Mica Working Design Document Naming Standards and Pillar Coding Conventions

Revision 0.6

13–January–1988

Issued by:
Kris K. Barker

digital™

TABLE OF CONTENTS

CHAPTER 1 NAMING STANDARDS AND PILLAR CODING CONVENTIONS 1–1

1.1 Introduction 1–1

1.2 Naming Standards 1–1
1.2.1 Goals 1–1
1.2.2 Scope 1–1
1.2.3 General Naming Standards 1–2
1.2.3.1 Case Sensitivity 1–3
1.2.3.2 What is a Facility? 1–3
1.2.4 Facility Names 1–3
1.2.5 Module Names 1–4
1.2.5.1 Definition Modules 1–4
1.2.5.2 Implementation Modules 1–4
1.2.5.3 Combination Modules 1–4
1.2.6 File Names and File Types 1–5
1.2.7 Procedures 1–5
1.2.7.4 System Routines 1–5
1.2.7.5 System Services and Executive Routines 1–6
1.2.7.6 Kernel Routines 1–6
1.2.7.7 Procedure Arguments 1–6
1.2.8 Types 1–6
1.2.8.8 Enumerated Type Element Names 1–6
1.2.8.9 Data Structure Types 1–6
1.2.9 Global Variables 1–7
1.2.10 Constants 1–7
1.2.11 Messages 1–7
1.2.12 Logical Names 1–8
1.2.13 Objects and Object Containers 1–8
1.2.14 Compile-time Facility Macros and Procedures 1–8

1.3 Pillar Coding Conventions 1–9
1.3.1 Goals 1–9
1.3.2 Indentation 1–9
1.3.3 Capitalization 1–9
1.3.4 Line Length 1–9
1.3.5 Multistatement Lines and Multiline Statements 1–10
1.3.6 Comments 1–10
1.3.6.1 Module Level Comments 1–10
1.3.6.2 Procedure Level Comments 1–11
1.3.6.3 Block Comments 1–11
1.3.6.4 Line Comments 1–12
1.3.7 "Whitespace" 1–12

1.3.8 Module Format 1–13
1.3.9 Copyright Formats 1–14
1.3.10 Procedure Format 1–14
1.3.11 Condition Handler Format 1–15
1.3.12 Order of Declarations 1–15
1.3.13 Statement Format 1–15
1.3.13.1 IF/THEN/ELSE 1–15
1.3.13.2 LOOP 1–15
1.3.13.3 CASE 1–16
1.3.13.4 Blocks (WITH Statement) 1–16
1.3.13.5 VALUE, TYPE, VARIABLE, BIND Declarations 1–16
1.3.13.5.1 RECORD Types 1–17
1.3.13.5.2 Enumerated Types 1–17
1.3.13.6 Procedure Declarations 1–18
1.3.13.7 Procedure Invocation 1–19
1.3.14 Message and Condition Declarations 1–19
1.3.15 Miscellaneous 1–19
1.4 OPEN ISSUES 1–21

INDEX

Revision History

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

CHAPTER 1

NAMING STANDARDS AND PILLAR CODING CONVENTIONS

1.1 Introduction

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

1.2 Naming Standards

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

1.2.1 Goals

Naming standards are important for several reasons:

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

1.2.2 Scope

This section covers the public naming standards for:

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

Module names
The names assigned to program source modules.

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

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

Program data and type names

Including:

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

Logical names

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

Compile-time facility macros and procedures

Macros and command procedures used during the compilation process.

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

1.2.3 General Naming Standards

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

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

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

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

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

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

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

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

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

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

1.2.3.1 Case Sensitivity

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

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

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

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

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

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

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

1.2.3.2 What is a Facility?

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

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

1.2.4 Facility Names

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

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

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

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

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

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

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

1.2.5 Module Names

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

1.2.5.1 Definition Modules

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

- "Internal" or "Private" definition modules

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

  Private definition module names are of the form:

  ```
  facility$module-name_DEF
  ```

- "External" or "Public" definition modules

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

  Public definition module names are of the form:

  ```
  facility$DEFINITION
  ```

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

1.2.5.2 Implementation Modules

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

```
facility$module-name
```

1.2.5.3 Combination Modules

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

1.2.6 File Names and File Types

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

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

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

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

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

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

1.2.7 Procedures

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

```
facility$entry-name
```

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

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

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

1.2.7.4 System Routines

System routines are those routines that are:

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

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

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

1.2.7.5 System Services and Executive Routines

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

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

1.2.7.6 Kernel Routines

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

1.2.7.7 Procedure Arguments

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

1.2.8 Types

The basic format for type names is:

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

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

1.2.8.8 Enumerated Type Element Names

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

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

1.2.8.9 Data Structure Types

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

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

1.2.9 Global Variables

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

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

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

1.2.10 Constants

Named constants have names of the following form:

```
facility$C_name
```

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

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

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

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

1.2.11 Messages

Message names are of the form:

```
facility$_status-name
```

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

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

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

```
EXEC$_status-name
```

1.2.12 Logical Names

Logical names are of the form:

```
facility$name
```

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

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

1.2.13 Objects and Object Containers

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

System object container names are of the form:

```
facility$name_OBJECT_CONTAINER
```

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

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

1.2.14 Compile-time Facility Macros and Procedures

TBD.

1.3 Pillar Coding Conventions

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

1.3.1 Goals

Writing code which follows these conventions has the following benefits:

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

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

1.3.2 Indentation

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

1.3.3 Capitalization

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

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

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

1.3.4 Line Length

The maximum source line length is 112 characters.

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

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

1.3.5 Multistatement Lines and Multiline Statements

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

1.3.6 Comments

These conventions describe formats for four different uses of comments.

1.3.6.1 Module Level Comments

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

```
MODULE module_name;

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

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

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

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

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

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

1.3.6.2 Procedure Level Comments

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

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

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

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

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

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

1.3.6.3 Block Comments

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

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

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

1.3.6.4 Line Comments

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

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

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

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

1.3.7 "Whitespace"

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

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

is preferable to

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

and:

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

is preferable to

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

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

For example:

```
VALUE
    value_name = some_value;
VARIABLE
    variable_name : variable_type = variable_initializer;
PROCEDURE foo (
            IN arg : arg_type;
            );
```

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

```
VALUE

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

is preferred to:

```
VALUE

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

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

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

or

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

1.3.8 Module Format

The general format for a Pillar module is:

```
MODULE module_name;

! Module-level comments

Interface section

Implementation section

Module linkage options

END module_name;
```

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

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

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

- Implementation section:

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

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

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

1.3.9 Copyright Formats

For internal sources, the copyright format is:

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

\The above copyright statement is available in:

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

For distributable sources, the copyright format is TBD.

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

1.3.10 Procedure Format

The general format for a Pillar procedure is:

```
PROCEDURE procedure_name...

!++
! Procedure-level comments
!--

VALUE, TYPE, VARIABLE, BIND Declarations

BEGIN
    statement-sequence ...

SUBPROCEDURES

    PROCEDURE sub_procedure_name...

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

    VALUE, TYPE, VARIABLE, BIND Declarations

    BEGIN
        statement-sequence ...
    END sub_procedure_name;

END procedure_name;
```

Procedure-level comments are described in Section 1.3.6.2.

1.3.11 Condition Handler Format

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

1.3.12 Order of Declarations

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

- VALUE
- TYPE
- VARIABLE
- BIND

1.3.13 Statement Format

The following sections describe preferred formats for several Pillar statements.

1.3.13.1 IF/THEN/ELSE

IF/THEN/ELSE statements are formatted as follows:

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

1.3.13.2 LOOP

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

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

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

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

For more complicated loops, use one of these formats:

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

or:

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

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

1.3.13.3 CASE

CASE statements are formatted as follows:

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

1.3.13.4 Blocks (WITH Statement)

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

```
WITH

VALUE, TYPE, VARIABLE, BIND declarations

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

1.3.13.5 VALUE, TYPE, VARIABLE, BIND Declarations

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

```
VALUE

    first_value = some_value;
    second_value = some_value;

TYPE

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

VARIABLE

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

BIND

    name = variable_name;
```

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

1.3.13.5.1 RECORD Types

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

```
TYPE

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

The field list is:

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

Within records, unions and variants are formatted as follows:

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

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

1.3.13.5.2 Enumerated Types

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

```
TYPE

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

1.3.13.6 Procedure Declarations

Procedure declarations are formatted as follows:

- External declarations:

```
PROCEDURE
    procedure_name1 (
        IN first_param : some_type;
        .
        .
        .
        ) RETURNS return_type;
        EXTERNAL;
    procedure_name2 (
        .
        .
        .
        ) RETURNS return_type;
        EXTERNAL;
```

or

```
PROCEDURE procedure_name1 (
            IN first_param : some_type;
                .
                .
                .
            ) RETURNS return_type;
            EXTERNAL;
PROCEDURE procedure_name2 (
                .
                .
                .
            ) RETURNS return_type;
            EXTERNAL;
```

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

- Normal declarations:

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

Section 1.3.10 describes the complete procedure format.

Note:

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

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

1.3.13.7 Procedure Invocation

The following format is used to invoke a procedure:

- Keyworded parameters:

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

- Positional parameters:

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

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

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

1.3.14 Message and Condition Declarations

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

1.3.15 Miscellaneous

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

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

```
TYPE

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

VARIABLE

    first_record, second_record : sample_record_pointer;

BEGIN

    !
    ! Allocate the records.
    !

    ALLOCATE first_record LOCAL;
    ALLOCATE second_record LOCAL;
```

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

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

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

second_record^ = first_record^;

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

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

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

- Others TBD.

1.4 OPEN ISSUES

The following issues are yet to be resolved.

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