Mica Working Design Document Status Values, Messages, and Text Formatting

Revision 0.5

29–January–1988

Issued by:
Kris K. Barker

digital™

TABLE OF CONTENTS

Revision History

Date	Revision Number	Author	Summary of Changes
5-NOV-1986	0.1	Kris Barker	Original.
11-DEC-1986	0.2	Kris Barker	Modifications prior to general review.
14-JAN-1987	0.3	Kris Barker	Modifications following general review.
14-JAN-1988	0.4	Kris Barker	Convert to SDML format and modify prior to primary review.
29-JAN-1988	0.5	Kris Barker	Misc. revisions following primary review.

CHAPTER 1

STATUS VALUES, MESSAGES, AND TEXT FORMATTING

1.1 Introduction

Status values pass information regarding the success or failure of a process, thread, or procedure back to the thread which created or called it. Status values are also used to organize and index messages that convey information about status values in textural form.

This chapter:

- Defines the format of status values.
- Describes the mechanisms used to translate status values in text strings.
- Describes the organization of messages and message files.
- Describes the use of messages and message files for internationalizing text.
- Outlines the text formatting support provided on Mica. While such support is an important part of message access and display, it is general purpose in nature and may be used in any programming situation where text formatting is required.

1.2 Goals

The primary goal of this implementation is to provide a consistent, easy-to-understand, and easy-to-use way of organizing definition of and access to status information, message text, or both. Within this general goal are the following specific goals:

- To provide a local message capability which allows message definition and access without the requirement of facility registration.
- To provide a convenient way of separating text from an image that uses it, and to allow the text to be rewritten in another natural language without affecting the image.
- To describe and encourage the use of the message capabilities for all user-displayed text in a program, not just messages, as a way to internationalize programs more easily.
- To provide a text formatting capability that addresses internationalization requirements.

1.3 Terminology

Table 1–1 summarizes key terms introduced in this chapter.

Table 1–1: Status Terminology

Term	Definition
Abbreviated condition name	A string of characters that briefly describes a particular condition.
Facility or Facility number	A 12-bit binary value that identifies the facility that produced the status value.
Facility name	A string of characters that identifies the facility that produced the status value.
Formatting directive	A command to the text formatting routine that specifies how a parameter to that routine is to be formatted.
Local message	A message local to a specific program. Local messages do not need to be registered, as access to them is through a single facility. Local messages are also used to internationalize message text.
Message section	A data structure that contains message text, severity information, abbreviated condition names, and facility names for the messages of a facility.
Message section descriptor	A data structure that contains information about a message section. It may contain a self-relative pointer to the message section itself (direct message section descriptor) or a self-relative pointer to a filename which contains the message section (indirect message section descriptor).
Message section descriptor table	A zero-terminated array of message section descriptors (direct or indirect).
Message string	A string of characters that describes a particular condition. It may contain message text, an abbreviated condition name, a severity character, and a facility name.
Message text	A string of characters that: • describes a particular condition in detail, or • contains noncondition information displayed to the user.
Message vector	A table of self-relative offsets, each of which points to a message section descriptor table.
Severity	Either a value or a single character (depending on the context) that describes the basic success or failure indicated by the condition.
Shared message	A system-wide message that inherits the facility name from the program that accesses it. Shared messages are used to provide consistency in message text across multiple programs. Shared messages also change the message searching rules; see Section 1.8.2 for a discussion on message searching.
Status value	A 32-bit numeric value containing information about the status of a thread, process, or procedure.

1.4 Status Values

Status values are longword values used to:

- Indicate the exit status of a process
- Indicate the exit status of a thread
- Return status from a remote procedure call
- Return status from a procedure or function call (such as a run-time library function)
- Organize *local messages,* that is, internal messages within a program

Additionally, values in status value format are used to organize and access nonmessage text local to a facility.

Throughout this chapter, the term *producer* is used to indicate the process, thread, or procedure returning or raising status and the term *consumer* is used to indicate the process, thread, or procedure which receives that status.

Status values have the following binary format:

Figure 1–1: exec$status_value

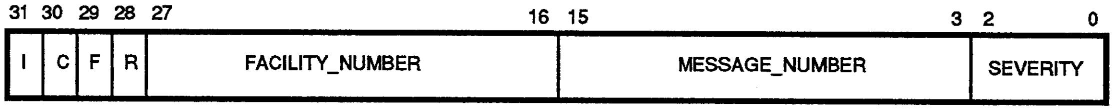

FIG1

```
exec$status_value : RECORD
    severity : integer[0..5] SIZE(BIT,3);
    message_number : integer[0..8191] SIZE(BIT,13);
    facility_number : integer[0..4095] SIZE(BIT,12);
    facility_specific : BIT;
    customer_facility : bit;
    inhibit_message_printing : bit;
    LAYOUT
        severity;
        message_number;
        facility_number;
        reserved : FILLER(BIT,*);
        facility_specific POSITION(BIT,29);
        customer_facility;
        inhibit_message_printing;
    END LAYOUT;
END RECORD;
```

Mica status values are similar to status values on VAX/VMS. The differences are the *inhibit_message_printing* bit (bit 31 on Mica) and the locations of the *customer_facility* and *facility_specific* bits (bits 30 and 29, respectively). Moving the *customer_facility* and *facility_specific* bits out of the *facility_number* and *message_number* fields effectively doubles the number of facility and message numbers over that allowed on VAX/VMS.

The sections below describe each field of a status value.

1.4.1 SEVERITY Field (bits <2:0>)

The severity field of a status value indicates the basic success or failure of the producer of the status. *Severity* is represented as a binary value in the range 0 to 4 (values in the range of 5 to 7 are reserved to DIGITAL).

Successful completion is indicated by an odd-valued severity.

Value	Meaning	Success
1	Success	This value indicates successful completion.
3	Information	This value indicates successful completion with some associated information for the consumer.

Even severity values indicate partial or complete failure.

Value	Meaning	Failure
0	Warning	This value indicates that the producer of the status encountered a nonfatal problem, but was able to complete processing the request. The status returned warns the consumer that the result of processing the request may not be what was expected.
2	Error	This value indicates that an error occurred, however, the error was not severe enough to force premature termination of the producer.
4	Fatal	This value indicates that a fatal error occurred. Such an error is severe enough that the producer of the status was forced to exit or return prematurely.

1.4.2 MESSAGE_NUMBER Field (bits <15:3>)

The *message_number* field of a status value is used to identifiy which of a set of several possible conditions this status value represents. The message routines use this value to index into a message section to obtain the corresponding message text. Message sections are described in Section 1.7.4.

1.4.3 FACILITY_NUMBER Field (bits <27:16>)

The *facility_number* field of a status value is used to identify the producer of the status value. With the exception of the local facility number, each facility must have its own unique facility number.

Status values with the facility number equal to 4095, the local facility number, are called local status values and allow a facility to define and maintain messages without being concerned about a unique facility number. Local status values and messages are discussed further in Section 1.5.6.

The facility number 0 is reserved for system-wide status values. The facility name corresponding to facility number 0 is SYSTEM.

1.4.4 Reserved Field (bit 28)

Bit 28 is reserved to DIGITAL and should be zero (SBZ).

1.4.5 FACILITY_SPECIFIC Field (bit 29)

The facility_specific field is used to indicate that the status value is specific to a single facility. Status values with this bit clear are used to identify system-wide status codes (for system and shared messages). Use of this bit for shared messages is described in Section 1.5.7.

1.4.6 CUSTOMER_FACILITY Field (bit 30)

The customer_facility field is used to indicate that the number specified in the facility number field is a customer facility. Status values for DIGITAL facilities have this bit clear.

1.4.7 INHIBIT_MESSAGE_PRINTING Field (bit 31)

The inhibit_message_printing field is used to inhibit display of the message by message output routines. This bit is set by system routines that display the resulting message, so that the message is not displayed twice.

1.5 Status Messages

Status messages are text strings used to describe a status value to the user in a natural language. A complete status message consists of:

- *Facility name*—A short string of characters indicating the facility to which the status is registered.
- Severity—A single letter indication corresponding to the severity of the status:

Field Value	Severity Letter	Meaning
1	S	Success
3	I	Information
0	W	Warning
2	E	Error
4	F	Fatal

- *Abbreviated condition name*—A short string of characters identifying the status in an abbreviated manner.
- Message text—A string of characters describing the status in detail.

1.5.1 Status Message Format

By default, status messages are assembled in the following format:

```
    %FACILITY-S-ACONDNAME, message text
```

"FACILITY" is the facility name, "S" is the severity, and "ACONDNAME" is the abbreviated condition name. A user or facility may request that certain parts of a status message be excluded when the message is assembled. The default message format may be changed with a CLI command (such as SET MESSAGE for DCL). The logical name MICA$MESSAGE_FORMAT is used to convey the current message format setting between a CLI running on a client system and a program running on the server.

The message access and display routines use the message format setting along with the following rules to determine the final format of a status message:

- The leading "%" is present only if the facility, severity, or abbreviated condition name are present (in other words, if only the message text is requested, no leading "%" will be returned).
- If only the message text is returned, the first character of the text string is converted to upper case.
- The message display routine *exec$display_message* supports display of multiple messages. In this case, the first message formatted is termed the primary message; successive messages are termed secondary messages. The *exec$get_message* routine provides an argument that allows the caller to specify that the message is to be formatted as a secondary message. The format of the secondary message is the same as that of a primary message except that the "%" sign (if present) is replaced by a "-" sign. The *exec$get_message* and *exec$display_message* routines are describe in Section 1.5.4 and Section 1.5.5.

1.5.2 Message Creation

Messages are created in text format using a text editor. A file consisting of a collection of facility names, abbreviated condition names, severity condition values, and message text is called a message source file. A message source file is processed by the Pillar message compilation facility into a message object module which is then linked with other object modules to form an image file.

1.5.3 Message Compilation

Message compilation is the process of creating a message object file from a message source file. Mica provides message compilation capabilities as part of the Pillar compiler.

The message compilation facility provides a way to internationalize messages by allowing the message text and formatting information to be separated from the image file. The message source file is compiled twice:

1. The first compilation produces a direct message object module containing the facility names, severities, abbreviated condition names, and message text. This module is then linked to form a message image file which is accessed when the message text is required. Note that this message image file must be linked by itself; resolution of indirect message section descriptors does not allow multiple direct object modules to be linked together unless they are linked into the program image. Section 1.8.2.4 discusses how and when these direct message image files are read.
2. The second compilation creates an indirect message object module which is linked with other program object modules to form the program image file. In this case, the compiler generates the message object file without the message text itself. Instead, the message section descriptors, which would normally point to message sections containing message text, contain the specification for the corresponding message image file that contains the message text. See Section 1.7 for a discussion of message data structures.

Once a particular message source file is translated into another natural language, the first step described above is repeated on the translated file. The result is a message image file in another language that can be accessed by the application without requiring that the application be relinked. The location of multiple language versions of message files is described in Section 1.9 and in Chapter 33, Layered Products and System Disk.

1.5.4 Obtaining and Formating Status Messages

The *exec$get_message* routine obtains and formats status messages. The interface to this procedure is:

```
PROCEDURE exec$get_message (
            IN condition_record : exec$condition_record;
            OUT message_buffer : varying_string(*);
            IN facility_name : string(*) OPTIONAL;
            IN format : boolean = true;
            IN flags : exec$message_options OPTIONAL;
            IN secondary : boolean = false;
            OUT argument_count : integer OPTIONAL;
            OUT user_value : integer OPTIONAL;
            ) RETURNS status;
```

Table 1–2 describes the parameters for this procedure.

Table 1–2: Parameters to exec$get_message

Parameter	Description
condition_record	Supplies a condition record containing the status value for which a message is to be returned. Only the message corresponding to the primary condition is returned. For local messages, the condition record contains the address of the message section descriptor which points to the message section containing the local message. The format and content of condition records is presented in Chapter 9, Condition, Exit, and AST Handling. Message section descriptors and message sections are discussed in Section 1.7.
message_buffer	Supplies the address of the buffer in which the message string is returned.
facility_name	Optionally supplies a facility name which overrides the facility name indicated by the facility number field of the status value. This parameter is useful when a program requests translation of a local or shared message and wants to replace the default facility name with a more meaningful facility name. See Section 1.5.6 and Section 1.5.7 for more information on local and shared messages.
format	Optionally supplies a Boolean value which, if TRUE, indicates that formatting directives in the message string are to be interpreted.
flags	Optionally supplies a set of type *exec$message_options* that indicates which portions of the status message are to be returned. Each element of the set that is supplied – *exec$facility*, *exec$severity*, *exec$condition_name*, *exec$message_text* – indicates that the corresponding field should be included in the formatted status message. If this argument is not supplied, the default format is used, as specified by the MICA$MESSAGE_FORMAT logical name supplied by the client. The data type *exec$message_options* is defined as:

```
exec$message_options_type : (
    exec$severity,
    exec$facility,
    exec$condition_name,
    exec$message_text
    );
exec$message_options : SET[exec$message_options_type];
```

Parameter	Description
secondary	Optionally supplies a Boolean value which, if true, specifies that the message should be formatted as a secondary message. By default, the message is formatted as a primary message.
argument_count	Optionally returns the number of parameters associated with the message.
user_value	Optionally returns the value associated with the message as specified in the message source file. The interpretation of this value is the responsibility of the caller of *exec$get_message*.

This routine searches the message sections pointed to by both the Image and System Message Vectors to obtain the message string corresponding to the specified status value. See Section 1.7 and Section 1.8 for more information on the organization of message vectors and message sections and the mechanisms used to traverse them.

1.5.5 Obtaining and Displaying Status Messages

The *exec$display_message* routine obtains and displays one or more status messages based on a specified condition record. The interface to this procedure is:

```
PROCEDURE exec$display_message (
            IN condition_record : exec$condition_record;
            IN flags : exec$message_options OPTIONAL;
            IN facility_name : string(*) OPTIONAL;
            IN action_routine : exec$action_procedure OPTIONAL;
            IN action_parameter : anytype = zero;
            ) RETURNS status;
```

Table 1–3 describes the parameters for this procedure.

Table 1–3: Parameters to exec$display_message

Parameter	Description
condition_record	Supplies a condition_record containing the status values to be formatted and output. Unlike *exec$get_message* described above, *exec$display_message* translates status values for the primary condition and all secondary conditions specified by the condition record.
flags	Optionally supplies a set of type *exec$message_options* that indicates which portions of the status message are to be returned. Each element of the set that is supplied – *exec$facility*, *exec$severity*, *exec$condition_name*, *exec$message_text* – indicates that the corresponding field should be included in the formatted status messages. If this argument is not supplied, the default format is used, as specified by the MICA$MESSAGE_FORMAT logical name supplied by the client.
	\This method of specifying the message formatting flags makes it impossible, using the *exec$display_message* routine, to specify different formatting for each status value in the specified condition record. This is possible on VAX/VMS.
facility_name	Optionally supplies a facility name which overrides the facility name indicated by the facility number portion for the primary condition.
action_routine	Optionally supplies the address of an action routine to be called after each message text line is formatted but before it is displayed. The two arguments to this routine are the formatted message string and the action parameter (see below) supplied in the call to *exec$display_message*. This routine must return a Boolean value: if TRUE, the message is output by *exec$display_message*; if FALSE, it is not.
action_parameter	Optionally supplies a value that is passed to the action routine.

This routine searches the message sections pointed to by both the Image and System Message Vectors to obtain the message string corresponding to the specified status value. See Section 1.7 and Section 1.8 for more information on the organization of message vectors and message sections and the mechanisms used to traverse them.

1.5.6 Local Messages

Local messages provide programs a way to store message and other text separately from the actual image file without the normal requirement to register a facility number. Status values whose facility number is 4095 are used to reference local messages.

The data structures used to organize local message data are the same as those used for nonlocal messages. Local message section descriptors, however, are specified explicitly—via condition record or procedure argument—rather than implicitly by their presence in a message section descriptor table whose address is in a message vector. When *exec$get_message* or *exec$display_message* is called to obtain the text for a local message, the supplied condition record specifies the address of the message section descriptor to be examined. When *exec$get_text* is called to obtain the text for a local message, the *message_section_descriptor* argument supplies the address of the message section descriptor to examine.

Only the specified message section descriptor is examined; if the local message number is not found in the section pointed to by the message section descriptor, the search fails. This is unlike the nonlocal message case, where the search continues by examining other message section descriptors.

See Section 1.8 for more information on the mechanisms used to translate status values to status and text messages.

1.5.7 Shared Messages

Shared messages are used to define status values and message text that can be shared by several facilities, thus providing a way to guarantee consistency of messages across facilities. These are different from system messages in that the name of the facility producing the status value is used as the facility name (as opposed to SYSTEM for system messages). Also, shared status values alter the default search order during message translation.

Shared status values are defined with a facility code and severity of 0. Within the status value, the *customer_facility* and *facility_specific* bits are clear. To use shared status values, a facility must merge its own facility code and the status severity with the shared status value. This is done as follows:

```
    status_value = facility_number * 65536 +
                   shared_status_value +
                   severity
```

This calculation yields a status value that contains the message code of a shared message, and the facility number and severity specified by the program producing the status value.

Section 1.8 describes the mechanisms used to translate status values to status and text messages. Section 1.8.1.3 describes how these mechanisms are affected by shared messages.

1.6 Text Messages

Text messages provide a way to define, organize, and access text that is user-visible and not related to a condition. This capability is required to provide support for internationalization of text displayed to users.

1.6.1 Relationship to Status Messages

Definition and organization of text messages is the same as that described for status messages in Section 1.5 with the following exceptions:

- The routine which accesses and formats text messages only returns the message text, not the severity, facility, or abbreviated condition name.
- Text messages are usually local; that is, the status value used to access them normally has the facility field equal to the local facility number.
- The routine which accesses and formats text messages does not require that access and parameter information be supplied in condition record format. This means that programs which use this functionality to organize user-visible text will not be required to handcraft condition records.

1.6.2 Obtaining and Formating Text Messages

The *exec$get_text* routine obtains and formats a text message based on a supplied status value and message section descriptor address. The interface to this procedure is:

```
PROCEDURE exec$get_text (
            IN status_value : status;
            IN message_section_descriptor : exec$message_section_desc;
            IN parameters : exec$argument_array(*);
            OUT message_buffer : varying_string(*);
            IN format : boolean = true;
            OUT argument_count : integer OPTIONAL;
            OUT user_value : integer OPTIONAL;
            ) RETURNS status;
```

Table 1–4 describes the parameters for this procedure.

Table 1–4: Parameters to exec$get_text

Parameter	Description
status_value	Supplies a status value used to locate the text message.
message_section_descriptor	Supplies the address of the message section descriptor to search.
parameters	Supplies an array of parameters to be formatted into the resultant string. The data type *exec$argument_array* is an array of *exec$argument_descriptor*. The data type *exec$argument_descriptor* is defined in Chapter 9, Condition, Exit, and AST Handling.
message_buffer	Supplies the address of the buffer in which the message text string is returned.
format	Optionally supplies a Boolean value which, if TRUE, indicates that the message string is to be formatted; that is, formatting directives are interpreted.
argument_count	Optionally returns the number of parameters associated with the message.
user_value	Optionally returns the value associated with the message as specified in the message source file.

1.7 Message Data Structures

The outputs of Mica's message compilation facility are *message object modules*. These modules contain message information in structures called *message sections*, *message section descriptors*, and *message section descriptor tables*. Once in memory, message information is organized into:

- *Message vectors*—These structures are tables of pointers, each of which points to a message section descriptor table.
- Message section descriptor tables—These structures are arrays of message section descriptors.
- Message section descriptors—These structures contain information about message sections including message section type (direct or indirect), facility number, a self-relative pointer to the message section, and, for indirect sections, a self-relative pointer to the name of the file that contains the actual message text (message file specification).
- Message sections—These structures contain a facility name, facility number and abbreviated condition names and message text. Message sections are organized by the message compilation facility so that they can be indexed by message number. Each message section contains the messages for one facility.

The following figure shows how all of these structures are related.

Figure 1–2: In-Memory Message Data Structure Organization

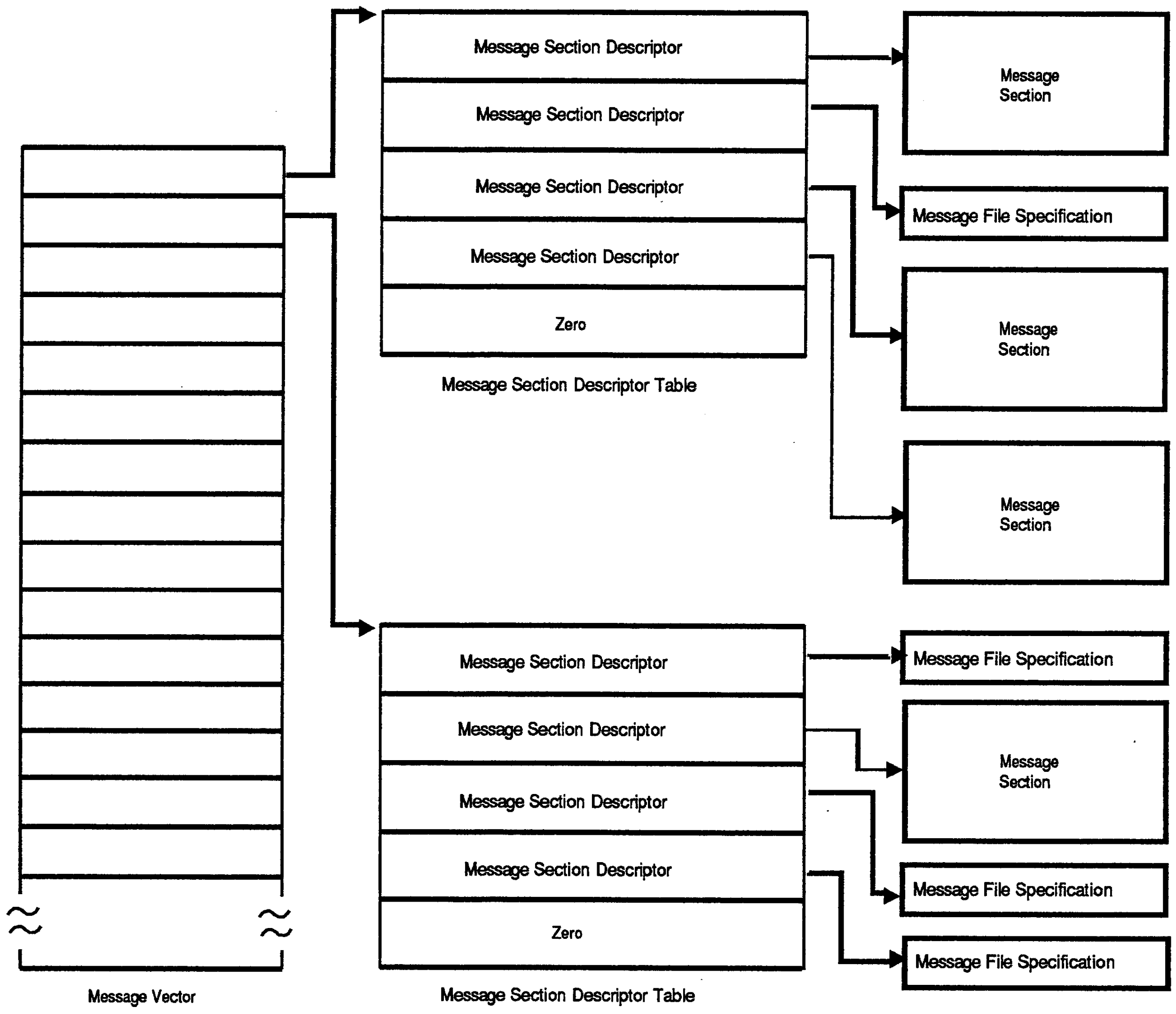
Message Section Descriptor
Message Section Descriptor
Message Section Descriptor
Message Section Descriptor
Zero
Message Section Descriptor Table
Message Section
Message File Specification
Message Section
Message Section
Message Section Descriptor
Message Section Descriptor
Message Section Descriptor
Message Section Descriptor
Zero
Message Section Descriptor Table
Message File Specification
Message Section
Message File Specification
Message File Specification
Message Vector

FIG2

The following sections describe the format and content of each of these data structures. In all cases, the data structure are aligned on natural boundaries.

1.7.1 Message Vectors

Within the address space of a given process, there are two message vectors: the System Message Vector and the Image Message Vector. When status code translation is requested, these two vectors supply pointers to tables of message section descriptors to be searched.

1.7.2 Message Section Descriptor Tables

A message section descriptor table is a zero-terminated list of message section descriptors. Message section descriptors are placed in either the *message$system_section_descriptor* or the *message$image_section_descriptor* PSECT. This PSECT is concatenated with like PSECTS from other *message object files* by the linker to form a message section descriptor table. An image section that contains message section descriptor tables will have a flag indicating this in the image section descriptor (ISD).

The terminator of a message section descriptor table is a zero longword. This zero longword resides in the overlaid *message$section_descriptor_table_end* PSECT so that only one such entry actually ends up in the resulting image file. Allowable values for the section_descriptor_type, system, and facility fields are such that all of these fields being zero is not a valid combination.

1.7.3 Message Section Descriptors

A message section descriptor is a data structure that describes the facility associated with a set of messages and provides a pointer to the message section where the message text is found. It has the following binary format:

Figure 1–3: exec$message_section_desc

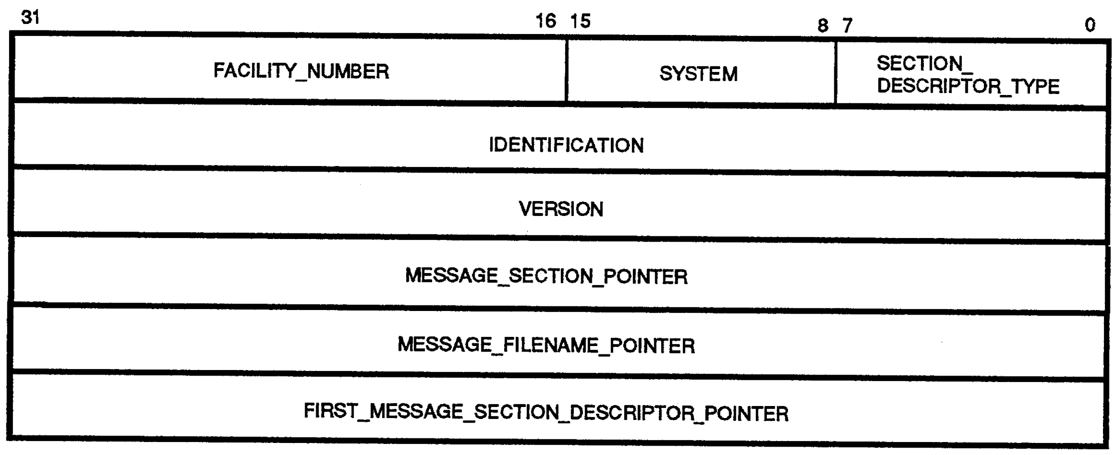

FIG3

```
exec$message_section_desc : RECORD
    section_descriptor_type : exec$message_section_desc_types[..] SIZE(BYTE);
    system : boolean;
    facility_number : exec$facility_number;
    ident : longword;
    version : longword;
    message_section_pointer : POINTER exec$message_section_pointer;
    message_filename_pointer : POINTER exec$counted_string;
    first_message_section_desc_pointer : POINTER exec$message_section_desc;
    LAYOUT
        section_descriptor_type;
        system POSITION(BYTE,1);
        facility_number POSITION(WORD,1);
        ident;
        version;
        message_section_pointer;
        message_filename_pointer;
        first_message_section_desc_pointer;
    END LAYOUT;
END RECORD;
```

```
exec$message_section_desc_types : (
    exec$direct_section_descriptor,
    exec$indirect_section_descriptor
    );

exec$facility_number : integer[0..4095] SIZE(WORD);

exec$message_section_pointer : POINTER exec$message_section;
```

These fields are defined as follows:

- *section_descriptor_type*—This field indicates the type of the message section descriptor:
  - *exec$direct_section_descriptor*—This type indicates that the message section is loaded into memory with the section descriptor. This means that the section descriptor and message section are part of the program image file. Message section descriptors whose *section_descriptor_type* is *exec$direct_section_descriptor* are referred to as *direct message section descriptors*.
  - *exec$indirect_section_descriptor*—This type indicates that the message section is contained in a separate message image file. The filename of the separate message image file is contained in a data structure pointed to by the *message_filename_pointer* field in the message section descriptor. Message section descriptors whose *section_descriptor_type* is *exec$indirect_section_descriptor* are referred to as *indirect message section descriptors*. When an indirect message section descriptor is first accessed, the corresponding direct message image file is read into memory in the image's address space, or *mapped*, and the *message_section_pointer* fields for all message section descriptors which refer to that direct message image file are set to point to the actual message sections.
- *system*—A boolean value that, when TRUE, indicates that the address of the message section descriptor table containing the message section descriptor is to be placed in the System Message Vector rather than the Image Message Vector. To avoid mixing system and nonsystem messages in the same message source file, a command line qualifier to the message compilation facility is used to indicate that this byte should be nonzero.

  \The intent is that this is for DIGITAL use only. PSECT naming conventions are be used to handle the case where system and nonsystem section descriptors are linked into the same file.\
- *facility_number*—The facility number associated with the messages in the section pointed to by this section descriptor. For local messages, this value is the local facility number.
- *ident*—This longword contains a binary identification value used in message section verification. This value is set by the message compilation facility and is used to verify that this data structure is actually a message section descriptor.
- *version*—This longword contains a binary version number of the message section. This value is set by the message compilation facility and provides a way to handle message data structure changes in future versions.
- *message_section_pointer*—A self-relative pointer to a self-relative pointer to the message section associated with this message section descriptor. For indirect message section descriptors, this pointer initially points to a pointer whose value is nil, indicating that the message image file containing the corresponding direct message section has not been mapped. Once the message image file is mapped, this pointer is updated to point to the message section.

  \This is a pointer (rather than actually placing the message section offset in the structure itself) so that write access to message section descriptors is not required. Note that due to size constraints, this extra level of indirection is not shown in Figure 1–2.\
- *message_filename_pointer*—A self-relative pointer to a data structure containing the filename of the message image file containing the message sections. For direct message section descriptors, this pointer is nil. This data structure is described in Figure 1–4.

- *first_message_section_desc_pointer*—A self-relative pointer to the first message section descriptor in the message section descriptor table produced by the message compilation facility. Because multiple message object modules may be linked together (and, therefore, multiple message section descriptor tables may be combined into one) in the image file, this pointer may not point to the first message section descriptor in the in-memory message section descriptor table. This pointer provides a way to get to the first section descriptor from the same message source file as the current section descriptor. It is used when mapping indirect message section descriptors to resolve all such descriptors which came from the same message source file.

Figure 1–4: exec$counted_string

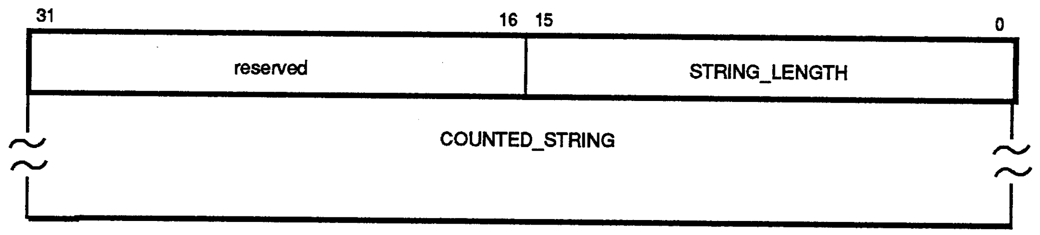

FIG8

```
exec$counted_string ( string_length : integer[0..65535] ) : RECORD
    CAPTURE string_length;
    counted_string : string(string_length);
    LAYOUT
        string_length;
        counted_string POSITION(BYTE,4);
    END LAYOUT;
END RECORD;
```

1.7.4 Message Sections

Message sections are generated by Mica's message compilation facility. Each contains messages defined for one facility. If the message source used to create the message section contains messages for more than one facility, the message compilation facility creates a separate message section for each facility.

Message sections always contain full message text. They are placed in normal read-only data PSECTs (readable, nowrite, noexecute) and contain pointers to the facility name, index table, and language in which the messages were written.

A message section has the following binary format:

Figure 1–5: exec$message_section

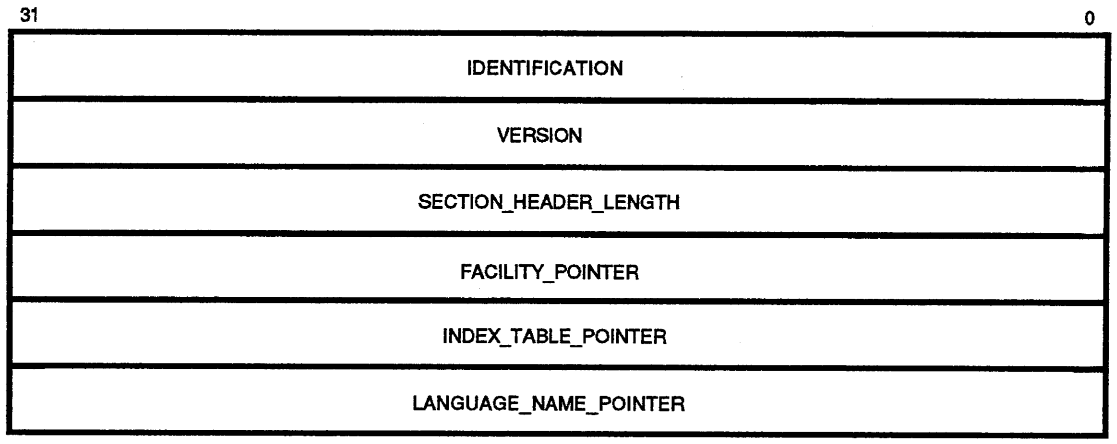

FIG4

```
exec$message_section : RECORD
    ident : longword;
    version : longword;
    section_header_length : integer;
    facility_pointer : POINTER exec$facility_name;
    index_table_pointer : POINTER exec$message_index_table;
    language_name_pointer : POINTER exec$counted_string;
    LAYOUT
        ident;
        version;
        section_header_length;
        facility_pointer;
        index_table_pointer;
        language_name_pointer;
    END LAYOUT;
END RECORD;
```

These fields are defined as follows:

- *ident*—This longword contains a binary identification value used in message section verification.
- *version*—This longword contains the binary version number of the message section.
- *section_header_length*—This longword contains the length of the message section header in bytes.
- *facility_pointer*—A self-relative pointer to a data structure of type *exec$facility_name* that contains the facility number and name for messages in the section, as shown in Figure 1–6.
- *index_table_pointer*—A self-relative pointer to the message index table that is used to index the messages themselves.
- *language_name_pointer*—A self-relative pointer to a data structure that contains the language in which this section was written.

  \It is TBD just what form will be used to indicate the language. It will most likely be a string; however, internationalization support may require that language name strings be expressible in different languages.\

Figure 1–6: exec$facility_name

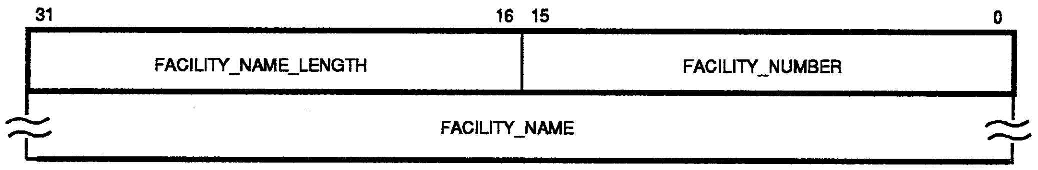

FIG5

```
exec$facility_name ( facility_name_length : integer[0..65535] SIZE(WORD)) : RECORD
    CAPTURE facility_name_length;
    facility_number : exec$facility_number;
    facility_name : string(facility_name_length);
    LAYOUT
        facility_number;
        facility_name_length;
        facility_name;
    END LAYOUT;
END RECORD;
```

The message index table is an ordered table containing message numbers and message record pointers for each record in the section.

The message index table has the following binary format:

Figure 1–7: exec$message_index_table

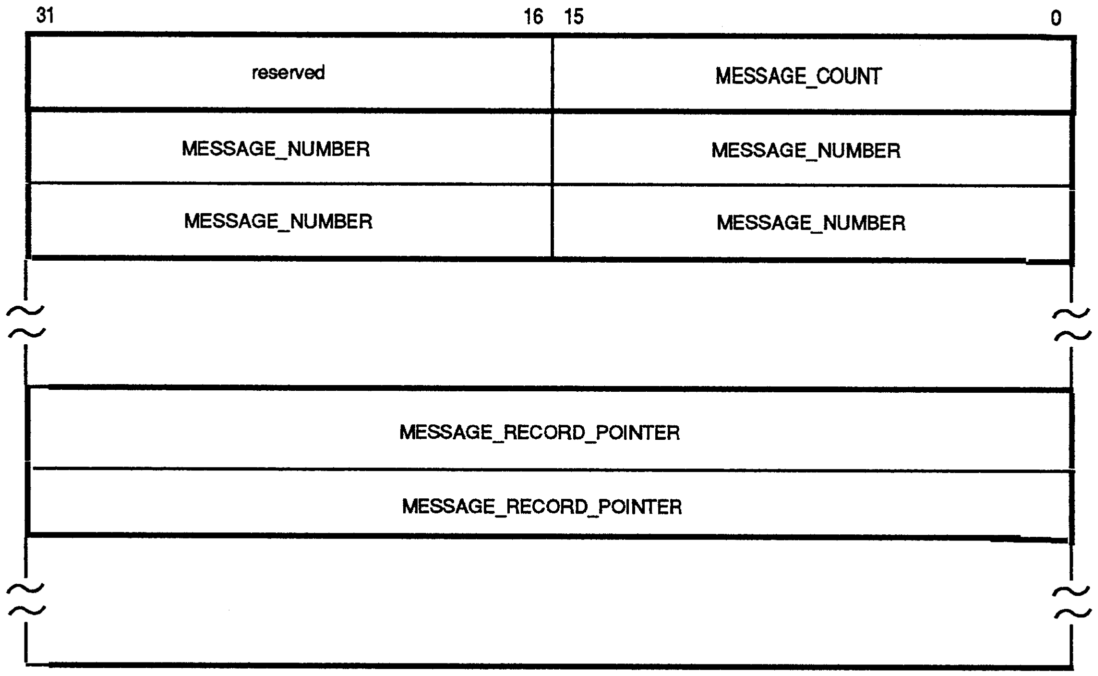

FIG6

```
exec$message_index_table ( message_count : integer [0..8191] SIZE(WORD) ) : RECORD
    CAPTURE message_count;
    message_number : ARRAY[0..message_count] OF word;
    message_record_pointer : ARRAY[0..message_count] OF POINTER exec$message_record;
    LAYOUT
        message_count;
        reserved1 : FILLER(WORD,*);
        message_number POSITION(WORD,2);
        reserved2 : FILLER(WORD,*);
        message_record_pointer;
    END LAYOUT;
END RECORD;
```

These fields are defined as follows:

- *message_count*—The number of messages indexed by this table.
- *message_number*—An array of message numbers. The message compilation facility generates this array in increasing message number order.
- *message_record_pointer*—An array of pointers to message records. Each pointer points to the message record corresponding to the message number at the same offset in the *message_number* array.

A binary search is done on the *message_number* array to locate a specific message. If found, the corresponding pointer in the *message_record_pointer* array is used to access the message record. The message record contains the message text and abbreviated condition name. Message records have the following binary format:

Figure 1–8: exec$message_record

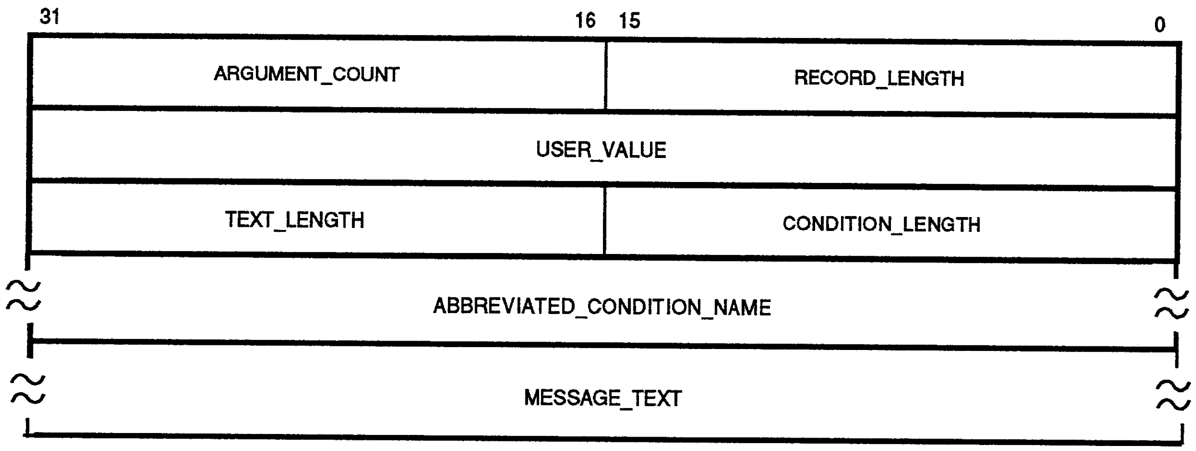

FIG7

```
exec$message_record ( condition_length, text_length : integer[0..65535] SIZE(WORD)) : RECORD
    CAPTURE condition_length, text_length;
    record_length : integer[0..65535] SIZE(WORD);
    argument_count : integer[0..65535] SIZE(WORD);
    user_value : exec$message_user_value;
    abbreviated_condition_name : string(condition_length);
    message_text : string(text_length);
    LAYOUT
        record_length;
        argument_count;
        user_value;
        condition_length;
        text_length;
        abbreviated_condition_name;
        message_text;
    END LAYOUT;
END RECORD;

exec$message_user_value : longword;
```

These fields are defined as follows:

- *record_length*—The length of the message record in bytes.
- *argument_count*—The number of arguments which are to be formatted into the message text.
- *user_value*—A user-defined longword value specified in the message source file which can be returned to the caller of *exec$get_message* or *exec$get_text*.
- *condition_length*—The length of the abbreviated condition name string in bytes.

- *text_length*—The length of the message text string in bytes.
- *abbreviated_condition_name*—The string of characters representing the abbreviated form of the condition.
- *message_text*—The actual message text itself.

1.8 Status Value to Message Translation

The following sections describe the mechanisms used to translate a status value into a status or text message. It discusses how message vectors, message section descriptors, and message sections are searched and how indirect message section descriptors are mapped.

The translation from status value to message involves examining one or more message section descriptors and searching message sections in an attempt to match both the *facility_number* and *message_number* fields in the status value with those in the message section descriptor and message section.

1.8.1 Which Message Sections are Searched

As mentioned above, the status value translation routines, *exec$get_message*, *exec$display_message*, and *exec$get_text*, will search one or more message sections to obtain the message text associated with a specified status value. The decision to search a particular message section is based on the *facility_number* and *facility_specific* fields in the status value. These fields indicate whether the message is local or nonlocal and whether or not it is shared.

This section describes which message sections are searched.

1.8.1.1 Nonlocal Messages

Message searching for nonlocal messages involves examining two sets of message section descriptors:

- Image message section descriptors—These are message section descriptors contained in message section descriptor tables whose addresses are contained in the Image Message Vector. Such message section descriptors are part of the executing process' image file or part of a shareable image's image file.
- System message section descriptors—These are message section descriptors contained in message section descriptor tables whose addresses are contained in the System Message Vector. Such message section descriptors contain system-wide message information.

A message search routine first examines the image message section descriptors and attempts to translate the status value. If no match is made, the routine continues by examining the system message section descriptors. If this search fails, the status value cannot be translated.

1.8.1.2 Local Messages

When a status value translation routine searches for a local message, the search is done on just one message section. The message section descriptor whose address is contained in the condition record or passed as a procedure argument indicates the section to search.

1.8.1.3 Shared Messages

When a shared status value translation is requested, the status value translation routines first look at the *facility_specific* bit in the status value. If this bit is clear and the facility number is not zero, the routines know that the status value is for a shared message. The translation from status value to message string is done in two parts:

1. First, a copy of the status value is made with the facility number set to zero. System message section descriptors are then examined to locate the section in which the message resides. Note that this is a different search order from the normal nonshared message case.
2. If the message is found and the facility name has been requested, all image message sections are searched to resolve the facility name. When a descriptor whose facility number matches that specified in the original status value is found, the facility name is immediately taken from the message section.

1.8.2 How Message Sections are Searched

This section describes how message sections are searched.

1.8.2.1 Deciding Which Message Section Descriptors to Examine

Based on the specified status value, the status value translation routines first determine which message section descriptors to examine. The following pseudocode describes how this is determined:

```
if facility indicates local message then
    examine message section descriptor specified by condition record or parameter
else
    if status value is not shared then
        for all message section descriptor tables pointed to by the Image Message Vector
            for all message section descriptors in this message section descriptor table
                examine message section descriptor
    if translation not successful or status value is shared then
        for all message section descriptor tables pointed to by the System Message Vector
            for all message section descriptors in this message section descriptor table
                if status value is shared then
                    examine message section descriptor with with modified status value
                else
                    examine message section descriptor
```

1.8.2.2 Examining a Message Section Descriptor

The following pseudocode describes the steps taken when a message section descriptor is examined:

```
if facility is local then
    if message section descriptor is indirect and not mapped then
        map corresponding message image file
    search message section for message
else
    if facility number matches facility in message section descriptor then
        if message section descriptor is indirect and not mapped then
            map corresponding message image file
        search message section for message
```

1.8.2.3 Searching a Message Section

The following pseudocode describes the steps taken when a message section is searched for a message:

```
if section language matches current language then
    search message index table for message number
    if message number found then
        copy requested portions of message into message string
        return message_found
    else
        return message_not_found
else
    return language_mismatch
```

1.8.2.4 Mapping Message Image Files

The following pseudocode describes the steps taken when a message image file is mapped to resolve an indirect message section descriptor:

```
acquire message mapping lock
open message image file specified in indirect message section descriptor
if file is found then
    map file into memory
    for all message section descriptors in mapped message image file
        calculate offset from indirect message section descriptor
            to mapped message section
        update offset pointed to by message_section_pointer field in the
            message section descriptor
    release message mapping lock
else
    release message mapping lock
    return file_not_found
```

The *first_message_section_desc_pointer* field in both the indirect message section descriptor and the newly mapped direct message section descriptor allows the mapping routine to walk the message section descriptor table resolving all indirect message section descriptors which refer to the file just mapped.

The process of mapping a direct message section descriptor must use a lock to prevent another thread from attempting to map the same message image file.

1.8.3 Initialization of Message Vectors and Loading of Message ISDs

As described in Section 1.8.1.1, the status value translation routines may examine two sets of message section descriptors when attempting to translate a status value. The Image and System Message Vectors are used to access these two sets of message section descriptors. The vectors are initially allocated at image startup by the a routine in the mica$fm_share shareable module. The mica$fm_share module also contains pointers to the message vectors, the status value translation routines, the *exec$install_message_isd* routine, and the message mapping lock used to prohibit race conditions between threads during message image file mapping. The *exec$install_message_isd* routine is called by the image loader whenever an image section descriptor is encountered with the message bit set. It is this routine that is responsible for installing message section descriptor table addresses in the message vectors. This routine is also responsible for allocation of larger message vectors should either vector become full. In this case, a new, larger vector is allocated, the entries from the old vector copied, the pointer to the vector changed to point to the new vector, and the old vector deallocated.

The *exec$install_message_isd* routine examines the first message section descriptor entry in the message section descriptor table. If the *system* field indicates a system message section descriptor, the address of the message section descriptor table is added to the System Message Vector; otherwise it is added to the Image Message Vector.

1.9 Internationalization

As mentioned previously, indirect message sections provide a way to keep message text separate from a program that references it, allowing it to be easily internationalized. Indirect message section descriptors contain the filename of the message image file which contains the corresponding direct message section descriptors and message sections. When this text is needed, the file containing the text is mapped, allowing it to be accessed. Mapping message image files is discussed in Section 1.8.2.4.

Message files on Mica are contained in the SYS$MESSAGE subdirectory of a directory tree set up for each language supported on the system. For example, the message subdirectory containing messages in English is [SYS$LANGUAGE.SYS$ENGLISH.SYS$MESSAGE].

The logical name MICA$LANGUAGE is used to specify the language tree to use to find message files. By default, the translation of MICA$LANGUAGE is used to build the complete file specification when the message image file is opened. For example, if the message image filename specified in the indirect message section descriptor is MY$MESSAGES and MICA$LANGUAGE translates to SYS$GERMAN, the complete file specification is:

```
[SYS$LANGUAGE.SYS$GERMAN.SYS$MESSAGE]MY$MESSAGES.IMAGE
```

The organization of the system directories and a list of logical names which point to them is presented in Chapter 33, Layered Products and System Disk.

1.10 Text Formatting

A text formatting capability is provided with Mica. As stated in Section 1.2, the overall goal is to provide a text formatting capability that addresses internationalization requirements. More specific goals for this functionality are:

- To move data type and access information out of the formatting control string, placing it with the arguments instead
- To provide full parameter positioning and formatting capabilities required for full internationalization support

The directives provide:

- Formatting information such as width, radix, and fill
- Positioning information that allows parameters to be positioned differently for different natural languages
- Special formatting requests such as system date and time
- A means of specifying that directives are to be repeated in a controlled fashon

The basic formatting process is to take zero or more parameters and a source string containing text and formatting directives and produce a resultant string containing the text and parameters formatted as specified by the directives in the source string.

1.10.1 Formatting Directives

A formatting directive is a string that specifies either how a parameter is to be formatted or what information is to be placed in the resultant string. Formatting directives are specified in the following form:

```
%directive[,directive...]%
```

In other words, a directive or comma-separated list of directives is enclosed in percent characters (%).

Following is a list of the formatting directives provided. In these examples, the following syntax notation is used:

- "N" is used to represent a number that is the number of the parameter to be formatted using this directive.
- "W" is used to represent a number that specifies the minimum width of the formatting field. If the formatted parameter requires more than "W" characters, a larger field is used.
- "Z" indicates that a numeric conversion will be done with leading zeros to fill to the specified width (leading blanks are used to fill by default).
- "N" may be specified in the format "N..M" in which case it refers to parameters "N" through "M" inclusive.
- If only certain bits of the specified parameter(s) are desired, the parameter number may be followed by:
  - [x:y]—this form indicates that "y" bits starting at bit "x" will be considered.
  - [x..y]—this form indicates that bits "x" through "y" will be considered.
  - [..x]—this form indicates that bits 0 through "x" will be considered.
  - [x..]—this form indicates that bits "x" through the most significant bit of the parameter will be considered.

  \This is certainly a poor solution to the problem of extracting bits. Ideally, field names should be used, however, this method provides a usable way to do this, if necessary.\

To better facilitate use of repeated directives, the formatting routine maintains a special internal parameter number which may be set, incremented, and decremented. This internal parameter number is acessed as if it were parameter 0 (zero). Upon entry to the *exec$format_text* routine, its value is set to 1. When the internal parameter number (0) is used in a formatting directive, it refers to the actual parameter whose parameter number is the internal parameter number. For example, if the directive *%left(0)%* is specified and the value of the internal parameter is 5, the 5th parameter would be formatted as a left justified string.

\Note that there are no plans to allow internationalization of the formatting directives themselves.\

- *decimal(N:WZ)*—the parameter is formatted in decimal. For floating point parameters, the width may optionally be specified as "W.P" where "P" specifies the precision. The field is zero filled if "Z" is present. Decimal is the only supported directive for formatting floating point values. For floating point parameters, the optional brackets used to select certain bits of a parameter are not allowed. For example:

  ```
  %decimal(5:10)%
  ```

  specifies that the 5th parameter is to be formatted in decimal in a field of width 10.

  \There are certain internationalization issues relating to characters used when formatting floating point. Having the desired language kept with the message text (in the message section) and the language parameter to the *exec$format_text* routine (see below) is intended to address these issues. If this is not sufficient, additional parameters may be added to this and other directives.\

- *hex(N:WZ)*—the parameter is formatted in hexadecimal. The field is zero filled if "Z" is present. Note that no leading characters indicating hexadecimal formatting are inserted. Example:

  ```
  %hex(2)%
  ```

  specifies that the 2nd parameter is to be formatted in hexadecimal in a field just large enough to hold the entire value.

- *octal(N:WZ)*—the parameter is formatted in octal. The field is zero filled if "Z" is present. Note that no leading characters indicating octal formatting are inserted.

- *binary(N:WZ)*—the parameter is formatted in binary. The field is zero filled if "Z" is present. Note that no leading characters indicating binary formatting are inserted.
- *date(N)*—the parameter is formatted in date/time format.

  \See the comment above for the "decimal" directive for internationalization issues.\
- *right(N:W)*—the string parameter is formatted in a right-justified field "W" characters wide. If the string parameter is longer than "W", it is not truncated.
- *left(N:W)*—the string parameter is formatted in a left-justified field "W" characters wide. If the string parameter is longer than "W", it is not truncated.
- *object(N)*—the parameter is the id of an object whose name is to be translated and inserted into the resultant string. If no name exists for the object, the id is output in hexadecimal.
- *plural(N,"singular string","plural string")*—this directive is used to control pluralization. If the value of the specified parameter is 1, "singular string" is inserted into the resultant string; otherwise "plural string" is inserted.

  If only the parameter number is supplied, "" and "s" are used by default.
- *system(item[,item...])*—insert the specified specified system item(s) into the resultant string at this location. Supported system items are:
  - date—current system date
  - time—current system time
  - date_time—current system date and time

  \See the comment above for the "decimal" directive for internationalization issues.\
- *control(item[,item...])*—insert the specified format control item(s) into the resultant string at this location. Supported format items are:
  - tab—tab character
  - new_line—new line indicator
  - form_feed—form feed indicator
- *character(n,c)*—insert the character "c" in the resultant string "n" times.
- *set(N)* or *set(#n)*—set the internal parameter value to the value of parameter "N" or to the numeric value "n". This is normally used prior to the repeat directive.
- *increment(n)*—increment the internal parameter value by "n". If "n" is not specified, the value 1 is assumed.
- *decrement(n)*—decrement the internal parameter value by "n". If "n" is not specified, the value 1 is assumed.
- *repeat(N,(directive,...))* or *repeat(#n,(directive,...))*—repeat the specified list of directives. The number of times to repeat may be specified by the value of a parameter ("N" form) or by an absolute number ("#n" form). The repeat directive in conjunction with the internal parameter value provides a short way to specify output of a list of parameters.
- *text("string of text")*—output the specified text string. This is useful in conjunction with the repeat directive.
- *if({op}N,directive,directive)* or *if({op}#n,directive,directive)*—execute the first directive if the operation specified by {op} is true, otherwise, execute the second directive. Operations compare the value of a specified parameter ("N" form) or a specified value ("#n" form) with the internal parameter value. All numeric values are interpretted as unsigned integers. The following comparison operations are supported:

Table 1–5: Comparison Operations

Operation	Description
=	true if the internal parameter value is equal to the value of parameter "N" or the value "n"
<>	true if the internal parameter value is not equal to the value of parameter "N" or the value "n"
<	true if the internal parameter value is less than the value of parameter "N" or the value "n"
>	true if the internal parameter value is greater than the value of parameter "N" or the value "n"
<=	true if the internal parameter value is less than or equal to the value of parameter "N" or the value "n"
>=	true if the internal parameter value is greater than or equal to the value of parameter "N" or the value "n"

- Two percent signs (%%) are used to insert a single percent sign at the current position in the resultant string.

\Are justification directives needed for entities other than strings (that is, numeric values, etc.)? If so, the direct formats could be enhanced to indicate such justification (use of "R" or "L", for example). This would eliminate the need for the "right" and "left" directives in favor of a more general "string" directive.\

1.10.2 Format Texting

The *exec$format_text* routine provides text formatting support described above. The interface to this procedure is:

```
    PROCEDURE exec$format_text (
                IN source_string : string(*);
                OUT resultant_string : varying_string(*);
                IN language : string(*) OPTIONAL;
                IN parameters : exec$argument_array(*);
                ) RETURNS status;
```

Table 1–6 describes the parameters for this procedure.

Table 1–6: Parameters to exec$format_text

Parameter	Description
source_string	Supplies the source string containing text and formatting directives
resultant_string	Returns the resultant formatted string.
language	An optional string that supplies a language name to override the current default language. Language is used to determine language-dependent formats for parameters formatted into the message string.
parameters	Supplies an array of parameters to be formatted into resultant string. The data type *exec$argument_array* is an array of *exec$argument_descriptor*. The data type *exec$argument_descriptor* is defined in Chapter 9, Condition, Exit, and AST Handling.

The *exec$format_text* routine copies text from the source string into the resultant string, formatting parameters as formatting directives are encountered.

1.10.2.1 Examples

Following are several examples using the formatting capabilities described above. In all cases, English is assumed to be the current language.

```
Control string: "Integer divide by zero at PC=%%x%hex(1)%, PSL=%%x%hex(2)%"
Parameters: %x4782a7, %x4003a2
Formatted string: "Integer divide by zero at PC=%x4782A7, PSL=%x4003A2"

Control string: "Undefined symbol %left(1)% referenced in "+
                "psect %left(2)%, offset %%x%hex(3)% in module "+
                "%left(4)%, file %left(5)%"
Parameters: FROBOTZ, MY$$PSECT, %x4896b, MY_MODULE, MY.OBJ
Formatted string: "Undefined symbol FROBOTZ referenced in
                   psect MY$$PSECT, offset %x4896B in module
                   MY_MODULE, file MY.OBJ"

Control string: "%decimal(1)% file%plural(1," was","s were")% deleted"
Parameters: 10
Formatted string: "10 files were deleted"
Parameters: 1
Formatted string: "1 file was deleted"

Control string: "%set(#2),repeat(1,(left(0),increment," +
                "if(=1,text(" "),text(", ")))% deleted"
Parameters: 4, file1.dat, another.file, third.one, last.file
Formatted string: "file1.dat, another.file, third.one, last.file deleted"

Control string: "Current system date and time is %system(date_time)%"
Parameters: none
Formatted string: "Current system date and time is 18-Dec-1986, 08:42.45"
```

1.11 Open Issues

There are several details which will eventually need further clarification.

- Some of the internationalization issues may need further detail.
- The mechanism for assigning and tracking facility numbers is TBD.
- Multi-threaded server processes running on the compute server require that each server thread potentially have a different default language. The indirect message section descriptor support could be expanded to allow this by:
  - Providing default language overrides to the *exec$get_message*, *exec$display_message*, and *exec$get_text* routines.
  - Allowing message sections in different languages to be chained; that is, a given message section may point to another message section containing the same messages written in another language.
- Policy for message translation when the message file does not exist in the requested language (error return vs. fallback to a default language).

1.12 Dependencies

The implementation of the status and message support described in this chapter depends on certain capabilities provided by other system components. These dependencies are listed below.

- Message Compilation Facility

  TBS

- Linker

  TBS

- Image Activation Mechanism

  TBS

- Condition Handling Data Structures

  TBS

- Client Context Server/Mica Job Controller Logical Name Transfer

  TBS