Mica Working Design Document Record Management Services

Revision 0.3

7–January–1988

Issued by:

Sumanta Chatterjee

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.