For random access by relative record number:
  - *get_key$seq_dsk_fixed*

For all data retrieval procedures, the first order of business is to determine if the requested record can be retrieved from the existing buffers. The steps are:

1. For keyed access, the procedure converts the relative record number to record's file address. All RFA procedures check if the offset value is within a block. To locate the record within a block of a buffer, the following steps are taken:
2. Checks if the VBN of the record is greater than or equal to the end of file block. If this is not true (the VBN is within bounds), then:
3. Gets the current buffer descriptor pointer. If the buffer descriptor is not available, then gets the next block. If the buffer descriptor is available, checks if the next record position (NRP) information is available. If the NRP is not available, then skips to next step. If the NRP is available, checks if the end of the buffer address is less than the NRP. If this is true, then the procedure skips to next step. Otherwise, the record is available immediately.
4. To get the next block, the data retrieval procedures takes the following steps:
   a. Gets the buffer descriptor address. If buffer descriptor is not available, does a read-ahead. If the buffer descriptor is available, it continues below.
   b. Computes relative VBN.
   c. Checks to see if the requested block is available within the buffer. If available, then maps the block, otherwise releases the current buffer and read ahead.
   d. Once the record offset within the block is determined, it performs checks according to the record format.
5. The above steps are common for sequential disk files, for all record formats. All data retrieval procedures listed above execute the common steps. Having found the record, each specific data retrieval procedure carries out specific checks depending upon the record format. For example, prior to returning the pointer to the data, to the user, *get_seq$seq_dsk_stm* performs the following checks:
   a. Ignores leading NUL characters.
   b. Tries to find a terminator. If a terminator is found, that is the end of the record. If a terminator is not seen before the end of the buffer, sets an indication.

c. If the last byte of the old buffer was a CR, and the first byte of the new buffer is a LF, then the procedure considers that a terminator for the record has been found.

6. Once the record is found, RMS sets the record pointer and the next record position information. RMS returns other user requested information.

7. The data retrieval service sets RMS status.

1.4.8 Data Output Services

Data outputs onto sequential disk files are described by the following procedures:

- If record access mode is sequential:
  - *get_seq$seq_dsk_var*
  - *get_seq$seq_dsk_vfc*
  - *get_seq$seq_dsk_stm* (for stream, stmcr, stmlf)
  - *get_seq$seq_dsk_fixed*
- For record access by relative record number:
  - *put_key$seq_dsk_fixed*

1.4.8.1 Sequential Record Output

The common steps for all data output procedures on sequentially organized disk files are listed below:

1. The record position must be at the end of the file. If at EOF, then output continues. Otherwise, checks if truncate on put option is set. If the option is set, checks if truncate access is also set. If anyof the two option checks fail, the output cannot be done.

2. The record has to be copied from the user's buffer to RMS buffer. Using the current record length RMS computes the number of bytes left in the RMS buffer, and checks to see if the record can be accomodated within a block. If the option records cannot cross block boundaries were set, and the computation showed that adding the current record would cause overflow onto the next block, it causes an exit with error.

3. If everything is correct, then the procedure copies the record, and updates the end of file data.

A few of the typical checks done in the specific procedures are described below.

If the record format is variable or variable with fixed control, the procedure:

- Ensures that the records are word aligned
- Determines the overhead size, and adds it to the record size
- For VFC format only, processes the header for control operations

If the record format is stream, stream related operations are performed. For example, the procedure sets the default terminators.

On a sequential file, data is usually inserted at the end of the file. However, for a sequential file with fixed record format, a random record can be modified. Records in such files are numbered in ascending order, starting with number 1. The user can refer to any relative record number, as long as it is within the current boundaries of the file (relative record number is less than or equal to the highest record number in the file). Basically, RMS converts the relative record number to the record's file address, and if all other checks (for example, the access permissible) are satisfactory, updates the record.

1.4.9 I/Os Through Client Context Server

If after translating the file name through the *clientcs$rms_translate_logical_name* an indication is received that the file is to be processed by way of the client context server routines, then the following significant steps are taken to establish, conduct and terminate such I/O sessions:

1. The file needs to be opened at the client site. Mica RMS procedure *rms$open* calls the client context server *clientcs$open* to initiate a remote procedure call (RPC) call in the client site procedure *clientcs$rms_open*, which in turn, calls VMS RMS $OPEN.
2. If the file is opened successfully, a VMS RMS $CONNECT is done. As Mica RMS user interface does not have a corresponding procedure, the VMS RMS $CONNECT call is made automatically, on the user's behalf, at the client site.
3. The *clientcs$rms_open* returns the *file_handle* and the specified output items. Typically, device characteristics are requested as output. Note, this *file_handle* is meaningful only to the *clientcs$rms_open*. The outputs are returned back to *rms$open*. The (*rms$open*) service saves the information received from the client site, and returns to its caller a *file_handle*. This *file_handle* points to the local file context area. If requested, the device characteristics, as defined in *rms$create*, is also returned to the user. A device is identified as a remote terminal if both, the terminal and unknown bits are set in *rms$device_characteristics*.
4. The Read/Write operations are performed by the following Mica RMS procedures:
   - To read from the client site, *get_seq$seq_unknown*
   - To write to the client site, *put_seq$seq_unknown*
5. The above procedures call *clientcs$get* and *clientcs$put* respectively. The *clientcs$get* calls *clientcs$rms_get_seq* at the client site. Similarly, *clientcs$put* calls *clientcs$rms_put_seq* at the client site.
6. Upon receiving a close request, Mica RMS Close service calls *clientcs$close* to close the file at the client site. The *clientcs$close* calls *clientcs$rms_close*, to make the VMS RMS $CLOSE call on the file.
7. If the file is closed at the client site, Mica RMS Close service deallocates the local file context and sets a nul value to the *file_handle*.

APPENDIX A

PRELIMINARY TEST PLANS

Testing of Mica RMS services is accomplished by the following:

1. Functional Tests—These tests exercise the various functions of a the RMS services. The tests validate the functionality of each Mica RMS service. These tests will be developed together with the RMS modules.
2. Fault Insertion Tests—These tests exercise the software's robustness. Ability to handle faulty inputs is established. Once a module passes the functional tests, fault insertion tests are done to determine how soundly the software handles such cases.
3. Regression Tests—These tests are developed as bugs are discovered and fixed in RMS software. These tests establish that the bug has been removed.
4. Performance Tests—These tests will be done to show RMS performance. Performance of simple sequential get and put operations will be initially tested.

APPENDIX B

OUTSTANDING ISSUES

The following list identifies the issues that are yet to be resolved:

- Item list definition—Mica system-wide definition of an *item* and *item_list* are not finalized.
- Protection options—The structure through which protection options are specified is not yet defined.