PRISM Common Layered Product Software Architecture

Types: A = part of AIA (*probably*)
P = PRISM-specific (necessarily)
N = Not by nature PRISM-specific, but not part of AIA

Note: "When" == When a product ships that needs this

Note: "*" == could temporarily finesse without it, but much better off with it.

Note: Some of these are just architectural specs, and some have real deliverable code associated with them.

Type	Description	When	Who Responsible
P	1. Extended Calling Standard	3/89	Nylander, Calling Std Group
P	2. PRISM condition handling	3/89	Bismuth and Calling Std Group
P	3. Extended object language	3/89	Peterson, Title, Grove, MacLaren, PCA, et. al.
N	4. Librarian Interface	3/89	Walp
?	5. Status codes, message files, message language, message formatting and reporting routines.	3/89	Ballenger
A	6. IPSE architectural spec (SCA data files, LSE diag files, callable interfaces)	3/90	Beander, et. al.
P	7. Name Space rules	3/89	Calling Standard
A	8. Common record management interfaces, including transperent filespec interpretation	3/89*	Chatterjee
N	9. Transperent command language interface	3/89*	Ballenger
A	10. Remote Procedure Calls	3/90	Corporate RPC architecture
A	11. Common Multithread Architecture	3/90	Conti and CMA working group
A	12. Common date/time format and run-time interfaces	3/89*	NAC and Al Simons
A	13. Common AIA utiility RTL	3/89*	Simons
A	14. Common Language RTL	3/89	Lapine
A	15. Common Math RTL	3/89	Wiener
A	16. Windows (DECwindows)	3/89	Corporate

?	17.	Dumb terminal I/O	3/89	Connors and Simons
A	18.	Memory allocation and freeing	3/89	Simons
A	19.	String format(s) and run-time interfaces	3/89	Simons
A	20.	%DIF architecture and run-time interfaces (DDIF, TDIF, etc.)	3/90	Corporate (and Core Applications?)

Outline of PRISM Common Layered Product Software Architecture

(2)

+---------------+
! d i g i t a l ! I n t e r o f f i c e M e m o r a n d u m
+---------------+

To: List

Date: 13 November 1987
From: Chip Nylander
Dept: Technical Languages
DTN.: 381-2057 Loc.: ZK 2-3/N30

Subj: Outline of PRISM Common Layered Product Software Architecture

Types: A = part of AIA (*probably*)
P = PRISM-specific (necessarily)
N = Not by nature PRISM-specific, but not part of AIA

Note: "When" == When a product ships that needs this

Note: "*" == could temporarily finesse without it, but much better off with it.

Note: Some of these are just architectural specs, and some have real deliverable code associated with them.

Type	Description	When	Who Responsible
P	1. Extended Calling Standard	3/89	Nylander and Calling Std Working Group
P	2. PRISM condition handling	3/89	Bismuth and Calling Std Working Group
P	3. Extended object language	3/89	Peterson, Title, Grove, MacLaren, PCA, et. al.
N	4. Librarian Interface	3/89	Walp
?	5. Status codes, message files, message language, message formatting and reporting routines.	3/89	Ballenger
A	6. IPSE architectural spec (SCA data files, LSE diag files, callable interfaces)	3/90	Beander, et. al.
P	7. Name Space rules	3/89	Calling Standard Working Group
A	8. Common record management	3/89*	Chatterjee

(3)

		interfaces, including transperent filespec interpretation		
N	9.	Transperent command language interface	3/89*	Ballenger
A	10.	Remote Procedure Calls	3/90	Corporate RPC Architecture
A	11.	Common Multithread Architecture	3/90	Conti and CMA Working Group
A	12.	Common date/time format and run-time interfaces	3/89*	NAC and Al Simons
A	13.	Common AIA utiility RTL	3/89*	Simons
A	14.	Common Language RTL	3/89	Lapine
A	15.	Common Math RTL	3/89	Wiener
A	16.	Windows (DECwindows)	3/89	Corporate
?	17.	Dumb terminal I/O	3/89	Connors and Simons
A	18.	Memory allocation and freeing	3/89	Simons
A	19.	String format(s) and run-time interfaces	3/89	Simons
A	20.	%DIF architecture and run-time interfaces (DDIF, TDIF, etc.)	3/90	CDA Program

From: TLE::RTL::SIMONS "Al Simons 381-2187 24-Nov-1987 1623" 24-NOV-1987 16:
To: DECWET::CUTLER,DECWET::DON,DECWET::SCHREIBER,TLE::NYLANDER,SIMONS,CLT::
Subj: The PRISM Software architecture: who, when?

I was asked by Dave Cutler for a summary of the PRISM software architecture indicating who was responsible for each component, and for those for which I had responsibility , when the spec would be available. The dates in this note indicate my current best guess. I plan to start formal scheduling at about the start of the new year.

Here is a list of the software architecture responsibilities, as we decided them on 02-Oct-87 at DECwest.

The list was initially typed in by Chip. I have verified the list against my notes, and have put in more information on items for which I think the RTL has full or partial responsibility.

-Al
****

PRISM Common Layered Product Software Architecture

Types: A = part of AIA (*probably*)
P = PRISM-specific (necessarily)
N = Not by nature PRISM-specific, but not part of AIA

Note: "When" == When a product ships that needs this

Note: "*" == could temporarily finesse without it, but much better off with it.

Note: Some of these are just architectural specs, and some have real deliverable code associated with them.

Type	Description	When	Who Responsible
P	1. Extended Calling Standard	3/89	Nylander, Calling Std Group
P	2. PRISM condition handling	3/89	Bismuth and Calling Std Group
P	3. Extended object language	3/89	Peterson, Title, Grove, MacLaren, PCA, et. al.
N	4. Librarian Interface	3/89	Walp
?	5. Status codes, message files, message language, message formatting and reporting routines.	3/89	Ballenger
A	6. IPSE architectural spec (SCA data files, LSE diag files, callable interfaces)	3/90	Beander, et. al.
P	7. Name Space rules	3/89	Calling Standard

A 8. Common record management interfaces, including transparent filespec interpretation — 3/89* Chatterjee

N 9. Transparent command language interface — 3/89* Ballenger

A 10. Remote Procedure Calls — 3/90 Corporate RPC architecture

A 11. Common Multithread Architecture — 3/90 Conti and CMA working group

+++

SDT will be implementing on VAX/VMS. I have seen papers saying that this is planned to be implemented by DECwest. I am currently NOT planning on implementing this for MICA at SDT.

A 12. Common date/time format and run-time interfaces — 3/89* NAC and Al Simons

+++

Common date/time format specification: NAC

Run time conversion interfaces: Simons

To/from U*X format (If this format is adopted for use in the Ultrix system also.)
To/from VAX/VMS quadword format (needed for file system in workgroups)
To/from text strings.

SDT design responsibility; undetermined implementation responsibility. (Some of the conversion routines could conceivably reside in the exec, I presume that DECwest would want to implement them.)

Actual timekeeping / storing / retrieving routines are DECwest responsibility.

Spec responsibility (conversions) is Simons.
Spec date is Apr 88.

A 13. Common AIA utility RTL — 3/89* Simons

+++

Common Utility RTL consists of:
OS interface routines
Utility routines

See the ARUS WDD chapter overview for more info. Joint design responsibility (SDT primary). SDT primary implementation group. DECwest assistance will probably be necessary in implementing some of the system interfaces.

Spec responsibility is Simons.
Spec date is probably Apr 88. (Some portions sooner.)

A 14. Common Language RTL 3/89 Lapine
+++

Currently working on FORTRAN and PASCAL specs.
SDT responsible for design and implementation.
Specs have been written for selected portions
of the language RTLs, for instance the Fortran
I/O system. These specs are available now by
contacting CLT::LAPINE.

Spec responsibility is Lapine.
Overall spec date is Feb '88.

A 15. Common Math RTL 3/89 Wiener
+++

Currently working on VAX/VECTORS. No current date for
PRISM work. Relying on the availability of PILLAR to
write routines for 3/89. SDT responsible for design
and implementation.

Spec responsibility is Wiener.
Spec date is TBD.

A 16. Windows (DECwindows) 3/89 Corporate

? 17. Dumb terminal I/O 3/89 Connors and Simons
+++

Simons is SDT contact person for this item. Actual
engineer working with Myles is Tom Scarpelli.
Joint design, undecided implementation.
Spec responsibility is Connors. (We understand that this
is currently under discussion at DECwest.)

A 18. Memory allocation and freeing 3/89 Simons
+++

Actually a part of the Utility RTL item above, but
important enough to be called out separately. We believe
that this is the most important piece of RTL code, because
of the frequency of use. It will be designed first.
SDT or DECwest implementation is undecided at this point.

We expect to begin the interface design in early January
so that we can have significant review.

Spec responsibility is Simons.
Spec date is Feb '88.

A 19. String format(s) and run-time interfaces 3/89 Simons
+++
Also part of the Utility RTL. Separate from that grouping because at least some parts of this package will be defined in the PRISM Extended Calling Standard as the system defined method for allocating/deallocating dynamic strings. [Decision of the calling std. meeting, Nov-87] The portions that directly affect the completion of the calling standard will be architected immediately after the memory allocation/deallocation routines. The remainder will be architected along with the rest of the Utility RTL, specifically the general utility routines portion. SDT implementation.

Spec responsibility is Simons.
Spec date is Mar '88 for Call Std. portion.

A 20. %DIF architecture and run-time interfaces (DDIF, TDIF, etc.) 3/90 Corporate (and Core Applications?

Digital Equipment Corporation - Confidential and Proprietary
For Internal Use Only

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

Revision 0.7

6–April–1988

Issued by:
Kris K. Barker

digital™

TABLE OF CONTENTS

INDEX

FIGURES

TABLES

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.
16-MAR-1988	0.6	Kris Barker	Revisions following architect review and 64- vs. 32-bit status meetings.
6-APR-1988	0.7	Kris Barker	Revisions following text formatting support discussions.

CHAPTER 1

STATUS VALUES, MESSAGES, AND TEXT FORMATTING

1.1 Introduction

Status values pass information regarding the success or failure of a process, thread, I/O service, 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 status 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 1.9.2 for a discussion on message searching.
Status	A 64-bit numeric value containing a 32-bit status value and 32 bits of additional information, the interpretation of which depends on the status value.
Status value	A 32-bit numeric value containing information about the status of a thread, process, procedure, or I/O request.

1.4 Status on Mica

Mica status is 64 bits. The first 32 bits are the status value; depending on the type of status, the second 32 bits may or may not be used. Mica defines three types of status: *facility-registered status*, *local status*, and *internal status*. The format of each type is shown in Figure 1–1. Status value formats are defined in Section 1.5.

- Facility-registered status—The status value contains a number indicating which facility generated the status. The second 32 bits are not used.

- Local status—The status value has the local status bit set. A full 25 bits are used for the message number as a facility number is not required. The second 32 bits of the status contain the address of a message data structure used to acquire the message text.
- Internal status—This type of status is used internally by a particular facility. The first 32 bits is a facility-registered status value. The second 32 bits may be used in whatever way the facility desires. An internal status normally does not appear outside the facility that uses it because outside the facility, the second 32 bits of the status are ignored.

Figure 1–1: Mica Status

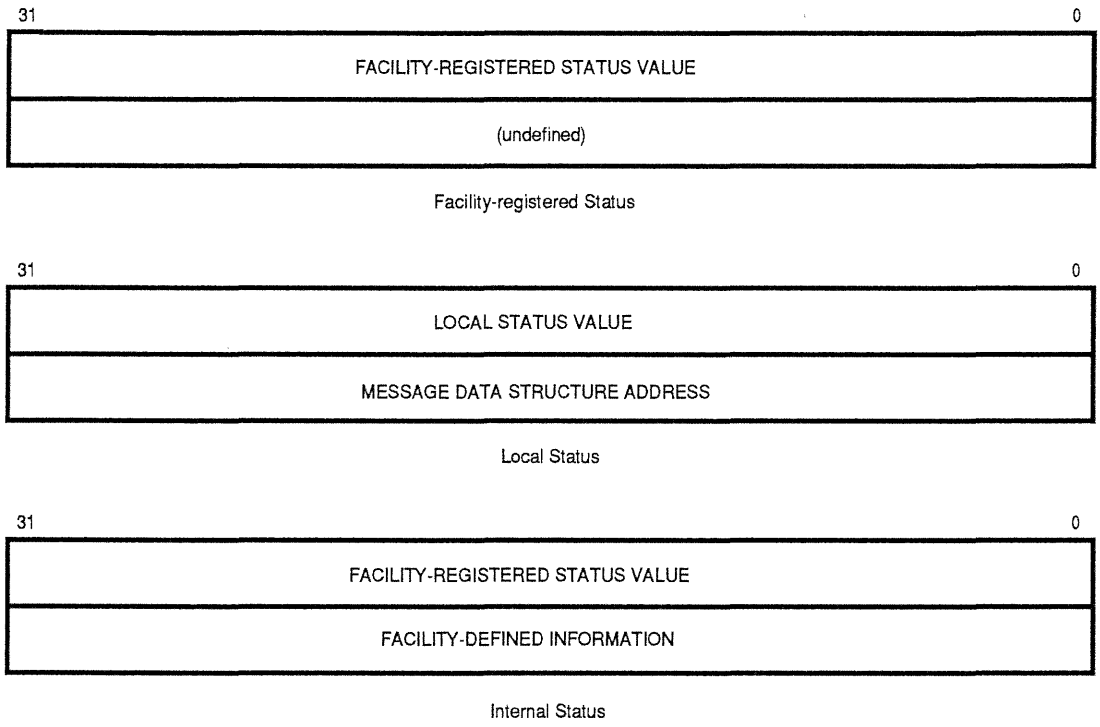


FIG0

Pillar's predefined data type STATUS is 64 bits. For languages which do not support 64-bit return values, such as C and FORTRAN, procedures which return local status should include an optional argument that returns a condition vector. This condition vector contains the full 64-bit status; the entire status is required for local message retrieval.

1.5 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 completion status from an I/O request
- 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 1–2: exec$status_value

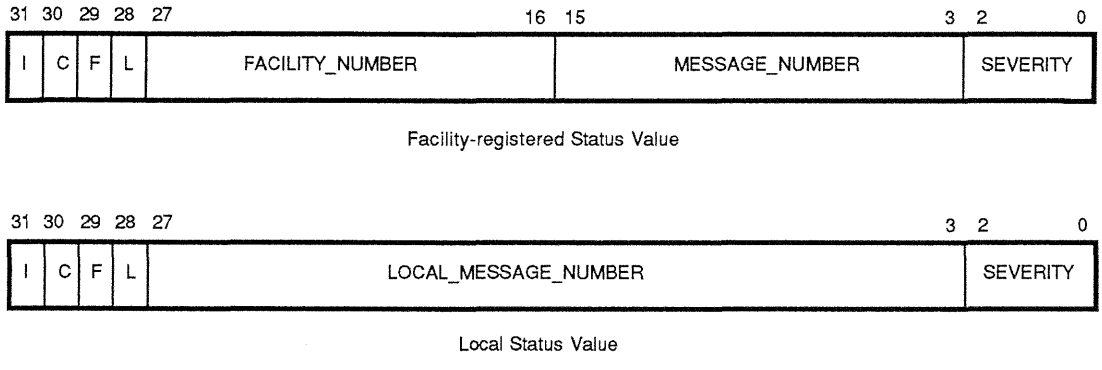


FIG1

```
exec$status_value : RECORD
    severity : integer[0..7] SIZE(BIT,3);
    UNION CASE *
        WHEN 1 THEN
            message_number : integer[0..8191] SIZE(BIT,13);
            facility_number : integer[0..4095] SIZE(BIT,12);
        WHEN 2 THEN
            local_message_number : integer[0..2**25-1] SIZE(BIT,25);
    END CASE;
    local_status : bit;
    facility_specific : bit;
    customer_facility : bit;
    inhibit_message_printing : bit;
    LAYOUT
        severity;
        UNION
            OVERLAY
                message_number;
                facility_number;
            OVERLAY
                local_message_number;
        END UNION;
        local_status POSITION(BIT,28);
        facility_specific POSITION(BIT,29);
        customer_facility POSITION(BIT,30);
        inhibit_message_printing POSITION(BIT,31);
    END LAYOUT;
END RECORD;
```

Mica facility-registered status values are similar to status values on VAX/VMS. The differences are the *inhibit_message_printing* bit (bit 31 on Mica), the *local_facility* bit (bit 28 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.

1.5.1 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 7 (values of 5 and 6 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.
7	Text	This value is used to indicate that the status value is being used to access a text message rather than a status message. In this case, the concept of severity does not apply. It is an error to attempt to obtain and format a text message with *lib$get_message* or *lib$display_message*.

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.

1.5.2 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 1.8.4. This field is defined only for facility-registered status values.

1.5.3 FACILITY_NUMBER Field (bits <27:16>)

The *facility_number* field of a status value is used to identify the producer of the status value. Each facility must have its own unique facility number. This field is defined only for facility-registered status values.

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

1.5.4 LOCAL_MESSAGE_NUMBER Field (bits <27:3>)

The *local_message_number* field of a status value is used to index into a message section to obtain message text for a local message. This field is defined only for local status values.

1.5.5 LOCAL_STATUS Field (bit 28)

The *local_status* field is used to indicate that the status value is local. Local status values have this bit set; facility-registered status values have this bit clear.

1.5.6 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 1.6.7.

1.5.7 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.

1.5.8 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.

1.5.9 Pending Status

A status value with all fields zeroed is reserved to indicate pending status. Processes, threads, I/O requests, and so on, which return status asynchronously should always initialize returned status to *status$_pending* (the zero status value). This informs the consumer that the final status value has not yet been set.

1.6 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, possibly with formatted parameters specific to the error occurrence.

1.6.1 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 SYSTEM$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 *lib$display_message* supports display of multiple messages. In this case, the first message formatted is termed the primary message; successive messages are termed secondary messages. The *lib$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 *lib$get_message* and *lib$display_message* routines are describe in Section 1.6.4 and Section 1.6.5.

1.6.2 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.

1.6.3 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 1.9.2.4 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 1.8 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 1.10 and in Chapter 35, System Volume Layout and Software Installation.

1.6.4 Obtaining and Formatting Status Messages—*lib$get_message*

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

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

Parameters:

Parameter	Description
condition_vector	Supplies a condition vector containing the status for which a message is to be returned. Only the message corresponding to the primary condition is returned. For local messages, the status in the condition vector contains the address of the message section descriptor which points to the message section containing the local message. The format and content of condition vectors is presented in Chapter 11, Condition, Exit, and AST Handling. Message section descriptors and message sections are discussed in Section 1.8.
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 1.6.6 and Section 1.6.7 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. See Section 1.11 for more information on formatting directives.

flags	Optionally supplies a set of type *lib$message_options* that indicates which portions of the status message are to be returned. Each element of the set that is supplied – *lib$c_facility*, *lib$c_severity*, *lib$c_condition_name*, *lib$c_message_text* – indicates that the corresponding field should be included in the formatted status message. If this argument is not supplied, the default format is used, as specified by the SYSTEM$MESSAGE_FORMAT logical name supplied by the client. The data type *lib$message_options* is defined as:
	`lib$message_options_type : (`
	`lib$c_severity,`
	`lib$c_facility,`
	`lib$c_condition_name,`
	`lib$c_message_text`
	`);`
	`lib$message_options : SET[lib$message_options_type];`

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 data type *lib$message_user_value* is defined as: `lib$message_user_value : longword;` The interpretation of this value is the responsibility of the caller of *lib$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 1.8 and Section 1.9 for more information on the organization of message vectors and message sections and the mechanisms used to traverse them.

1.6.5 Obtaining and Displaying Status Messages—*lib$display_message*

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

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

Parameters:

Parameter	Description
condition_vector	Supplies a condition_vector containing the status values to be formatted and output. Unlike *lib$get_message* described above, *lib$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 *lib$message_options* that indicates which portions of the status message are to be displayed. Each element of the set that is supplied – *lib$c_facility*, *lib$c_severity*, *lib$c_condition_name*, *lib$c_message_text* – indicates that the corresponding field should be included in the formatted status messages. If this argument is not supplied, the default format is used, as specified by the SYSTEM$MESSAGE_FORMAT logical name supplied by the client. \This method of specifying the message formatting flags makes it impossible, using the *lib$display_message* routine, to specify different formatting for each status value in the specified condition vector. 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.
	`PROCEDURE lib$action_procedure (`
	`IN message_string : string(*) CONFORM;`
	`IN action_parameter : lib$action_parameter = zero;`
	`) RETURNS boolean;`
	The two arguments to this routine are the formatted message string and the action parameter (see below) supplied in the call to *lib$display_message*. This routine must return a Boolean value: if TRUE, the message is output by *lib$display_message*; if FALSE, it is not.

action_parameter	Optionally supplies a value of type *lib$action_parameter* that is passed to the action routine. The data type *lib$action_parameter* is defined as: `lib$action_parameter : longword;`

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 1.8 and Section 1.9 for more information on the organization of message vectors and message sections and the mechanisms used to traverse them.

1.6.6 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 with the *local_status* bit set are used to reference local messages. Note that since a facility number is not needed, a full 25 bits are used as the message number for local status values.

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 address in the status contained in condition vector or passed by procedure argument—rather than implicitly by their presence in a message section descriptor table whose address is in a message vector. When *lib$get_message* or *lib$display_message* is called to obtain the text for a local message, the status contained in the supplied condition vector specifies the address of the message section descriptor to be examined. When *lib$get_text* is called to obtain the text for a local message, the *status_argument* argument supplies the local status value and 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 1.9 for more information on the mechanisms used to translate status values to status and text messages.

1.6.7 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*, *facility_specific*, and *local_status* 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 * lib$c_facility_offset +
               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 1.9 describes the mechanisms used to translate status values to status and text messages. Section 1.9.1.3 describes how these mechanisms are affected by shared messages.

1.7 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.

1.7.1 Relationship to Status Messages

Definition and organization of text messages is the same as that described for status messages in Section 1.6 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 *local_status* bit set.
- The routine which accesses and formats text messages does not require that access and parameter information be supplied in condition vector format. This means that programs which use this functionality to organize user-visible text will not be required to handcraft condition vectors.

1.7.2 Obtaining and Formatting Text Messages—*lib$get_text*

The *lib$get_text* routine obtains and formats a text message based on a supplied status. The interface to this procedure is:

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

Parameters:

Parameter	Description
status_argument	Supplies a status value and, for local messages, the address of a message section descriptor, used to locate the text message.
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 PRISM Calling Standard.
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.

1.8 Message Data Structures

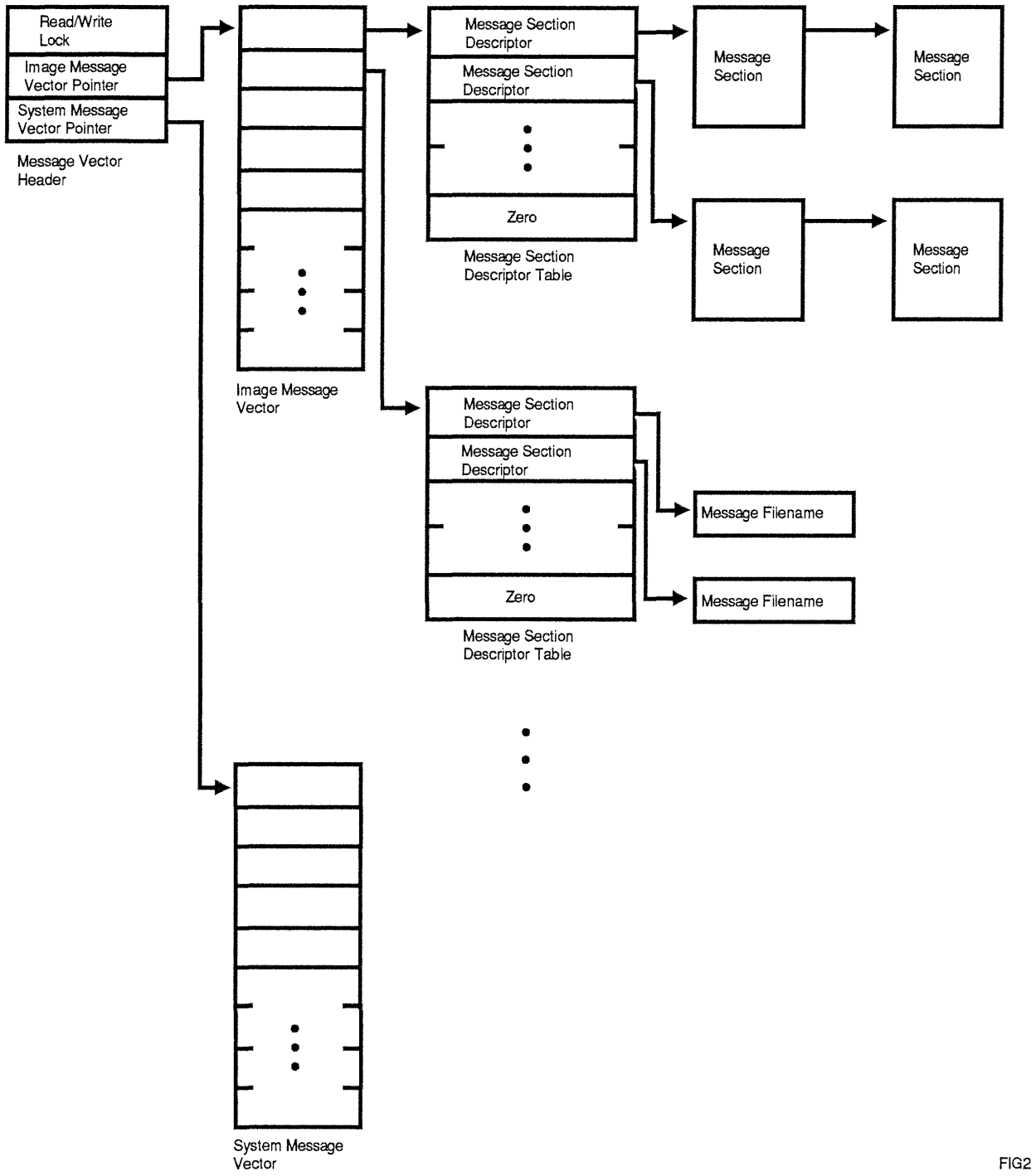
The outputs of Mica's message compilation facility are *message object modules*. These modules contain message information in structures called *message sections*, *message section descriptors*, and *message section descriptor tables*. Once in memory, message information is organized into:

- *Message vector header*—This structure provides a multiple reader/single writer lock to control updating of message structures. It also contains pointers to the system and image message vectors for the process.
- *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 in one natural language. Message sections may be chained together to provide support for multiple languages within one process.

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

Figure 1–3: In-Memory Message Data Structure Organization



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

1.8.1 Message Vector Header and Message Vectors

The message vector header is the top level structure used to access all other message data structures. It contains a multiple reader/single writer lock used to control updating of message data structures and pointers to two 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.

1.8.2 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.

1.8.3 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 1–4: lib$message_section_desc

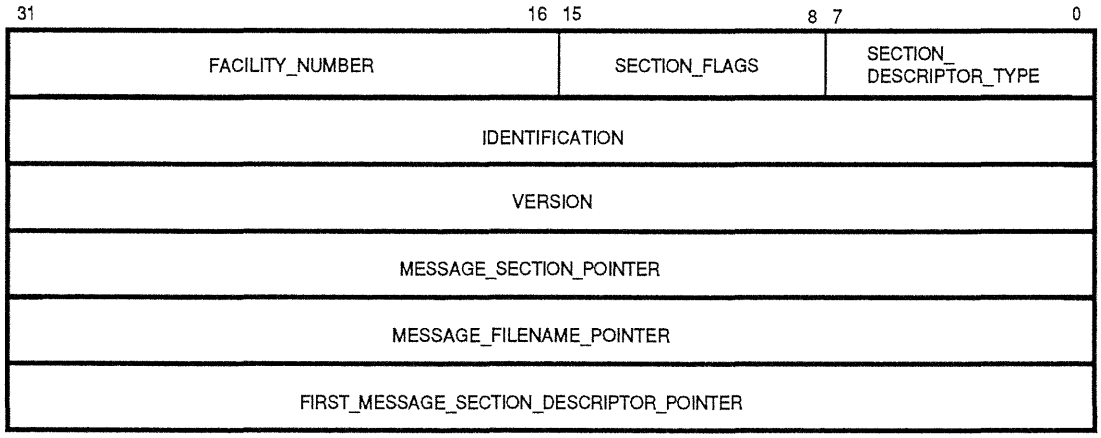


FIG3

```
lib$message_section_desc : RECORD
    section_descriptor_type : lib$message_section_desc_types[..] SIZE(BYTE);
    section_flags : lib$message_section_desc_flags SIZE(BYTE);
    facility_number : lib$facility_number;
    ident : longword;
    version : longword;
    message_section_pointer : POINTER lib$message_section_pointer;
    message_filename_pointer : POINTER lib$counted_string;
    first_msg_section_desc_pointer : POINTER lib$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;

lib$message_section_desc_types : (
    lib$c_direct_section_desc,
    lib$c_indirect_section_desc
    );

lib$msg_section_desc_flag_type : (
    lib$c_system_section_desc,
    lib$c_local_section_desc
    );
lib$message_section_desc_flags : SET lib$msg_section_desc_flag_type[..];

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

lib$message_section_pointer : POINTER lib$message_section;
```

These fields are defined as follows:

- *section_descriptor_type*—This field indicates the type of the message section descriptor:
  - — *lib$c_direct_section_desc*—This type indicates that the message section is loaded into memory with the section descriptor. This means that the section descriptor and message section are part of the program image file. Message section descriptors whose *section_descriptor_type* is *lib$c_direct_section_desc* are referred to as *direct message section descriptors*.
  - — *lib$c_indirect_section_desc*—This type indicates that the message section is contained in a separate message image file. The filename of the separate message image file is contained in a data structure pointed to by the *message_filename_pointer* field in the message section descriptor. Message section descriptors whose *section_descriptor_type* is *lib$c_indirect_section_desc* 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.
- *section_flags*—Flags used to identify system and local message section descriptors.
  - — If the *lib$c_system_section_desc* bit is TRUE, 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.

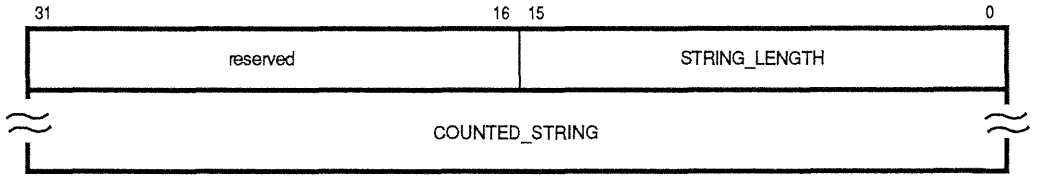
    \The intent is that this is for DIGITAL use only. PSECT naming conventions are be used to handle the case where system and nonsystem section descriptors are linked into the same file.\

— If the *lib$c_local_section_desc* bit is TRUE, the message section contains local messages rather than facility-registered messages.

- *facility_number*—The facility number associated with the messages in the section pointed to by this section descriptor. For local messages, this value is not used.
- *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 1–3.\
- *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 1–5.
- *first_msg_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 1–5: lib$counted_string




```
lib$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;
```

1.8.4 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 1–6: lib$message_section

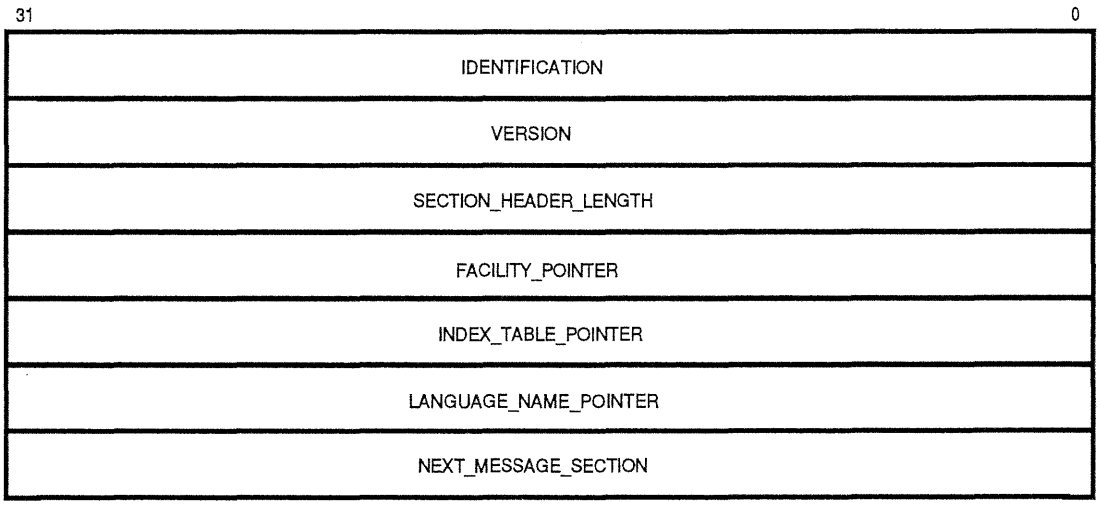


FIG4

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

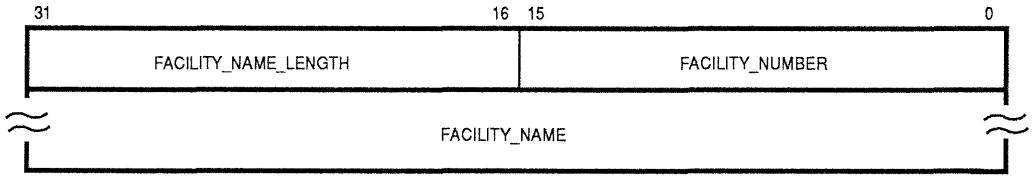
These fields are defined as follows:

- *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 *lib$facility_name* that contains the facility number and name for messages in the section, as shown in Figure 1–7. For local message sections, this pointer is NIL.
- *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.

  The language name is a string of uppercase characters which expresses the name of the language spelled in English.
- *next_message_section*—A pointer to another message section. This field is used to chain message sections of different languages together. Such support is required for multithreaded server processes that serve multiple clients, each of which may have a different default language.

Figure 1–7: lib$facility_name




```
lib$facility_name ( facility_name_length : integer[0..65535] SIZE(WORD)) : RECORD
    CAPTURE facility_name_length;
    facility_number : lib$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 1–8: lib$message_index_table

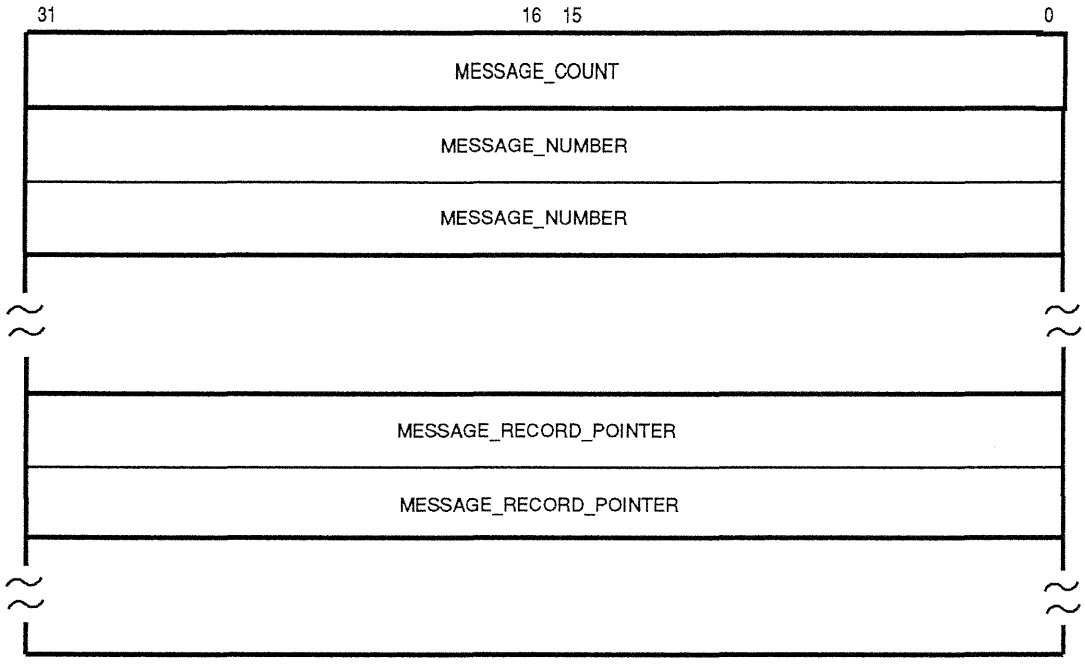


FIG6

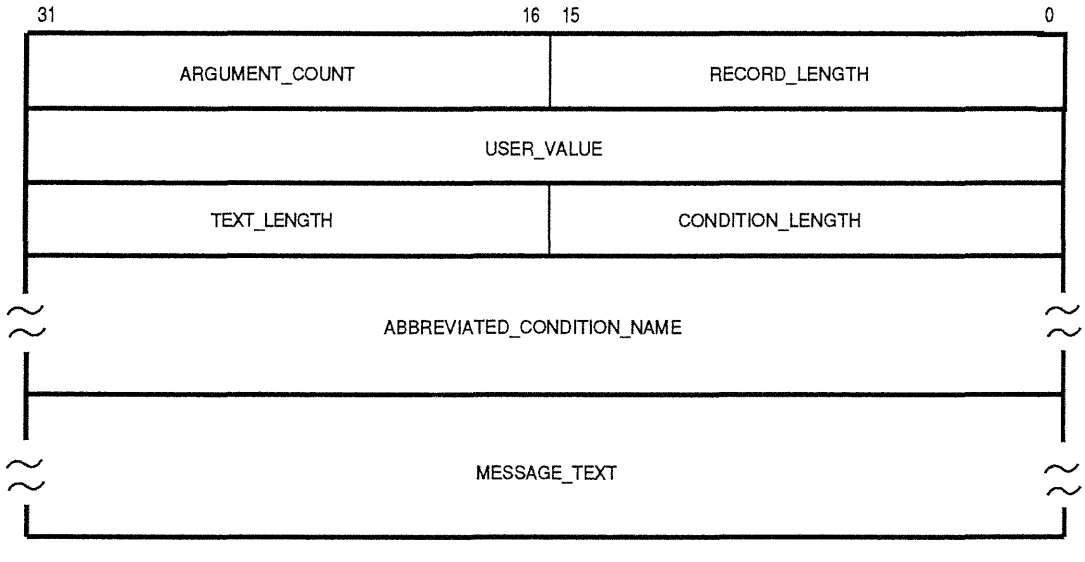
```
lib$message_index_table ( message_count : integer [0..2**25-1] ) : RECORD
    CAPTURE message_count;
    message_number : ARRAY[0..message_count] OF integer;
    message_record_pointer : ARRAY[0..message_count] OF POINTER lib$message_record;
    LAYOUT
        message_count;
        reserved1 : FILLER(WORD,*);
        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 1–9: lib$message_record




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

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 value of type *lib$message_user_value* specified in the message source file which can be returned to the caller of *lib$get_message* or *lib$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. The message text is stored in Digital Data Interchange Syntax (DDIS) form. This will allow future versions of message support to store not only ASCII text, but graphical information as well. For Mica FRS, however, only ASCII text is supported.

1.9 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.

1.9.1 Which Message Sections are Searched

As mentioned above, the status value translation routines, *lib$get_message*, *lib$display_message*, and *lib$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*, *facility_specific*, and *local_status* 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.

1.9.1.1 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.

1.9.1.2 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 to search is specified by an address in the second 32 bits of a status. This status is contained in a condition vector or passed as a procedure argument.

1.9.1.3 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.

1.9.2 How Message Sections are Searched

This section describes how message sections are searched.

1.9.2.1 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:

```
set lock mode to read
acquire lock specified by lock mode              ! lock point
if status value indicates local message then
    examine message section descriptor specified by condition vector 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
release lock specified by lock mode
```

1.9.2.2 Examining a Message Section Descriptor

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

```
if status value indicates local message then
    if message section descriptor is indirect then
        if message section for desired language is 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 then
            if message section for desired language is not mapped then
                map corresponding message image file
        search message section for message
```

1.9.2.3 Searching a Message Section

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

```
if local_status bit in status value matches local field in message section descriptor then
    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
        search English message section for message
        return warning indicating language is not what was requested
else
    return local/facility-registered status translation mismatch
```

1.9.2.4 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:

```
if lock mode is read then
    release lock
    update lock mode to write
    goto lock point (uplevel goto)
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
        if indirect message section descriptor not mapped then
            update offset pointed to by message_section_pointer
        else
            update next_message_section field in last message section in chain
else
    return file_not_found
```

The *first_msg_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 to resolve all indirect message section descriptors which refer to the file just mapped.

If the indirect message section descriptor has already been mapped, the newly mapped message sections are chained off of the last message sections in the chain.

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

1.9.3 Initialization of Message Vectors and Loading of Message ISDs

As described in Section 1.9.1.1, 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 a pointer to the message vector header, the status value translation routines, and the *lib$install_message_isd* routine. The *lib$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, the write lock is acquired, a new, larger vector is allocated, the entries from the old vector are copied, the pointer to the vector is changed to point to the new vector, the old vector is deallocated, and the write lock is released.

The *lib$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.

1.10 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 1.9.2.4.

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

The *arus$get_user_language* routine is used to obtain the user's default natural language. This default language is then used to build the complete file specification when the message image file is opened. For example, if the message image filename specified in the indirect message section descriptor is MY$MESSAGES and *arus$get_user_language* returns "GERMAN" as the user's default language, the complete file specification is:

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

The organization of the system directories and a list of logical names which point to them is presented in Chapter 35, System Volume Layout and Software Installation.

1.11 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 date and time
- A means of specifying that directives are to be repeated in a controlled fashon
- A means of including text based on parameter value or length

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.

1.11.1 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 within percent characters (%).

Table 1–2 describes each formatting directive. 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.

  These bit forms are only allowed with parameters of type libc_byte_data, libc_word_data, lib$c_longword_data, and lib$c_quadword data. See the PRISM Calling Standard for a description of parameter data types.

  \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.\
- "V" is used to represent either a parameter number or a numeric or string constant. Parameter numbers are specified by a number only. Numeric constants are specified by preceeding the value with a "#" sign. String constants are enclosed in double quotation marks ("string"). The *length* directive returns a value that can be used wherever a numeric constant is allowed.

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 accessed as if it were parameter number 0 (zero). Upon entry to either the *lib$format_single_string* or *lib$multiple_strings* routine, its value is set to 1. When the internal parameter number (0) is used in an inserting formatting directive, such as %left% or %binary%, its value is used to refer to a 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. When the internal parameter number is used in a comparison or controlling formatting directive, such as %if% or %repeat%, its value is used directly.

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

Table 1–2: Formatting Directives

Directive[1]	Description
decimal(N[:W[Z]][,"RT"])	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. Normally, floating point parameters are formatting with the user's preferred radix point and thousands separator character. This may be optionally overridden by specifying "RT" as part of the directive, where "R" is the radix point character and "T" is the thousands separator character. For example: `%decimal(5:10.2,",.")%` specifies that the 5th parameter is to be formatted in decimal in a field of width 10 and a precision of 2. Additionally, the "," character is used to specify the radix point and the "." character is used as the thousands seperator. 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.
hex(N[:W[Z]])	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[:W[Z]])	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[:W[Z]])	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][:F][,ALIGN])	The specified parameter is formatted in date format. If no parameter is supplied, the current system date is formatted.[2]Normally, the date is formatted using the user's preferred date format. If ":F" is specified, the date is formatted using date format "F". See Chapter 58, Application Run-Time Utility Services for more information on date formats. If "ALIGN" is specified, the formatting routine will force the date field width to be the maximum produced by the specified date format. This is useful when text is formatted in columns.
time([N][:F][,ALIGN])	The specified parameter is formatted in time format. If no parameter is supplied, the current system time is formatted.[2]Normally, the time is formatted using the user's preferred time format. If ":F" is specified, the time is formatted using time format "F". See Chapter 58, Application Run-Time Utility Services for more information on time formats. If "ALIGN" is specified, the formatting routine will force the time field width to be the maximum produced by the specified time format. This is useful when text is formatted in columns.

[1]Table 1–3 presents the rules for which parameters and constants are allowed with which directives.

[2]The *lib$format_single_string* and *lib$format_multiple_strings* routines use the ARUS date/time formatting services to format the date and time. These services provide full internationalization capabilities as well as support for multiple date/time formats. See Chapter 58, Application Run-Time Utility Services for more information.

Table 1–2 (Cont.): Formatting Directives

Directive[1]	Description
date_time([N][:F:G][,ALIGN])	The specified parameter is formatted in date/time format. If no parameter is supplied, the current system date and time are formatted.[2]Normally, the date and time are formatted using the user's preferred date and time formats. If ":F:G" is specified, the date is formatted using date format "F" and the time is formatted using time format "G". See Chapter 58, Application Run-Time Utility Services for more information on date and time formats. If "ALIGN" is specified, the formatting routine will force the date/time field width to be the maximum produced by the specified date/time formats. This is useful when text is formatted in columns.
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.
center(N[:W])	The string parameter is centered in a field "W" characters wide. If the string parameter is longer than "W", it is not truncated.
length(directive)	The specified directive is evaluated and the formatted string is returned as an integer constant. Note that this is the only directive that does not insert characters into the resultant string. This directive may be used wherever a numeric constant is allowed.
plural(N[,"zero string"[,"singular string"[,"2 string",...,"n string"]]])	This directive is used to control pluralization. The directive allows specification of different strings to be inserted into the resultant string for different values of the specified parameter. The first string corresponds to a value of zero, the second string to a value of one, the third to a value of two, and so on. The final string is used for values greater than or equal to "n". If only the parameter number is supplied, "" is used for the singular case and "s" is used for the zero and more than singular case.
system(item[,item...])	Insert the specified system item(s) into the resultant string at this location. System items are typically specific to a particular operating system and should be avoided in cases where format strings are used across multiple systems. Supported system items on Mica are: • *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.
control(item[,item...])	Insert the specified format control item(s) into the resultant string at this location. Supported format items are: • tab—insert <*tab*> character • new_line—new line indicator; for the *lib$format_single_string* routine, this inserts <*carriage_return*><*line_feed*> characters; for the *lib$format_multiple_strings* routine, this advances to the next output string in the resultant string array • form_feed—insert <*form_feed*> character
character(V,c)	Insert the character "c" in the resultant string n times, where n is the value of the specified parameter or the specified constant.

[1]Table 1–3 presents the rules for which parameters and constants are allowed with which directives.

[2]The *lib$format_single_string* and *lib$format_multiple_strings* routines use the ARUS date/time formatting services to format the date and time. These services provide full internationalization capabilities as well as support for multiple date/time formats. See Chapter 58, Application Run-Time Utility Services for more information.

Table 1–2 (Cont.): Formatting Directives

Directive[1]	Description
set(V)	Set the internal parameter value to the value of the specified parameter or constant. This is normally used prior to the repeat directive.
increment([V])	Increment the internal parameter value by the value of the specified parameter or the specified constant. If "V" is not specified, the constant value 1 is assumed.
decrement([V])	Decrement the internal parameter value by the value of the specified parameter or the specified constant. If "V" is not specified, the constant value 1 is assumed.
repeat(V,directive[,directive...])	Repeat the specified list of directives. The number of times to repeat may be specified by the value of a parameter or by constant value. The repeat directive in conjunction with the internal parameter value provides a short way to specify output of a list of parameters.
text(V)	Output the specified text string. This is useful in conjunction with the repeat directive.
if(V{op}V,directive[,directive])	Execute the first directive if the operation specified by {op} is true; otherwise, execute the second directive, if specified. Operations compare the value of the first specified parameter or constant with the second parameter or constant. The following comparison operations are supported:

Operation	Description
=	true if the first parameter or constant is equal to the second parameter or constant
<>	true if the first parameter or constant is not equal to the second parameter or constant
<	true if the first parameter or constant is less than the second parameter or constant
>	true if the first parameter or constant is greater than the second parameter or constant
<=	true if the first parameter or constant is less than or equal to the second parameter or constant
>=	true if the first parameter or constant is greater than or equal to the second parameter or constant

[1]Table 1–3 presents the rules for which parameters and constants are allowed with which directives.

Table 1–2 (Cont.): Formatting Directives

Directive[1]	Description
case(V,{op}V:directive [,{op}V:directive,...])	Case on value. The value of the first parameter or constant is compared sequentially with each other parameter or constant according to the specified operator {op}. If the comparison is true, the directive is executed and the comparisons stop. The table below lists the supported comparison operations:

Operation	Description
=	true if the first parameter or constant is equal to the specified parameter or constant
<>	true if the first parameter or constant is not equal to the specified parameter or constant
<	true if the first parameter or constant is less than the specified parameter or constant
>	true if the first parameter or constant is greater than the specified parameter or constant
<=	true if the first parameter or constant is less than or equal to the specified parameter or constant
>=	true if the first parameter or constant is greater than or equal to the specified parameter or constant

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

[1]Table 1–3 presents the rules for which parameters and constants are allowed with which directives.

\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.\

1.11.2 Formatting Text

Two routines are provided to support the text formatting directives described above. These are described in the following sections.

1.11.2.1 Single String Text Formatting—*lib$format_single_string*

The *lib$format_single_string* routine provides text formatting support producing a single resultant string. The interface to this procedure is:

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

Parameters:

Parameter	Description
source_string	Supplies the source string containing text and formatting directives.
resultant_string	Returns the resultant formatted string.
resultant_length	Returns the number of characters used in the resultant string.
parameters	Optionally 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 the PRISM Calling Standard.
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.

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

1.11.2.2 Multiple String Text Formatting—*lib$format_multiple_strings*

The *lib$format_multiple_strings* routine provides text formatting support producing multiple resultant strings. The interface to this procedure is:

```
PROCEDURE lib$format_multiple_strings (
            IN source_string : string(*);
            IN array_size : integer;
            OUT resultant_string : lib$string_array(array_size);
            OUT string_lengths : ARRAY [1..array_size] of integer;
            OUT strings_used : integer;
            IN parameters : exec$argument_array(*) OPTIONAL;
            IN language : string(*) OPTIONAL;
            ) RETURNS status;
```

Parameters:

Parameter	Description
source_string	Supplies the source string containing text and formatting directives.
array_size	Supplies the number of strings in the *resultant_string* array.
string_size	Supplies the length of each string in the *resultant_string* array.
resultant_string	Returns the resultant formatted strings.
string_lengths	Returns the length of each of the resultant formatted strings.
strings_used	Returns the number of strings in the array that were actually used in the formatting.
parameters	Optionally 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 the PRISM Calling Standard.
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.

The *lib$format_multiple_strings* routine copies text from the source string into the *resultant_string* array, formatting parameters as formatting directives are encountered. Formatting begins by using the first string in the array. When a *new_line* control item is encountered, formatting continues with the next element in the array.

1.11.2.3 Allowable Parameter and Constant Types for Directives

\The current version of the PRISM Calling Standard does not include a list of supported data types for the *exec$argument_descriptor* data type. Following is a list of data types required for the text formatting routines:

```
    lib$data_types : (
        lib$c_integer,          ; Signed integer
        lib$c_large_integer,    ; Signed 64-bit integer
        lib$c_real,             ; 32-bit real
        lib$c_double,           ; 64-bit real
        lib$c_byte_data,        ; Byte array
        lib$c_word_data,        ; Word array
        lib$c_longword_data,    ; Longword array
        lib$c_quadword_data,    ; Quadword array
        lib$c_string,           ; Fixed length string
        lib$c_varying_string,   ; Varying string
        lib$c_asciz_string,     ; ASCIZ string
        lib$c_absolute_time,    ; 128-bit absolute time
        lib$c_relative_time     ; 128-bit relative time
        );
\
```

The following is a list of which parameter and constant data types are allowed for each formatting directive:

Table 1–3: Data Type Rules for Formatting Directives

Directive	Allowable Data Types for Parameters and Constants
decimal	*lib$c_integer, lib$c_large_integer, libc_real, libc_double*
hex	*lib$c_integer, lib$c_large_integer, libc_byte_data, libc_word_data, lib$c_longword_data, lib$c_quadword_data, libc_string, libc_varying_string, lib$c_asciz_string*
octal	*lib$c_integer, lib$c_large_integer, libc_byte_data, libc_word_data, lib$c_longword_data, lib$c_quadword_data*
binary	*lib$c_integer, lib$c_large_integer, libc_byte_data, libc_word_data, lib$c_longword_data, lib$c_quadword_data*
date	*lib$c_absolute_time*
time	*lib$c_absolute_time, lib$c_relative_time*
date_time	*lib$c_absolute_time, lib$c_relative_time*
left, right, center	string constant, *libc_string, libc_varying_string, lib$c_asciz_string*
length	N/A—argument must be decimal, hex, octal, binary, date, time, date_time, left, right, center, plural, system, control, character, text, if, or case formatting directive
plural	*lib$c_integer*
system	System items are of type *libc_byte_data, libc_word_data, lib$c_longword_data, lib$c_quadword_data*, or ??
control	N/A—arguments are keywords
character	positive integer constant, *lib$c_integer*
set, increment, decrement	signed integer constant, *lib$c_integer*
repeat	positive integer constant, *lib$c_integer*
text	string constant, *libc_string, libc_varying_string, lib$c_asciz_string*
if, case	signed integer constant, real constant, string constant, *lib$c_integer, lib$c_large_integer, libc_real, libc_double, libc_string, libc_varying_string, lib$c_asciz_string*

1.11.2.4 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,"s were"," was","s were")% deleted"
Parameter: 10
Formatted string: "10 files were deleted"
Parameter: 1
Formatted string: "1 file was deleted"

Control string: "%set(#2),repeat(1,(left(0)," +
                "if(0<1:text(","),text(" ")),increment)% 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"

Control string: "We shipped %decimal(1)% units in the last %decimal(2)% months." +
                "%control(new_line)%" +
                "           %character(length(decimal(1)),-)%"
Parameters: 41000, 6
Formatted strings: "We shipped 41,000 units in the last 6 months."
                   "           ------"
```

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
- Linker
- Image Activation Mechanism
- Condition Handling Data Structures
- Client Context Server/Mica Job Controller Logical Name Transfer

Digital Equipment Corporation - Confidential and Proprietary
For Internal Use Only

Mica Working Design Document
Object Module And Image File Format

Revision 1.4

26–February–1988

Issued by:
Kim Peterson

digital™

TABLE OF CONTENTS

FIGURES

Revision History

Date	Revision Number	Author/Summary of Changes
26-Mar-1986	0.1	B. Schreiber / Original
6-May-1986	0.2	B. Schreiber / Name table, etc.
13-May-1986	0.3	B. Schreiber / Changes from review
	0.4	B. Schreiber / Compiler status
13-Jun-1986	0.5	B. Schreiber / Changes from review
1-Aug-1986	0.6	B. Schreiber / Remove SYSTEM_SERVICE, add lengths to all structures
27-Jan-1987	0.7	B. Schreiber / ISD message section, remove global ISDs, fixup table descriptions
8-Mar-1987	0.8	B. Schreiber / Support for thread local storage
8-Sep-1987	0.9	K. Peterson / 64-bit revisions, breakout generic module format from object module/image file specific format, make goals explicit, abbreviate module-local and internal module-local symbols
2-Oct-1987	1.0	K. Peterson / Editorial changes, minor data structure changes, explicit rules for module name table, universal symbol support added to GST, added linker checking of entity consistency check table, optimized image file data structures for the target instruction size, and positive integers and large integers changed to longword and quadword
20-Nov-1987	1.1	K. Peterson / Final review draft; editorial corrections; addition of virtual address size and count of demand zero sections in image header; incorporate semantics agreed to at Nov 1987 calling standard committee for initialization routines; fuller specification of linker directive table; elimination of object module's initialization routine table; addition of global/local PSECT attribute in GST PSECT definition; addition of PILLAR declarations for autoload data structures
26-Jan-1988	1.2	K. Peterson / ECO RKP008 / Incorporated support for qualified names in the module name table
28-Jan-1988	1.3	K. Peterson / ECO RKP009 / Removal of references to environments; null entry in global symbol table; support for new calling sequence in global symbol table, data relocation table, image activation table, and autoload vector; expand size of module header system id and module type; modify field names for consistency and translation to BLISS; addition of facility code names; modification of data relocation TLS entries

Date	Revision Number	Author/Summary of Changes
25-Feb-1988	1.4	K. Peterson / ECO RKP022 / Restriction on compound name ordering; prohibit duplicate names; word align name table; change name_size to entry_size in name table entry; add name_index field to all entries in GST; remove null entry from GST; add GST entry for referencing symbols and psects; change common symbol def to own entry in GST and allow it to reference symbols; explicitly define psect attributes for FORTRAN common; change semantics of SHARE psect attribute; add general psect initialization mechanism; redo TLS support; add field prefixing for BLISS; modify autoloading and ECC; specify rules for object module layout

CHAPTER 1

OBJECT MODULE AND IMAGE FILE FORMAT

1.1 Overview

1.1.1 Requirements

There are three requirements for this chapter:

- The chapter specifies the image file format required by the image activator.
- The chapter specifies the object module format required by the linker and the object module loader.
- The chapter specifies the generic module format required by the librarian.

The first two requirements are related to each other because object modules share the same format as image files. The first two requirements are related to the third requirement because object modules and image files are examples of the types of files maintained in libraries.

These formats were designed for the following goals:

- The format allows clean extensions in future releases to add functions.
- The format allows for files greater than four gigabytes.
- The module format allows different types of modules to be mixed in one library.
- The object module format allows object modules to be run without linking.
- The object module format allows separate object modules to be combined into one object module.
- The image file format allows fast image activation.

The increasing division of programs into separate object modules leads to an increasing number of object modules, with a concomitant increase in overhead during linking. Combining separate modules into a single module is a means of controlling this overhead.

The tendency in VAX/VMS has been to add functionality to object modules and image files, resulting in larger files. If this trend towards larger files continues, it will become desirable to separate the different parts of an object module or image file into separate files, and yet maintain them in a common library. A common module format allows for this separation. A common module format also allows for a single implementation of the librarian.

The data structures in this chapter allow for both 64-bit addresses, and file sizes greater than four gigabytes. Systems that do not support 64-bit addresses, or file sizes greater than four gigabytes, must have the upper longword of these fields zero to ensure that the values stored in the fields are valid 64-bit values.

1.1.2 Description

An object module and an image file are both examples of a module. A module can define names, refer to names, and be in a library. An object module and an image file share additional similarities because both can be activated, and because image files are created from object modules.

The primary users of object modules are compilers, the linker, and the loader. The primary users of image files are the linker, the debugger, and the image activator. Generally, references to the linker refer to both the linker and the loader. The loader is activated when an object module is run without first linking it.

All modules have a common header format and a common name table format so that a common librarian utility can be used for different types of libraries. All modules contain within their header an index to their different sections. Different types of modules may contain different types of sections, but the module header contains an index that provides a means of accessing sections specific to a module type.

Module specific sections that are common to both object modules and image files are:

- Global symbol table
- Debug symbol table
- Entity consistency check table
- Data sections
- Code sections

The object module specific sections are:

- Linker directive table
- Data relocation table

Image file specific sections are:

- Image section descriptor table
- Image relocation tables
- Activation tables
- Transfer vector table
- Debug module table

Figure 1–1: Object Module and Image File Format

1.1.3 Dependencies

- Object module format affects compiler development.
- Image file format affects image activation.
- Image file format affects debugger development.

The requirements of compilers, image activation, and debuggers are reflected in this chapter.

1.2 Module Format

All PRISM modules are constructed as block-oriented files. The file attributes are fixed-length 512-byte records (similar to library files and image files on VMS). All PRISM modules have a common module header, which is at the beginning of the first block of the module (zero byte offset from the starting virtual block number). The module header contains sufficient information to permit all consumers of the module to locate the various other sections and tables within it. The module header consists of the following fields:

- Type
- System identification
- Module size
- Module name table offset
- Module name table size
- Module specific item list offset
- Module specific item list size

The module header has the following declaration and layout:

```
! \BLISS uses prefix MODULE$HDR_\
module$header : RECORD
    system_identifier : longword;
    module_type : word;
    module_block_size : quadword;
    name_table_offset : quadword;
    name_table_size : quadword;
    item_list_offset : quadword;
    item_list_size : quadword;
    END RECORD;
```

\To fully specify record field names, a prefix is specified for each record for use with languages like BLISS. The prefix is added to the record's field name as part of the automatic translation of PILLAR declarations to BLISS.\

Figure 1–2: Module Header Layout

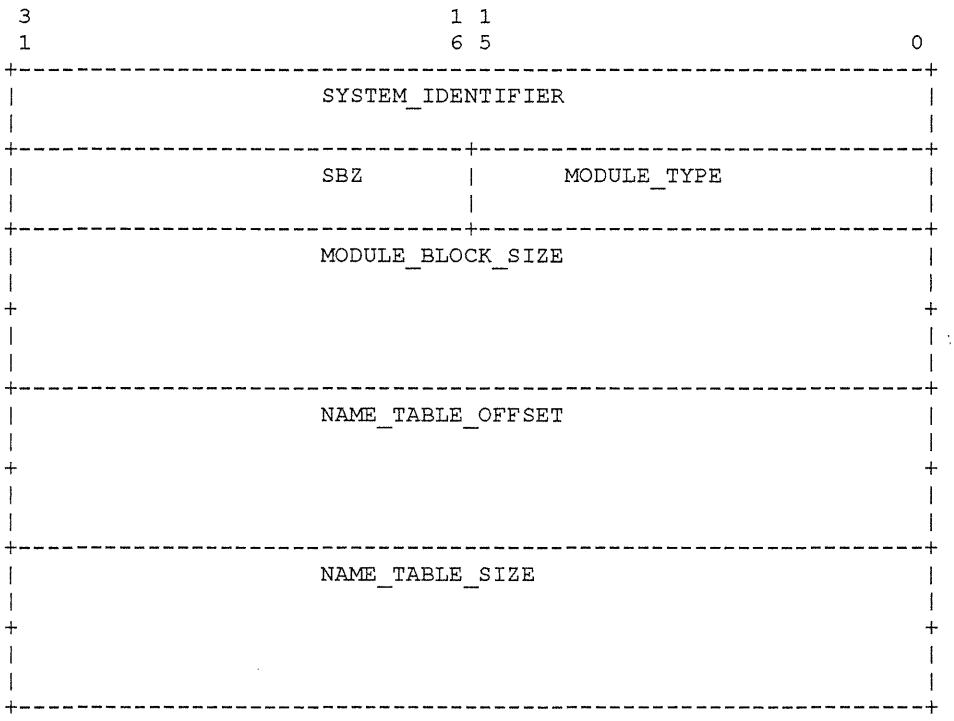
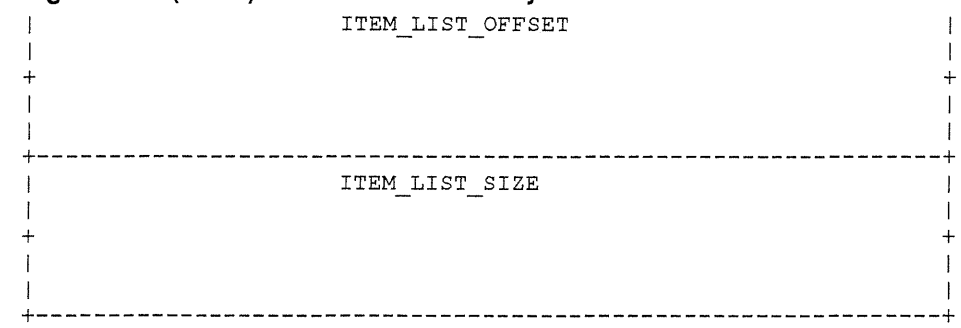


Figure 1–2 Cont'd. on next page

Figure 1–2 (Cont.): Module Header Layout



The following types are used throughout this document:

```
module$entry_type : byte;
module$entry_size : integer[0..65535];
module$entry_size_word : module$entry_size[..] SIZE(word);
module$string_size : integer[0..65535];
module$string_size_word : module$string_size[..] SIZE(word);
module$name_size : integer[0..255];
module$name_size_byte : module$name_size[..] SIZE(byte);
```

\The following types are used only as a convenience in this document:

```
unsigned_byte : integer[0..255] SIZE(byte);
unsigned_word : integer[0..65535] SIZE(word);
```

Also as a convenience in this document, Pillar layouts are not specified for data structures laid out in figures. All such data structures will be declared with Pillar layouts in their definition modules to correspond with their figures.\

1.2.1 System Identification

The SYSTEM_IDENTIFIER field contains the identification MODULE$C_PRISM_IDENTIFIER. This identification is used to distinguish PRISM objects and images from VMS and ULTRIX objects and images.

In VMS, the first word of an object module is the RMS byte count of the first record in the object module. The third byte is the record type (symbolically, OBJ$B_RECTYP). In order to give PRISM the ability to differentiate VMS object modules from PRISM object modules and images, the third and fourth bytes of MODULE$C_PRISM_IDENTIFIER are distinct from the values used in VMS object modules.

Because VMS object modules are variable-length record-oriented files, while PRISM object modules and images are fixed-length 512-byte records, object modules could be differentiated by their record attributes. However, since this is not also true for images, the SYSTEM_IDENTIFIER field is used.

In ULTRIX, the first longword of an image is used to identify image types. The value MODULE$C_PRISM_IDENTIFIER is distinct from the values used in ULTRIX.

\The values for SYSTEM_IDENTIFIER can be chosen so that they cannot be mistaken for part of ULTRIX magic information. The values that may be mistaken for ULTRIX magic information are those that can be interpreted as printable ASCII characters (which are used by ULTRIX to indicate shell scripts).\

\Note that SYSTEM_IDENTIFIER identifies only the target system; it does not identify the system that created the module.\

1.2.2 Module Type

The MODULE_TYPE field specifies the type of module described by the header. This chapter describes the following types of modules in detail:

- MODULE$C_OBJECT_MODULE, which specifies an object module
- MODULE$C_EXECUTABLE_IMAGE, which specifies an executable image
- MODULE$C_SHAREABLE_IMAGE, which specifies a shareable (linkable) image

1.2.3 Module Block Size

The MODULE_BLOCK_SIZE field specifies the total number of virtual blocks in the module. This allows modules to be concatenated without combining them in a single module.

1.2.4 Module Name Table

The module name table contains all of the module's names that can be accessed by a consumer of the module. In the case of an object module, this includes all global symbols and PSECTs (program sections) defined or referenced in the module.

The NAME_TABLE_OFFSET field specifies the offset of the module's name table in bytes from the beginning of the module. The NAME_TABLE_SIZE field specifies the size of the module's name table in bytes. The module name table is not at a fixed location within the module in order to optimize the location of the module name table for the specific module.

Names are gathered in one general table to represent them independently of any module-type-specific data structure. This general table provides a clean interface (which VMS does not have) between librarian functions and module specific functions; for instance, in VMS, object module names are contained within the global symbol table, which is specific to object modules. Inserting an object module into a library requires an understanding of object module data structures.

The module name table supports names that are qualified by other names in the module name table through the use of compound names. The module name table defines two types of names: atomic and compound. Atomic names define the string of characters that make up a name. Compound names are an ordered tuple of atomic names. The linker and librarian handle compound names as they handle atomic names; for example, the librarian would use a compound name as a key to a module just as it would use an atomic name as a key.

The module name table is a byte-stream-oriented table with word-aligned entries, each entry representing a name. Each entry contains a byte that specifies the format of the entry, the size of the entry in bytes, and the library index(es) of the name, if the name is a key in a library. The name can be either a library key, or invisible in a library. If the name is a library key, it can be in any of the library's indexes. Each entry for an atomic name also contains the name string itself. Each entry for a compound name contains instead the indexes to two other entries in the module name table. The first entry indexed can be either atomic or compound, but the second entry indexed must be atomic.

\The type field would be used for future extensions to the name table for such things as multiple byte character sets. To make such extensions as easy as possible, the meaning of the value in the length field is based upon its entry's type.\

The following rules apply to the module name table:

- Atomic names cannot contain the NUL character (numeric value zero).
- The maximum length of atomic names is 255.
- The first name in the module name table has an index of one. The index is incremented by one for each subsequent name.
- The indexes used in a compound name entry must be less than the index of the compound name's entry. In other words, a compound name entry cannot make a forward reference.

- The maximum number of atomic names in a compound name is 255.
- The module name table can be sorted or unsorted.
- The module name table can not contain duplicate names.
- Case-sensitive names retain their case in the module name table, but case-insensitive names must be lowercase.
- A name's index(es) is a matter of convention among the users of the module. The convention for object modules and image files is that the first index keys are module names, and the second index keys are global and universal symbol definitions.

\Case-insensitive names are lowercased to allow case-insensitive languages to match the ULTRIX case conventions naturally.\

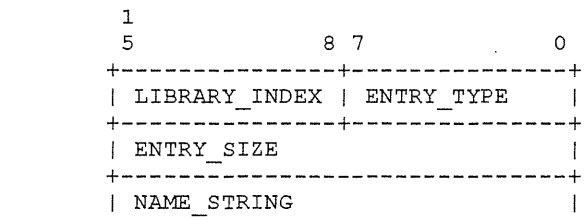
The name table is word aligned, and each entry in it has the following declaration:

```
module$name_table_type : (
    module$c_name_atomic,
    module$c_name_compound_byte,
    module$c_name_compound_word,
    module$c_name_compound_long
    );

module$library_index : (
    module$c_library_index1,
    module$c_library_index2,
    module$c_library_index3,
    module$c_library_index4,
    module$c_library_index5,
    module$c_library_index6,
    module$c_library_index7,
    module$c_library_index8
    );

! \BLISS uses prefix MODULE$NTE_\
module$name_table_entry(
    entry_type : module$name_table_type[..] SIZE(byte);
    entry_size : unsigned_byte ): RECORD
    CAPTURE entry_type, entry_size;
    library_index : SET module$library_index[..] SIZE(byte);
    VARIANTS
        CASE entry_type
            WHEN module$c_name_atomic THEN
                name_string : string(entry_size-4);
            WHEN module$c_name_compound_byte THEN
                byte_index1 : unsigned_byte;
                byte_index2 : unsigned_byte;
            WHEN module$c_name_compound_word THEN
                word_index1 : unsigned_word;
                word_index2 : unsigned_word;
            WHEN module$c_name_compound_long THEN
                long_index1 : longword;
                long_index2 : longword;
    END VARIANTS;
END RECORD;
```

1.2.4.1 Atomic Name Entry

Atomic name entries contain the string that defines the atomic name. The layout of atomic names is shown in Figure 1–3.

Figure 1–3: Module Name Table Entry Layout/Atomic Names



- ENTRY_TYPE is the value MODULE$C_NAME_ATOMIC.
- LIBRARY_INDEX is the set of all key indexes in the library in which that name occurs. MODULE$C_LIBRARY_INDEX1 is the index where the module names reside in an object module library. MODULE$C_LIBRARY_INDEX2 is the index where the global names reside in an object module library. The other indexes are not used in an object module library. If no bits are set, the name is not a library key and is not visible in a library.
- ENTRY_SIZE is the size of the entry itself. The size of the name string is ENTRY_SIZE-4.
- NAME_STRING is the characters that make up the name. Name look-up for object modules and image files is case-sensitive.

1.2.4.2 Compound Name Entries

Compound name entries contain the indexes to two other name entries that define the compound name. Compound names can be nested, so that either or both of the indexes can refer to other compound names. Compound name entries have the layouts shown in Figure 1–4.

Figure 1–4: Module Name Table Entry Layout For Compound Names

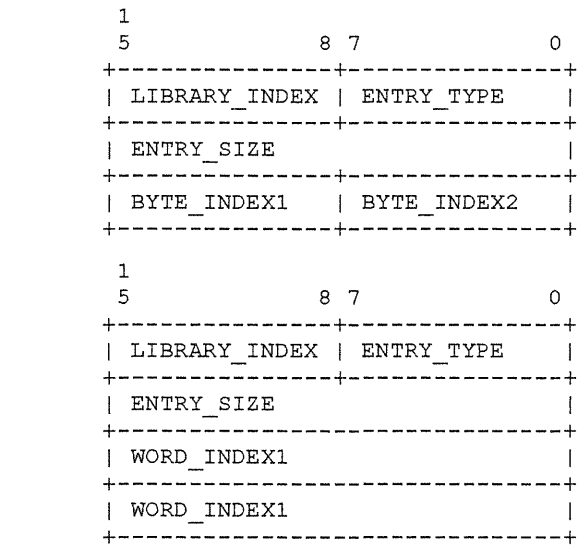
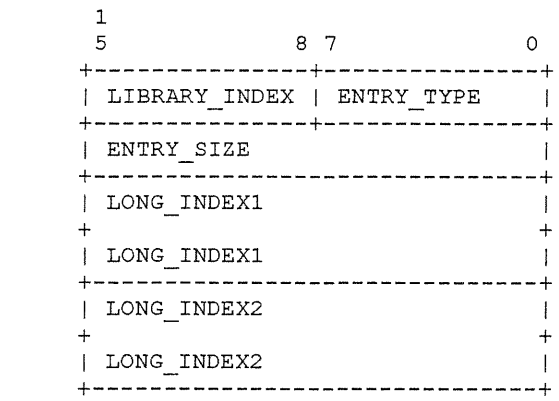


Figure 1–4 Cont'd. on next page

Figure 1–4 (Cont.): Module Name Table Entry Layout For Compound Names



- ENTRY_TYPE is the value MODULE$C_NAME_COMPOUND_BYTE, MODULE$C_NAME_COMPOUND_WORD, or MODULE$C_NAME_COMPOUND_LONG.
- LIBRARY_INDEX is the set of all key indexes in the library in which that name occurs. MODULE$C_LIBRARY_INDEX1 is the index where the module names reside in an object module library. MODULE$C_LIBRARY_INDEX2 is the index where the global names reside in an object module library. The other indexes are not used in an object module library. If no bits are set, the name is not a library key and is not visible in a library.
- ENTRY_SIZE is six for MODULE$C_NAME_COMPOUND_BYTE, eight for MODULE$C_NAME_COMPOUND_WORD, and twelve for MODULE$C_NAME_COMPOUND_LONG.
- BYTE_INDEX1, WORD_INDEX1, and LONGWORD_INDEX1 are each an index to a name that makes up the first part of the current name. The indexed name can be an atomic name or a compound name.
- BYTE_INDEX2, WORD_INDEX2, and LONGWORD_INDEX2 are each an index to a name that makes up the last part of the current name. The indexed name can only be an atomic name.

1.2.5 Module Item List

Modules can contain any number of data structures that describe the specific module layout for users of that module. These data structures are themselves described within the module item list. Each item list entry contains the length of the item entry, the facility that defines the item, the type of item within the facility, flags that define the format of the entry, and optionally the offset and length of the data structure, if it is not contained within the item list entry.

The ITEM_LIST_OFFSET field specifies the offset of the module's item list in bytes from the beginning of the module. The ITEM_LIST_SIZE field specifies the size of the module's item list in bytes. The module's item list is not at a fixed location within the module in order to optimize the location of the item list for the specific module.

1.2.5.1 Item List Entry

Each entry in the item list is quadword aligned and has the following declaration and layouts:

```
! \BLISS uses prefix MODULE$ILE_\
module$item_list_entry(
        item_entry_size : unsigned_word;
        offset_format : bit ) : RECORD
    CAPTURE item_entry_size,offset_format;
    item_facility : word;
    item_code : word;
    item_flags : SET integer[0..7] SIZE(byte);
    VARIANTS CASE offset_format
        WHEN true THEN
            item_offset : quadword;
            item_size : quadword;
        WHEN false THEN
            item_specific:
                quadword_data((item_entry_size-1)/8);
    END VARIANTS;
END RECORD;
```

Figure 1–5: Module Item List Entry Layout with Offset Format

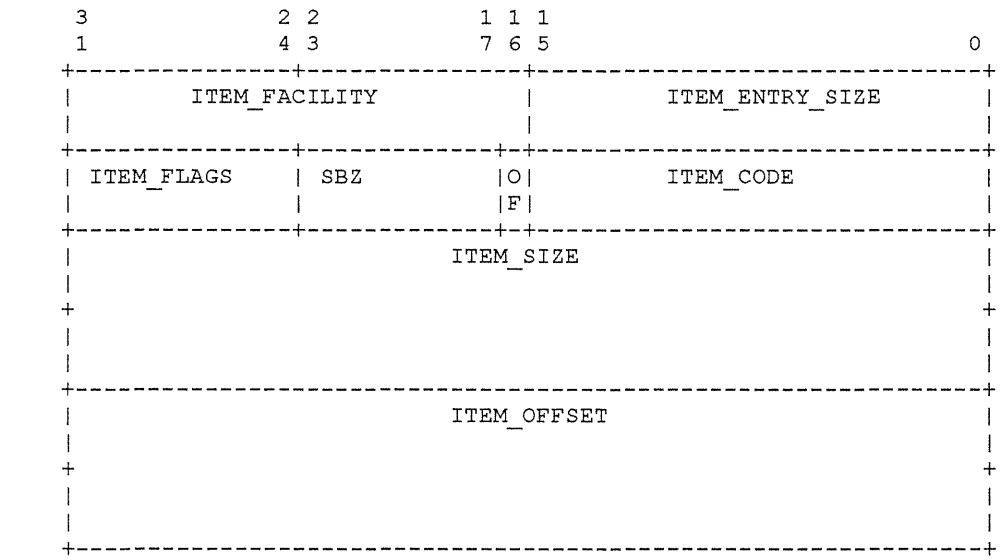
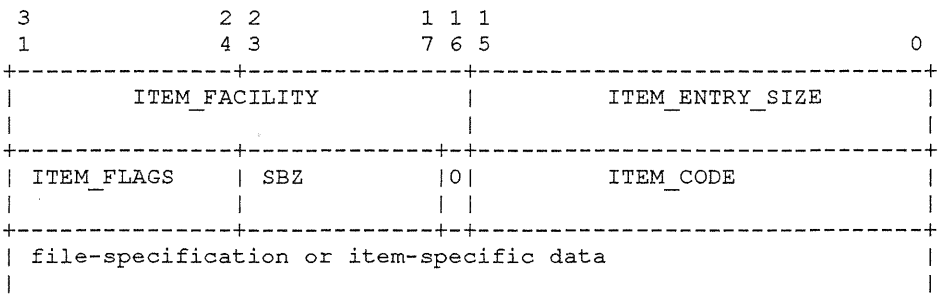


Figure 1–5 Cont'd. on next page

Figure 1–5 (Cont.): Module Item List Entry Layout with Offset Format

Figure 1–6: Module Item List Entry Layout with Nonoffset Format



- ITEM_ENTRY_SIZE contains the size (in bytes) of the entire entry, including the words ITEM_ENTRY_SIZE, ITEM_FACILITY, ITEM_CODE, and ITEM_FLAGS. This field has the value of 24 when OFFSET_FORMAT is set. Note that the size of the item list entry does not include any padding used to maintain the quadword alignment of the entries.
- ITEM_FACILITY contains the facility code. Facility codes are described in Chapter 3, Status Values, Messages, and Text Formatting. The following facilities each have their own unique facility code and are defined in this chapter:
  - — Generic module facility, which has names prefixed by MODULE$ and has the facility code MODULE$C_FACILITY.
  - — Object module facility, which has names prefixed by MODOBJ$, and also includes items common with image files and has the facility code MODOBJ$C_FACILITY.
  - — Image file facility, which has names prefixed by MODIMG$ and has the facility code MODIMG$C_FACILITY.
- ITEM_CODE contains the item code. The item code assignments are facility specific. Item codes specific to the module, object module, and image file facilities are listed in Section 1.2.5.2, Section 1.2.5.3, and Section 1.2.5.4. The following sections describe the data structures associated with many of the listed items.
- OFFSET_FORMAT (OF) when set specifies that the item-specific data is in offset format. Offset format specifies the size (in bytes) of the item-specific information within the current module, and the offset (in bytes) from the beginning of the module to the data. When this flag is set, the value of the ITEM_ENTRY_SIZE field must be 24. The layout of the item entry is shown in Figure 1–5.
- \File format is a possible extension to the module format after FRS.

  FILE_FORMAT (bit 17 in the second longword) when set specifies that the item-specific data is contained within the specified file. If the OFFSET_FORMAT flag is clear, the file specification is within the item list entry. If the OFFSET_FORMAT flag is set, the file specification for the

item-specific data is of the size and at the offset specified by the ITEM_SIZE and ITEM_OFFSET fields. Interpretation of information within the file is facility dependent.\

- ITEM_FLAGS contains item-specific flags. The following item-specific flags are specified in this chapter:
  - — MODOBJ$C_WEAK_TRANSFER_ADDRESS is specific to object modules. If it is set, the transfer address specified in the item list entry is weak. Only valid on MODOBJ$C_ACTIVATE_INFORMATION entry.
  - — MODIMG$C_DEBUG_FLAG. The image was linked /DEBUG. When this image is run, the debugger is started unless RUN /NODEBUG was specified. Only valid on MODIMG$C_IMAGE_SECTION entry.
  - — MODIMG$C_ALIGN_FLAG. The image was linked with page alignment less than 64K. When this image is run, the image activator must check that the page size of the system is compatible with the page size that the image was linked with. Only valid on MODIMG$C_IMAGE_SECTION entry.

1.2.5.2 Defined Module Item Codes

The following item codes provide general information and are useful to different types of modules:

- MODULE$C_CREATE_STATUS contains the severity of the creation of the module. This item is a longword in nonoffset format.
- MODULE$C_CREATOR_NAME contains a string for identifying the creator of the module. This would be the compiler's name for object modules, and the linker's name for images. This item is a string in nonoffset format.
- MODULE$C_ENTITY_CONSISTENCY contains the entity consistency check table, which provides information about the dependencies of this module on other modules. This table is described in Section 1.3.6. It is in offset format.
- MODULE$C_CPU_TARGET contains the target CPU type, the target's subset of the PRISM architecture, and the register size. This record is described in Section 1.3.4.
- MODULE$C_CREATION_TIME contains the date and time at which the module was created. This item is a quadword in nonoffset format.
- MODULE$C_MATCH contains the segment match control along with the major and minor identifier specified by the user when the module was created. This record is described in Section 1.3.5.
- MODULE$C_NAME contains a string that can be supplied to identify the module. This item is in nonoffset format.
- MODULE$C_VERSION contains a string that can be supplied to identify the module version. This item is in nonoffset format.

1.2.5.3 Defined Object Module Item Codes

The following item codes provide information that may be specific to object modules or specific to both object modules and image files.

- MODOBJ$C_ACTIVATE_INFORMATION contains the transfer address for the module. This item is a longword in nonoffset format. It contains the index for the global symbol that defines the invocation descriptor of the routine. This item is specific to object modules. The MODOBJ$C_WEAK_TRANSFER_ADDRESS flag can be set for this item code if the transfer address is weak. A weak transfer address can be overridden by a nonweak transfer address. If the transfer addresses of all modules included in the link are weak, the first transfer address becomes the image transfer address.

- MODOBJ$C_RELOCATE contains the data relocation table, which allows the linker to do link-time fixups. This table is described in Section 1.4.2. It is in offset format and is specific to object modules.
- MODOBJ$C_DEBUG_SYMBOL contains the information needed by the debugger to understand symbols used in the image. This table is described in Section 1.3.7. It is in offset format and is common to both object modules and image files.
- MODOBJ$C_GLOBAL_SYMBOL contains the information needed by the linker and debugger to understand symbolic references. This table is described in Section 1.3.3. It is in offset format and is common to both object modules and image files.
- MODOBJ$C_LINKER_DIRECTIVE contains the linker directives provided by the compiler to the linker, which allow compilers to specify initialization routines and additional files for the link. This table is described in Section 1.4.1. It is in offset format and is specific to object modules.

1.2.5.4 Defined Image File Item Codes

The following item codes provide information that is specific to image files:

- MODIMG$C_ACTIVATE_DEFERRED contains information about the images that can be referenced by the current image. This table is described in Section 1.5.5. It is in offset format.
- MODIMG$C_ACTIVATE_IMMEDIATE contains information about the images that must be activated concurrently with the current image. This table is described in Section 1.5.6. It is in offset format.
- MODIMG$C_AUTOLOAD contains all of the image's autoload vectors. This item supports the image analyzer, but is not needed for autoloading since the autoloader accesses the autoload vectors through the linkage pairs that refer to it.
- MODIMG$C_DEBUG_MODULE contains the information needed by the debugger to understand each module's contribution to the image and the virtual address of each PSECT within a module. This table is described in Section 1.5.9. It is in offset format.
- MODIMG$C_DEMAND_ZERO_COUNT contains the number of demand zero sections in the image. This item is a longword in nonoffset format.
- MODIMG$C_INITIAL_ROUTINE contains the information needed by the image activator to identify the initialization routines. This table is described in Section 1.5.8. It is in offset format.
- MODIMG$C_IMAGE_SECTION contains the information needed by the image activator to set up users' virtual address space correctly. This table is described in Section 1.5.1. It is in offset format.
- MODIMG$C_LINK_VPN specifies the virtual address at which the image has been based. If the image activator finds itself loading this image at a different address, all fixups described in the local relocation table must be performed. This item is a quadword in nonoffset format.
- MODIMG$C_RELOCATE_EXTERNAL contains the external relocation table, which allows the image activator to perform fixups at image activation that depend upon external images. This table is described in Section 1.5.4. It is in offset format.
- MODIMG$C_RELOCATE_LOCAL contains the local relocation table, which allows the image activator to perform fixups at image activation within the current image. This table is described in Section 1.5.3. It is in offset format.
- MODIMG$C_RELOCATE_TLS contains the thread local storage count relocation table, which allows the image activator to perform fixups dependent upon the number of thread local storage regions already created at image activation. This table is described in Section 1.5.2. It is in offset format.

- MODIMG$C_TLS_INDEX_COUNT specifies the number of TLS (thread local storage) regions defined in the image. This item is a longword in nonoffset format and is specific to object modules.

- MODIMG$C_TRANSFER_VECTOR contains the offsets to the routines that can be called from outside the shareable image (see Section 1.5.7). This item is in offset format.
- MODIMG$C_VIRTUAL_ADDRESS_SIZE contains the total size of the virtual address space mapped by the image. This item is a longword in nonoffset format.

1.3 Object Module and Image File Data Structures

Object modules and image files are modules, and both follow the module format specified above. In addition, object modules and image files share many common data structures. These common data structures allow object modules to be executed without being linked, and allow images to be linked with other images.

\The object module should be laid out according to the following rules. These rules result in an object module that is optimized for the linker to access.

- The module's item list should immediately follow the module's header
- The module's name table and the global symbol table should be grouped together in that order
- The module should not duplicate item list entries

1.3.1 Code Section

On PRISM, compilers can generate code sections directly; there is no need for any fixups by the linker. Therefore, a code section for each PSECT appears in the object file exactly as it will be laid out in memory. The linker gathers up all modules' contributions for each PSECT and concatenates them at link time. In other words, the STACK/ADD/STORE commands used in object files on VMS are not found in PRISM object files. The compiler is responsible for generating code directly.

\The lack of the STACK/ADD/STORE commands makes linker fixups of instruction literals, branch displacements, and load/store displacements impossible.\

Code sections are not listed in the module item list; in image files, the first code section is found through the image section descriptors, and a code section is aligned on a virtual block number (512-byte unit) boundary.

In object files, a code section is found through the PSECT definitions in the global symbol table. A code section typically starts at a block boundary, but a code section can be located anywhere in the object module.

The compiler cannot make assumptions about the relative virtual address assignments of different PSECTs. References to addresses in other PSECTs must be made using the data relocation table. PSECTs are different if the name is different. It is an error if the same PSECT has different attributes. Furthermore, it is an error if the same PSECT has different alignments within a module.

1.3.2 Data Section

Like code sections, data sections can appear in the object file as *memory-ready*, except that certain fixups may need to be performed before it can be accessed at run time. Data sections may also appear in the object file in compressed format, which requires expansion before it can be accessed at run time. The compilers cannot make assumptions about the relative virtual address assignments of other PSECTs. PSECTs are different if the name is different. It is an error if the same PSECT has different attributes. Furthermore, it is an error if the same PSECT has different alignments within a module.

Data sections are not listed in the module item list. In image files, data sections are found through the image section descriptors, and data sections are aligned on a virtual block number (512-byte unit) boundary.

In object files, data sections are found through the PSECT definitions in the global symbol table and a data section can be located anywhere in the object module.

1.3.3 Global Symbol Table

The global symbol table is built by the compiler (in object modules) and the linker (in images) and identifies all symbols defined or referenced in the module. The items in the global symbol table are associated with the names in the module name table. Entries in the global symbol table contain an index to the symbol's name.

In addition, each symbol and PSECT is assigned an index by its position in the global symbol table. The global symbol table can be viewed as having multiple index counters, one for global symbols and one for PSECTs, which allows named entities to be identified by their index number.

Global symbols are assigned their indexes by assigning one to the first global symbol reference or definition. The index is increased by one for each subsequent global symbol reference or definition. Duplicate indexes for the same global symbol should not occur, but if they do (for example a reference and a definition within the same module) they have distinct indexes.

PSECTs are assigned their indexes by assigning "1" to the first uninitialized, standard, or compressed PSECT entry. The index is increased by one for each subsequent uninitialized, standard, or compressed entry. Duplicate indexes for the same PSECT are not an error, but all entries for the same PSECT must have the same attributes and alignment. \This restriction saves a lot of rules about figuring offsets in PSECTs that have multiple allocations with separate alignments.\

A PSECT definition can be repeated within the module to permit multiple allocations to the PSECT to be built in a straightforward way. Both concatenated and overlaid PSECTs are built in this way, because PSECTs are only overlaid with PSECTs from other modules. However, all references to a PSECT should index the first entry of the PSECT. Multiple PSECT entries build a PSECT, but only the first PSECT entry references it.

The first field of each entry in the global symbol table defines the entry's type. The second field of each entry in the global symbol table defines the entry's sub-type, and is used in an entry specific way. The third field of each entry in the global symbol table contains the size in bytes of the entry, including the entry's type, sub-type, and size, but not including any padding at the end of the entry. The entries within the global symbol table are longword aligned. Global symbol table entry types are shown in the next table, and their declarations follow. The following sections discuss each type of entry.

Entry Type Name (prefixed MODOBJ$C_)	Value	Interpretation
Not named	0	Reserved to DIGITAL
Symbol_module	1	Module definition
Symbol_definition	2	Global symbol definition
Symbol_reference	3	Intolerant global symbol reference
Symbol_PSECT_reference	4	Tolerant global symbol or PSECT reference
PSECT_definition	5	Strong PSECT definition
PSECT_symbol_definition	6	Weak PSECT definition
Not named	7-127	Reserved to DIGITAL
Not named	128-255	Reserved to Customers/CSS

```
! \BLISS uses prefix MODOBJ$GSE_\
modobj$global_symbol_entry(
    entry_type : module$entry_type;
    entry_subtype : module$entry_type;
    entry_size : module$entry_size[..] SIZE(word)
    ) : RECORD
    CAPTURE entry_type, entry_subtype, entry_size;
    name_index : longword;
    VARIANTS CASE entry_type
        WHEN modobj$c_symbol_module,
             modobj$c_symbol_psect_reference,
             modobj$c_symbol_reference THEN
            NOTHING;
        WHEN modobj$c_symbol_definition THEN
            symbol_definition :
                modobj$symbol_definition(
                    entry_subtype,
                    entry_size-8 );
        WHEN modobj$c_psect_symbol_definition,
             modobj$c_psect_definition THEN
            psect_definition : modobj$psect_definition(
                entry_subtype,
                entry_size-8 );
        WHEN OTHERS THEN
            symbol_unknown : longword_data((entry_size-1)/4);
    END VARIANTS;
END RECORD;
```

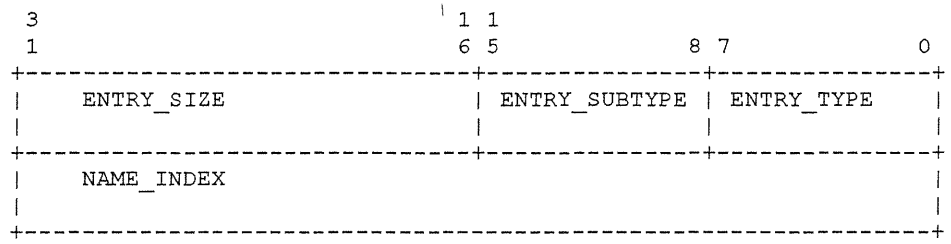
1.3.3.1 Module

This entry represents the module name in the module's name table. Module entries have the following format:

Figure 1–7: GST Module Entry

Figure 1–7 Cont'd. on next page

Figure 1–7 (Cont.): GST Module Entry



These fields are defined as follows:

- ENTRY_TYPE is the value MODOBJ$C_SYMBOL_MODULE.
- ENTRY_SUBTYPE should be zero.
- ENTRY_SIZE is eight.
- NAME_INDEX is the index of the symbol's name in the module name table. The index can not be zero.

1.3.3.2 PSECT Definition Entry

A PSECT definition is also known as a strong PSECT definition, because it always defines a program section. PSECT (program section) definitions are used by the compilers to describe the contents of the various parts of memory to the linker. The linker gathers PSECTs into image sections based on PSECT attributes. There are three kinds of PSECT entries in the global symbol table : uninitialized, standard, and compressed; their descriptions follow:

- Uninitialized PSECTs have the value MODOBJ$C_PSECT_UNINITIALIZED. These definitions merely describe an allocation of memory. If the psect is not for thread local storage, the memory is initialized to zero by the linker if no other initialization is specified. If the psect is for thread local storage, the memory is initialized according to the thread local storage design.
- Standard PSECTs have the value MODOBJ$C_PSECT_STANDARD. These definitions describe an allocation of memory and its initial value. The entry contains the offset to where the initial value of the PSECT is stored.
- Compressed PSECTs have the value MODOBJ$C_PSECT_COMPRESSED. These definitions describe data that exists in the module in a compressed format. The entry contains the offset and size to the compressed data in the module. If the psect is not for thread local storage, the data is restored to the image by the linker. If the psect is for thread local storage, the data is restored according to the thread local storage design. (See the PRISM calling standard for a complete description of TLS regions.)

  \The representation of the compressed data in the object module may be included when the TLS design is finalized.\

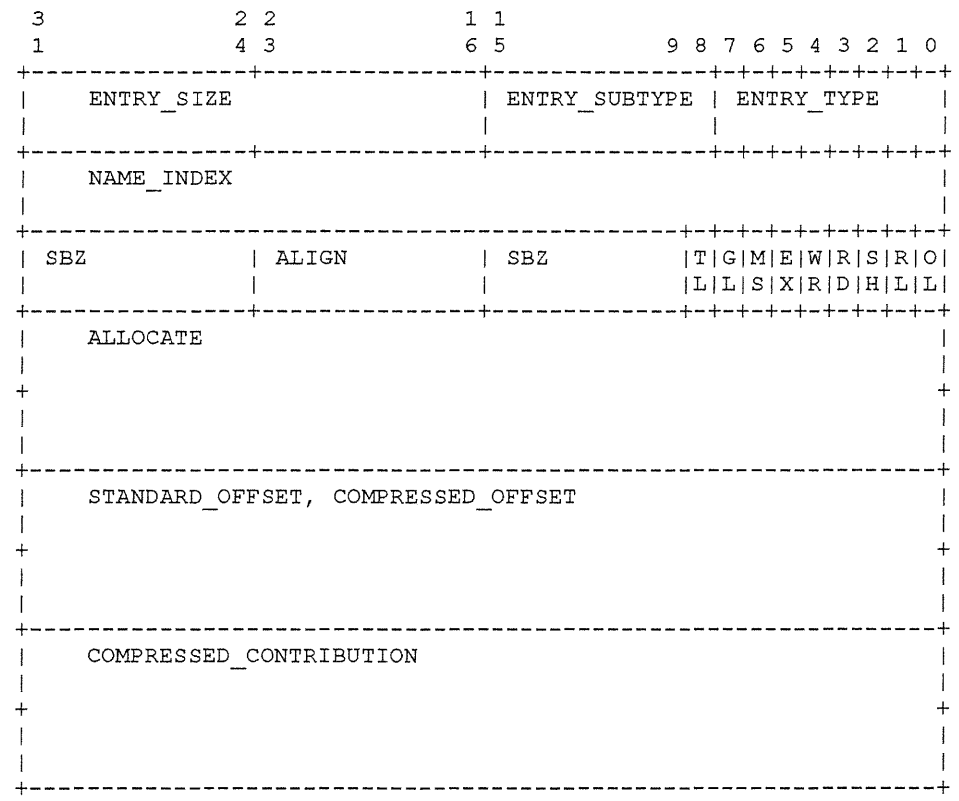
Note that PSECT definitions are referenced by symbol definitions through the PSECT's ID, which is the relative position of the PSECT definition in the module's name table and global symbol table. The declaration, values, and layouts of the PSECT part of a PSECT definition entry follows, with the values of *module$data_size* defined in Section 1.3.4.

```
    modobj$psect_attribute : (
        modobj$c_psect_overlaid,
        modobj$c_psect_relocatable,
        modobj$c_psect_shareable,
        modobj$c_psect_readable,
        modobj$c_psect_writable,
        modobj$c_psect_executable,
        modobj$c_psect_message,
        modobj$c_psect_global,
        modobj$c_psect_tls
        );

    ! \BLISS uses prefix MODOBJ$PD_\
    modobj$psect_definition(
        subtype : module$entry_type;
        subsize : module$entry_size[..] SIZE(byte)
        ) : RECORD
        attributes : SET modobj$psect_attribute[..] SIZE(WORD);
        align : module$data_size[..] SIZE(byte);
        allocate : quadword;
        VARIANTS CASE subtype
            WHEN modobj$c_psect_uninitialized THEN
                NOTHING;
            WHEN modobj$c_psect_standard THEN
                standard_offset : quadword;
            WHEN modobj$c_psect_compressed THEN
                compressed_offset : quadword;
                compressed_size : quadword;
        END VARIANTS;
    END RECORD;
```

Figure 1–8: GST PSECT Definition Entry

Figure 1–8 Cont'd. on next page

Figure 1–8 (Cont.): GST PSECT Definition Entry



PSECT entries are described by the following fields:

- ENTRY_TYPE is the value MODOBJ$C_PSECT_DEFINITION.
- ENTRY_SUBTYPE is the value MODOBJ$C_PSECT_UNINITIALIZED, MODOBJ$C_PSECT_STANDARD, or MODOBJ$C_PSECT_COMPRESSED.
- ENTRY_SIZE is 20 for uninitialized PSECTs, 28 for standard PSECTs, and 36 for compressed PSECTs.
- NAME_INDEX is the index of the PSECT's name in the module name table. The index can not be zero.
- ATTRIBUTES specifies the attributes of the PSECT. All contributions to the same PSECT must have the same attributes.
  - MODOBJ$C_PSECT_OVERLAID (OL) is set if the PSECT is overlaid, and clear if it is concatenated.
  - MODOBJ$C_PSECT_RELOCATABLE (RL) is set if the PSECT is relocatable, and clear if it is absolute. Absolute PSECTs do not have any memory allocation; they are used only for defining absolute symbols.
  - MODOBJ$C_PSECT_SHAREABLE (SH) is set if the PSECT is accessible from outside a shareable image. It is clear if the PSECT is private to a shareable image. The linker can allow a shareable PSECT to be shared between two or more active processes through a qualifier specified in Chapter 30, Linker.
  - MODOBJ$C_PSECT_READABLE (RD) is set if the PSECT is readable.
  - MODOBJ$C_PSECT_WRITABLE (WR) is set if the PSECT is writable.
  - MODOBJ$C_PSECT_EXECUTABLE (EX) is set if the PSECT contains executable code.

  - MODOBJ$C_PSECT_MESSAGE (MS) is set if the PSECT contains message section descriptors.
  - MODOBJ$C_PSECT_GLOBAL (GL) is set if the PSECT is global across clusters of program sections. Only global, shareable, overlaid PSECTs resolve to the same global, shareable, overlaid PSECT in another image.
  - MODOBJ$C_PSECT_TLS (TL) is set if the PSECT represents a template for thread local storage. A thread local storage template provides initialization for memory that is private for each thread.

- ALIGN specifies the PSECT alignment (see Section 1.3.4 for the values).

  \In order to guarantee the memory layout for PSECT allocations of the same name within the same module, all PSECT allocations for a like-named PSECT in the same module must use the same alignment.\

- ALLOCATION specifies in bytes the amount of memory required to represent the PSECT. For uninitialized PSECTs, this represents the amount of memory contributed by this definition. For standard PSECTs, this represents both the amount of memory and the amount of PSECT data or code in the module. For compressed PSECTs, this represents the amount of memory that the compressed PSECT data will initialize.
- STANDARD_OFFSET and COMPRESSED_OFFSET specify the byte offset from the beginning of the module to the start of the PSECT data or code.
- COMPRESSED_SIZE specifies the size of the compressed data in the module.

1.3.3.3 FORTRAN Common Definition

Process-wide FORTRAN common is defined as a PSECT with only the following attributes.

- MODOBJ$C_PSECT_OVERLAID
- MODOBJ$C_PSECT_RELOCATABLE
- MODOBJ$C_PSECT_SHAREABLE
- MODOBJ$C_PSECT_READABLE
- MODOBJ$C_PSECT_WRITABLE
- MODOBJ$C_PSECT_GLOBAL

Thread local FORTRAN common is defined as a PSECT with only the following attributes.

- MODOBJ$C_PSECT_OVERLAID
- MODOBJ$C_PSECT_RELOCATABLE
- MODOBJ$C_PSECT_READABLE
- MODOBJ$C_PSECT_WRITABLE
- MODOBJ$C_PSECT_GLOBAL
- MODOBJ$C_PSECT_TLS

1.3.3.4 PSECT Definition and Symbol Reference Entry

A PSECT definition and symbol reference is also known as a weak PSECT definition, because it can resolve to a symbol without defining a program section. This entry is used by the C compiler to describe the an external reference that may be resolved to either a global symbol or a FORTRAN common. The semantics of this reference require that the FORTRAN common be created if it doesn't already exist.

\The definition of this entry is bounded by the requirements of C, and many of the features of a standard PSECT definition are not used because C does not require them.\

Note that this is a special form of a PSECT definition, and is referenced in the global symbol table through a PSECT index, and not a symbol index. The declaration is the same for a standard PSECT (see Section 1.3.3.2).

The linker implements PSECT definition/symbol reference in the following way. During pass 1 of the linker, the linker treats these entries as symbol references, and searches for symbol definitions. At the completion of pass 1, the linker searches for a PSECT definition of the same name.

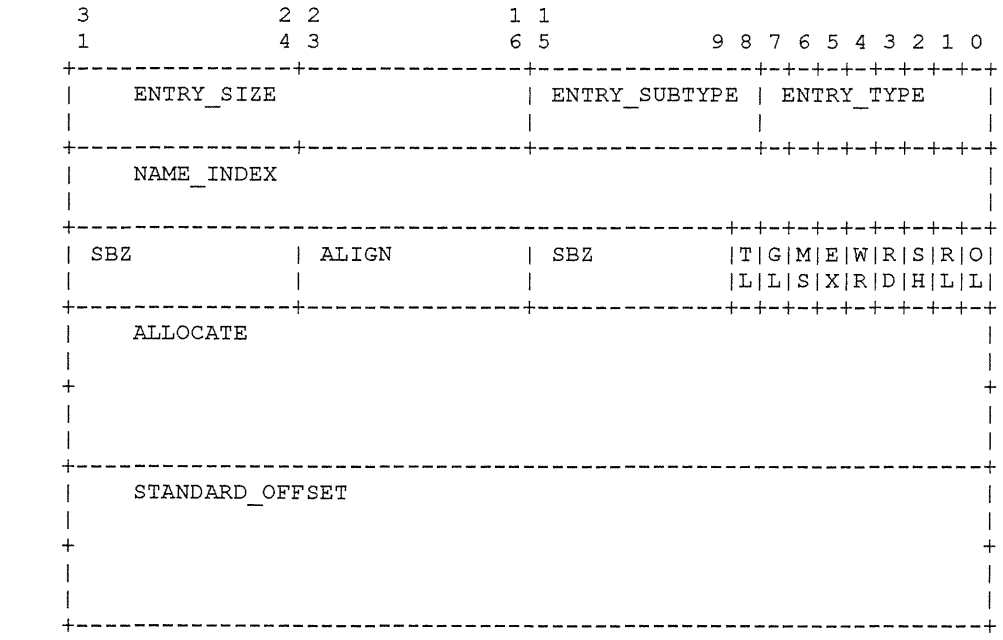
- If a symbol definition is found, and no matching PSECT definition is found, the linker resolves the reference(s) to the symbol definition
- If a PSECT definition is found, and no matching symbol definition is found, the linker resolves the reference(s) to the PSECT definition if the attributes match.
- If no symbol or PSECT definition is found, the linker defines the PSECT using the information in the entry. Note that it only defines a PSECT and does not define a symbol.
- If both symbol and PSECT definitions are found, the linker resolves the reference to the PSECT definition and issues a warning.

When no symbol or PSECT definition is found, the linker not only defines the PSECT using the information in the entry, but it also describes it in the image's global symbol table with the PSECT definition/symbol reference entry. This allows the linker to issue a warning in the second of the following cases.

1. Module A has a global symbol definition of "foo" and module B has a global common symbol definition of "foo," and object module A links with object module B.
2. Module A has a global symbol definition of "foo" and module B has a global common symbol definition of "foo," and object module A links with shareable image B.

A layout of the PSECT part of a PSECT definition entry follow:

Figure 1–9: GST PSECT Definition/Symbol Reference Entry



PSECT entries are described by the following fields:

- ENTRY_TYPE is the value MODOBJ$C_PSECT_SYMBOL_DEFINITION.
- ENTRY_SUBTYPE is the value MODOBJ$C_PSECT_UNINITIALIZED in object modules (because C does not allow this types of reference to be initialized), and MODOBJ$C_PSECT_ STANDARD in image files.
- ENTRY_SIZE is 20 for uninitialized PSECTs, 28 for standard PSECTs.
- NAME_INDEX is the index of the PSECT's name in the module name table. The index can not be zero.
- ATTRIBUTES specifies the attributes of the PSECT.

\The attributes match FORTRAN common because C only uses this type of reference to match FORTRAN common.\

— MODOBJ$C_PSECT_OVERLAID (OL) is set.

— MODOBJ$C_PSECT_RELOCATABLE (RL) is set.

— MODOBJ$C_PSECT_SHAREABLE (SH) is set, but might be cleared by the linker.

— MODOBJ$C_PSECT_READABLE (RD) is set.

— MODOBJ$C_PSECT_WRITABLE (WR) is set.

— MODOBJ$C_PSECT_EXECUTABLE (EX) is clear.

— MODOBJ$C_PSECT_MESSAGE (MS) is clear.

— MODOBJ$C_PSECT_GLOBAL (GL) is set.

— MODOBJ$C_PSECT_TLS (TL) is clear.

- ALIGN specifies the PSECT alignment.
- ALLOCATION specifies in bytes the amount of memory required to represent the PSECT. The linker will maximize this with other module's allocations because this is an overlaid PSECT.

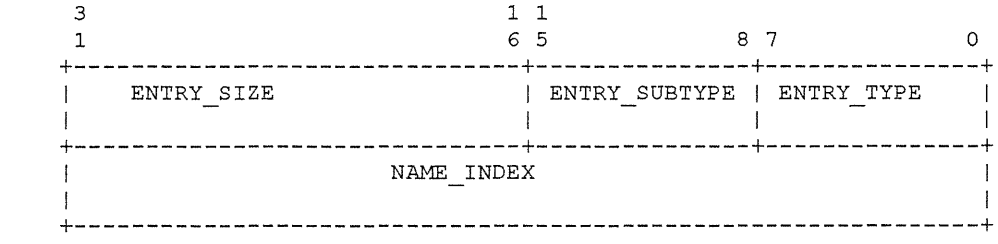
1.3.3.5 Global Symbol Reference Entry

A global symbol reference is also known as an intolerant symbol reference because it can not reference a PSECT definition. Global symbol reference entries specify references to global symbols. There are two types of global symbol references:

- Standard references, which have the value MODOBJ$C_SYMBOL_STANDARD. If these references are unresolved, the linker reports an error.
- Weak references, which have the value MODOBJ$C_SYMBOL_WEAK. If these references are unresolved, the linker treats them as zero value. These references do not cause the linker to search new modules in a library to resolve them.

Global symbol reference entries have the following format:

Figure 1–10: GST Global Symbol Reference Entry



These fields are defined as follows:

- ENTRY_TYPE is the value MODOBJ$C_SYMBOL_REFERENCE.
- ENTRY_SUBTYPE is MODOBJ$C_SYMBOL_STANDARD or MODOBJ$C_SYMBOL_WEAK.
- ENTRY_SIZE is eight.

- NAME_INDEX is the index of the symbol's name in the module name table. If the index can not be zero.

Note that global symbol reference entries do not contain the PSECT index or symbol value fields.

1.3.3.6 Global Symbol Or PSECT Reference Entry

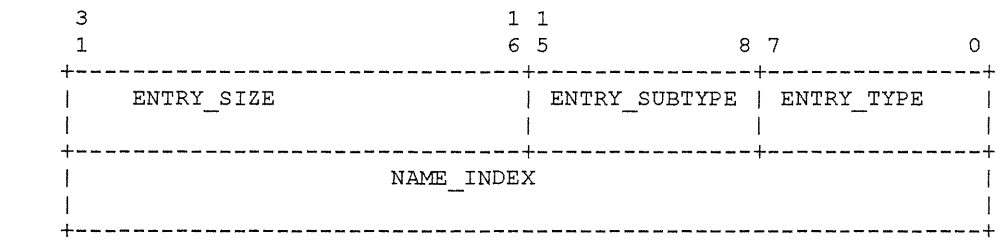
A global symbol or PSECT reference is also known as a tolerant symbol reference because it can reference a PSECT definition if there is no matching symbol definition. Global symbol or PSECT reference entries specify references to global symbols or PSECTs. The linker implements symbol or PSECT references in the following way. During pass 1 of the linker, the linker treats these entries as symbol references, and searches for symbol definitions. At the completion of pass 1, if no matching symbol definition was found, the linker searches for a PSECT definition of the same name. Note that the reference resolves to the matching symbol definition without an error if both a symbol definition and a PSECT definition match.

There are two types of these references.

- Standard references, which have the value MODOBJ$C_SYMBOL_STANDARD. If these references are unresolved, the linker reports an error.
- Weak references, which have the value MODOBJ$C_SYMBOL_WEAK. If these references are unresolved, the linker treats them as zero value. These references do not cause the linker to search new modules in a library to resolve them.

Global symbol/PSECT reference entries have the following format:

Figure 1–11: GST Global Symbol/PSECT Reference Entry



These fields are defined as follows:

- ENTRY_TYPE is the value MODOBJ$C_SYMBOL_PSECT_REFERENCE.
- ENTRY_SUBTYPE is MODOBJ$C_SYMBOL_STANDARD or MODOBJ$C_SYMBOL_WEAK.
- ENTRY_SIZE is eight.
- NAME_INDEX is the index of the symbol's name in the module name table. The index can not be zero.

Note that symbol/PSECT reference entries do not contain the PSECT index or symbol value fields.

1.3.3.7 Global Symbol Definition

This entry defines a symbol that can be referenced from any other module. The symbol can be absolute, relocatable, procedural, or common. The types of local and internal symbols are a subset of global symbols. The declaration, values, and layouts of the symbol definition portion of a global symbol definition entry follow:

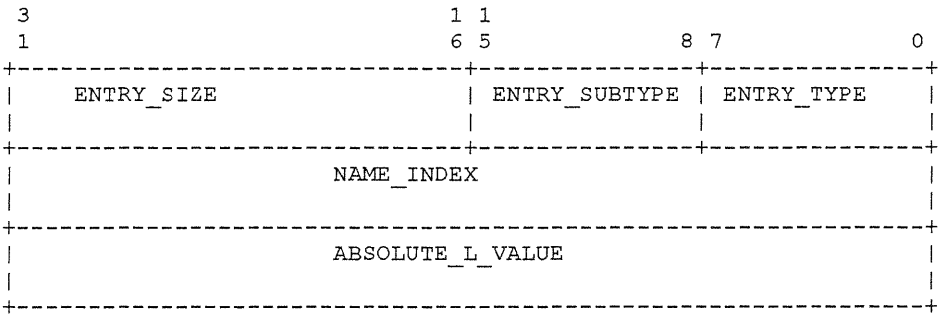
```
modobj$procedure_argument(
    modobj$c_argument_unknown,
    modobj$c_argument_value,
    modobj$c_argument_reference,
    modobj$c_argument_descriptor
    );

modobj$procedure_descriptor( argument_list_count : module$entry_size ):
    ARRAY [1..argument_list_count] OF
        modobj$procedure_argument[..] SIZE(byte);

! \BLISS uses prefix MODOBJ$SD_\
modobj$symbol_definition(
    subtype : module$entry_type;
    subsize : module$entry_size[..] SIZE(word)
    ) : RECORD
    VARIANTS CASE subtype
        WHEN modobj$c_symbol_absolute_long THEN
            absolute_l_value : integer;
        WHEN modobj$c_symbol_absolute_quad THEN
            absolute_q_value : large_integer;
        WHEN modobj$c_symbol_relocate_long THEN
            relocate_l_psect : longword;
            relocate_l_offset : longword;
        WHEN modobj$c_symbol_relocate_quad THEN
            relocate_q_psect : longword;
            relocate_q_offset : quadword;
        WHEN modobj$c_symbol_procedure_long THEN
            procedure_l_psect : longword;
            procedure_l_offset : longword;
            procedure_l_code_psect : longword;
            procedure_l_code_offset : longword;
            procedure_l_arguments : modobj$procedure_descriptor(subsize-16);
        WHEN modobj$c_symbol_procedure_quad THEN
            procedure_q_psect : longword;
            procedure_q_offset : quadword;
            procedure_q_code_psect : longword;
            procedure_q_code_offset : quadword;
            procedure_q_arguments : modobj$procedure_descriptor(subsize-24);
        WHEN modobj$c_symbol_transfer THEN
            transfer_index : longword;
            transfer_arguments : modobj$procedure_descriptor(subsize-4);
    END VARIANTS;
END RECORD;
```

Note that the semantics of a VMS weak global definition are equivalent to defining a global symbol in the global symbol table while its name in the module name table has no name index.

1.3.3.7.1 Absolute Global Symbol Definition with Longword Value

Global absolute symbol definition entries having a longword symbol value have the following format:

Figure 1–12: GST Absolute Global Symbol Definition Entry with Longword Value



These fields are defined as follows:

- ENTRY_TYPE is the value MODOBJ$C_SYMBOL_DEFINITION.
- ENTRY_SUBTYPE is the value MODOBJ$C_SYMBOL_ABSOLUTE_LONG..
- ENTRY_SIZE is 12.
- NAME_INDEX is the index of the symbol's name in the module name table. If the index is zero, the symbol is unnamed.
- ABSOLUTE_L_VALUE is the value of the symbol.

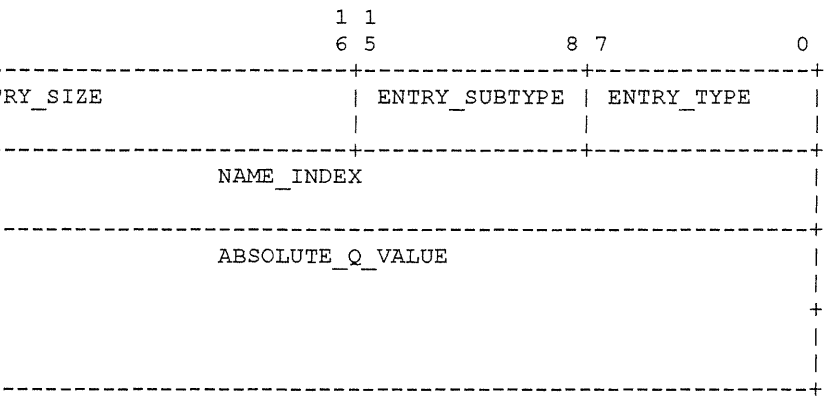
1.3.3.7.2 Absolute Global Symbol Definition with Quadword Value

Global absolute symbol definition entries having a quadword symbol value have the following format:

Figure 1–13: GST Absolute Global Symbol Definition Entry with Quadword Value

Figure 1–13 Cont'd. on next page

Figure 1–13 (Cont.): GST Absolute Global Symbol Definition Entry with Quadword Value



These fields are defined as follows:

- ENTRY_TYPE is the value MODOBJ$C_SYMBOL_DEFINITION.
- ENTRY_SUBTYPE is the value MODOBJ$C_SYMBOL_ABSOLUTE_QUAD.
- ENTRY_SIZE is 16.
- NAME_INDEX is the index of the symbol's name in the module name table. If the index is zero, the symbol is unnamed.
- ABSOLUTE_Q_VALUE is the value of the symbol.

1.3.3.7.3 Relocatable Symbol Definition with Longword Value

Global symbol definition entries having a longword symbol value have the following format:

Figure 1–14: GST Global Symbol Definition Entry with Longword Value

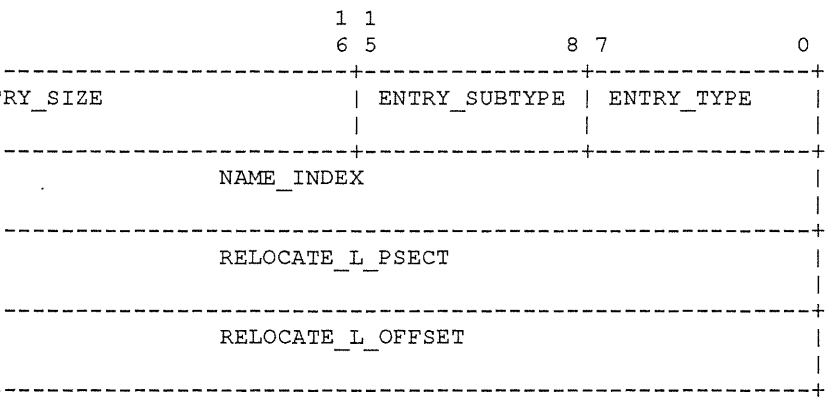


Figure 1–14 Cont'd. on next page

Figure 1–14 (Cont.): GST Global Symbol Definition Entry with Longword Value

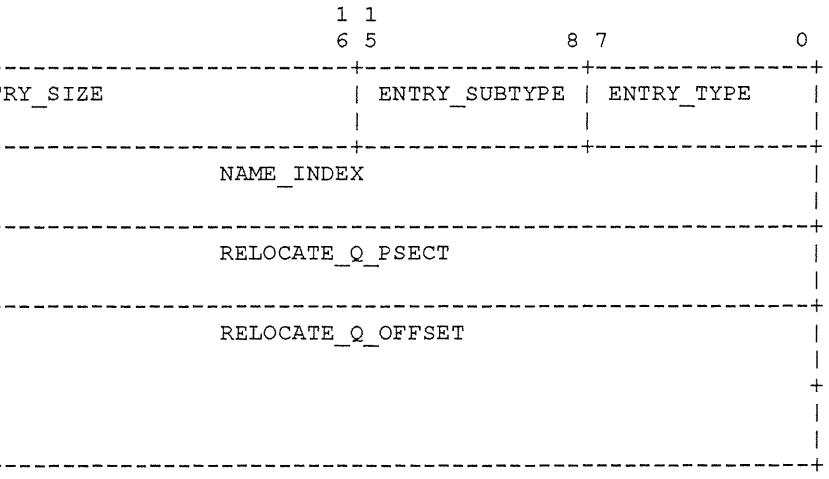
These fields are defined as follows:

- ENTRY_TYPE is the value MODOBJ$C_SYMBOL_DEFINITION.
- ENTRY_SUBTYPE is the value MODOBJ$C_SYMBOL_RELOCATE_LONG.
- ENTRY_SIZE is 16.
- NAME_INDEX is the index of the symbol's name in the module name table. If the index is zero, the symbol is unnamed.
- RELOCATE_L_PSECT contains the PSECT index in which the symbol is defined.
- RELOCATE_L_OFFSET is the value of the symbol, which is the offset into the specified PSECT.

1.3.3.7.4 Relocatable Symbol Definition with Quadword Value

Global symbol definition entries having a quadword symbol value have the following format:

Figure 1–15: GST Global Symbol Definition Entry with Quadword Value



These fields are defined as follows:

- ENTRY_TYPE is the value MODOBJ$C_SYMBOL_DEFINITION.
- ENTRY_SUBTYPE is the value MODOBJ$C_SYMBOL_RELOCATE_QUAD.
- ENTRY_SIZE is 20.

- NAME_INDEX is the index of the symbol's name in the module name table. If the index is zero, the symbol is unnamed.
- RELOCATE_Q_PSECT contains the PSECT index in which the symbol is defined.
- RELOCATE_Q_OFFSET is the value of the symbol, which is the offset into the specified PSECT.

1.3.3.7.5 Global Procedure Definition Entry with Longword Value

The global procedure definition entry fulfills three functions: it defines the invocation descriptor of a procedure, it defines the entry point (code address) of a procedure, and it describes arguments for FORTRAN string argument coercion. Note that the formal argument description is needed only by the linker for FORTRAN to coerce string arguments at link time, and other languages need not generate any formal argument description. Global procedure definition entries with longword value have the following format:

Figure 1–16: GST Global Procedure Definition Entry

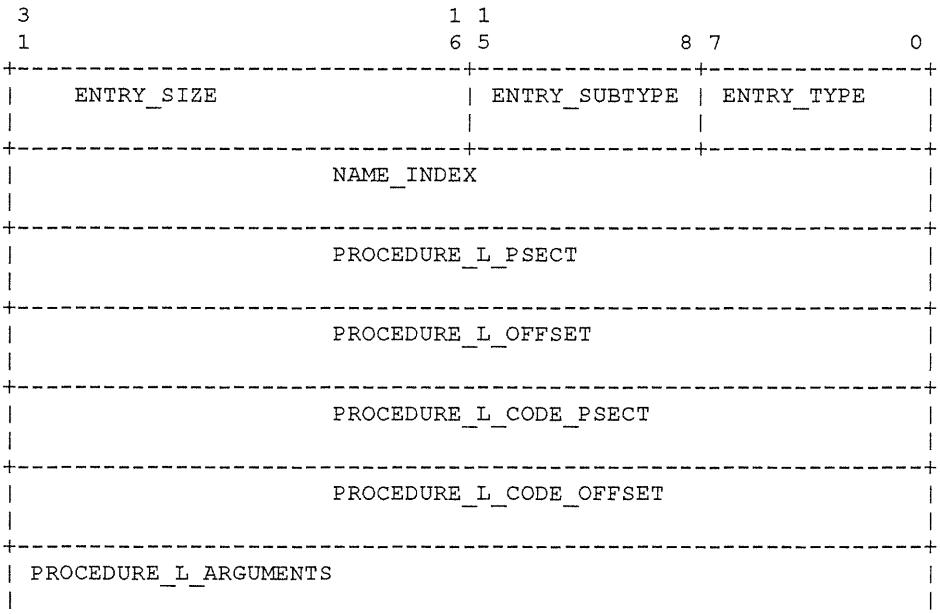


Figure 1–16 Cont'd. on next page

Figure 1–16 (Cont.): GST Global Procedure Definition Entry

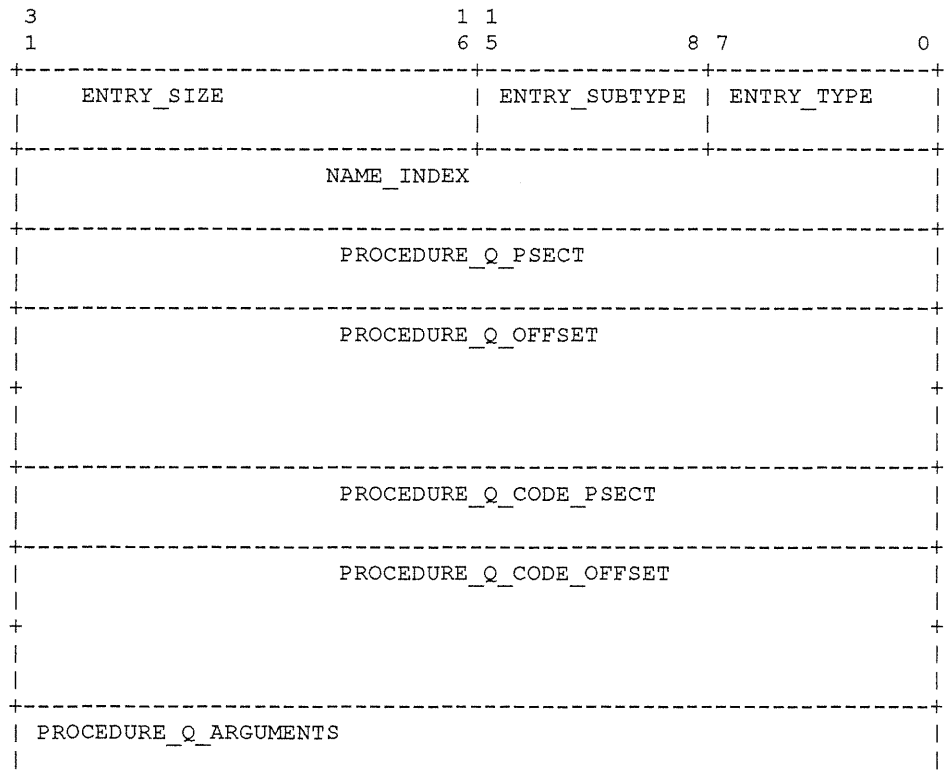
These formal arguments are defined as follows:

- ENTRY_TYPE is the value MODOBJ$C_SYMBOL_DEFINITION.
- ENTRY_SUBTYPE is the value MODOBJ$C_SYMBOL_PROCEDURE_LONG.
- ENTRY_SIZE is 24 plus the size of the argument descriptors.
- NAME_INDEX is the index of the symbol's name in the module name table. If the index is zero, the symbol is unnamed.
- PROCEDURE_L_PSECT contains the PSECT index in which the procedure's invocation descriptor is defined.
- PROCEDURE_L_OFFSET contains the offset into the specified PSECT of the procedure's invocation descriptor.
- PROCEDURE_L_CODE_PSECT contains the PSECT index in which the procedure's entry point is defined.
- PROCEDURE_L_CODE_OFFSET contains the offset into the specified PSECT of the procedure's entry point.
- PROCEDURE_L_ARGUMENTS describes the specified number of arguments in a byte-stream oriented format. Each formal argument in PROCEDURE_L_ARGUMENTS is represented by a byte that contains one of the following values:
  - MODOBJ$C_ARGUMENT_UNKNOWN
  - MODOBJ$C_ARGUMENT_VALUE
  - MODOBJ$C_ARGUMENT_REFERENCE
  - MODOBJ$C_ARGUMENT_DESCRIPTOR

1.3.3.7.6 Global Procedure Definition Entry with Quadword Value

The global procedure definition entry fulfills three functions: it defines the invocation descriptor of a procedure, it defines the entry point (code address) of a procedure, and it describes arguments for FORTRAN string argument coercion. Note that the formal argument description is needed only by the linker for FORTRAN to coerce string arguments at link time, and other languages need not generate any formal argument description. Global procedure definition entries with quadword value have the following format:

Figure 1–17: GST Global Procedure Definition Entry, Quadword Value



These formal arguments are defined as follows:

- ENTRY_TYPE is the value MODOBJ$C_SYMBOL_DEFINITION.
- ENTRY_SUBTYPE is the value MODOBJ$C_SYMBOL_PROCEDURE_QUAD.
- ENTRY_SIZE is 32 plus the size of the argument descriptors.

- NAME_INDEX is the index of the symbol's name in the module name table. If the index is zero, the symbol is unnamed.
- PROCEDURE_Q_PSECT contains the PSECT index in which the procedure's invocation descriptor is defined.
- PROCEDURE_Q_OFFSET contains the offset into the specified PSECT of the procedure's invocation descriptor.
- PROCEDURE_Q_CODE_PSECT contains the PSECT index in which the procedure's entry point is defined.
- PROCEDURE_Q_CODE_OFFSET contains the offset into the specified PSECT of the procedure's entry point.
- PROCEDURE_Q_ARGUMENTS describes the specified number of arguments in a byte-stream oriented format. Each formal argument in PROCEDURE_Q_ARGUMENTS is represented by a byte that contains one of the following values:
  - MODOBJ$C_ARGUMENT_UNKNOWN
  - MODOBJ$C_ARGUMENT_VALUE
  - MODOBJ$C_ARGUMENT_REFERENCE
  - MODOBJ$C_ARGUMENT_DESCRIPTOR

1.3.3.7.7 Global Transfer Definition Entry

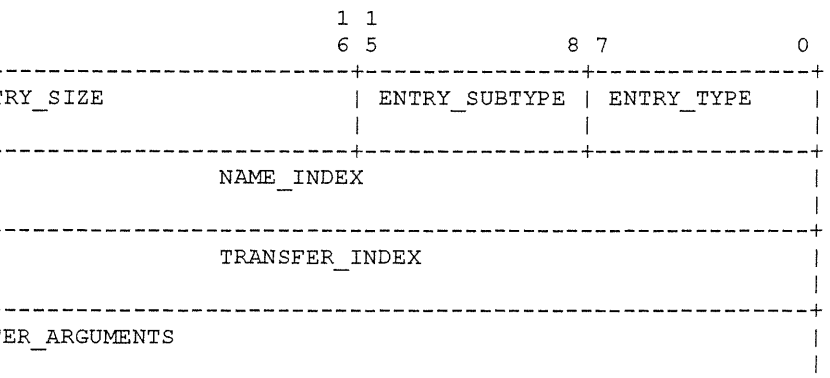
A global transfer definition entry only occurs in shareable images' global symbol tables and it identifies the transfer vector of a routine in a shareable image. This entry is created by the linker when it makes a universal procedure name in a shareable image. Unlike VMS, it is impossible to have a universal procedure without a transfer vector. (See Section 1.6 for details on symbol resolution in shareable images.)

The layout of the global transfer definition entry is similar to the global procedure definition entries. As with global procedure definition entries, there is a description of the formal arguments defined by the procedure. Global transfer definition entries have the following format:

Figure 1–18: GST Global Transfer Definition Entry

Figure 1–18 Cont'd. on next page

Figure 1–18 (Cont.): GST Global Transfer Definition Entry



These formal arguments are defined as follows:

- ENTRY_TYPE is the value MODOBJ$C_SYMBOL_DEFINITION.
- ENTRY_SUBTYPE is the value MODOBJ$C_SYMBOL_TRANSFER.
- ENTRY_SIZE is 12 plus the size of the argument descriptors.
- NAME_INDEX is the index of the symbol's name in the module name table. If the index is zero, the symbol is unnamed.
- TRANSFER_INDEX is the value of the symbol, which is its index into the shareable image's transfer vector table.
- TRANSFER_ARGUMENTS describes the specified number of arguments in a byte-stream oriented format. Each formal argument in TRANSFER_ARGUMENTS is represented by a byte that contains one of the following values:
  - MODOBJ$C_ARGUMENT_UNKNOWN
  - MODOBJ$C_ARGUMENT_VALUE
  - MODOBJ$C_ARGUMENT_REFERENCE
  - MODOBJ$C_ARGUMENT_DESCRIPTOR

1.3.4 Target Record

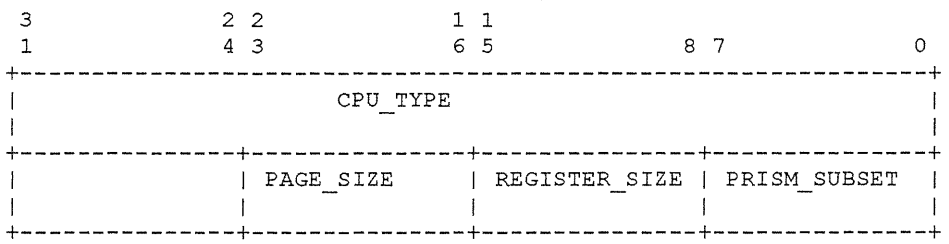
The target record contains the target's CPU type, its subset of the PRISM architecture, its page size, and its register size. The values and declarations of the fields in the match record follow:

```
module$data_size : (
    module$c_byte,
    module$c_word,
    module$c_longword,
    module$c_quadword,
    module$c_octaword,
    module$c_32byte,
    module$c_64byte,
    module$c_128byte,
    module$c_256byte,
    module$c_512byte,
    module$c_1kbyte,
    module$c_2kbyte,
    module$c_4kbyte,
    module$c_8kbyte,
    module$c_16kbyte,
    module$c_32kbyte,
    module$c_64kbyte,
```

```
        module$c_page
        );

    module$target_type : (
        module$c_target_stone,
        module$c_target_shrike,
        module$c_target_osprey
        );

    module$target_subset : (
        module$c_target_integer,
        module$c_target_floating,
        module$c_target_vector,
        module$c_target_coprocessor
        );

    ! \BLISS uses prefix MODOBJ$TR_\
    module$target_record : RECORD
        cpu_type : module$target_type;
        prism_subset : SET module$target_subset[..] SIZE(byte);
        register_size : module$data_size[..] SIZE(byte);
        page_size : module$data_size[..] SIZE(byte));
    END RECORD;
```

\MODULE$C_PAGE denotes whatever the existing page size is.\

Figure 1–19: Target Record



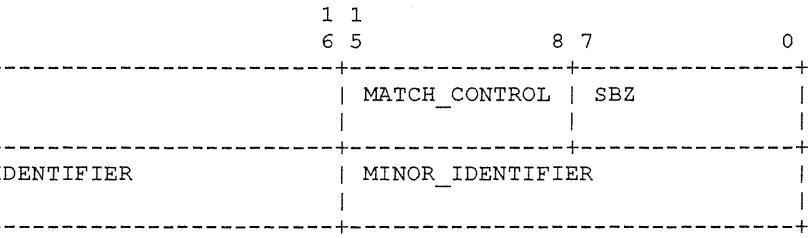
1.3.5 Match Record

The match record contains the segment match control along with the major and minor identifier specified by the user when the module was created. The values and declarations of the fields in the match record follow:

```
    module$match_identity : integer[0..65535];

    module$match_control : (
        module$c_match_always,
        module$c_match_less_equal,
        module$c_match_equal
        );
```

```
! \BLISS uses prefix MODULE$MR_\
module$match_record : RECORD
    match_control : module$match_control[..]SIZE(byte);
    minor_identifier : module$match_identity[..] SIZE(word);
    major_identifier : module$match_identity[..] SIZE(word);
END RECORD;
```

Figure 1–20: Match Record



The match record describes a module's compatibility with previous and subsequent versions of itself. As an example, this record is used by the linker and the image activator to check the compatibility of different versions of shareable images. The linker stores in the executable image the shareable image's match record values, along with the shareable image's specification. The image activator compares the major and minor identifier values stored in the executable image with the values of the current version of the shareable image. The image activator then uses the match control value stored in the executable image to determine the compatibility of the shareable image. For more information on the linker's use of this record, see Chapter 30, Linker.

If the match control value is MODULE$C_MATCH_EQUAL, the major IDs of the two versions must match exactly, and the minor IDs must also match exactly. If the match control value is MODULE$C_MATCH_LESS_EQUAL, the major IDs that are compared must match exactly, but the minor identifier of the previous version must be less than or equal to the current version. If the match control value is MODULE$C_MATCH_ALWAYS, neither the major IDs nor the minor IDs are required to match. The absence of this item is the same as specifying MODULE$C_MATCH_ALWAYS.

1.3.6 Entity Consistency Check Table

The entity consistency check table provides a means of ensuring that the exact version of a file used by a compiler is being used by the linker. During pass 1, the linker reads the entity consistency check table. It then searches an internal table for a matching entity name. If a matching entity name is found, the identifications are compared, and an error message is issued if they differ. If not found, the linker simply adds the entity to its internal table.

After pass 1, the linker checks each entry's identification with the object's identification. The object's identification is found by searching the linker's internal table for an entry whose entity name matches the object name of the original entry. The match control of the original entry is used to compare the original entry's identification with the object's identification. An entity can have multiple entries, but all of an entity's entries should have the same consistency check type. An entry can have a zero-length object name, which allows an entity to be defined without a check.

The table consists of a byte-stream of entries, and each entry is word aligned. The values and declaration of the entity consistency check entries follows:

- MODULE$C_ENTITY_BINARY—Entity check with binary identification

- MODULE$C_ENTITY_ASCII—Entity check with ASCII identification

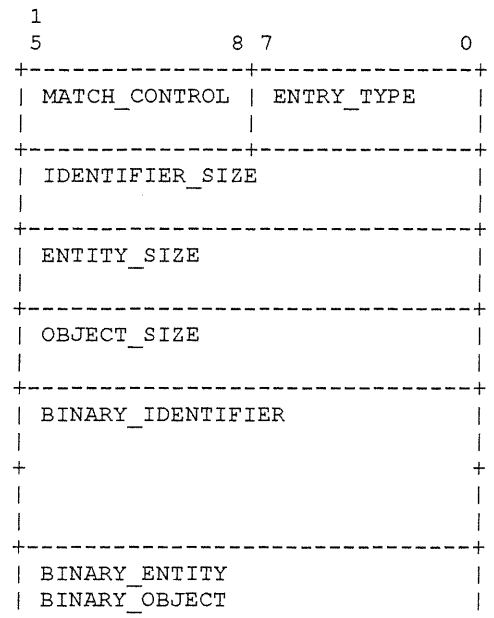
```
module$entity_consistency_type : (
    module$c_entity_binary,
    module$c_entity_ascii
    );

! \BLISS uses prefix MODULE$ECE_\
module$entity_consistency_entry(
    entry_type : module$entity_consistency_type
                                [..] SIZE(byte);
    identifier_size : unsigned_word;
    entity_size : unsigned_word;
    object_size : unsigned_word
    ) : RECORD
    CAPTURE
        entry_type,identifier_size,entity_size,object_size;
    match_control : module$match_control[..] SIZE(byte);
    VARIANTS CASE entry_type
        WHEN module$c_entity_binary THEN
            binary_identifier : integer;
            binary_entity : string(entity_size);
            binary_object : string(object_size);
        WHEN module$c_entity_ascii THEN
            ascii_identifier : string(identifier_size);
            ascii_entity : string(entity_size);
            ascii_object : string(object_size);
    END VARIANTS;
END RECORD;
```

1.3.6.8 Entity Check with Binary Identification

Figure 1–21: Entity Check with Binary Identification

Figure 1–21 Cont'd. on next page

Figure 1–21 (Cont.): Entity Check with Binary Identification

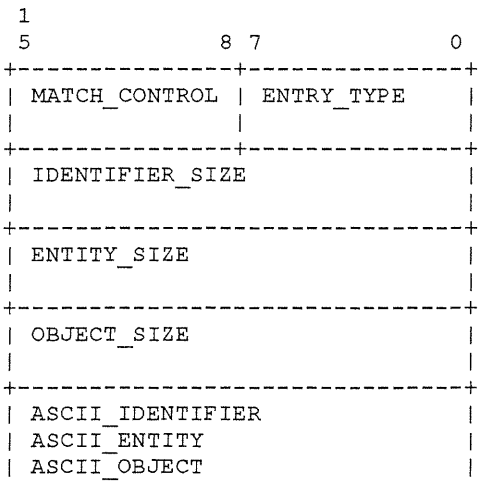


These fields are defined as follows:

- ENTRY_TYPE is MODULE$C_ENTITY_BINARY.
- MATCH_CONTROL specifies the match control on the binary identification. Values are MODULE$C_LESS_EQUAL and MODULE$C_EQUAL.
- IDENTIFIER_SIZE should be zero.
- ENTITY_SIZE is the length of the entity's name.
- OBJECT_SIZE is the length of the object's name, or zero if there is no object.
- BINARY_IDENTIFIER specifies the binary identification.
- BINARY_ENTITY is the entity's name.
- BINARY_OBJECT is the object's name.

1.3.6.9 Entity Check with ASCII Identification

Figure 1–22: Entity Check with ASCII Identification



These fields are defined as follows:

- ENTRY_TYPE is MODULE$C_ENTITY_ASCII.
- MATCH_CONTROL is MODULE$C_EQUAL.
- IDENTIFIER_SIZE is the length of the identifier's name.
- ENTITY_SIZE is the length of the entity's name.
- OBJECT_SIZE is the length of the object's name or zero if there is no object.
- ASCII_IDENTIFIER is the identifier's name.
- ASCII_ENTITY is the entity's name.
- ASCII_OBJECT is the object's name.

1.3.7 Debug Symbol Table

The debug symbol table allows the debugger to interpret user commands and display memory contents in "the current programming language." The debug symbol table is created by the compiler and interpreted by the debugger in conjunction with the debug module table. The linker does not fix up the debug symbol table.

\The debug symbol table is being designed by SDT. The specification of the debug symbol table in the object module may be included when its design is finalized.\

1.4 Data Structures Specific to Object Modules

Object modules contain data structures that are not shared with image files or other types of modules. These data structures contain information that is useful to compilers and the linker, but is not needed for image activation.

1.4.1 Linker Directive Table

The linker directive table provides a means for the compiler to "program" the linker with initialization routines, additional libraries to search, and additional files to include in the link. This table consists of simple counted ASCII strings—word aligned—each string being a linker command. The following are valid linker commands (see Chapter 30, Linker for more details):

- IMAGE_FILE specifies the inclusion of a shareable image in the link.
- INITIAL_ROUTINE specifies the initialization routines that are called when the image is activated. Routines are specified with the name of their global symbol.
- LIBRARY_FILE specifies the inclusion of a library in the link.
- MODULE specifies the inclusion of modules in a library file in the link.
- OBJECT_FILE specifies the inclusion of an object module file in the link.

\The syntax for these commands is not yet determined, but it will be common to all systems.\

1.4.2 Data Relocation Table

The data relocation table is generated by the compiler, and contains sufficient information to direct the linker to fix up a data section. This table is a collection of command-oriented structures. The first byte of the command structure is the type, indicating the relocation to be performed. Following the type is the variable-length additional information required for the command.

The data relocation table consists of a series of variable-length entries, and each entry is longword aligned. The declarations of the data relocation values and entries follow:

```
modobj$relocate_type : (
    modobj$c_relocate_data,
    modobj$c_relocate_procedure,
    modobj$c_relocate_argument,
    modobj$c_relocate_psect_size,
    modobj$c_relocate_tls_offset
    );
```

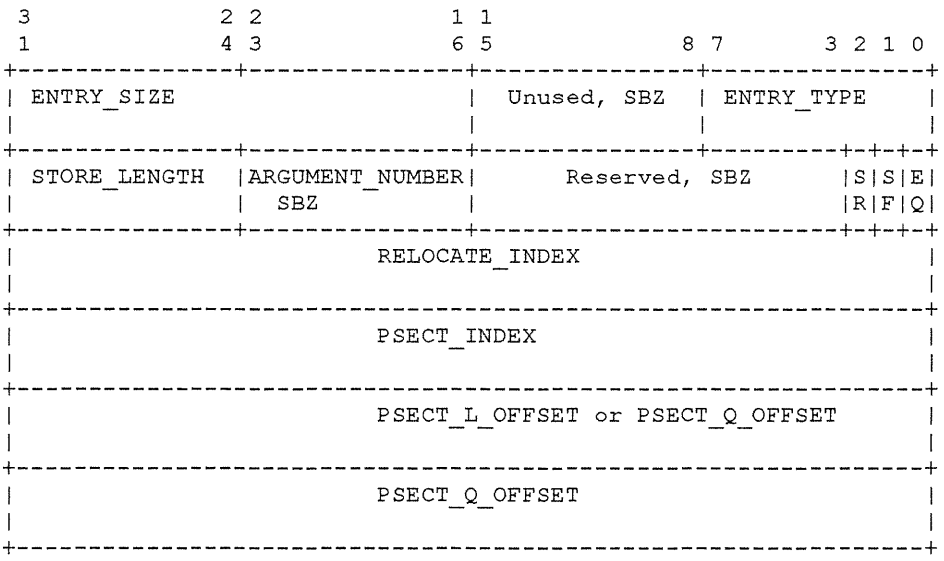
```
! \BLISS uses prefix MODOBJ$RE_\
modobj$relocate_entry(
    entry_type : modobj$relocate_type[..] SIZE(byte);
    entry_size : module$entry_size[..] SIZE(word);
    entry_quadword : bit
    ) : RECORD
    CAPTURE entry_type, entry_size, entry_quadword;
    symbol_fixup : bit;
    self_relative : bit;
    argument_number : unsigned_byte;
    store_length : module$data_size[..] SIZE(byte);
    relocate_index : longword;
    psect_index : longword;
    VARIANTS CASE entry_quadword
        WHEN false THEN
            psect_l_offset : longword;
            VARIANTS CASE entry_type
                WHEN modobj$c_relocate_argument THEN
                    descriptor_l_index : longword;
                    descriptor_l_offset : longword;
                    string_l_index : longword;
                    string_l_offset : longword;
                WHEN OTHERS THEN
                    NOTHING;
            END VARIANTS;
        WHEN true THEN
            psect_q_offset : quadword;
            VARIANTS CASE entry_type
                WHEN modobj$c_relocate_argument THEN
                    descriptor_q_index : longword;
                    descriptor_q_offset : quadword;
                    string_q_index : longword;
                    string_q_offset : quadword;
                WHEN OTHERS THEN
                    NOTHING;
            END VARIANTS;
    END VARIANTS;
END RECORD;
```

The contents of the data relocation section are described in the following subsections.

1.4.2.1 Global Symbol and PSECT Relocations

Global symbol and PSECT relocations direct the linker to add the value of an external symbol or base address of a PSECT to either the contents of a memory location within the data section, or the contents of a memory location minus the location's address. The contents of these relocation items are shown in Figure 1–23.

Figure 1–23: DRT Global Symbol and PSECT Entry



- ENTRY_TYPE is MODOBJ$C_RELOCATE_DATA. The relocation item is either a global symbol relocation or a PSECT relocation, specified by SYMBOL_FIXUP.
- ENTRY_SIZE is 20 if the entry_quadword is clear, or 24 if it is set.
- ENTRY_QUADWORD (EQ) is set if the entry contains the quadword field PSECT_Q_OFFSET. ENTRY_QUADWORD is clear if the entry contains the longword field PSECT_L_OFFSET.
- SYMBOL_FIXUP (SF) is set if the entry describes a global symbol fixup. SYMBOL_FIXUP is clear if the fixup is to a PSECT.
- SELF_RELATIVE (SR) is set if the fixup is self-relative to the location being fixed. SELF_RELATIVE is clear if the fixup is absolute. If SELF_RELATIVE is set, STORE_LENGTH must be either MODULE$C_LONGWORD (32-bit architecture) or MODULE$C_QUADWORD (64-bit architecture).
- STORE_LENGTH specifies the size of the value to be stored (see Section 1.3.4).
- RELOCATE_INDEX is the relative symbol number (if SYMBOL_FIXUP is set) or PSECT number to use in the fixup.
- PSECT_INDEX is the PSECT number containing the location to be fixed up.

- PSECT_L_OFFSET or PSECT_Q_OFFSET is the relative byte offset within the PSECT identified by PSECT_INDEX for the current module to fix up.

\When several object modules are combined into one module, these fixups will be modified to refer to the new, improved relative symbol and PSECT numbers.\

\Note that the compiler can do external link-time fixups using this mechanism. To add two (or more) external literals together and store the result in a location, the compiler initializes the location to zero, and generates the appropriate RELOCATE_DATA fixups. The linker always picks up the previous value from the location for the fixup. If the referenced symbols are not known at link time (for example, the symbol is resolved to a shareable image), an error occurs.\

1.4.2.2 Procedure Relocations

Procedure relocations direct the linker to fixup a linkage pair of a procedure, which consists of the addresses of the procedure's invocation descriptor and entry point. Procedure relocations direct the linker to add the location of the procedure's invocation descriptor to either the contents of the specified longword or quadword or the contents minus the longword's or quadword's location. It further directs the linker to add the location of the procedure's entry point to either the contents of the next longword or quadword, or the contents minus the next longword's or quadword's location. This fixup allows a caller of a procedure to load the address of both the invocation descriptor and the entry point at once. The format of this relocation item is shown in Figure 1–24.

Figure 1–24: DRT Procedure Entry

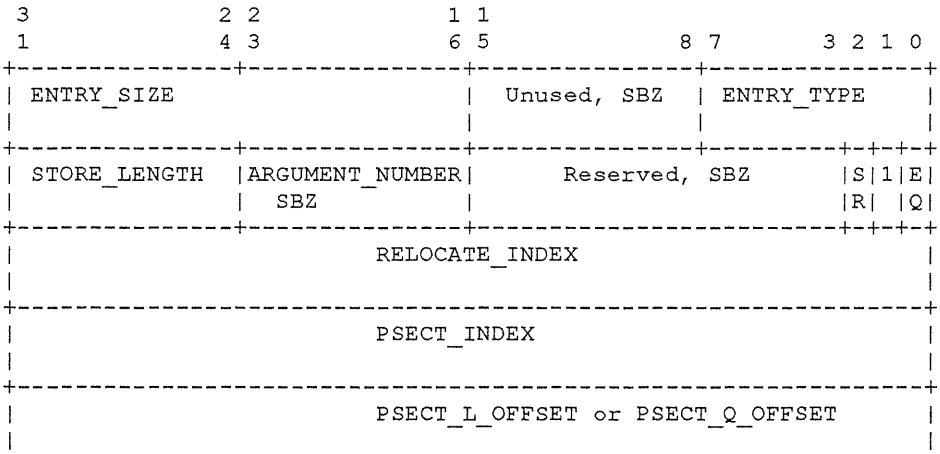
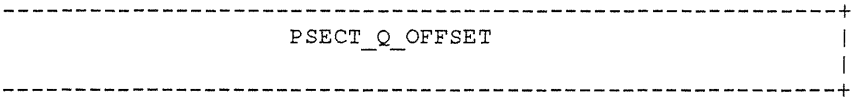


Figure 1–24 Cont'd. on next page

Figure 1–24 (Cont.): DRT Procedure Entry



These fields are defined as follows:

- ENTRY_TYPE is MODOBJ$C_RELOCATE_PROCEDURE.
- ENTRY_SIZE is 20 if the entry_quadword is clear, or 24 if it is set.
- ENTRY_QUADWORD (EQ) is set if the entry contains the quadword field PSECT_Q_OFFSET. ENTRY_QUADWORD is clear if the entry contains the longword field PSECT_L_OFFSET.
- SELF_RELATIVE (SR) is set if both the invocation descriptor fixup and the entry point fixup are self-relative to their own locations.
- STORE_LENGTH is either MODULE$C_LONGWORD (32-bit architecture) or MODULE$C_ QUADWORD (64-bit architecture) (see Section 1.3.4).
- RELOCATE_INDEX is the relative symbol number to use in the fixup.
- PSECT_INDEX is the PSECT number containing the location to be fixed up.
- PSECT_L_OFFSET or PSECT_Q_OFFSET is the relative byte offset within the PSECT identified by PSECT_INDEX for the current module to fix up.

1.4.2.3 FORTRAN String Argument Coercion

In FORTRAN, strings can be passed by descriptor or reference. The mechanism used for the arguments in a procedure is specified in the formal argument descriptors in the called procedure definition (found in the global symbol table of the called procedure module). The calling routine must include the proper data structures to pass the string either way, and the linker will make the proper choice.

\This mechanism is required because FORTRAN programs can accept string arguments by descriptor (declared as a string) or by reference (declared as a linear array of bytes). The FORTRAN compiler is unable to determine the correct passing mechanism when compiling references to such procedures.\

FORTRAN string relocation descriptors have the following format :

Figure 1–25: DRT FORTRAN String Entry

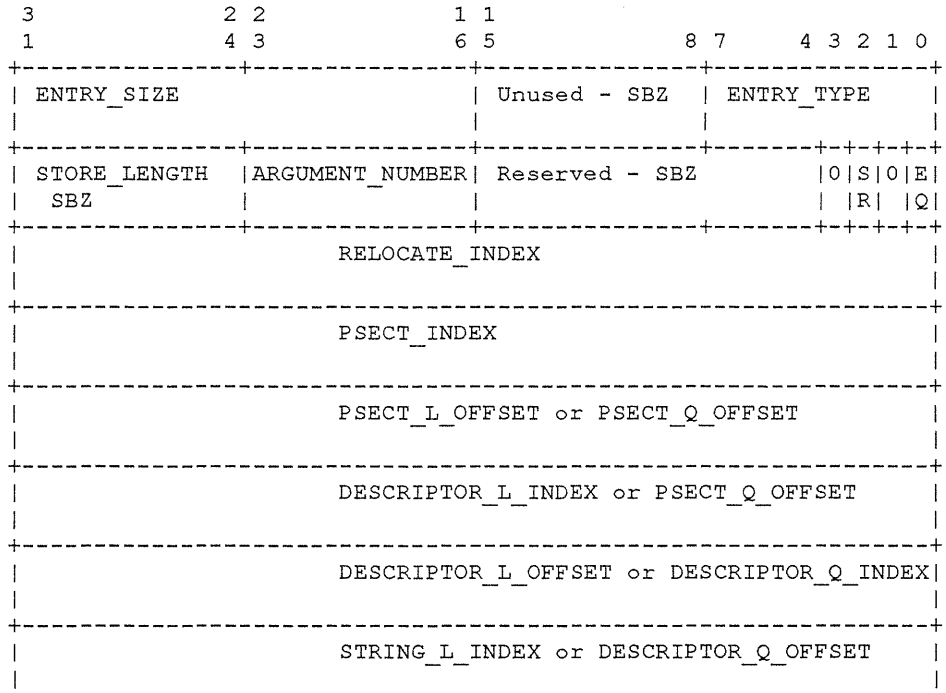
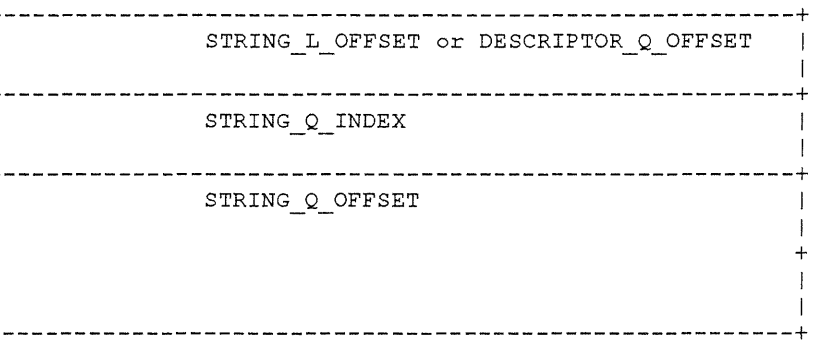


Figure 1–25 Cont'd. on next page

Figure 1–25 (Cont.): DRT FORTRAN String Entry

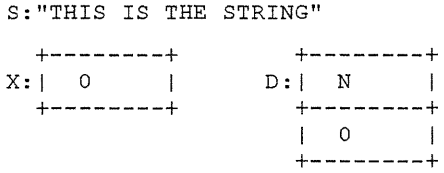


These fields are defined as follows:

- ENTRY_TYPE is MODOBJ$C_RELOCATE_ARGUMENT. The relocation item describes alternate methods of passing strings to FORTRAN.
- ENTRY_SIZE is 36 if the entry_quadword is clear, or 48 if it is set.
- ENTRY_QUADWORD (EQ) is set if the entry contains the fields with _Q_ in their names. ENTRY_QUADWORD is clear if the entry contains the fields with _L_ in their names.
- SELF_RELATIVE (SR) is set if the fixup is self-relative to the location being fixed. SELF_RELATIVE is clear if the fixup is absolute. For FORTRAN string coercion, this flag applies only to the value stored in the location described by PSECT_OFFSET and PSECT_INDEX.
- ARGUMENT_NUMBER is the positional value in the argument list of the string being passed. Arguments are numbered beginning with one.
- RELOCATE_INDEX is the relative symbol number of the procedure symbol reference in the calling module.
- PSECT_INDEX is the PSECT number containing the location to be fixed up.
- PSECT_L_OFFSET or PSECT_Q_OFFSET is the relative byte offset within the PSECT identified by PSECT_INDEX for the current module to fix up.
- DESCRIPTOR_L_INDEX or DESCRIPTOR_Q_INDEX is the PSECT number containing the full string descriptor for the string.
- DESCRIPTOR_L_OFFSET or DESCRIPTOR_Q_OFFSET is the relative byte offset within the PSECT identified by DESCRIPTOR_INDEX for the full string descriptor.
- STRING_L_INDEX or STRING_Q_INDEX is the PSECT number containing the string itself.
- STRING_L_OFFSET or STRING_Q_OFFSET is the relative byte offset within the PSECT identified by STRING_INDEX for the body of the string.

FORTRAN string relocation works as described in the following steps:

1. The compiler must set up data storage as follows:



2. The compiler generates a MODOBJ$C_RELOCATE_ARGUMENT data relocation describing location X in PSECT_INDEX and PSECT_OFFSET, location D (the string descriptor) in DESCRIPTOR_INDEX and DESCRIPTOR_OFFSET, and location S in STRING_INDEX and STRING_OFFSET.
3. The linker initializes the address pointer in D to point to the string, using the information in STRING_INDEX and STRING_OFFSET.
4. The linker examines the formal argument descriptor for the argument specified by ARGUMENT_NUMBER in the procedure described by RELOCATE_INDEX. If the formal argument specifies "pass by descriptor," the linker will initialize X to contain the address of D. If the formal argument specifies "pass by reference," the linker will initialize X to contain the address of the string constant S.
5. The calling routine then simply picks up the address contained in X, which contains the address of the full string descriptor or the address of the string constant S. Note that, in this case, the compiler must not generate a MODOBJ$C_RELOCATE_DATA fixup for X.

1.4.2.4 Store PSECT Size

The Store PSECT Size relocation directs the linker to add the size of the specified PSECT to the contents of a memory location within a data section. Store PSECT Size relocation descriptors have the following format:

Figure 1–26: DRT PSECT Size Entry

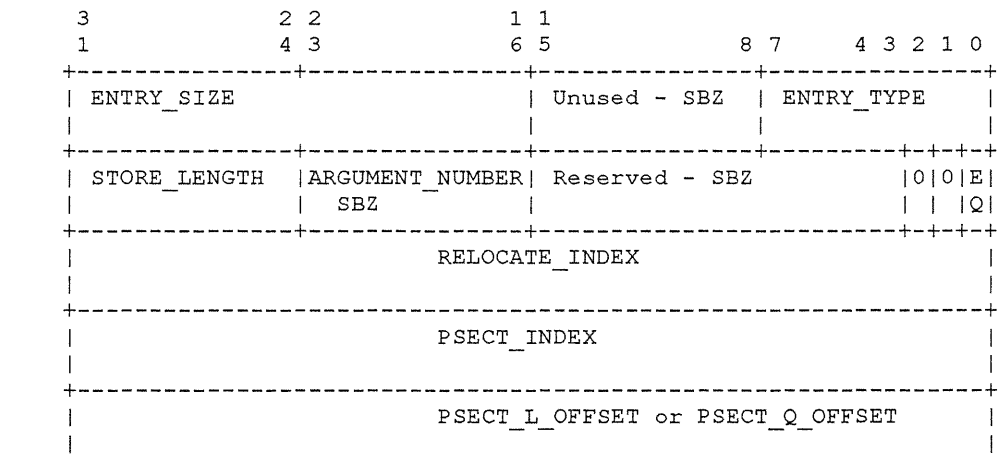
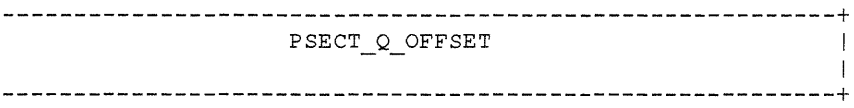


Figure 1–26 Cont'd. on next page

Figure 1–26 (Cont.): DRT PSECT Size Entry



These fields are defined as follows:

- ENTRY_TYPE is MODOBJ$C_RELOCATE_PSECT_SIZE. The relocation item is the size of a PSECT.
- ENTRY_SIZE is 20 if ENTRY_QUADWORD is clear, or 24 if it is set.
- ENTRY_QUADWORD (EQ) is set if the entry contains the quadword field PSECT_Q_OFFSET. ENTRY_QUADWORD is clear if the entry contains the longword field PSECT_L_OFFSET.
- STORE_LENGTH specifies the size of the value to be stored (see Section 1.3.4).
- RELOCATE_INDEX is the PSECT number whose size is to be stored.
- PSECT_INDEX is the PSECT number containing the size of the PSECT identified by INDEX.
- PSECT_L_OFFSET or PSECT_Q_OFFSET is the relative byte offset within the PSECT identified by PSECT_INDEX for the current module to fix up.

\Note that self-relative fixups are not allowed for this type of fixup.\

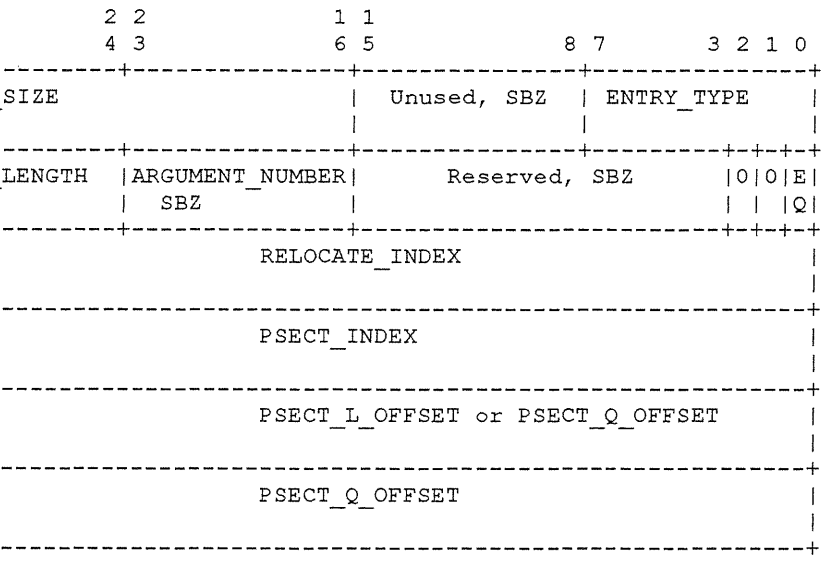
1.4.2.5 Store TLS Offset

The Store TLS (thread local storage) Offset relocation directs the linker to add the offset of the module's contribution to a thread local storage region to the contents of a memory location within a data section. This allows access to the module's portion of a concatenated TLS region. Store TLS Offset relocation descriptors have the following format:

Figure 1–27: DRT TLS Index Entry

Figure 1–27 Cont'd. on next page

Figure 1–27 (Cont.): DRT TLS Index Entry



These fields are defined as follows:

- ENTRY_TYPE is MODOBJ$C_RELOCATE_TLS_OFFSET.
- ENTRY_SIZE is 20 if the entry_quadword is clear, or 24 if it is set.
- ENTRY_QUADWORD (EQ) is set if the entry contains the quadword field PSECT_Q_OFFSET. ENTRY_QUADWORD is clear if the entry contains the longword field PSECT_L_OFFSET.
- STORE_LENGTH specifies the size of the value to be stored (see Section 1.3.4).
- RELOCATE_INDEX is the PSECT number of the TLS PSECT whose TLS offset is to be stored.
- PSECT_INDEX is the PSECT number of the location to be fixed up by the offset of the TLS PSECT identified by INDEX.
- PSECT_L_OFFSET or PSECT_Q_OFFSET is the relative byte offset within the PSECT identified by PSECT_INDEX to be fixed up.

\Note that self-relative fixups are not allowed for this type of fixup.\

1.5 Data Structures Specific to Image Files

Image files contain data structures that are not shared with object modules or other types of modules. These data structures provide information optimally formatted for the image activator, autoloader, and debugger. The image activator data structures are designed for the 32-bit PRISM architecture because programs that run on the 32-bit PRISM architecture cannot run on the 64-bit PRISM architecture without recompiling and relinking.

\The 64-bit image activator data structures will be specified later.\

1.5.1 Image Section Descriptor Table

Image section descriptors are used by Mica to load the various sections of code and data into memory. Image section descriptors are only present in image files and are aligned on longword boundaries. Because page protection is done on 64K-byte boundaries, the sections defined by image section descriptors must start on 64K-byte boundaries. The linker ensures 64K-byte boundary alignment.

\Demand zero compression will be adversely affected when a page is sparsely initialized because all memory up to the last initialized location must be present on disk.\

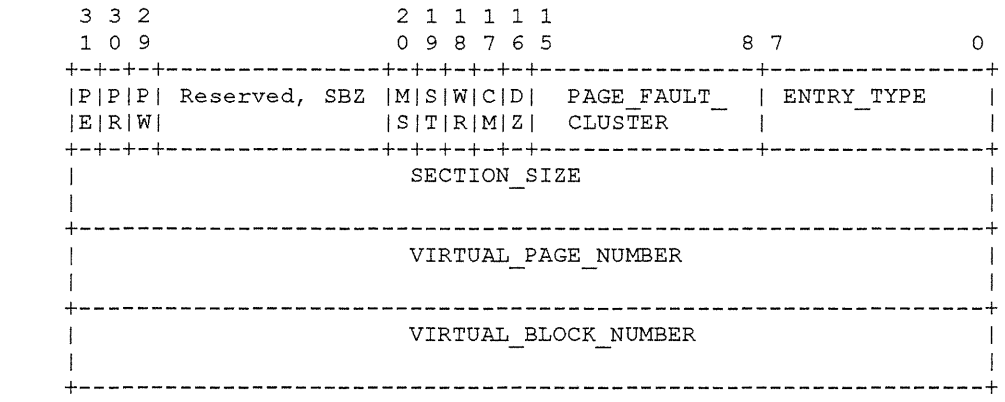
\A linker option will provide 8K-byte boundary alignment for linking the executive.\

The stack image section descriptor should be the last of the local image section descriptors to allow the image activator to allocate the stack's virtual address last. The linker will generate a default stack section of 128 pages, which can be changed by the user at link time.

The ISD (image section descriptor) table consists of an array of fixed-length entries, and each entry is quadword aligned. The declaration of the ISD entry and its format follows:

```
modimg$image_section_attribute : (
    modimg$c_isd_demand_zero,
    modimg$c_isd_copy_on_modify,
    modimg$c_isd_write,
    modimg$c_isd_stack,
    modimg$c_isd_message_section,
    modimg$c_isd_bit5,
    modimg$c_isd_bit6,
    modimg$c_isd_bit7,
    modimg$c_isd_bit8,
    modimg$c_isd_bit9,
    modimg$c_isd_bit10,
    modimg$c_isd_bit11,
    modimg$c_isd_bit12,
    modimg$c_isd_prot_read,
    modimg$c_isd_prot_write,
    modimg$c_isd_prot_execute
    );

! \BLISS uses prefix MODIMG$ISD_\
modimg$image_section_descriptor(
    entry_type : module$entry_type
    ) : RECORD
    CAPTURE entry_type;
    page_fault_cluster : unsigned_byte;
    attributes : SET modimg$image_section_attribute
        [..] SIZE(WORD);
    section_size : longword;
    virtual_page_number : longword;
    virtual_block_number : longword;
END RECORD;
```

Figure 1–28: Image Section Descriptor



These fields are defined as follows:

- ENTRY_TYPE contains zero.
- PAGE_FAULT_CLUSTER specifies the number of pages to be read for this image section when a page fault occurs.
- ATTRIBUTES specifies the attributes of the image section.
  - — MODIMG$C_ISD_DEMAND_ZERO is set if this section is demand zero. No pages are allocated in the module. VIRTUAL_BLOCK_NUMBER must be zero.
  - — MODIMG$C_ISD_COPY_ON_MODIFY is set if this section is copy on modify. All sections will have MODIMG$C_ISD_COPY_ON_MODIFY set, except for those sections containing PSECTs that are WRITABLE and SHAREABLE.
  - — MODIMG$C_ISD_WRITE is set if this section is writable. This is set if this section needs to be fixed up during image activation.
  - — MODIMG$C_ISD_STACK describes the stack. MODIMG$C_ISD_DEMAND_ZERO must also be set.
  - — MODIMG$C_ISD_MESSAGE_SECTION contains message section descriptors.
  - — MODIMG$C_ISD_PROT_WRITE is set if the owner has write access. A section with MODIMG$C_ISD_PROT_WRITE set and MODIMG$C_ISD_COPY_ON_MODIFY clear is a writable shareable section (created from PSECTs that are WRITABLE and SHAREABLE).
  - — MODIMG$C_ISD_PROT_READ is set if the owner has read access.
  - — MODIMG$C_ISD_PROT_EXECUTE is set if the owner has execute access.
- SECTION_SIZE specifies the number of 512-byte units in the section.

- VIRTUAL_PAGE_NUMBER specifies the starting relative virtual page number (in 8K-byte pages) for the section. The VPNs are specified relative to a zero-based image. The image activator must relocate these VPNs based on where the image is actually laid out in memory.

  \8K-byte is used because it is the smallest page size allowed in the PRISM architecture. The linker will ensure that each section's starting relative virtual page number is on a 64K-byte boundary for user mode images. The 64K-byte alignment may cause unexpectedly large image files because the linker cannot compress to demand zero uninitialized pages that are between initialized pages within a single section; for example, a 64 Kbyte array that has only the last byte initialized will require 64 Kbytes (128 blocks) of image file. On the other hand, a 128 Kbyte array with only the byte at offset 64K + 1 initialized will require only 512 bytes (1 block) of image file, because the first 64 Kbytes and the last 64 Kbytes can be compressed to demand zero sections.\

- VIRTUAL_BLOCK_NUMBER specifies the starting virtual block number (in 512-byte units) in the image file for the section.

1.5.2 Thread Local Storage Relocation Table

The TLS relocation table defines TLS index fixups (that is, a location that references a TLS region). The TLS-region-count fixup directs the image activator to add the count of TLS regions contributed by previously activated images to the specified locations. TLS regions are identified by their index into an array of addresses that are pointed to by a field in the TEB. This fixup allows TLS regions created in separately activated shareable images to have unique indexes.

The TLS relocation table is an array of longwords describing the memory locations to be fixed up. Each longword is a relative address from the beginning of the image to the target location. The image activator adds the number of TLS regions contributed by the previously activated images to the longword value at the memory location. The following is the declaration of the table:

```
modimg$relocate_tls_table(
        table_size : integer[0..]):
    ARRAY [1..table_size/4] OF longword;
```

1.5.3 Local Relocation Table

The local relocation table defines image-internal fixups (that is, a location that references another location within the same image). The local fixup directs the image activator to add the difference between the base address and expected base address to the specified locations. This fixup allows shareable images to be activated at any virtual address.

The local relocation table is an array of longwords that describe the memory locations to be fixed up. Each longword is a relative address from the beginning of the image to the target location. The image activator adds a correction value to the longword value at the memory location. The correction value is the actual base address of the image minus the expected base address of the image. If the actual base address is the same as the expected one, then local fixups are not done. The following is the declaration of the table:

```
modimg$relocate_local_table( table_size : integer[0..] ) :
    ARRAY [1..table_size/4] OF longword;
```

1.5.4 External Relocation Table

The external relocation table defines image-external fixups (that is, locations that reference a location within a different image). The external fixup directs the image activator to add to the specified locations the difference between the base address of the specified image and its expected base address. This fixup allows shareable images to share data with other shareable images, and any shareable image specified in this table must be specified in the immediate activation table. This fixup is not needed to call a procedure in another shareable image because that fixup is done dynamically by the autoloader (see Section 1.6).

The external relocation table is an array of longword pairs that describes the external image and the memory locations to be fixed up. The first longword in the pair is the byte offset of the external image's entry in the immediate activation table. Unlike other offsets, it is not from the beginning of the module, but from the beginning of the immediate activation table. The second longword is a relative address from the beginning of the image to the target location. The image activator adds a correction value to the longword value at the memory location. The correction value is the actual base address of the specified image minus its expected base address. The external relocation table is sorted by the image containing the target of the reference. The following is the declaration of the table:

```
! \BLISS uses prefix MODIMG$REE_\
modimg$relocate_external_entry : RECORD
    image_offset : longword;
    location_offset : longword;
    LAYOUT
        image_offset;
        location_offset;
    END LAYOUT;
END RECORD;

modimg$relocate_external_table( table_size : integer[0..] ) :
    ARRAY [1..table_size/8] OF modimg$relocate_external_entry;
```

1.5.5 Deferred Activation Table

The deferred activation table contains a list of images that can be activated on demand during the course of the image execution. These entries are the image descriptors referenced by the image autoload vectors. The following is the declaration and layout of an entry, which is quadword aligned:

```
modimg$activate_linkage(
    linkage_count : integer[0..]
    ) : ARRAY [1..linkage_count] OF RECORD
    linkage_pair_offset : longword;
    transfer_vector_index : longword;
    END RECORD;

! \BLISS uses prefix MODIMG$AE_\
modimg$activate_entry(
    linkage_table_count : integer[0..];
    image_name_size : module$string_size
    ) : RECORD
    CAPTURE linkage_table_count, image_name_size;
    base_address : longword;
    image_match : module$match_record;
    expected_address : longword;
    image_name : string( image_name_size );
    linkage_table : modimg$activate_linkage( linkage_table_count );
    END RECORD;
```

Figure 1–29: Activation Table Entry

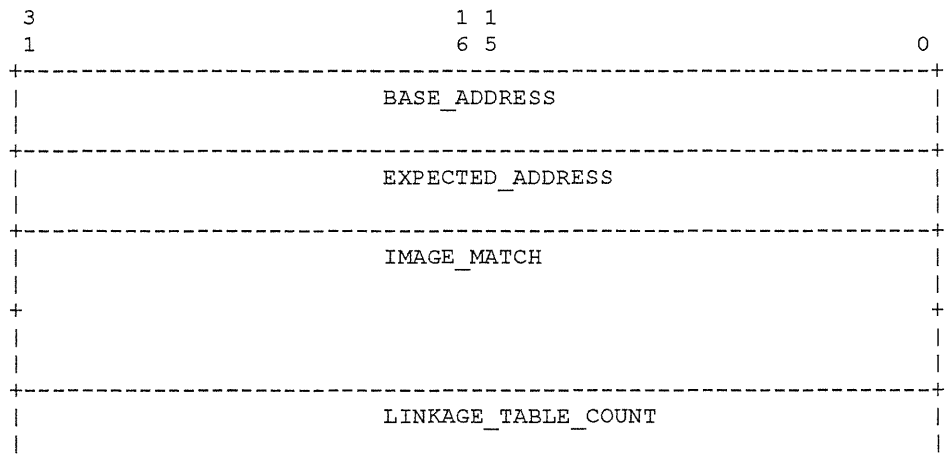
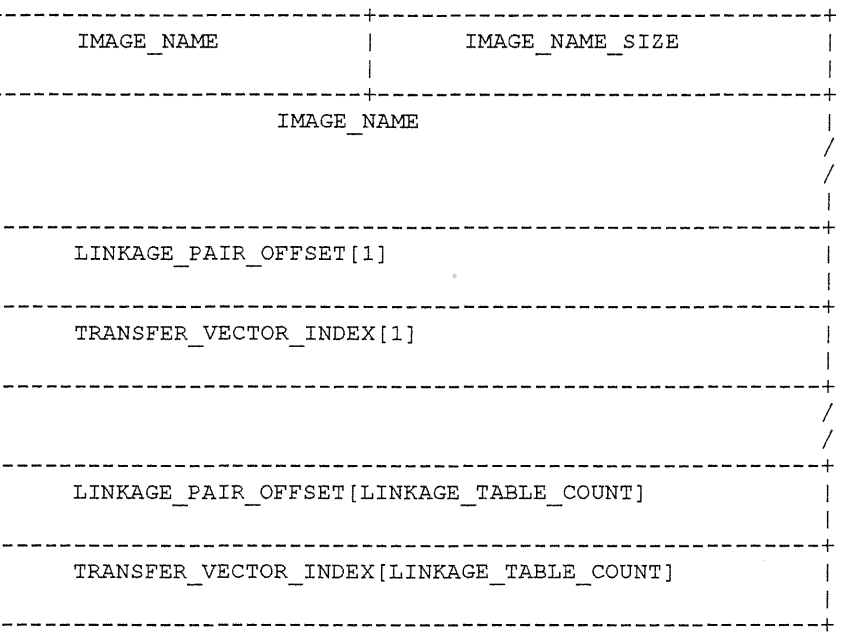


Figure 1–29 Cont'd. on next page

Figure 1–29 (Cont.): Activation Table Entry



- BASE_ADDRESS is the base address of the image that contains this entry. This address is required for autoloading and is fixed up by the image activator, if necessary.
- EXPECTED_ADDRESS is the optimal load address for the described image (which is the base address the linker has used in preassigning virtual addresses).
- IMAGE_MATCH is the value at link time of the described image's match control record.
- LINKAGE_TABLE_COUNT is the number of entries in the linkage table.
- IMAGE_NAME_SIZE is the size of the image name.
- IMAGE_NAME is the name of the image (LBRSHR, LIBRTL, and so on).
- LINKAGE_TABLE contains the information needed to fixup all linkage pairs that refer to routines in the described image.
  - — LINKAGE_PAIR_OFFSET is the offset (in bytes) from the base address of the image that contains this entry to a linkage pair that references a routine in the described image.
  - — TRANSFER_VECTOR_INDEX is the index of the routine in the described image's transfer vector table (see Section 1.5.7).

1.5.6 Immediate Activation Table

The immediate activation table contains a list of images that are to be automatically activated when the containing image is activated. It has the same format as the deferred activation table (see Section 1.5.5.)

1.5.7 Transfer Vector Table

The transfer vector table provides a level of indirection between the caller's shareable image and the called shareable image. The transfer vector table is an array of longwords containing the offsets from the beginning of the mapped file to the invocation descriptor for the routine it represents. Transfer vector zero will contain a count of the number of transfer vectors in the image, and therefore an image offset of zero is illegal.

The transfer vector table is read by the autoloader when it fixes up a shareable image dynamically. The transfer vector table is placed in the image header because the image header is always mapped anyway and because the table needs to be found before any symbol fixups can be done. The linker places the transfer vector table at the constant offset MODIMG$C_TRANSFER_VECTOR_OFFSET from the beginning of the image file to optimize the dynamic fixup of procedures. The declaration and layout of the transfer vector table follows:

```
! \BLISS uses prefix MODIMG$TVT_\
modimg$transfer_vector_table(
    transfer_count : integer[0..]
    ) : RECORD
    CAPTURE transfer_count;
    transfer_vector : ARRAY [1..transfer_count]
        OF longword;
END RECORD;
```

Figure 1–30: Transfer Vector Table

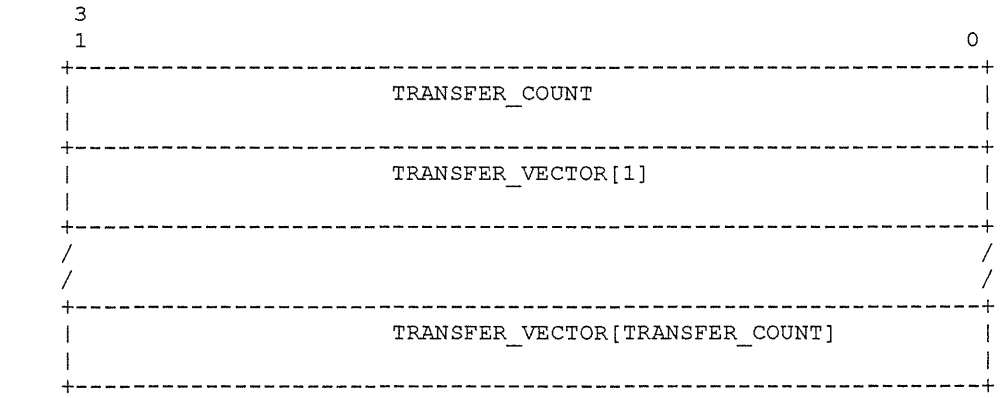


Figure 1–30 Cont'd. on next page

Figure 1–30 (Cont.): Transfer Vector Table

- TRANSFER_VECTOR_COUNT is the number of transfer vectors in the transfer vector table. It is equal to the number of procedures within the shareable image that are accessible from other images.
- TRANSFER_VECTOR is the offset in bytes from the beginning of the image to a routine's invocation descriptor.

1.5.8 Initialization Routine Table

The initialization routines of an image are called at image activation. Initialization routines within an image are called in order with no parameters. (See Chapter 30, Linker for details on the ordering.) There is no ordering between initialization routines of separate images. The initial routine table is an array of longwords that contains the offset from the beginning of the image to the invocation descriptor's of the initialization routines. The declaration of the table is:

```
modimg$initial_routine_table( table_size : integer[0..] ) :
    ARRAY [1..table_size/4] OF longword;
```

1.5.9 Debug Module Table

The debug module table is built by the linker and is used by DEBUG. This table contains a list of modules in the image and, for each module, the PSECTs and their base addresses.

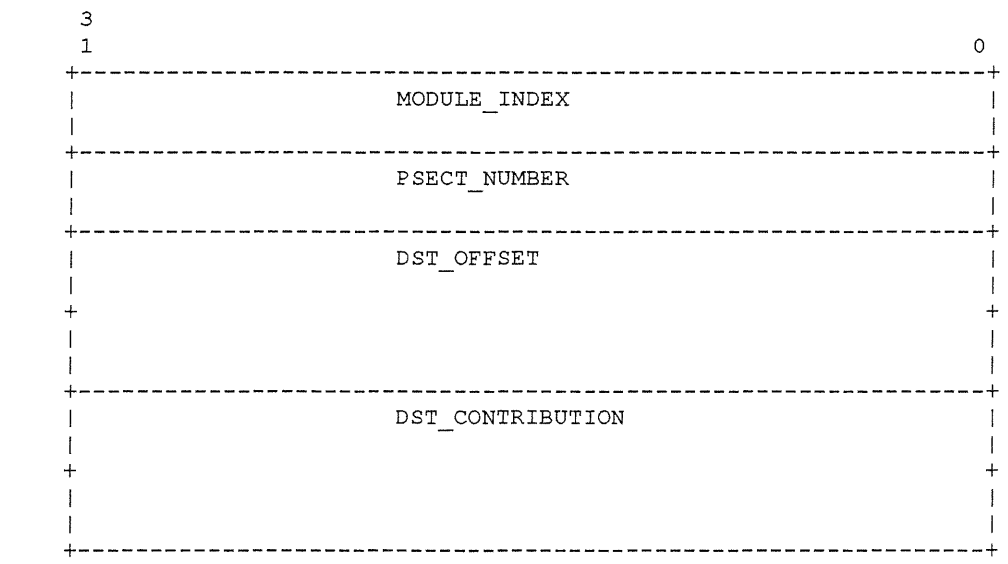
The declaration and layout for a debug module table entry is:

```
! \BLISS uses prefix MODIMG$DME_\
modimg$debug_module_entry( psect_number : integer[0..] ) : RECORD
    CAPTURE psect_number;
    module_index : longword;
    dst_offset : quadword;
    dst_size : quadword;
    entry_item : ARRAY [1..psect_number] OF modimg$debug_module_item;
    END RECORD;
```

Figure 1–31: Debug Module Table Entry

Figure 1–31 Cont'd. on next page

Figure 1–31 (Cont.): Debug Module Table Entry



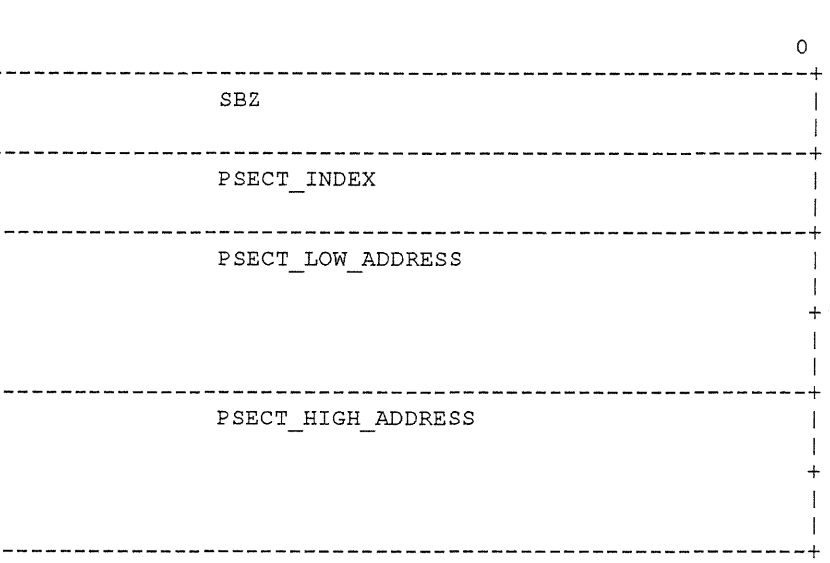
Following the header for a module is a triplet of quadwords for each PSECT within the module. This triplet contains the PSECT index, and address range for the PSECT in the following format:

```
! \BLISS uses prefix MODIMG$DMI_\
modimg$debug_module_item : RECORD
    psect_index : longword;
    psect_low_address : quadword;
    psect_high_address : quadword;
    END RECORD;
```

Figure 1–32: Debug Module Table Entry Item

Figure 1–32 Cont'd. on next page

Figure 1–32 (Cont.): Debug Module Table Entry Item



The length of the DMT for each module is determined by the count of triplets in PSECT_NUMBER field.

1.6 Linker

The following sections discuss how the linker operates on the contents of object modules and shareable images in order to produce PRISM image files. This information is provided in this chapter to aid the reader in understanding the object and image formats; it is not intended to provide complete information on the linker's operation. See Chapter 30, Linker for a complete description.

1.6.1 Symbol References

Symbol references are generated when a source module refers to a symbol that is not found in the current module. The symbol can be either absolute or relocatable, and either data or procedure. In the following sections, only relocatable symbols are treated. The linker handles symbol references differently, depending on whether the symbol is a procedure or data, and depending on whether an object module or a shareable image defines the symbol.

1.6.1.1 Building an Executable Image with Object Modules

When the linker is building an executable image, it assigns a base address to the image. Hence, the virtual addresses of all global symbols found in object modules are known at link time. The linker can simply fill in the longword with the correct virtual address.

1.6.1.2 Building a Shareable Image with Object Modules

When the linker is building a shareable image, it does not know where the shareable image will be placed in virtual memory at run time. To resolve these symbol references, the linker substitutes its best guess for the symbol's value, and then generates an entry in the local relocation table for the image under construction. This entry directs the image activator in fixing up the correct location in a data section at image activation.

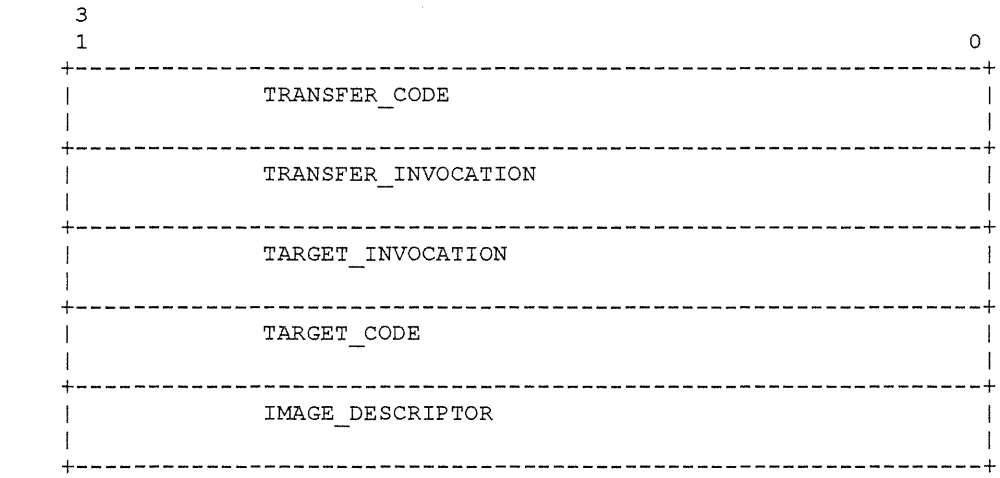
1.6.1.3 Resolving Procedure Symbols from Shareable Images

If a procedure symbol is found in a shareable image, the linker performs the following actions:

1. The linker enters the shareable image into the deferred activation table if it is not already in an activation table.
2. The linker generates an image autoload vector in the image header. This autoload vector is quadword aligned and has the following declaration and format:

```
! \BLISS uses prefix MODIMG$AV_\
modimg$autoload_vector : RECORD
    transfer_code : POINTER anytype;
    transfer_invocation : POINTER anytype;
    target_invocation : POINTER anytype;
    target_code : POINTER anytype;
    image_descriptor : POINTER modimg$activate_entry;
    END RECORD;
```

Figure 1–33: Image Autoload Vector



These fields are defined as follows:

- TRANSFER_CODE contains the virtual address of the transfer code. The transfer code is code generated by the linker to transfer the routine to either the autoloader or the target routine:

```
LDQ     8(R10), R4          ;Load addresses for autoloader
OR      R4, R0, R10         ;Move autoload vector address
JSR     R0, (R5)            ;Jump to autoloader/routine
```

\The PRISM calling standard specifies registers R4 and R5 as registers that can be overwritten between the calling procedure and the called procedure.\

Note that if the image being generated is a shareable image, this location may need to be fixed up at image activation.

- TRANSFER_INVOCATION contains the address of the transfer routine's invocation descriptor. The invocation descriptor is needed only for the requirements of the calling standard. It is not actually referenced by the transfer code. Note that if the image being generated is a shareable image, this location may need to be fixed up at image activation.
- TARGET_INVOCATION initially contains the address of its own autoload vector. Eventually, it will contain the address of the invocation descriptor for the called routine. Note that if the image being generated is a shareable image, this location may need to be fixed up at image activation.
- TARGET_CODE initially contains the address of the autoloader code. Eventually, it will contain the address of the entry point for the called routine. Note that if the image being generated is a shareable image, this location may need to be fixed up at image activation.
- IMAGE_DESCRIPTOR is a self-relative pointer that points to the entry in the current image's immediate or deferred activation table of the called routine's shareable image.

3. The linker uses the location of the autoload vector and the location of the transfer code to fix up all references to a routine in the autoloaded image. (All references to the routines in the autoloaded image are listed in the linkage table in its image descriptor table entry [see Section 1.5.5]).
4. The program begins execution. The first call to the routine in the shareable image results in a call to the transfer code. The autoload routine finds the image descriptor for the image. If the image has not been loaded, it loads the image into memory and fixes up all references to the image. Additionally, TARGET_INVOCATION and TARGET_CODE are modified to contain the addresses of the routine's invocation descriptor and entry point. This allows subsequent calls to routines in the image to work even if they use stale invocation descriptors and entry points.

1.6.1.4 Resolving Data Symbols from Shareable Images

If a data symbol is found in a shareable image, the linker performs the following actions:

1. The linker enters that shareable image in the immediate activation table if not already entered (and removes it from the deferred activation table if it was entered there)
2. The linker calculates the expected value of the symbol, using the image's optimal load address and stores it at the required location.
3. The linker generates an entry in the external relocation table for the image under construction. This entry directs the image activator in fixing up the correct location in a data section at image activation.

1.6.2 Overlaid PSECT References

In general, PSECT references are only generated by compilers when referencing PSECTs that are overlaid (for example, FORTRAN COMMON-like PSECTs). The treatment of overlaid PSECTs depends upon whether a shareable image is built and whether a shareable image defines the overlaid PSECT.

1.6.2.1 Building an Executable Image with Object Modules

When the linker is building an executable image, it assigns a base address to the image. Hence, the virtual addresses of all PSECTs in the image are known at link time. The linker can simply fill in the longword with the correct virtual address.

1.6.2.2 Building a Shareable Image with Object Modules

When the linker is building a shareable image, it does not know where the shareable image will be placed in virtual memory at run time. To resolve these PSECT references, the linker generates an entry in the local relocation table for the image under construction. This entry directs the image activator in fixing up the correct location in a data section at image activation.

1.6.2.3 Referencing an Overlaid PSECT in a Shareable Image

If the PSECT is found in a shareable image, the linker generates an entry in the external relocation table, directing the image activator to make the correct fixup at image activation. The shareable image that contains the overlaid PSECT is activated at the same time as the image that references it. References to a PSECT with a greater allocation than the size of the overlaid PSECT in the shareable image are errors.

1.6.3 Virtual Address Preassignment for Shareable Images

In order to reduce the number of fixups that must actually be done at image activation, the linker will "prefixup" the locations, using a best guess as to where the image will be laid out at run time.

\For images installed and permanently activated, this will turn out to be a big win, especially if the mechanism for installing them also allows specification of the address at which the image is to be installed. It is more difficult to predict the benefit of miscellaneous user shareable images. Clearly, the image activator can try to put images where the linker thought they would be (in virtual address space).\

1.6.4 Image Header Mapping

The linker will always create an image section descriptor that maps the image header as read only in the user's virtual address space. The header is mapped to allow the image activator, autoloader, and possibly the debugger to access the image file's data structures in a uniform manner. It also avoids the overhead of another channel to the image file when the header is read, since the pager already has the file open.

1.7 Open Issues

Some open issues:

- The debug symbol table must be more fully specified by SDT.
- The design of TLS regions is under review.
- The syntax for linker directives is dependent upon the Mica command line parser.
- The design of the entity consistency check table is subject to further requirements from:
  - — PILLAR
  - — PASCAL
  - — IPSE
- Image and object files are not sparse. Demand zero sections are described by demand zero image section descriptors; this complicates image activation somewhat, because the image activator must create demand zero prototype page tables entries for demand zero section descriptors. The lack of sparse files also causes much larger files to be created for sparsely initialized data. However, this eliminates the dependency for sparse file support in both VMS and in Mica.

GLOSSARY

activated image: An image file that has been laid out in the address space of a process. All relocations and fixups have been performed, and control can be transferred to defined entry points within the image.

atomic name: A name in the module name table that is not qualified by another name.

autoloader: A routine supplied with the MICA system that performs the dynamic activation of shareable images at run time.

autoload routine: See *autoloader*.

autoload vector: An autoload vector contains the information needed by the transfer routine to dispatch to either the autoloader or the target routine. It also contains a self-relative pointer to the information needed by the autoloader to fixup the target routine's image.

calling standard: See *PRISM calling standard.*

code section: A section containing all the executable code for a module. It is directly generated by the compiler and is not modified by the linker, except to combine like-named PSECT (program section) contributions into image sections.

composite object module: A module created as the result of merging multiple object modules into a single object module; when this is done, all intermodule relationships are resolved, PSECTs are concatenated, and a new symbol table is generated.

compound name: A name in the module name table that is qualified by other names. Compound names provide a means for languages to implement multiple name spaces in a way supported by both the linker and librarian.

data relocation table: A table describing all fixups that must be performed by the linker to the data and linkage sections of the module, based on program section addresses.

data section: A section containing all the data defined in the module. Some of this data is read only and some is read/write. This section also contains the linkage ($LINK) section and all entry descriptors for routines defined in the module.

debug symbol table: A symbol table built by a compiler containing sufficient information for the debugger to interpret user commands and display memory contents in "the current programming language."

dynamic activation: Delaying the activation of an image (into memory) until it is actually referenced.

executable image: An image produced by the linker, with a base address assigned to the image (value TBD). Executable images must have a transfer address or the linker generates a warning at link time.

fix-up: An action taken by the linker to alter an image so that it becomes memory-ready.

global symbol: A symbol (value or location) defined in one object module, whose value is made available by the linker to other object modules.

global symbol table: A table describing symbols defined or referenced in a module. The global symbol table parallels the module name table. That is, programs must walk both tables at the same time to obtain all the attributes of an element in the global symbol table.

image: A file resulting from linking several object modules together. PSECTs are gathered into image sections, and there are no unresolved external references.

image activator: The part of the system responsible for loading image files into memory and preparing them for execution.

image autoload vector: See *autoload vector*.

image fix-up: See *fix-up*.

image relocation tables: A relocation table within an image describing how memory locations within the data section are fixed up once the image has been activated. The linker generates relocation tables for symbols defined within the image, symbols defined in other images, and TLS region counts.

image section: A collection of PSECTs with like protection attributes, found only in images.

invocation descriptor: A quadword-aligned data structure that provides basic information about a routine. This structure is used in calls between separately compiled routines, and in interpreting the call stack that exists at any point in the execution of an image. Entry descriptors are defined by the PRISM calling standard.

linkage pair: A linkage pair consists of the addresses of a procedure's invocation descriptor and entry point.

linkage ($LINK) section: The portion of the module data section that contains pointers to data. The linkage section is generated by the compiler, and address relocations to this section are performed by the linker, using information in the address relocation table. The linkage section must not be shareable, as it contains process-private addresses.

loader: The part of the system responsible for loading object modules into memory, resolving external references, and preparing object modules for execution. The loader may be implemented as part of the image activator.

memory-ready: Ready to be loaded into memory. A memory-ready image is one requiring no fix ups.

module: A file, containing names and related information, that conforms to the described module format.

module header: The first record in a module. All information in a module can be located directly or indirectly through information in the module header.

module name table: A table containing the names of all symbols and PSECTs defined or referenced in the module. It contains both atomic and compound names. Entries in the module name table correspond one-to-one with entries in the global symbol table.

object module: The output of a compiler, a single module generated from the source language.

PRISM calling standard: The standard sequence used to call a routine. The PRISM calling standard is defined in the *PRISM Calling Standard*.

From: BECALM::TLE::GROVE 7-MAR-1988 16:03
To: DECWET::PETERSON,DECWET::SCHREIBER,DECWET::KIMURA
Subj: PSECT data proposal - 26-Feb better than 3-March

References:
1. Proposed PSECT change ... from Gary 26-Feb
2. New idea from Darryl, Gary,... from Kim 3-March

Kim -

I favor the approach to Psect data outlined in Gary's 26-Feb note, but I am STRONGLY OPPOSED to the new format described in your 3-March memo. The aspect that I don't like is separating the data from the PSD table entry. My objections are based on the following:

1. It seems to me that we are getting carried away with levels of indirection here. The object format is already awkward to produce. Let's not add more levels of indirection and chicken wire!

2. It's not apparent to me that there is much gain in Linker processing. Since all of the data is separated in the PDT, there is no need for the linker to scan it at all during Pass 1. All the information that the Linker needs to allocate virtual memory is contained in the Psect definition in the GST.

3. The 26-Feb version of the PDT makes the structure of the PDT and the DRT more similar.

4. In your 3-March proposal, it seems to me that the actual data is "naked" in the object module. My understanding of the original object module design is that there is a sort of tree-structured hierarchy, consisting of the module header, tables pointed to from the header, and item lists. All of the "leaves" are described by one of these structures. This is important, because the object module format is supposed to be extensible, and becuase it allows language processors to include items that are not really understood by the linker. So all the leaf data should be encapsulated in a wrapper (e.g. item list or table) whose structure (starting point and length at least) are understood.

So, it seems to me you need yet another wrapper around the data. Let's just use the 26-Feb version.

5. In the 3 March version, I think you would need a flag in the PSD entry to indicate whether the format is "image" or "compressed". Depending on "compressed" to be smaller doesn't seem reliable.

In summary, I liked the PDT of 26-Feb, and I am strongly opposed to the 3-March version.

Please let me know what you think.
Thanks
Rich Grove

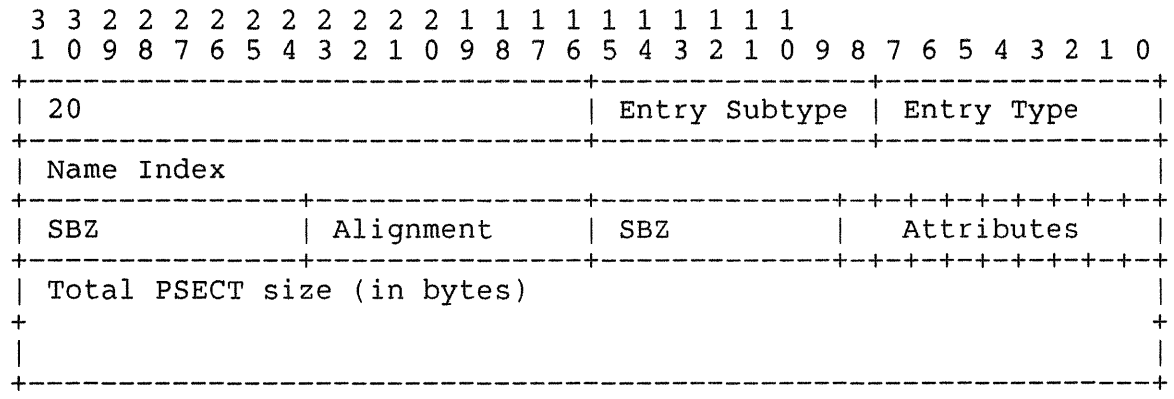
From: TLE::DECWET::PETERSON "Kim Peterson, DECwest Engineering 03-Mar-1988 13
To: @[.WORK]CALLING_STANDARD
Subj: New idea from Darryl, Gary, and myself

Darryl, Gary, and myself believe we have come up with a better counter proposal for treating PSECTs in a PRISM Module. We still will move denoting the data a PSECT out of the GST. But instead of having the new PSECT Data Table contain variable sized records, each with a header followed by data, we will follow the Image Section Descriptor (ISD) table paradigm.

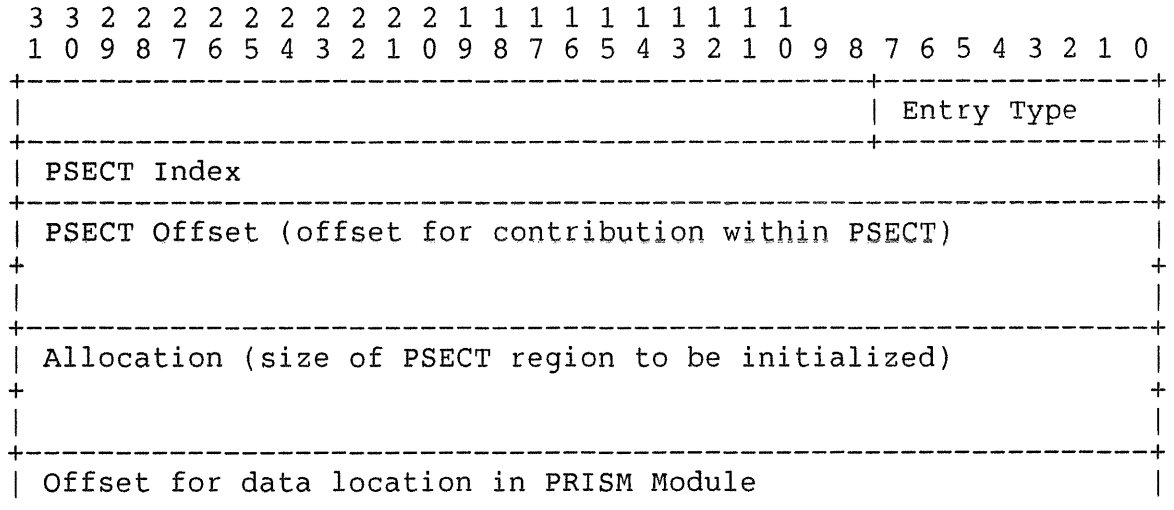
Again we would like to have GEM's opinion on this proposed change.

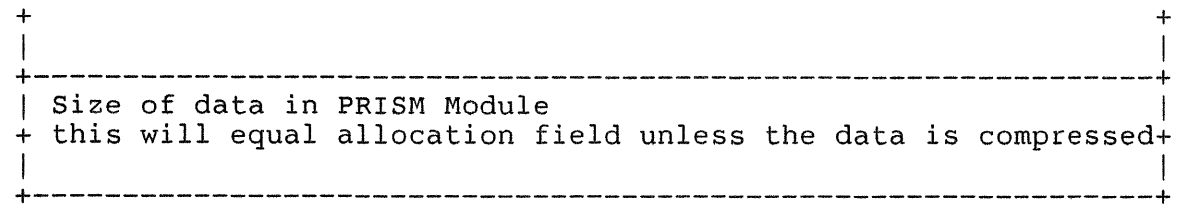
This new method is sufficient to completely replace the old form of PSECT definitions, unlike the previous proposal.

In this new proposal the GST contains only one PSECT definition record per PSECT. The old GST PSECT definition entry will still be eliminated. This new record defines the PSECT's name (via a name index), its type (UNINITIALIZED or INITIALIZED), its attributes, alignment, and overall size. It will not be used to denote the data within the PSECT. The STANDARD_OFFSET, COMPRESSED_OFFSET, and COMPRESSED_SIZE fields are eliminated. A PSECT definition record would then look like:



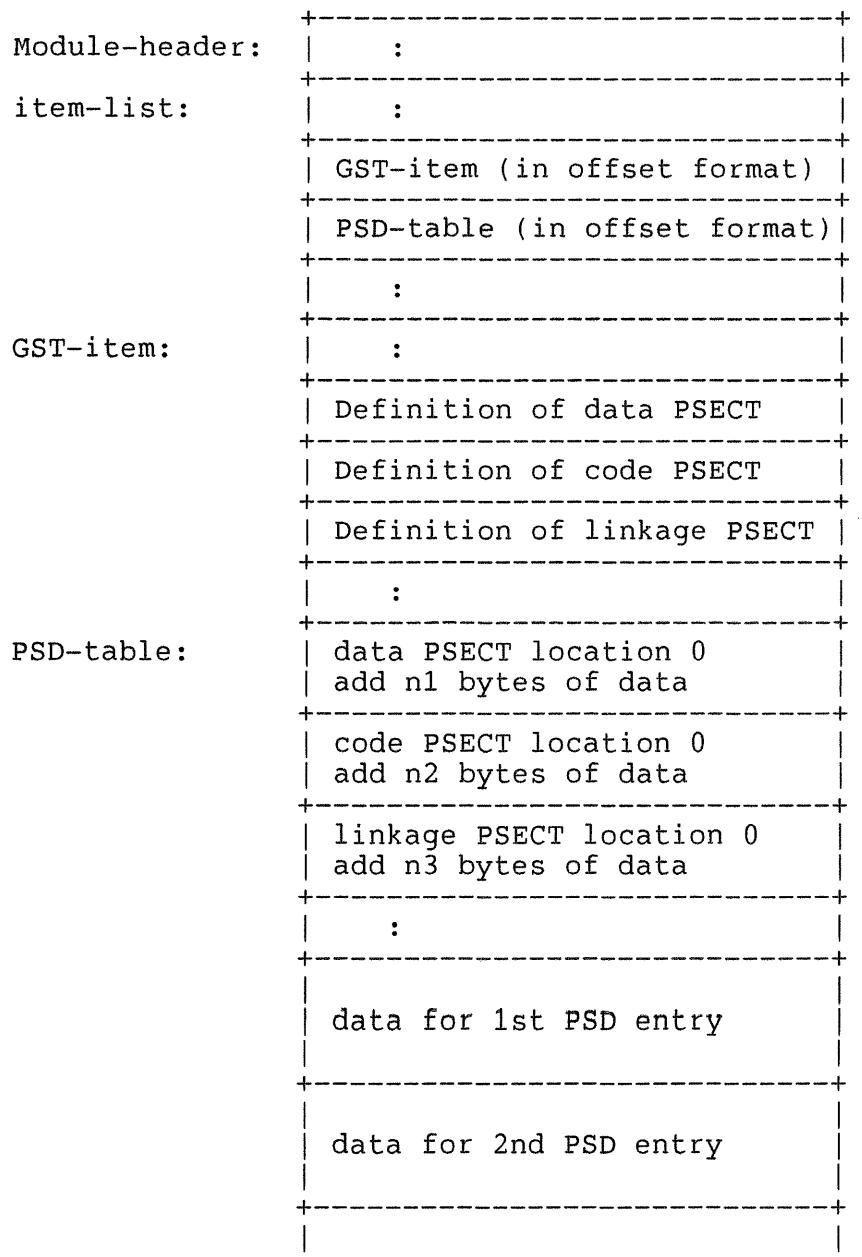
The data within a PSECT will be specified within a new item called a Program Section Descriptor (PSD) table. The PSD table will contain a list of fixed sized records, each record will identify a PSECT, an offset within the PSECT, the size of the data contribution, and a pointer to the data along with its stored size. The record will also indicate whether the data is in standard or compressed form. A PSD record would then look like:

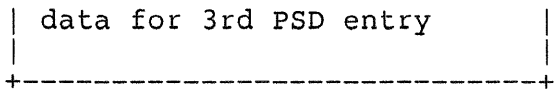




We will still identify PSECTs similar to the current PRISM Module format. Each PSECT, as it is defined in the GST, will be assigned a PSECT index. This index will be used in PSD entries to identify a specific PSECT.

An example of how this will work is, assume a compiler is trying to write an object module. In the GST it will insert three PSECT definition records for the data, code, and linkage sections. Each will be assigned a PSECT index. To assign data to a PSECT the compiler would construct a PSD table and add a PSD records for each "large" chunk of data. A sample PRISM module might look like:





In this figure I've left off the Module name table and some of the other items. Note that this figure is different from the previous proposal's figure in that the data for each PSECT is moved out of the PSECT table.

Here are the advantages we see with this newer scheme.

1. It unifies the treatment of PSECT data with Image data.

2. The PRISM Module I/O Package will be able to easily handle PSD and ISD tables. We can do this by simply extending the data types used by the PRISM Module I/O Package to include ISDs and PSDs. Which implies that everyone using the shell to read or write an PRISM Module can probably benefit from using this package.

3. It can completely replace the GST old PSECT definition records.

4. It does not slow down the linker. In the previous proposal the linker was going to scan the entire PSECT data table twice, once to collect the header information for each contribution (this involves skipping over the data) and a second time to read the data. This new proposal eliminates having to reread the PSECT's data twice.

5. It still provides a simple eloquent model of a PRISM Module. We believe that technically this is a cleaner better design.

Cost of any change (good or bad) cannot be ignored. We believe that the costs of doing this change are:

1. It will add time to the linker schedule and all projects dependent upon the linker. The additional time is a couple of days.

2. It will eliminate time from the compiler schedule because one package will be available and tested for reading and writing PRISM modules that can be used by everyone using the compiler shell.

If GEM agrees with the technical merits of this proposal we believe the cost is worth the benefits and it should be adopted.

From: BECALM::TLE::DECWET::PETERSON "Kim Peterson, DECwest Engineering 11-Mar
To: @[.WORK]CALLING_STANDARD
Subj: If we are agreed, I'll put this in the chapter.

```
---------------
| | | | | | | |
|d|i|g|i|t|a|l|        I N T E R O F F I C E  M E M O R A N D U M
| | | | | | | |
---------------
```

TO: @Calling Standard

DATE: 24-Feb-1988
FROM: Kim Peterson
DEPT: DECwest Engineering
EXT: 206-865-8704
LOC: ZSO
ENET: DECWET::PETERSON

CC: DECwest Distribution
John Bishop
Gerald Sacks

Digital Equipment Corporation -- Confidential and Proprietary

SUBJECT: ECO to the Object Module/Image File Format Chapter

REFERENCES:
1. Proposed PSECT change mail message from Gary Kimura, 26-Feb-1988

The following is the proposed changes to the Object Module and Image File Format caused by adding the PSECT Data Table proposed in Gary's note. These changes fall under two categories: the definition of the PSECT data table and changes to the global symbol table.

1 PSECT DATA TABLE

The data within a PSECT will be specified within a new item called a PSECT Data Table (PDT). The PDT will contain a list of entries, and each entry contains an index to a PSECT, an offset with in the PSECT, and data contribution to the PSECT in either uninterpreted or compressed form. If the PDT entry contains compressed data, the entry also contains the size the data should expand to.

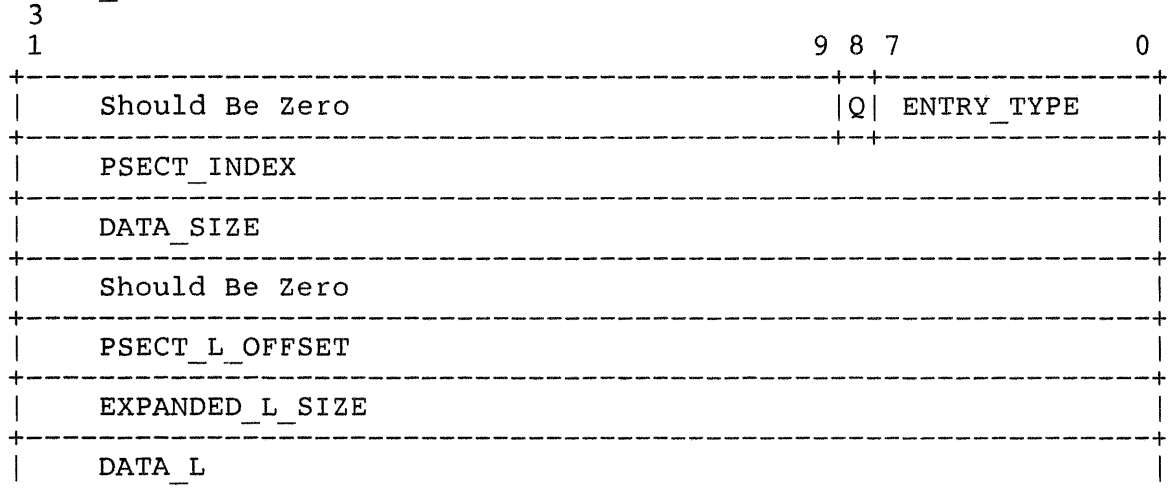
The Global Symbol Table (GST) still contains PSECT definition records which defines the PSECT's name (via a name index), its attributes, alignment, and overall size. It will not be used to denote the data within the PSECT. Each PSECT should only have one PSECT definition in the GST, but there can be multiple PDT entries for a PSECT.

As before, an index is associated with each PSECT definition in the GST. This index is used in the PDT to identify PSECTs. (This index continues to be used in the GST and data relocation table to identify PSECTs as well.)

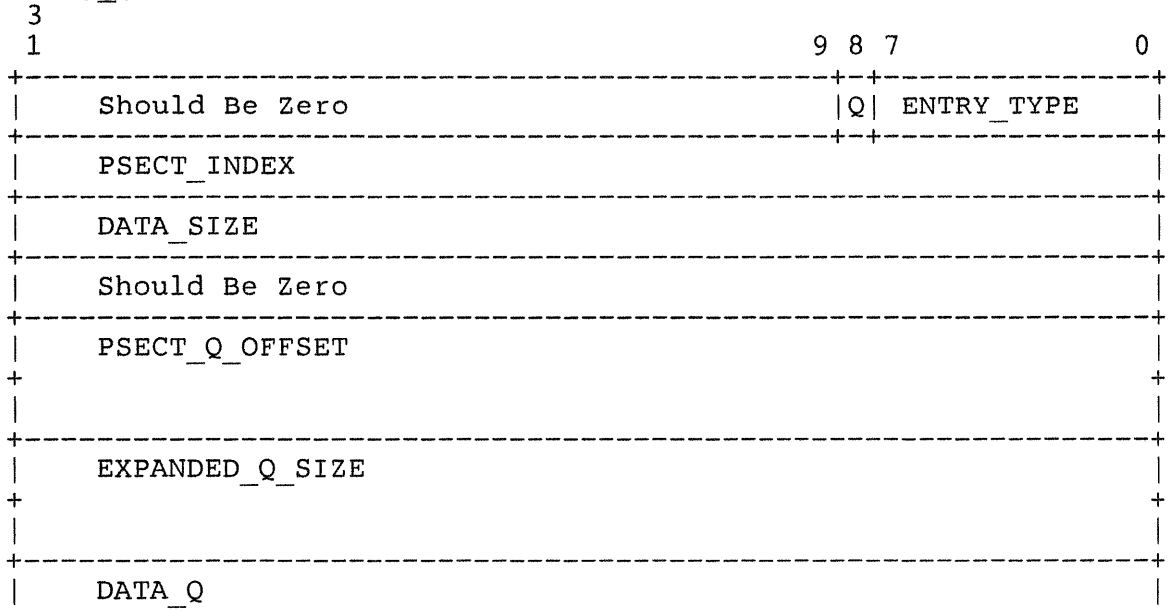
The PDT has an item code of MODOBJ$C_PSECT_DATA. Each entry in the PDT is quadword aligned, and has the following declarations:

```
modobj$psect_data_entry(     ! BLISS prefix modobj$pde_
    entry_type : modobj$psect_data_type[..] SIZE(byte);
    entry_quadword : bit;
    data_size : integer[0..] ): RECORD
    CAPTURE entry_type, entry_quadword, data_size;
    psect_index : longword;
    VARIANTS CASE entry_quadword
        WHEN false THEN
            psect_l_offset : longword;
            expanded_l_size : longword;
            data_l : byte_data(data_size);
        WHEN true THEN
            psect_q_offset : quadword;
            expanded_q_size : quadword;
            data_q : byte_data(data_size);
    END VARIANTS;
END RECORD;

modobj$psect_data_type : (
    modobj$c_data_expanded
    modobj$c_data_compressed,
    );
```

entry_quadword (Q) is clear



entry_quadword (Q) is set



Note that the data field is always quadword aligned. Note that data_size is the size of the data field, and that the size of the entry is either ((data_size+24+7)/8)*8 or ((data_size+32+7)/8)*8.

2 GLOBAL SYMBOL TABLE CHANGES

The MODOBJ$C_PSECT_COMPRESSED variant of the MODOBJ$PSECT_DEFINITION record is eliminated.

The MODOBJ$C_PSECT_STANDARD variant of the MODOBJ$PSECT_DEFINITION record is renamed to MODOBJ$C_PSECT_IMAGE and is restricted to image files.

The MODOBJ$C_PSECT_UNINITIALIZED variant of the MODOBJ$PSECT_DEFINITION record is renamed to MODOBJ$C_PSECT_STANDARD, but is otherwise unchanged. Note that all parts of a non-TLS PSECT not initialized through the PDT are zeroed by the linker.

From: TLE::DECWET::KIMURA "Gary D. Kimura - DECwest Engineering 26-Feb-1988 1
To: @CALLSTD,KIMURA
Subj: Proposed PSECT change within PRISM Modules

We believe (i.e., Don, Darryl, and myself) that the treatment of PSECTs in the current and purposed PRISM Module format can be simplified and improved. This improvement involves treating PSECTs similar in fashion to how the Data Relocation Table is currently treated.

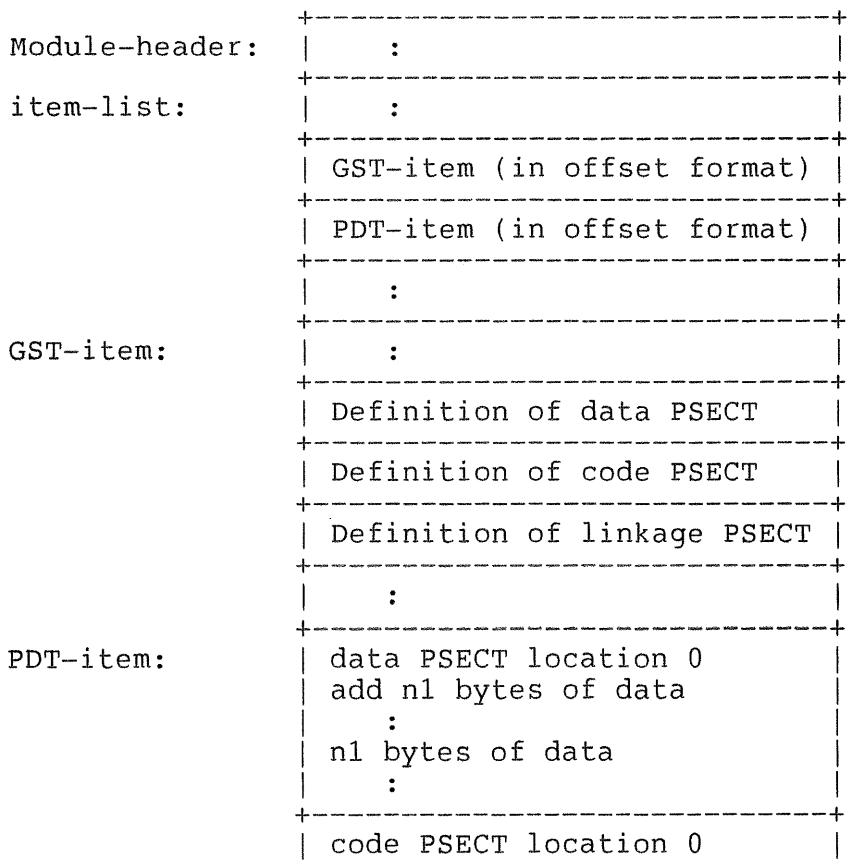
We would like a quick opinion from the GEM project on this proposed change.

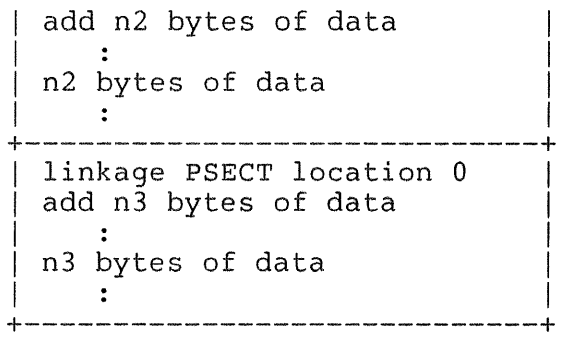
In this new scenario the GST contains only one PSECT definition record per PSECT. The old GST PSECT definition entry will be eliminated. This new record defines the PSECT's name (via a name index), its attributes, alignment, and overall size. It will not be used to denote the data within the PSECT.

The data within a PSECT will be specified within a new item called a PSECT Data Table (PDT). The PDT will contain a list of records, each record will identify a PSECT, an offset with in the PSECT, and its data contribution to the PSECT in either uninterpreted or compressed form.

We will still identify PSECTs similar to the current PRISM Module format. Each PSECT, as it is defined in the GST, will be assigned a PSECT index. This index will be used in PDT entries to identify a specific PSECT.

An example of how this will work is, assume a compiler is trying to write an object module. In the GST it will insert three PSECT definition records for the data, code, and linkage sections. Each will be assigned a PSECT index. To assign data to a PSECT the compiler would construct a PDT and add new records for each "large" chunk of data. A sample PRISM module might look like:





In this figure I've left off the Module name table and some of the other items.

The advantages that we see with this approach are:

1. It provides for a more consistent and logical treatment of items (now including PSECTs) in a PRISM Module.

2. Because it's a more consistent and logical layout the PRISM Module I/O Package provided by the Compiler Shell will be simpler, more efficient, and easier to use.

3. The DECwest compiler will not be using the old PSECT definition method. Instead we will emit only one GST entry for each PSECT. This should then fix the size of the GST before we need to emit the PSECT data, so if need be the GST can be moved ahead of the PSECT data in the PRISM Module.

From: TLE::GROVE 26-FEB-1988 13:27
To: NYLANDER
Subj: my 2 cents to Gary

From: TLE::GROVE 26-FEB-1988 13:19
To: GROVE
CC:
Subj: cc of comments to Gary

After a quick read thru your note, I like it.
I will circulate it around here to the GEM and BLISS folks
and see what their reaction is.

I would expect that GEM too would only have one contribution per module
for most psects. Although we have been planning to put out code on
a routine-by-routine basis in multiple contributions.

More comments early next week.

Rich

program section: See *PSECT*.

PSECT: Program section. PSECTs describe a contiguous piece of memory. With concatenated PSECTs, all contributions for a particular PSECT are gathered contiguously in memory. If the PSECT is overlaid, all contributions for a particular PSECT begin at the same virtual address, and the module that has the largest contribution to a PSECT defines the length of the PSECT.

shareable image: A special form of executable image that contains a global symbol table and can be input to the linker in subsequent linking operations.

transfer address: The address of an invocation descriptor in an executable image that is called when the image is run.

transfer code: The code generated by the linker that transfers a call to a routine in another image to the autoloader.

transfer vector: The offset from the beginning of a mapped shareable image to the invocation descriptor of the routine it represents.

thread-local storage: See *TLS*.

TLS: Thread-local storage. TLS is per-thread storage with FORTRAN COMMON semantics, and storage allocated at run time.

VBN: Virtual block number. VBNs are 512-byte entities on the disk. If the size of the virtual blocks of the on-disk structure changes, software must convert 512-byte VBN numbers to the new values, which should be a multiple of 512 bytes.

virtual block number: See *VBN*.

INDEX

M

N

O

P

R

S

T

U

V

W

From: WILBUR::DON "Don MacLaren -- DECwest" 8-FEB-1986 15:30
To: TLE::GROVE,TLE::NYLANDER,TLE::DON,@SOFTWARE
Subj: Object Language and Symbol Tables

P/VMS object modules will be quite different from those in VMS. It's too early to define the object language in detail, but some context is needed in order to begin the DST design, DSTs being a part of the object language. This note is my current guess as to some characteristics of the object language. As a design sketch, it reflects my interest in

- Unifying various structures into a coherent whole and
- Promoting sharing of things like symbol tables

1 OBJECT MODULES -- GENERAL

A "simple object module" is the normal output of a compilation. A simple module is obtained by compiling one Pillar source module, or its equivalent in another language. A "composite object module" is a set of simple modules bound together by a compiler or linker.

Simple- and composite- modules have the same structure. Either can be an input to the linker for further binding; either can be loaded and executed, provided the necessary environment is in place. The linking/binding that takes place during loading is similar to what happens now during image activation.

An object module may contain quite a variety of components:

1. Code Psects.
2. Data Psects.
3. A relocation table a giving relocation information for some longwords in the data psects.
4. A global symbol table declaring global symbols defined or referenced in this module.
5. An exported symbol table containing information about exported names. This may be used in compilation of other modules or by the debugger, etc. This is needed for Pillar and may well be used by other languages.
6. A build table containing information about the source files and modules used in compiling this module. This information can be used by a program management system. The signature information used for module consistency checking in Pillar probably goes here.

7. A cross reference table.

8. A profile table containing information needed to interpret an execution profile to be obtained by executing this module.

9. Universal optimization table. Information provided for or by the universal optimizer.

10. Debug Symbol Table.

At this stage in the design, we should consider "containment in an object module" as a logical notion. It's not required that the components all be in the same file. On the other hand, it's nice if commands such as COPY and LIB operate on the entire thing.

Note: if all relevant components are in an object module, a complete listing could be generated from it -- more complete than anything we generate now. The listing could serve either as the .LIS output of a compiler or the .MAP output of a linker.

2 CODE PSECTS.

A PRISM code Psect is pure code that can be placed at any properly aligned memory location. ANAL/OBJ can detect any violations.

A code Psect in a composite module may be the concatenation of Psects from simple modules. The structure of this can be determined by examining the module's symbol table. Psect origins are symbols.

3 DATA PSECTS.

A PRISM data Psect can be placed at any memory address. It has memory management attributes such as read-write, copy on modify, etc. An unusually interesting question is

does the Psect have relocation information?

Psects containing linkage sections and their overflows do have relocation information. It's best if others do not.

Note that a linkage section contains entry-description information, and the form of this is specified by the calling standard. However the layout of an entire linkage section is arbitrary; it can be distributed over various Psects. The entry descriptors can be found by consulting the object module's symbol tables.

4 RELOCATION TABLE

This table defines how some of the longwords in the module's data

Psects are to be "relocated" during loading in accordance with symbol values not known when the module was bound (i.e., when it was compiled or linked). Most relocation is simply the addition of a symbol's address value to the longword's contents. Rather general values can be specified using symbol expressions (explained below), which replace the stack machine language of the VMS object language. However this will be rare in P/VMS; "keep it simple" is the motto.

Speed of name binding and relocation during loading is a P/VMS priority item, and there are many interesting possibilities for acheiving it. We may want a fancy data structure here for any production module. For the moment, I am assuming that the compilers will put out a good but straightforward structure. If appropriate, the linker can generate something fancier; and, as a consequence, it may be reasonable to link a single object module against nothing.

5 PROFILE TABLE

Compiling a module with /PROFILE generates code in such a way that counts will be accumulated for basic blocks or some other stuctured unit of code. The counts are elements of a static array. The profile table relates that array to

1. the source code as seen by the user, e.g. to the THEN block in an IF statement

2. the IL code used for optimization and code generation in the compiler

I suppose that either or both relationships might be in the table depending on the exact /PROFILE option used.

6 THE DEBUG SYMBOL TABLE

This has at least three components

1. PC Correlation Table.

2. Code Explanation.

3. DST Table.

This relates code locations to the source module structure, i.e. to source lines, block structure, etc.

A code location in this table is relative to a Psect origin. The binding that occurs during linking or loading does not require any modification to this table.

It might be nice, and also efficient, if the table were designed around of the typical nested structure in languages, in effect using something like END to terminate the range of an inline expansion, include file, or whatever.

It should be able to identify things like the prolouge of a routine.

(I've lost track of the structure used on VMS).

6.1 Code Explanation

Highly optimized PRISM code will not be easy to read and there is probably no hope of describing all of a symbol's life span in the DST table. The code explanation helps by revealing some simple facts

1. The source text location that generated an instruction.
2. The hard to see flow graph information, helpful for trying to find the source of an imprecise exception, the source of a value, etc.
3. The symbol, if any, associated with a register operand.
4. The symbol, if any, that's the source/target for a load/store.

A nice display of object code can be based on this.

Linking or loading does not require any modification to this table.

6.2 DST Table

The DST table for a simple module contains type-, location-, and value- information about symbols declared in the source module -- no doubt other information also.

Linking or loading does not require any modification to this table. The DST table is a tree whose leaves are the DST tables for simple modules.

A DST table is more complicated than the other symbol tables in an object module, and there is a great deal of variation across langauges. Nevertheless, unification is possible along the following lines.

1. The symbol tables are nested.
   - The exported symbol table can reference the global symbol table. It may elaborate on the definition of a global symbol, but it doesn't repeat anything.

- The DST table can reference the global table or the exported table, and may elaborate on definitions in them.

2. A common language is used in all tables for definitions and references to values and locations. The global- and exported- tables use only a subset of this language, and there will be escapes, but almost everything is handled in a uniform fashion.

3. A common data type language is used as far as possible. We catalog most of the data types in use. A symbol's data type is then given as a cataloged type with the possible but unlikely addition of language specific information.

7 OBJECT SYMBOL LANGUAGE

This is the common language used to define and reference symbols in the object-module symbol tables. It has escapes so that strange things can be managed without having to extend the "standard" language.

I'm using the term "symbol" in a general sort of way to mean something that needs to be defined one place and referenced in other places. A symbol may have a name: global name, exported name, name within a block, or it can be strictly internal such as a compiler temporary generated to hold the location of a dynamically allocated variable.

Examples of symbols:

- A global data item with storage
- A global constant value. (In VAX MACRO "X == 27" defines X as a global value equal to 27).
- One of the standard register symbols.
- The origin of a Psect.
- An entry-descriptor symbol.
- A variable declared in a block.

7.1 Symbol Properties

The properties of symbols need to be classified in some way so that definitions can be broken down into pieces. A symbol need not have all of these properties, and the properties it does have need not be given in single definition.

1. Symbol name.

2. Symbol scope -- how it fits, if at all, into the standard name-lookup structure.

3. Symbol value. Note: definition of the space of values and it's type structure has to be part of the symbol language, but it can be relatively simple because type information in user terms is separate.

4. Symbol location.

5. Symbol type -- data type.

6. Symbol class -- what sort of symbol is it.

   - o Value. The symbol directly denotes a value defined somehow in the total symbol table structure. Such a symbol cannot have a location.

   - o Data item. The symbol denotes a data item existing at run time. Such a symbol has (usually) a location and a value. The value may only be obtainable at runtime.

   - o Others such as Psect, entry descriptor and label. It's always hard to figure out this list.

7. Misc.

7.2 Symbol Expressions

A symbol expression denotes a value or location. (Actually we may want to take a more general view, but values and locations are the main thing.) Symbol expressions are in reverse Polish. A terminal operand is a literal, a symbol value, or a symbol location. (The two have to be distinguished for sanity). Literals are normally integers, maybe that's the only possibility.

Whatever escapes are in the language, the form is such that an expression can be scanned without understanding the escape.

The typical uses of a symbol expression are to define the location or value of a symbol or to define an extent value in a data type. In these contexts it is easy to have short forms for some common expressions, e.g. a literal, or a symbol's value or location.

7.3 Parametric Symbols

A symbol may be parametric. When a reference to such a symbol is

interpreted, its binding is determined by the "current environment". Here the environment is something that's constantly maintained by any program that's trying to interpret a symbol table. While coding you hardly notice this -- once you get used to it. Examples:

1. The length of a string or decimal data type is a parametric symbol.

2. The origin of a Psect is a parametric symbol.

3. A register symbol is parametric. It's binding depending on the call frame stack.

Parametric symbol expressions can be evaluated in an environment that binds only some of the symbols. The result is another parametric symbol expression.

7.4 On-Disk Vs. In-Memory Structure, Sharing

As in VMS the symbol language, and the object language in general, describes the structure of an object module in the file system. When a symbol table is "loaded", the structure in memory depends on whose doing the loading. For example, the compiler will load an exported symbol table one way, the debugger another.

The loading process can add detail and bind things, e.g. it can replace symbol indices by pointers. This is advantageous, but I think the debugger should load symbol tables in such a way that they are sharable in different address spaces: use relative pointers within a single table (or component of a table) and keep the parametric structure for global references.

From: TLE::NYLANDER "Chip 29-Apr-1986 2159" 29-APR-1986 22:00
To: PUDDLE::SCHREIBER,NYLANDER
Subj: RE: Preliminary object language spec

Benn,

Comments on the "P.VMS Object and Image File Format" follow. Thanks for the opportunity to preview it.

To Ultrix or not to Ultrix?

The document seems a little muddled about whether it is describing the Ultrix object and image file formats or not.

1. If the document describes file formats that are intended for Ultrix as well as P.VMS, you can globally replace most instances of "P.VMS" in the document with "PRISM" and have a document that more accurately reflects that intent.

   Last I heard, everyone (Don, Ray, etc.) felt that PRISM/VMS and PRISM/ULTRIX should and would have compatible object formats, calling standard, etc.

2. The module format (pages 3 - 4) and the IHDL_CODE_xxx, IHDL_DATA_xxx and IHD$L_UDATA_xxx fields in the module header are straight out UNIX.

   The concept of image sections, PSECTS, and demand-zero sections that can be scattered in the object module are straight out of VMS.

   They don't always mix very well.

   The document does not make clear whether, for example, PSECT definitions with GPS$L_VBN = 0 are allowed in the section of the object file described by IHD$L_DATA_xxx (which would make the "uninitialized data section" non-contiguous if IHD$L_UDATA_BYTES was non-zero).

   My bias is to get rid of the UNIX-like properties of the object and image file formats, unless Ray has technical requirements from the Ultrix debugger, profiler, etc. for such UNIX-like properties.

3. Are executable images re-linkable? Don originally had a concept of "simple" object modules (e.g. compiler output) and "composite" object modules (e.g. linker output). "Composite" object modules could be used as input to another link operation.

   Note that UNIX works this way - an executable image can be input to a subsequent link, as long as the relocation information has not been stripped out.

Tool-related information

We might want to discuss the inclusion of such information as the Build Table,

Cross Reference Table, Profile Table, and Universal Optimization Table in the object file.

Reason is, for good or for bad, SDT is defining a "Tools Integration Architecture" to get all the programming and development tools to play together. The idea is to port it to PRISM someday.

The model that holds it all together is that everything produced at various stages of development and programming is an "element"; source files, object code, executable images, profiling results, cross-reference tables, etc. are all elements. Performing processing on an element to produce a new element (compiling a module, tracing a program's execution, doing a cref, etc.) is a transformation of one element into another.
There is supposed to be a master "element database" that keeps track of each element and the path of transformations that produced it.

Now, if one believes all this, then it is necessry to keep elements where neither their substance nor their addressability by the element database will get lost when a transformation occurs.

So there might be a problem keeping certain tool-related info in the object file, if it can't be addressed by the element database if or the addressability is lost when a transformation (like a link) occurs.

I'm not sure this is a significant issue, but I thought it would be worth mentioning in case you want to address issues like that at this stage in the game.

NITS

Page 3: The PRISM calling standard will be defined seperately from the PILLAR reference manual. It'll be a seperate document, maintained seperately, possibly part of a "software architecture notebook".

(At least, that's the last word I heard from Don).

Page 6: The purpose of the IHD$V_PSMULT field is not clear. Perhaps you could elaborate.

Page 8: The intent of the ISD$V_WRITE comment about FORTRAN COMMON isn't clear. FORTRAN COMMON is (on VMS) implemented with global overlayed writable PSECTs (GBL+OVR+WRT). Is the intent of the comment that it should apply to overlaid writable PSECTs in general?

Likewise, the ISD$V_IMMEDIATE_ACTIVATE comment about FORTRAN BLOCK DATA is not clear.

Perhaps you could also explain in terms of PSECT and symbol attributes instead of language constructs?

The NOTE on page 9 also.

Page 9: Since the page size is processor-implementation-dependant, and page protection boundries should always be on 64Kb boundries, why not always interpret ISD$L_VPN as 64Kb pages (VPN 0, 1, 2, etc.)?

Page 15: You could unify the format of global symbol definition subrecords with that of global symbol reference subrecords at the cost of 16 reserved bits in each subrecord occurance.

Page 21: The auto-load of a shareable image only works in this scheme if the reference to the symbol defined in the shareable image is a procedure call. Data references will not work correctly by this mechanism.

Is the ISD$V_IMMEDIATE_ACTIVATE image section attribute intended to make auto-loading of sharable images with global non-procedure symbols unnecessry?

Page 22: I'm concerned about the performance of procedure calls to auto-loaded shareable images. If I read the text right, it will take several extra instructions each time a procedure in an auto-loaded shareable image is called. (Fetch IAV$L_ROUTINE from the dummy entry descriptor, examine it, jump to the target routine).

The software architecture is going to some trouble to get rid of even unnecessary single instructions on procedure calls. (PILLAR, for example, will not load the argument-count register unless the interface is explicitly declared as a public interface for which the argument count must be loaded. Last I heard, we were reserving a register for the RTL base address to make RTL calls faster).

Several extra instructions for each procedure call seems excessive. Am I missing something?

Another PSECT Attribute?

The PSECT attributes look about right.

However, we defined another PSECT attribute for the "VMS/ULTRIX" (if you'll excuse the term) object language when we ported VAX FORTRAN and the VMS LINKER to Ultrix.

The UNMOD attribute tells the linker that a data PSECT, which is not necessarily demand-zero, has no non-zero static data in it, and can therefore be placed in the BSS ("uninitialized data") image section. This allows the linker to produce smaller image files on Ultrix.

(We believe that it would be a win for this attribute to be supported on VMS as well, since it would allow the linker to reduce the number of image sections and speed up image activation, but that's another story.)

I commend it for your consideration. Paul Winalski's write-up of it follows:

From: BABEL::WINALSKI 8-AUG-1985 15:50
To: NYLANDER
Subj: New PSECT Attribute Bit in GSD PSC Object Records for Ultrix

To: VFU Developers From: Paul S. Winalski

Dept: Technical Languages and Environments
DTN: 381-2022
MS: ZKO2-3/N30
NET: EIFFEL::WINALSKI

Subject: New FLAGS Bit in GSD PSECT Definitions

This document describes a change to the VAX Object Language Global Symbol Definition records processed by the VAX Ultrix Linker (LK). This change does not affect the VAX/VMS Linker at this time. The VMS Linker ignores the new bit flag definition.

On VMS, the VMS Linker attempts to reduce final image size by creating demand zero (DZERO) image sections. The Ultrix object language has a similar feature called the BSS section (also known as the uninitialized data segment). The difference is that, while VMS DZERO image sections may occur anywhere in an image, components of the BSS section must occur in a contiguous block at the end of the image. The Ultrix object language has a way for compilers to direct the linker to define a specified amount of space in the BSS section. Unfortunately, the VMS Object Language has no similar feature. The Linker does not know that a PSECT is uninitialized until after it runs the TIR stack machine in pass 2. By then, it is too late to move the uninitialized PSECTs to the end of the image so that they can be in the BSS section.

The upshot of all this is that when programs compiled by our ported compilers are linked using LK, there is a risk that the image will be substantially larger than the same program either linked on VMS or compiled using the native Ultrix compilers. To get around the problem, it is proposed that there be a new PSECT attribute to allow compilers to inform the Linker that a PSECT is never initialized, and therefore can be in the BSS section.

The new PSECT attribute bit occurs in the Program Definition Subrecord (type GSD$C_PSC) of the Object Language GSD Record (see section 6.3.1 of the VAX/VMS Linker Reference Manual, AA-Z420A-TE). In field GPS$W_FLAGS, bit 10 (formerly reserved) is now GPS$V_UNMOD. If GPS$V_UNMOD is zero, the PSECT may be stored into and therefore cannot be allocated in BSS. IF GPS$V_UNMOD is one, the PSECT is never stored into and therefore can be allocated in BSS. Having the sense of the bit defined this way allows LK to process existing VMS objects correctly.

NOTE

> The VMS Group, who own the VAX/VMS Linker, have neither reviewed nor approved this change. The VMS Linker currently ignores GPS$V_UNMOD. Nonetheless, only Ultrix compilers should set this bit until the VMS Group sanction its use in VMS compilers.

When LK encounters a PSC subrecord, it records the settings of all of the PSECT attribute bits in its internal PSECT tables. If other PSECTs of the same name are overlayed on or concatenated to this PSECT, the values of GPS$V_UNMOD from all contributions will be ANDed to obtain the final value. Thus, if any contribution to the PSECT is stored into, GPS$V_UNMOD will be zero and LK will allocate the entire PSECT in the DATA segment instead of in BSS.

Compilers for Ultrix are urged to take advantage of setting GPS$V_UNMOD to allocate data in BSS wherever this is possible, to minimize the final image size.

[End of Document]

From: TLE::DECWET::DON "Don MacLaren -- DECwest 06-Oct-1987 1434" 6-OCT-1987 20:39
To: PETERSON,KIMURA,PALMER,@CALLSTD
Subj: Case Sensistivity In Object Language

I agree with the attached proposal by Steve Hobbs. I think we should also consider the possible value of a bit to say that a reference or definition comes from a case-insensitive language.

- Don

From: TLE::HOBBS 5-OCT-1987 13:58
To: DECWET::PETERSON
CC: GROVE,WINALSKI,LAGASSE,HOBBS
Subj: RE: Please scan these changes for anything adverse Thanks

Subj: Case Insensitive Names in Object Files

+---------------+
! d i g i t a l ! I n t e r o f f i c e M e m o r a n d u m
+---------------+

To: Kim Peterson

Date: 2 October 1987
From: Steven Hobbs
Dept: Technical Languages
Ext.: 381-2066 Loc.: ZKO2-3/N30

CC: Rich Grove, Chip Nylander, Paul Winalski

Your recent mail on Object/Image file format for Prism mentioned that case insensitive names must be capitalized. I believe that compatibility between case sensitive and case insensitive software would be improved if the convention were that case insensitive names must be down cased.

The usual way to write case sensitive names in C programs and in Ultrix is to use exclusively lower case. In particular, all of the important Ultrix system services and library routines have all lower case names. If the case insensitive languages (such as FORTRAN, Pascal and Ada) were to use lower case external symbols then these languages could access these Ultrix entry point without needing special name escape conventions. Special name escape conventions would only be necessary to access a C or Ultrix name containing an upper case letter (and upper case is a very rare occurrence in either C or Ultrix). The only other case sensitive language we have is Modula-2 but Modula-2 is not frequently used on Digital machines so compatibility with C and Ultrix is more important.

Since it is desirable to write one set of compatible compilers and language utilities to run on both Prism/Mica and Prism/Ultrix, the use of lower case by case insensitive languages seems very desirable. Upper case names were used on VMS because upper case is considered aesthetically more pleasing in symbol table listings. If there are nonaesthetic reasons for preferring upper case over lower case then you should let me know.

From: TLE::DECWET::DON "Don MacLaren -- DECwest 13-Oct-1987 1414" 13-OCT-1987
To: PETERSON,@DIS$COMPILER_TEAM,@REVTEAM,@CALLSTD
Subj: Pillar Modules, Names, and the PRISM Object Language

It turns out that the VAX object language is not helpful in the implementation of Pillar modules. There are two problem areas; both have simple solutions in the PRISM object language because names can be long (up to 255 characters ?). I propose we do the simple thing on PRISM.

For VAX Pillar, the treatment of system dependent things, such as external names, descriptors, message files, and linkage options, will be specified later -- during the development of the VAX Pillar compiler. The treatments may not be as elegant as in PRISM, but I don't anticipate any significant loss of function.

PROBLEM 1. MODULE QUALIFIED NAMES.

A Pillar module can export the names of EXTERNAL declarations. These names do not have global scope. The normal form for referencing an exported item is a module-qualified name

module_name.item_name

In general, the EXTERNAL items exported from a module APLHA are implemented in one or more modules whose names are not known when ALPHA is compiled. For all practical purposes this requires that the name of the name of the global symbol representing the item be qualified by the name (ALPHA) of the module exporting its declaration.

On PRISM, Pillar will simply use the full module-qualified name (ALPHA.item_name) as the global symbol name.

PROBLEM 2. HIDDEN SYMBOLS.

Pillar permits a module to export the names of HIDDEN declarations. Again, a reference to such an exported item is via a module qualified name. The difference from the EXTERNAL case is that, when compiling a module that uses a HIDDEN declaration exported from ALPHA, the compiler will get information from the (compiled) Pillar definition module, BETA, that implements the hidden declaration. It finds BETA using the item's module-qualified name (ALPHA.item_name) as a key for library search. This is just what the linker does, but a different library index is searched. (In the PRISM object langauge, the indicies into which a name is entered are specified by the "name_level" field in a name table entry.)

So far, so good. However the implementation (in BETA) of a HIDDEN declaration may involve references to symbols that are internal to BETA, i.e. they have no global-symbol name in the normal course of things. The most interesting case is that of a HIDDEN inline procedure, which can reference all sorts of things visible only within BETA. The VAX object language provides a relevant construction: "environments" and "module local symbols", but such a symbol can't become a universal symbol in a shared image. This would be a severe restriction in the use of Pillar inline procedures.

On PRISM, for an internal item used in this way, Pillar will use a global symbol with the name of the form

BETA.1$.item_name

SUMMARY.

Because global names can be long on PRISM, Pillar only needs global names, and their relationship to the name used by the programmer will be obvious.

From: TLE::DECWET::PETERSON "Kim Peterson, DECwest Engineering 04-Jan-1988 09
To: @TEMP
Subj: Originally sent out last week. These are my proposals. Unresolved issu

d i g i t a l

I N T E R O F F I C E M E M O R A N D U M

TO: @Calling Standard

DATE: 23-Dec-1987
FROM: Kim Peterson
DEPT: DECwest Engineering
EXT: 206-865-8704
LOC: ZSO
ENET: DECWET::PETERSON

CC: Mica O.S. Group

Digital Equipment Corporation -- Confidential and Proprietary

SUBJECT: Additional Changes to the Mica Module Format

1

The following changes in the module's global symbol table are prompted by the changes to the module name table that allow qualified names to be specified in the name table. (These changes are described in my previous memo dated 8-Dec-1987.)

- Environment entries, local symbol entries, and internal entries will be removed from the global symbol table because they are superceded by the use of qualified names in the module name table.

- A null entry will be added as a place holder because an entry in the module name table may not represent a symbol. (For example, a module name table entry may be an atomic name that only represents a symbol when it is qualified with another atomic name.) This null entry will be a longword in size.

An alternative to using null entries in the symbol table is to index all entries in the global symbol table with their name's index in the module name table. This alternative would cost an extra longword on each entry, and would be more costly as long as the number of symbol names was greater than half of the total number of names.

In the case of languages that have no qualification, using the null entry will result in the greater space savings. In the case of languages that have one level of qualification (such as PILLAR), the two alternatives are require approximately equal space since the

number of compound names (which are meaningful) approximately equals the number of atomic names (which are meaningless). In the case where

there are more levels of qualification, using the null entry will result in greater space wastage.

The use of null entries in the symbol table is proposed because requires less space than the alternative for many DEC languages, and it never requires more space.

2

The following changes in the module's data relocation table are prompted by both the changes to the module name table and the change in the calling sequence.

- Local symbol relocation entries will be removed from the relocation table because they are superceded by the use of qualified names in the module name table.

- A cleared data_fixup bit in a symbol relocation entry will mean that a procedure's entry point as well as invocation descriptor will be fixed up. A set data_fixup bit in a symbol relocation entry will mean that only a procedure's invocation descriptor will be fixed up.

The calling sequence was changed to allow a caller to maintain the address of the called routine's entry point as well as its invocation descriptor. The caller can do this using the symbol relocation entry to fixup both the invocation descriptor and its entry point at once. This fixup requires that the location of the entry point fixup immediately follows the location of the invocation descriptor fixup.

The linker handles the case of relocating the invocation descriptor of a procedure with the same method as it does now. If the procedure is defined in an object module, the linker adds the location of the procedure's invocation descriptor to the contents of the specified location and stores the sum at the specified location. If the procedure is defined in a shareable image, the linker adds the location of the autoload vector to the contents of the specified location and stores the sum at the specified location. (The linker creates the autoload vector and stores in it the index of the procedure's transfer vector.)

The linker handles the case of relocating both the invocation descriptor and the entry point by first relocating the invocation descriptor as described above. Then if the procedure is defined in a shareable image, the linker adds the location of the autoload routine's entry point to the contents at the succeeding fixup location. (The succeeding fixup location is determined by the original fixup location and the size of the fixup (which is specified in the store_length field of the relocation)). Otherwise, if the procedure is defined in an object module, the linker puts the location of the invocation descriptor, the location of the fixup, and the size of the fixup on an internal relocation list. When all of the other

data relocations are done, the linker goes through the list and for each fixup location, it adds the value at the invocation descriptor's location to the value at the succeeding fixup location.

The entry point fixups are done this way:

1. To ensure that the entry point location stored with the invocation descriptor location matches the value in the invocation descriptor

2. To ensure that the entry point is never separated from its invocation descriptor

3. To avoid special symbols for procedure entry points

4. To minimize entries in the data relocation table

3

The following miscellaneous changes are prompted by the ongoing implementation of the linker.

- The module header type field will be enlarged from a byte to a word and placed in the second longword of the module. The first longword will have a value for the system id in its first longword that distinguishes it from VMS and Ultrix object modules and images.

- The use of procedure symbols will be better described. All languages must use procedure symbols to define procedures in order to allow the linker to check data relocations. Languages other than Fortran would not specify any argument descriptors.

- The use of PSECT entries in the data relocation table will be better described. A reference to an offset in a PSECT can be fixed up by placing the offset at the fixup location, and then using the PSECT data relocation entry to add the location of the PSECT to the offset at the fixup location.

From: TLE::DECWET::PETERSON "Kim Peterson, DECwest Engineering 15-Dec-1987 12
To: @CALLING_STANDARD
Subj: Comments? (on the proposed change to allow qualified names in PRISM modu

---

|d|i|g|i|t|a|l|

I N T E R O F F I C E M E M O R A N D U M

TO: @Calling Standard

DATE: 15-Dec-1987
FROM: Kim Peterson
DEPT: DECwest Engineering
EXT: 206-865-8704
LOC: ZSO
ENET: DECWET::PETERSON

CC:

Digital Equipment Corporation -- Confidential and Proprietary

SUBJECT: Incorporation of Qualified Name Proposal in Mica WDD

In a memo dated 8-Dec-1987, I proposed modifying the module name table in PRISM modules to allow qualified names. The consensus of opinion in SDT and DECwest seems to be to adopt this proposal. This proposal will be incorporated into the Object Module and Image File Format chapter of the Mica WDD, unless I hear otherwise by 21-Dec-1987. It is important to move quickly on this proposal because if it is adopted the linker must incorporate it by the middle of January, 1988.

The following is the substance of the proposal, which is repeated from my 8-Dec-1987 memo.

The proposed change would allow names to be qualified by other names in the module name table. The module name table would contain two types of names, atomic and compound. Atomic names are equivalent to the names in the current design of the module name table -- they are a string of characters. Compound names have no equivalent in the current design and are an ordered tuple of atomic names. The linker and librarian would handle compound names as they handle atomic names. For example, the librarian would use a compound name as a key to a module just as it would use an atomic name as a key.

The format of the module name table entries would change in the following ways:

1. A type field a byte in size would be added to distinguish atomic names from compound names

2. The name length field in the entry would reduced from a word to a byte

3. The alignment of each entry is reduced from word boundaries to byte boundaries

4. Compound names would be denoted by a pair of indexes into the module name table -- each index can denote either an atomic name or another compound name

5. There are three types of entries to denote compound names:

   - an entry with both indexes byte values

   - an entry with both indexes word values

   - an entry with both indexes longword values

\The type field would be used for future extensions to the name table for such things as multiple byte character sets. To make such extensions as easy as possible, the meaning of the value in the length field is based upon its entry's type.\

In addition, the following rules would be adopted for names contained within the module name table:

- The null character (byte value 0) cannot occur in an atomic name. The first reason for this restriction is that strings that contain nulls are not expressible in C, and the second reason is to maintain a character that can be used as a separator of atomic names.

- The maximum length of atomic names is 255 characters.

- The maximum number of atomic names in a compound name is 255. The reason for this restriction is to allow the length of an expanded compound name to fit in a word.

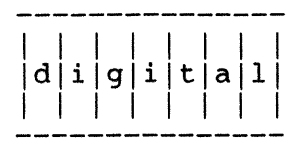
- The first name in a module name table has an index of one. The index is incremented by one for each subsequent name.

- The indexes used in a compound name entry must be less than the compound name's entry's index. In other words, a compound name entry can not make a forward reference. The reason for this restriction is to make life easier for the reader of the module name table.

- The name index field continues to specify in which library index the name is used as a key. Module names are entered into index one, and defining instances of global symbols are entered into key index two.

  \Note that both atomic names and compound names can be global symbols.\

From: TLE::DECWET::PETERSON "Kim Peterson, DECwest Engineering 08-Dec-1987 14
To: @CALLING_STANDARD
Subj: Proposed change to module name table to support qualified names



INTEROFFICE MEMORANDUM

TO: @Calling Standard

DATE: 8-Dec-1987
FROM: Kim Peterson
DEPT: DECwest Engineering
EXT: 206-865-8704
LOC: ZSO
ENET: DECWET::PETERSON

CC: Dave Walp

Digital Equipment Corporation -- Confidential and Proprietary

SUBJECT: Implementation of Qualified Names in PRISM Modules

The current design of naming in PRISM object modules is not adequate for the needs of modern programming languages. I propose that the current naming design by replaced by the following design, which was suggested by Don McLaren.

The proposed change would allow names to be qualified by other names in the module name table. The module name table would contain two types of names, atomic and compound. Atomic names are equivalent to the names in the current design of the module name table -- they are a string of characters. Compound names have no equivalent in the current design and are an ordered tuple of atomic names. The linker and librarian would handle compound names as they handle atomic names. For example, the librarian would use a compound name as a key to a module just as it would use an atomic name as a key.

The format of the module name table entries would change in the following ways:

1. A type field a byte in size would be added to distinguish atomic names from compound names

2. The name length field in the entry would reduced from a word to a byte

3. The alignment of each entry is reduced from word boundaries to byte boundaries

4. Compound names would be denoted by a pair of indexes into the module name table -- each index can denote either an atomic name or another compound name

5. There are three types of entries to denote compound names:

- an entry with both indexes byte values
- an entry with both indexes word values
- an entry with both indexes longword values

\The type field would be used for future extensions to the name table for such things as multiple byte character sets. To make such extensions as easy as possible, the meaning of the value in the length field is based upon its entry's type.\

In addition, the following rules would be adopted for names contained within the module name table:

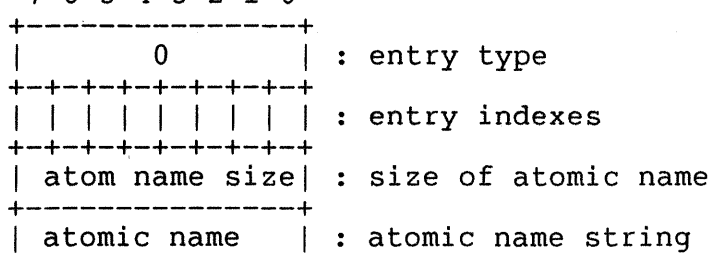
- The null character (byte value 0) cannot occur in an atomic name. The first reason for this restriction is that strings that contain nulls are not expressible in C, and the second reason is to maintain a character that can be used as a separator of atomic names.
- The maximum length of atomic names is 255 characters.
- The maximum number of atomic names in a compound name is 255. The reason for this restriction is to allow the length of an expanded compound name to fit in a word.
- The first name in a module name table has an index of one. The index is incremented by one for each subsequent name.
- The indexes used in a compound name entry must be less than the compound name's entry's index. In other words, a compound name entry can not make a forward reference. The reason for this restriction is to make life easier for the reader of the module name table.
- The name index field continues to specify in which library index the name is used as a key. Module names are entered into index one, and defining instances of global symbols are entered into key index two.

  \Note that both atomic names and compound names can be global symbols.\

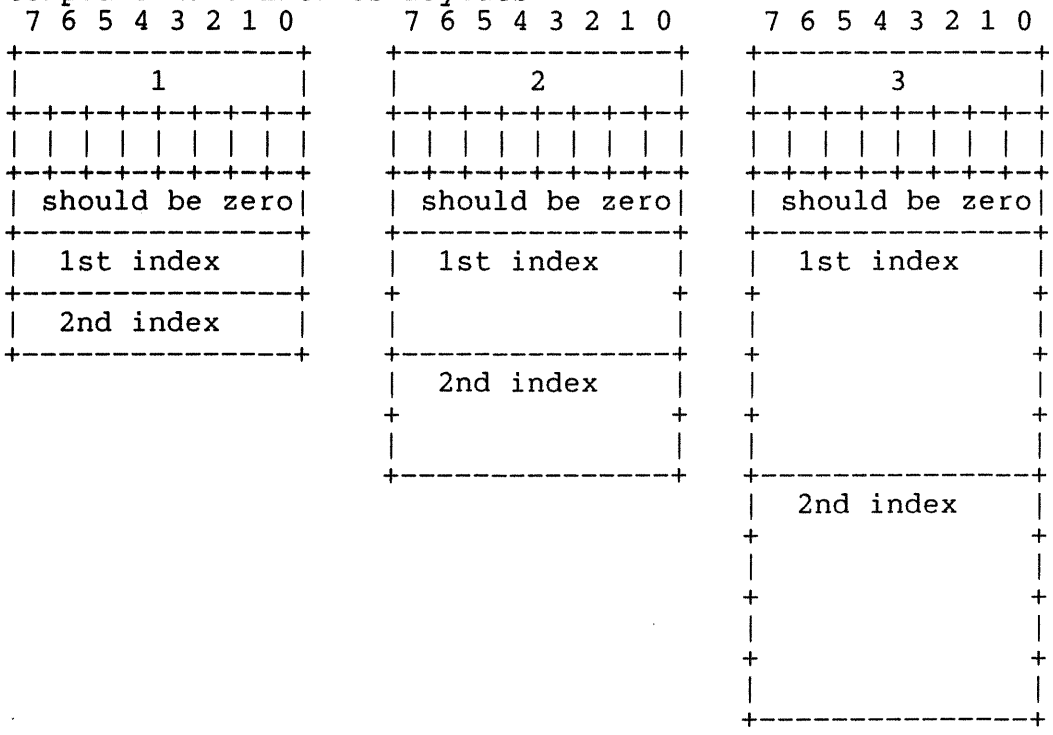
The following are the proposed declaration and layout of module name table entries.

```
module$name_table_entry(
    name_size : unsigned_byte;
    name_type : module$name_table_type[..] SIZE(byte) ): RECORD
    CAPTURE name_size, name_type;
    name_index : SET integer[1..8] SIZE(byte);
    VARIANTS
        CASE name_type
            WHEN module$c_mnt_atomic THEN
                name_string : string(name_size);
            WHEN module$c_mnt_compound_byte THEN
                byte_index1 : unsigned_byte;
                byte_index2 : unsigned_byte;
            WHEN module$c_mnt_compound_word THEN
                word_index1 : unsigned_word;
                word_index2 : unsigned_word;
            WHEN module$c_mnt_compound_longword THEN
                longword_index1 : longword;
                longword_index2 : longword;
    END VARIANTS;
END RECORD;

module$name_table_type : (
    module$c_mnt_atomic,
    module$c_mnt_compound_byte,
    module$c_mnt_compound_word,
    module$c_mnt_compound_longword );
```

Atomic Name Entry Layout



Compound Name Entries Layouts



From: TLE::GROVE 22-FEB-1988 11:13
To: SACKS,JBISHOP,@GEM$DIS
Subj: News about the PRISM object language

This note is a quick summary of changes in the PRISM object langauge that we discussed last week in Seattle, with Kim Peterson et al.

It is a DECwest goal to put the object language chapter under ECO control by the end of March. Kim expects to have an updated version of the object language chapter in our hands by 29 Feb. We should do a thorough review and quickly send any comments or other changes to they can be considered before the end of March.

The following items were discussed and changes accepted:

1. Structure of the MNT. It was agreed to change from having parallel MNT and GST tables, to going to a scheme were names in the MNT are referred to from the GST by index values. There will be a single index numbering of the MNT; first name in MNT has index value 1, etc. The index value 0 can be used in a GST entry to indicate that the GST item has no name.

   This means that the "null" GST item can be eliminated, because there is no need for a filler since the MNT and GST are no longer parallel.

2. It was agreed that the syntax of compound names would be normalized to the form <compound name>.<simple name>.

3. All names in the MNT must be unique.

4. Data initialization. It was agreed to provide a data initialization mechanism similar to the DRT table. This may actually be part of the DRT table, or it may be a separate table. (Details from Kim)
   You can initialize a piece of storage by specifying the psect, offset, item length, and value. There will be some abiltity to specify a repeat count.

5. Transfer address will be specified (as in the current spec) as a symbol rather than as Psect+Offset on VAX. It will be possible to have a GST entry with no name, so to specify a transfer address to a local symbol or unnamed location, you create a GST entry with no name.

6. The BLISS group request the ability to do global arithmetic and store values smaller than a longword (with overflow checking). We discussed this and concluded that elaborate global arithmetic facilities are not generally needed on PRISM and not appropriate. We recommend that PRISM BLISS restrict global arithmetic to that subset supported by the PRISM object language and linker. We do not expect that this will affect many BLISS programs.

7. Following is note from Kim describing new name matching rules that are intended to support FORTRAN, C, etc and to allow them to work harmoniously in a mixed language environment.

If you have questions about any of these items, please ask me or check the forthcoming object language chapter.

Rich Grove

! From: DECWET::PETERSON "Kim Peterson, DECwest Engineering 21-Feb-1988 2259"

! To: @CALLING_STANDARD
! Subj: Changes in global symbol table for better access to FORTRAN common

The following implements the new semantics of symbol and psect references agreed to during the recent calling standard meeting.

The value modobj$c_symbol_reference is replaced by the values:

- modobj$c_symbol_reference - satisfied only by symbol definition; error if unsatisfied
- modobj$c_symbol_psect_reference - satisfied by symbol or psect definition; error if unsatisfied

The meaning for the value modobj$c_symbol_definition remains unchanged.

The value modobj$c_symbol_psect is replaced by the values:

- modobj$c_psect_definition - defines a PSECT just as modobj$c_symbol_psect did; in the terminology of the meeting, it is a strong common definition when the overlaid attribute is set
- modobj$c_psect_symbol_definition - defines an overlaid PSECT just as modobj$c_psect_definition does, but only if no overlaid PSECT or global symbol of the same name is found; in the terminology of the meeting, it is a weak common definition.

The value modobj$c_symbol_null is eliminated.

From: TLE::DECWET::PETERSON "Kim Peterson, DECwest Engineering 24-Feb-1988 18
To: TLE::JBISHOP,TLE::SACKS,@CALLING_STANDARD
Subj: Proposed changes to object module format from Calling standard meeting

```
---------------
| | | | | | | |
|d|i|g|i|t|a|l|        I N T E R O F F I C E  M E M O R A N D U M
| | | | | | | |
---------------
```

TO: @Calling Standard

DATE: 24-Feb-1988
FROM: Kim Peterson
DEPT: DECwest Engineering
EXT: 206-865-8704
LOC: ZSO
ENET: DECWET::PETERSON

CC: DECwest Distribution
John Bishop
Gerald Sacks

Digital Equipment Corporation -- Confidential and Proprietary

SUBJECT: ECO to the Object Module/Image File Format Chapter

The following represent changes to the Mica Object Module and Image File Format agreed to at the February calling standard meeting. These changes fall under four broad categories: name table, global symbol table, psect definition, and miscellaneous minor modifications.

1 NAME TABLE

1.1 Restriction Of Qualified Name Ordering

The current design allows both indexes in a qualified name entry to specify either an atomic name or another qualified name. This allows qualified names to be built in any order. The proposed change is to restrict the last index to specify an atomic name only. This only allows qualified names to be built sequentially starting with the first two atomic names and ending with the last. This restriction allows easier and more efficient implementation of qualified name support.

1.2 Change Alignment Of Name Table Entries

The current design has each entry byte-aligned. The proposed change is to word-align each entry. This change allows for more efficient access to the qualified name entries. Qualified name entries with word index fields were thought to be the most common name table entry after atomic names, and efficient access to them was required.

The change in alignment allowed the size field to be expanded from a byte to a word. The following is the resulting declaration:

```
module$name_table_entry(
    entry_type : module$name_table_type[..] SIZE(byte);
    entry_size : unsigned_word ): RECORD
    CAPTURE entry_size, entry_type;
    name_index : SET integer[0..7] SIZE(byte);
    VARIANTS
        CASE name_type
            WHEN module$c_mnt_atomic THEN
                name_string : string(entry_size-4);
            WHEN module$c_mnt_compound_byte THEN
                byte_index1 : unsigned_byte;
                byte_index2 : unsigned_byte;
            WHEN module$c_mnt_compound_word THEN
                word_index1 : unsigned_word;
                word_index2 : unsigned_word;
            WHEN module$c_mnt_compound_longword THEN
                longword_index1 : longword;
                longword_index2 : longword;
    END VARIANTS;
END RECORD;
```

1.3 Change NAME_SIZE Field To ENTRY_SIZE

The current design requires the name_size field to be dependent upon the type of name entry. The proposed change is to make this field the size of the entire entry. This change allows for easier addition of new entry types to the name table, and standardizes the structure of the module name table with other tables in the chapter.

1.4 Restrict Duplicate Names

The current design allows duplicate names to be entered in the module name table. The proposed change is to disallow duplicate names. This change allows an easier and more efficient implementation of name table support.

2 GLOBAL SYMBOL TABLE

2.1 Name Index

The current design requires that the global symbol table be written and read in lock step with the name table. The proposed change is to add a name index field to global symbol table entries. The index field would contain an index to an entry in the name table. A value of zero in this field signifies an unnamed symbol, which can only be used within a module, and does not show up in a map of the image. This change allows an easier implementation for creating both name tables and global symbol tables. This change also allows unnamed symbols, which are necessary in specifying transfer addresses in BLISS.

2.2 Null Entry Removed

The current design requires a null entry to maintain coherency between the name table and the global symbol table. With explicit name table indexing in the global symbol table, this entry is useless.

2.3 New Symbol Referencing Semantics

The current design allows symbol references to be resolved only with other symbols. This means that languages cannot access FORTRAN common without explicit language features that direct the compiler to resolve to a psect instead of a symbol. The proposed change is to keep the current semantics and add another symbol reference type (MODOBJ$C_SYMBOL_PSECT_REFERENCE) that could resolve to a FORTRAN common if a FORTRAN common definition was present and a symbol definition was not. It remains an error if no symbol or common definition is present.

To resolve each symbol/psect reference, the linker would first search each module for a symbol definition that matched the symbol/psect reference. If no symbol definition is found in any of the modules used in the link, the linker would then search the psect table(s) for an overlaid psect definition that matched.

If a symbol definition is found, the linker would still search the psect table(s) for a possible match. If a match was found, the linker would report a warning that the reference is ambiguous.

2.4 Rename PSECT Definition Entry

The current design uses MODOBJ$C_SYMBOL_PSECT to name a psect definition entry. The code is renamed to MODOBJ$C_PSECT_DEFINITION to more appropriately name it.

2.5 Modify Global Common Definition

The current design specifies that the symbol definition subtypes MODOBJ$C_SYMBOL_COMMON_LONG and MODOBJ$C_SYMBOL_COMMON_QUAD defines a global common symbol that matches an overlaid psect of the same name. The current design also specifies that the symbol cannot resolve to a standard global symbol. The proposed change is to not define a global symbol for it, but to allow it to resolve to a global symbol if a global symbol exists. In addition, since global common definitions are no longer a symbol definition, but a mixed symbol reference and psect definition, a new entry type (MODOBJ$C_PSECT_SYMBOL_DEFINITION) will speicify it. This change is required to support C semantics in ULTRIX.

3 PSECT DEFINITION

3.1 FORTRAN Common PSECT Definition

The current design does not specify how FORTRAN common is defined. The proposed change is to specify that FORTRAN common is defined as a PSECT with overlaid, relocatable, global, readable, and writable attributes. This is required because the linker and multiple languages need to implement FORTRAN common compatibly.

The use of SHARE is covered by the next item.

3.2 SHARE Attribute

The current design specifies that SHARE signifies inter-process sharing of read/write data. The proposed change is to specify that SHARE signifies inter-image sharing. The use of a PSECT attribute to specify inter-process sharing is replaced by the linker qualifier INSTALL. This change is required to allow access to FORTRAN common exported from shareable images using process private memory. Sharing of FORTRAN common between processes is still available through the use of the linker qualifier INSTALL. The INSTALL qualifier causes the linker to set SHARE attribute in the image sections resulting from shareable PSECTs. It also guarantees that images so linked must be installed /write before they can be activated. In the absence of the SHARE qualifier, the linker leaves the image sections copy-on-modify.

This represents several changes from VAX/VMS behavior.

- VAX/VMS does not allow FORTRAN common to be shared between shareable images using process-private memory
- VAX/VMS FORTRAN common has by default the SHARE attribute, which seems to be incorrect

The share attribute would be set by the linker on psects that are specified by the linker qualifier EXPORT_PSECT. The export_psect qualifier is analogous to the UNIVERSAL qualifier in that it controls access to PSECTs within the shareable image.

3.3 TLS Template PSECTs

The current design specifies that TLS template psects have their own subtype. The proposed change is to specify TLS template psect through a new PSECT attribute (MODOBJ$C_PSECT_TLS). The linker would create TLS template regions using PSECT definitions with this attribute set.

3.4 PSECT Initialization

The current design allows inadequate means of initializing data psects. The proposed change specifies a compressed method of representing data that the linker instantiates in the image. The data compression is the same as used in TLS templates, and is specified in the TLS design. The change involves adopting the PSECT entry format currently used for subtype MODOBJ$C_PSECT_TLS to the new subtype MODOBJ$C_PSECT_COMPRESSED. In other words, the current psect entry subtype MODOBJ$C_PSECT_TLS is renamed to MODOBJ$C_PSECT_COMPRESSED.

For non-TLS psects, the linker instantiates the compressed date into the image, and for TLS template psects, the linker creates a template region in the image.

4 MISCELLANEOUS

4.1 Object Module Organization

Rules for laying out object modules are not specified in the current design. The following rules will be added so that compiler writers have guidance on how an object module should be structured for the linker to optimally process.

- The module's item list should immediately follow the module's header.
- The module's name table and the global symbol table should be grouped together in that order.
- The module should not duplicate item list entries.

4.2 Field Prefixing For BLISS

Prefixes will be specified that allow record fields to be unique in languages like BLISS that have a flat name space.

4.3 Entity Consistency Check Table

The current design specifies that the match control field in the entity consistency check table is ignored for ASCII identification. The proposed change is to require that the field be MODULE$C_EQUAL.

Additionally, the following may have additional requirements for entity consistency checking:

- PILLAR (signature checking)
- PASCAL (environment checking)
- IPSE

This is an open issue that is added to the open issues section.

4.4 Autoloading

The current description of autoloading is not completely up to date. The proposed change is to bring it up to date with the image activation chapter.

4.5 File Format

The current design specifies a file format for an item list entry. The design is not used and won't be implemented for FRS. The proposed change is to make the field SBZ, and put the description of file format into backslash comments.