Data Monitoring Reference Manual

0890493-000
December 2003

The information contained in this document is believed to be correct at the time of publication. It is subject to change without notice. Concurrent Computer Corporation makes no warranties, expressed or implied, concerning the information contained in this document.

To report an error or comment on a specific portion of the manual, photocopy the page in question and mark the correction or comment on the copy. Mail the copy (and any additional comments) to Concurrent Computer Corporation, 2101 W. Cypress Creek Road, Ft. Lauderdale, FL 33309–1892. Mark the envelope **"Attention: Publications Department**." This publication may not be reproduced for any other reason in any form without written permission of the publisher.

Printed in U. S. A.

Revision History:	Level:	Effective With:
Original Release -- May 1999	000	MAXAda 3.2

Contents

Chapter 4 FORTRAN Interface

Appendix A MAXAda Examples

Appendix B C Examples

Appendix C FORTRAN Examples

1
Data Monitoring

This chapter presents the concepts and requirements of Data Monitoring. Data Monitoring allows you to specify executable programs that contain Ada, C, or FORTRAN variables to be monitored, obtain and modify the values of selected variables by specifying their names, and obtain such information about the variables as their virtual addresses, types, and sizes.

Three interfaces are available:

Ada The `Real_Time_Data_Monitoring` package and compilation environment (**`/usr/ada/default/rtdm`**) is bundled and shipped with the MAXAda product.

C,C++ The Data Monitoring library header file (**`/usr/lib/libdatamon.a`** and **`/usr/include/datamon.h`**) are provided via the **`datamon`** PowerMAX OS package and the **`ccur-datamon`** RedHawk Linux RPM.

FORTRAN The Data Monitoring library and FORTRAN header file (**`/usr/lib/libdatamon.a`** and **`/usr/include/datamon_.h`**) are provided via the **`datamon`** PowerMAX OS package and the **`ccur-datamon`** RedHawk Linux RPM.

Subsequent chapters in this manual describe each of the above interfaces. The remaining portion of this chapter deals with Data Monitoring requirements which are common to all of the interfaces.

Requirements

Data Monitoring uses symbolic information generated by compilers; it requires the use of the **`-g`** option (to generate debug information) when compiling source files containing variables to be monitored.

Data Monitoring supports monitoring variables from programs built with the following compilers:

PowerMAX OS:

- Concurrent MAXAda
- Concurrent C/C++
- Concurrent Fortran

RedHawk Linux:

- Concurrent MAXAda

- Concurrent Fortran
- GNU C/C++
- GNU Fortran (limited support)

Many of the subprograms within Data Monitoring require that the target program be executing. For statically linked programs, however, the target program, in general, does not need to be executing if the only subprograms invoked are the following:

- `open_program`, `dm_open_program`
- `info_only`
- `get_type_name`, `dm_get_type_name`
- `get_enum_image`, `dm_get_enum_image`
- `get_enum_val`, `dm_get_enum_val`
- `get_array_info`
- `get_constraints`
- `get_real_time_monitoring_error`, `dm_get_error_string`
- `get_real_time_monitoring_error_code`, `dm_get_error_code`
- `close_program`, `dm_close_program`
- instantiations of `list.list` and `list.global_list`, `dm_list`

If Data Monitoring is to be used only to obtain symbolic information about variables within a target program, that target program does not need to be executing unless it uses shared libraries. If the target program is not executing, the variables must have addresses that are calculated without access to the memory image of an executing process—that is, their addresses, size, and shape must be completely static (i.e. determined at compile or link time).

Data Monitoring subprograms use the **`usermap(3)`** library routine to create address mappings between the monitoring process and the target process. Once pages from the target process are mapped into the monitoring process, the monitoring process assumes that the target pages will not change their physical location. The physical location of the pages can change in the following circumstances:

- The target process terminates.
- The target process un-maps the target address.
- The target address is in a private, writable page, and the target process calls **`fork(2)`** and then writes to or locks the target address before the child process does.
- The target process has a private, read-only mapping at the time of the **`usermap(3)`** call, subsequently calls **`mprotect(2)`** to make the mapping writable, and then writes to the target address.
- The target process explicitly maps the target address to a new physical page.

In such situations, the monitoring process is unaware of the change in mapping; the results of subsequent Data Monitoring subprogram calls that access target process addresses are undefined. For further explanation of what is meant by the terms *private*, *writable*, see the information on **MAP_PRIVATE** and **PROT_WRITE** in the **`mmap(2)`** system manual page.

NOTE

Data Monitoring requires that the monitoring process have read access to the executable files associated with the processes being monitored. Further, if values of variables are to be obtained or modified, you must have read access or write access to the **`/proc`** files (see **`proc(4)`**) associated with the processes being monitored.

Variable Eligibility

Throughout this text, the term *target program* denotes an application that is being monitored. The term *target process* denotes the executing program that is being monitored. The term *target program file* denotes the disk image of the target program.

The term *package* denotes an Ada package, which is a grouping of variables, type declarations, subprograms, and tasks. The term *variable* denotes the symbolic name of any of the following:

- A non-composite variable (for example, a scalar)
- An element of an array variable
- A component of a record or structure variable
- A member of a common block
- A composite variable (for example, an array or record)

The term *target variable* refers to a variable in the address space of a process for which you wish to perform Data Monitoring.

The terms *variable* and *target variable* are further constrained by the following:

- The variable must have a static base address.
- The variable must have a static shape or the target program must be executing.
- The variable must have a static size or the target program must be executing.

The following variables are eligible for monitoring:

- Variables in library-level Ada packages (including nested packages)
- C variables whose storage class is `static` or `extern`

- FORTRAN variables within subroutines
- FORTRAN common block members

The following variables are not eligible for monitoring:

- Variables allocated on a program stack

  Examples include Ada variables within subprograms, C variables with storage class `auto`, and `procedure`, `function`, and `subroutine` parameters.

- Elements of array variables whose offsets are variable (for example, array[variable])

Expanded Name Notation

You must specify variables in symbolic expanded notation. The expanded notation used by Data Monitoring is similar to that specified by the Ada programming language. It has been extended for use with C and FORTRAN and is as follows:

```
expanded_name          ::= scope
expanded_name          ::= scope '.' variable_name
expanded_name          ::= variable_name

scope                  ::= [ file_scope '.' ] language_scope

file_scope             ::=  ' “simple_file_name” '
language_scope         ::= package_scope | subprogram_scope | common_scope
package_scope          ::= identifier { '.' language_scope }
subprogram_scope       ::= identifier
common_scope           ::= subprogram_scope '/' common_block '/'

common_block           ::= identifier | <null>

variable_name          ::= identifier | selected_component | indexed_component

selected_component     ::= prefix '.' selector
selector               ::= identifier | 'all'

indexed_component      ::= prefix '(' index {',' index } ')'
indexed_component      ::= prefix '[' index {',' index } ']'
index                  ::= numeric_literal | Ada_enumeration_literal

prefix                 ::= identifier | selected_component | indexed_component
```

In the rules just presented:

- <null> signifies absence of notation.
- Single quotation marks surround keywords and syntactic tokens.

> Note that you must not supply the single quotation marks when you are using expanded notation to specify variables.

Although the canonical form of a `scope` includes the file name enclosed in double quotation marks (as noted above in file_scope), it is often unnecessary to specify the file name. In many cases, the remaining portion of the scope, if any, unambiguously identifies the item of interest. A C `extern` variable, for example, can usually be identified by an expanded_name that solely includes the identifier denoting the variable. Similarly, a C `extern` or FORTRAN subroutine can usually be identified by an expanded_name that solely includes the identifier denoting the function or subroutine. And a library-level Ada package can usually be identified by an expanded_name that solely includes the identifier denoting the package. The file_scope portion of a scope is required only when one of the following is `true`:

- The item of interest is not globally visible (for example, C `static` functions or variables, variables within functions or subroutines)
- Another item exists with the same identifier at the same visibility level

The ".all" notation has been borrowed from the Ada language and represents pointer indirection. It must be used in place of the "*" operator in the C language; however, ".all" is placed after the pointer, whereas in the C language, the "*" precedes the pointer.

The ".all" notation is not required between pointers and selected components or between pointers and indexing; for example, the following are equivalent:

```
ptr_to_structure.all.component
ptr_to_structure.component
```

The following are also equivalent:

```
ptr_to_array.all[5]
ptr_to_array[5]
```

Consider the following Ada, FORTRAN, and C source program segments contained in source files `ada_source.a`, `fortran_source.f`, and `c_source.c`, respectively:

```
package pkg is
     type scalar_type is range 0..10 ;
     type enum_type is (class, object, auto) ;
     type record_type is
          record
               a : enum_type ;
               b : string (1..5) ;
          end record ;
     type array_type is
        array (enum_type, scalar_type) of integer ;
     type integer_ptr_type is access integer ;
     type record_ptr_type is access record_type ;
     Ada_scalar : scalar_type ;
     Ada_composite : array_type ;
     package nested_pkg is
          var : record_ptr_type := new record_type ;
          ptr : integer_ptr_type ;
     end nested_pkg ;
end pkg ;

package pkg.child is
     item : integer ;
end pkg.child ;

...
subroutine fortran_sub
common /named_common/ x, y, z
common dummy, item_in_blank_common, another_dummy
integer*4 subroutine_var(20)
end

subroutine sub
integer*4 int_var
end
...

int c_global_var ;
int sub ;
static int c_static_var ;

void c_func (void)
{
     static int ***ptr ;
```

```
        static int run[10][10] ;
        {
             static int nested_routine_var ;
        }
}
```

All of the following are eligible variables expressed in proper expanded notation:

- `pkg.ada_scalar`
- `pkg.ada_composite(class,4).b(3)`
- `pkg.nested_pkg.var.a`
- `pkg.nested_pkg.ptr.all`
- `pkg.child.item`
- `fortran_sub.subroutine_var(5)`
- `fortran_sub/named_common/y`
- `fortran_sub//item_in_blank_common`
- “`fortran_source.f`”`.sub.int_var`
- `c_global_var`
- `“c_source.c”.c_static_var`
- `c_func.run[3][5]`
- `c_func.nested_routine_var`
- `c_func.ptr.all.all.all`

Note that Ada child packages must be specified by their `expanded_name`, not the direct name which is just the child portion of the name; i.e. “parent.child”, not “child”.

Note the lack of `file_scope`s in most of the `expanded_name`s shown above. Although specification of a `file_scope` is always allowed, in the above examples, it is required only for the file-level `static` variable `c_static_var` in the C source file `c_source.c` (because file-level `static` variables are not globally visible) and the variable `int_var` within the FORTRAN subroutine `sub` (because another identifier `sub` appears in the program and is globally visible).

Note:

The GNU Fortran compiler does not describe common blocks in its debug information. Attempts to locate variables using the common block syntax shown above will fail on programs built with the GNU Fortran compiler. Individual components of common blocks can be located by omitting the common block name and enclosing ’/ characters.

Note:

The GNU Fortran compiler generates mangled names in its debug descriptions. Most variables and functions are named with one ro two trailing underscores. Attempts to locate variables using the simple name supplied in the source code will often fail.

2
MAXAda Interface

This chapter presents the MAXAda `Real_Time_Data_Monitoring` package. This package provides you with a flexible interface to the key features of Data Monitoring. It contains subprograms that allow you to specify executable programs that contain Ada, C, or FORTRAN variables to be monitored, obtain lists of eligible variables that can be monitored, obtain and modify the values of selected variables by specifying their names, and obtain such information about the variables as their virtual addresses, types, and sizes

The `Real_Time_Data_Monitoring` package and compilation environment is bundled and shipped with the MAXAda product. Access to the subprograms in the `Real_Time_Data_Monitoring` package is granted to user's MAXAda compilation environments via the command:

```
/usr/ada/bin/a.path -a rtdm
```

The specification of the package can be found in "**`/usr/ada/default/rtdm/rtm.a`**".

Organization

In the sections that follow, all of the Data Monitoring subprograms contained in the MAXAda `Real_Time_Data_Monitoring` package are grouped and presented according to their functionality. For each subprogram, the following information is provided:

- A description of the subprogram or subprograms
- The Ada declarations
- Detailed descriptions of each parameter
- Conditions upon which errors can occur

Procedures for compiling and linking user programs are presented in "Compilation and Linking Instructions" on page A-1 in Appendix A.

To perform Data Monitoring, you may use either of two methods for invoking the subprograms from an application. Figure 2-1 illustrates the first method and shows the order in which you might invoke the subprograms.

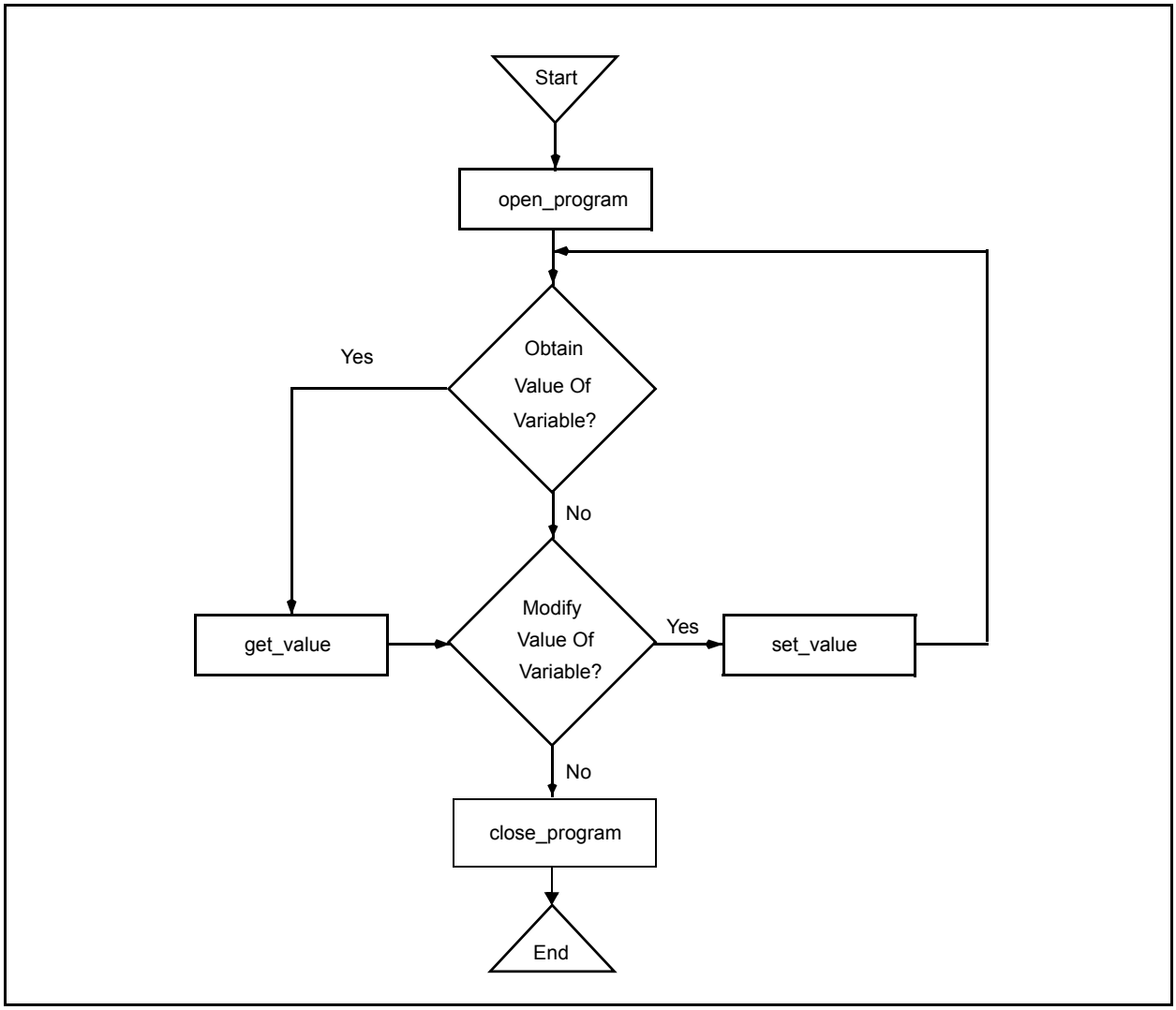

Figure 2-1. MAXAda Data Monitoring Call Sequence: Method 1

With the method illustrated by Figure 2-1, you specify the name of a target variable on each call to `get_value` and `set_value`. On each invocation of `get_value` and `set_value`, the following operations occur:

- The target program's symbol table is searched for the specified variable.
- The type, size, shape, and address of the variable are obtained.
- A mapping is created between the monitoring process's virtual address space and the final address of the target variable.
- The value of the variable is obtained or modified.

For time-critical applications, it is recommended that the second method be used, which is illustrated by Figure 2-2.

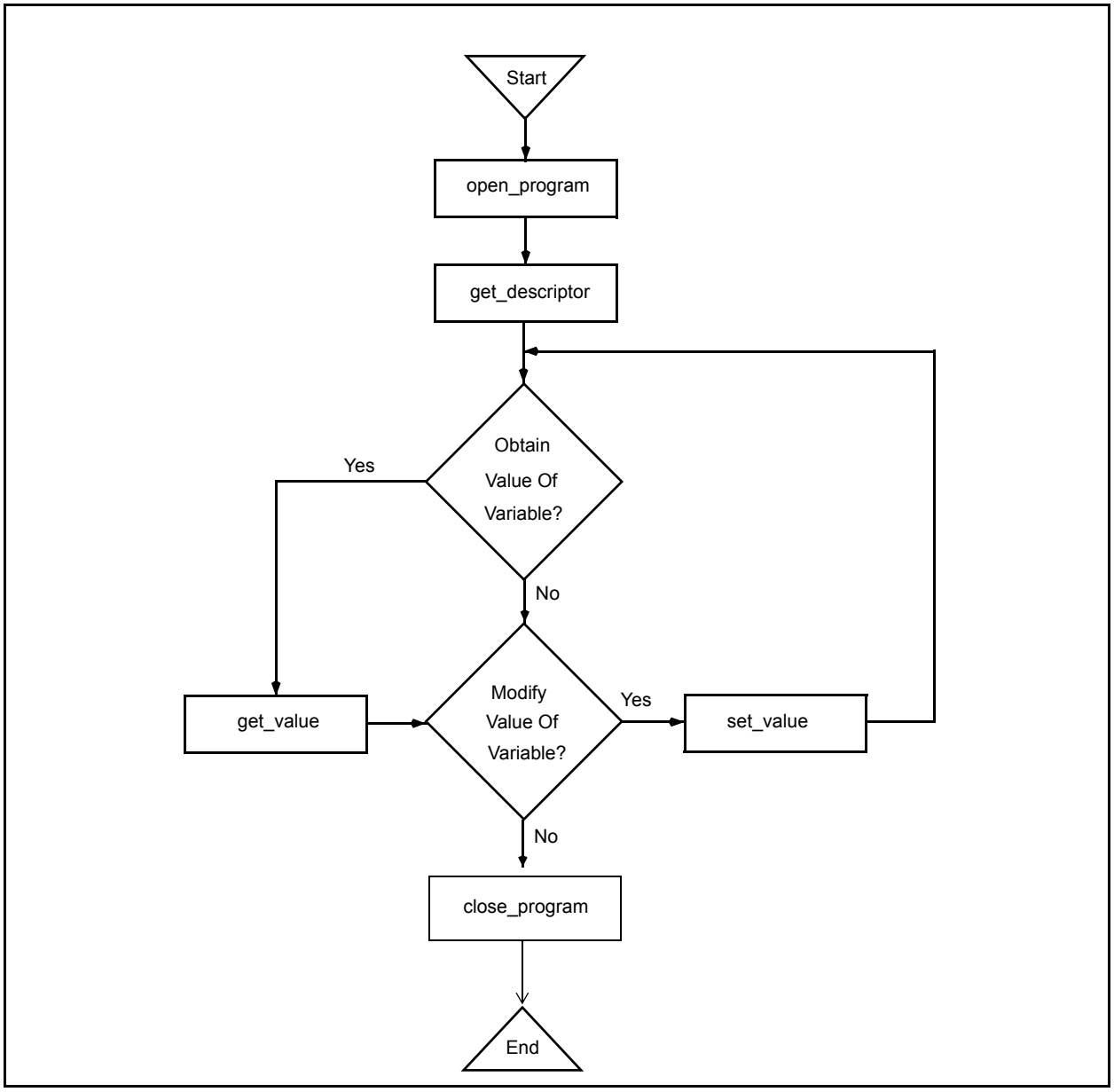

Figure 2-2. MAXAda Data Monitoring Call Sequence: Method 2

With the method illustrated by Figure 2-2, you first obtain the internal descriptors for the target variables whose values you wish to obtain or modify; subsequently, you specify an internal descriptor on each call to `get_value` or `set_value`. Obtaining the internal descriptors requires a considerable amount of time. For time-critical applications, it is recommended that you invoke `get_descriptor` during application initialization and then use the resultant descriptor(s) on subsequent `get_value` and `set_value` calls during the time-critical sections of your monitoring application.

An additional consideration with this method is that at the time of the `get_descriptor` call, the size, shape, type, and address of the specified variable are frozen; subsequent uses of the returned descriptor will utilize the frozen information, even if the actual variable

underwent subsequent size, shape, type, or address changes. See the description of Obtaining Internal Descriptors for Variables on page 2-18 for more information.

Error Processing

When a call to one of the `Real_Time_Data_Monitoring` subprograms fails, the following steps are performed:

- The error code for the last failure associated with the current subprogram call is recorded.

  When available, a description of the error is also recorded. This description may include a system call, an `errno` value, or other information that is specific to the parameters supplied on the subprogram call.

- The exception `real_time_monitoring_error` is raised.

Both the error code and the description of the error can be retrieved as shown by the Ada declarations related to error processing. These declarations, which are provided in the file **`/usr/ada/default/rtdm/rtm.a`**, are as follows:

```
real_time_monitoring_error      : exception ;

type error_codes is (
RTME_NOMEM,     --  Insufficient program memory for operation
RTME_EXCEPT,    --  Exception raised during operation
RTME_BADENUM,   --  Illegal or unexpected enumeration literal/value
RTME_SYNTAX,    --  Illegal char. in expanded name or expression
RTME_NODWARF,   --  Insufficient debug information (DWARF) available
RTME_NOTVAR,    --  Specified name is not a variable or named constant
RTME_DYNAMIC,   --  Object has dynamic size, shape, or address
RTME_NOTRECORD, --  Object is not a record, structure, or common block
RTME_NOTARRAY,  --  Object is not an array
RTME_NOTFOUND,  --  Could not find package, module, var., or component
RTME_RANGE,     --  Specified value/subscript is out-of-range for type
RTME_BADDIM,    --  Insufficient or extra subscripts for array
RTME_NOELF,     --  Unrecognized/Illegal ELF object file format
RTME_BADPID,    --  Invalid (or missing) pid for file using shared libs
RTME_USRMAP,    --  usermap(3C) failed to map process; bad pid?
RTME_SYMBOLS,   --  Insufficient symbol table information for operation
RTME_BADDWARF,  --  Unexpected/illegal/missing debug (DWARF)information
RTME_AMBIG,     --  Specified identifier is ambiguous
RTME_SERVICE,   --  System/library service call failed
RTME_NAME2BIG,  --  Expanded name too long
RTME_NOTOPEN,   --  open_program call skipped or was unsuccessful
RTME_NOFILE,    --  Could not open specified program file
RTME_BADPROG,   --  Bad program descriptor specified
RTME_BADDESC,   --  Bad object descriptor specified
RTME_UNSUP,     --  Unsupported (or unsupported type for) operation
RTME_COMPOSIT,  --  Composite type/object not allowed for operation
RTME_BUF2SMALL, --  User-specified buffer too small
RTME_NOBITS,    --  Operation requires byte-aligned types
RTME_BADREG     --  Illegal regular expression
) ;

function get_real_time_monitoring_error return string ;

function get_real_time_monitoring_error_code return error_codes ;
```

Invoke the `get_real_time_monitoring_error_code` function to obtain an enumeration value that indicates the type of error that has occurred. Invoke the `get_real_time_monitoring_error` function to obtain a string that more fully describes the error that has occurred.

A set of examples that demonstrates use of the `Real_Time_Data_Monitoring` package is provided in *Appendix A - MAXAda Examples*. Included in the examples are: (1) the Ada source code for a simple target program, (2) the Ada source code for the monitoring program, (3) the instructions for compiling and linking the target program, and (4) sample output from the example programs.

Package Types and Objects

This section describes type and object declarations that are defined and used by the `Real_Time_Data_Monitoring` package. *Descriptors* presents declarations for descriptors and constants that represent objects that the `Real_Time_Data_Monitoring` package manipulates. *Enumerations* presents declarations for types that help interpret the type and image of variables.

Descriptors

The following declarations define descriptors and constants that represent objects that the `Real_Time_Data_Monitoring` package manipulates.

```
type program_descriptor is private ;
current_program : constant program_descriptor ;

type internal_descriptor is private ;
```

`program_descriptor` — a private type that is used to represent a distinct target program or process. Information within this type is not directly visible to the user. A `program_descriptor` is created by `open_program`, destroyed by `close_program`, and consulted by several other subprograms (see pages 2-8 and 2-11 for explanations of `open_program` and `close_program`, respectively).

`current_program` — a pseudo constant that always represents the current program. Normally the current program is the `program_descriptor` that has most recently been created via `open_program` and has not yet been destroyed via `close_program`. It is supplied as a default parameter to several subprograms; thus, for applications that operate only on a single target program at once, it is not necessary to specify a `program_descriptor` on calls to most subprograms.

`internal_descriptor` — a private type that is used to represent a distinct target variable associated with a distinct target program or process. It contains type, size, and address information about the target variable. An internal_descriptor is created by `get_descriptor` and is used by several subprograms. It holds sufficient information to make subsequent modification or reference of the associated target variable very efficient.

Enumerations

The following type and object declarations aid in interpreting the type and image of variables.

```
type enumeration_image_case is (lower_case, upper_case) ;
enumeration_case : enumeration_image_case := lower_case ;

type codes is (
   code_enumeration,
   code_float,
   code_fixed,
   code_integer,
   code_record,
   code_array,
   code_char,
   code_pointer,
   code_complex,
   code_common,
   code_unknown) ;

type atomic_types is (
   discrete_1byte_signed,
   discrete_2byte_signed,
   discrete_4byte_signed,
   discrete_1byte_unsigned,
   discrete_2byte_unsigned,
   discrete_4byte_unsigned,
   fixed_1byte,
   fixed_2byte,
   fixed_4byte,
   float_4byte,
   float_8byte,
   aggregate_record,
   aggregate_array,
   complex_8_byte,
   complex_16_byte) ;
```

`enumeration_image_case` a type that defines the choices available for the ASCII representation of enumerated types

`enumeration_case` a variable that defines the current choice for the ASCII representation of enumerated types. It controls the case of enumeration images returned by the `get_value` subprogram. It does not affect the translation of user-supplied enumeration images; all such translations are done in a case-insensitive manner (e.g. an enumeration constant supplied by the user as an array index value in an expanded name).

`codes` a type that presents the categories of language-defined types for a variable. A variable's code and atomic type aid in interpreting the bits associated with the variable. Codes are as follows:

`code_enumeration`	Ada or C enumerated types
`code_float`	Floating point types
`code_fixed`	Ada fixed point types
`code_integer`	Integer types
`code_record`	Ada record or C structure types
`code_array`	Array types.
`code_char`	Ada character, C `char`, and FORTRAN `character`
`code_pointer`	Ada access types, C pointer types
`code_complex`	FORTRAN complex types
`code_common`	FORTRAN common blocks
`code_unknown`	Reserved for unrecognized types

`atomic_types` — a type that presents the list of low-level machine types associated with a variable. A variable's atomic type and code aid in interpreting the bits associated with the variable—for example, a typical 32-bit signed integer has an atomic type of `discrete_4byte_signed`.

Target Program Selection and Identification

This section presents the subprograms that allow you to (1) specify the target program for Data Monitoring, (2) obtain and close a program descriptor, (3) obtain and change the current program descriptor, and (4) obtain information about a program descriptor.

Open_Program – Obtaining Program Descriptors

This subprogram is invoked to specify the target program for Data Monitoring. You must invoke `open_program` prior to invoking any other subprogram in the `Real_Time_Data_Monitoring` package. Subsequent calls to `get_descriptor` to obtain an internal descriptor for a target variable require an open program descriptor. Internal descriptors that you have obtained following a previous `open_program` call continue to be valid; you may use them to obtain or modify the values of the target variables with which they are associated.

The `open_program` call requires that portions of the target program file be read from disk into memory and that an internal symbol table be built. These procedures can use significant amounts of memory; the amounts used depend upon the size of the target program and the number of variables that can be monitored. You are advised not to invoke `open_program` from time-critical sections of your application.

Ada Declarations

```
procedure open_program (
   program_name           : in string ;
   pid                    : in integer := 0 ;
   in_same_address_space  : in boolean := false ;
   interest_threshold     : in integer := 0) ;

function open_program (
   program_name           : in string ;
   pid                    : in integer := 0 ;
   in_same_address_space  : in boolean := false ;
   interest_threshold     : in integer := 0)
   return program_descriptor ;
```

Parameters

program_name — refers to a string that contains a standard UNIX path name identifying the target program file in which the variables are found. A full or relative path name of up to 1024 characters can be specified.

pid — refers to an integer value representing the process identification number of the target executable program specified by the *program_name* parameter. If the value of *pid* is 0, then `open_program` will attempt to locate a process that is executing on the system with the specified path name. If successful, the corresponding process identification number of that process is used, otherwise, it is as if an invalid value for *pid* has been specified.

Under specific conditions, the value of *pid* may be specified as -1. In this case, the target program does not need to be executing. These conditions are as follows: 1) the target program is statically linked (that is, it does not contain any shared libraries); 2) the variables of interest have static addresses, sizes, and shapes; and 3) subsequent use of `Real_Time_Data_Monitoring` subprograms is confined to one or more of the following:

- `info_only`
- `get_type_name`
- `get_array_info`
- `get_constraints`
- `list.list`
- `list.global_list`
- `get_real_time_monitoring_error`
- `get_real_time_monitoring_eror_code`
- `open_program`
- `close_program`

Use of modes involving interpretation of class-wide variables (see `set_class_interpretation` page 2-15) and active record variants (see `set_variant_handling` page 2-14) are also prohibited if the target program is not executing.

in_same_address_space refers to a boolean flag that indicates whether or not the `Real_Time_Data_Monitoring` package is being executed in the same application as that containing the variables whose values are to be obtained or modified. The default value for this flag is `false`. If the monitoring process and the target process are the same (that is, the monitoring is done within the target process), set the flag to `true`. In this case, the overhead of address space mapping is avoided.

interest_threshold refers to an integer value which specifies the interest threshold for the specified target program. The default value for this setting is **`0`**. All eligible variables have an interest value which is set by their compiler. By default, all eligible variables have an interest value of zero. The Ada compiler allows the user to set the interest value of selected variables via the implementation-defined pragma INTERESTING. (See Annex M of the *MAXAda Reference Manual* (0890516) for more information on pragma INTERESTING). The interest threshold controls whether an otherwise eligible variable is visible to the subprograms in the `Real_Time_Data_Monitoring` package. If the interest value of a variable is below the interest threshold it is as if the variable did not exist. The interest threshold may also be set via the `set_interest_threshold` subprogram (see page 2-13).

Return Value

The function form of the `open_program` subprogram returns the newly-created program descriptor. For either form, the `current_program` becomes the newly-created program descriptor.

Error Conditions

When an error is detected, the exception `real_time_monitoring_error` is raised. Possible error conditions include the following:

- The file associated with *program_name* could not be located or opened for read.
- The specified *pid* was a value other than -1 and did not identify an executing process.
- The specified *pid* was -1 but the target program associated with *program_name* requires shared libraries.
- The specified *pid* was 0 but no target process associated with *program_name* could be located.

- The file associated with *program_name* is not a valid ELF executable file.
- The file associated with *program_name* contains no symbolic information.

Close_Program – Closing Program Descriptors

This subprogram is invoked to free internal storage that is being used to hold symbolic information associated with the specified program descriptor. After making this call, you may not call any other subprograms with the specified program descriptor. Internal descriptors for target variables that have already been obtained via calls to `get_descriptor`, however, are still valid—for example, `get_value` and `set_value` operations can still occur using those descriptors.

Ada Declarations

```
procedure close_program ;
procedure close_program (program : program_descriptor);
```

Parameters

(null) The subprogram form without an argument refers to the `current_program`.

program refers to a program descriptor that has been returned from a previous call to `open_program` and has not yet been closed

Error Conditions

When an error is detected, the exception `real_time_monitoring_error` is raised. Possible error conditions include the following:

- No parameter is specified, and there is no valid `current_program`
- *Program* is not a valid, open program descriptor

Get_Current_Program – Referencing the Current Program

This subprogram is invoked to obtain the program descriptor that is represented by the `current_program`. The `current_program` represents the program descriptor associated with the last valid `open_program` or `set_current_program` call if the descriptor has not been closed since the call.

This subprogram is rarely used since all subprograms which require a program descriptor have a default value associated with that formal parameter which specifies the `current_program`. It is only provided because the constant `current_program` is really just a marker which abstractly represents the "current program"; the actual value of that constant is not a valid `program_descriptor`.

Ada Declaration

```
function get_current_program return program_descriptor ;
```

Return Values

The program descriptor associated with the last valid `open_program` or `set_current_program` call is returned if the descriptor has not been closed since the call.

Error Conditions

When an error is detected, the exception `real_time_monitoring_error` is raised. Possible error conditions include the following:

- There is no valid `current_program`

Set_Current_Program – Changing the Current Program Descriptor

This subprogram is invoked to associate a previously obtained program descriptor with `current_program`.

Ada Declaration

```
procedure set_current_program
   (program : in program_descriptor) ;
```

Parameters

program refers to a program descriptor that has been returned from a previous call to `open_program` and has not yet been closed

Error Conditions

When an error is detected, the exception `real_time_monitoring_error` is raised. Possible error conditions include the following:

- *Program* is not a valid, open program descriptor

Info_Program – Obtaining Information from a Program Descriptor

This subprogram returns basic information about a specified program descriptor including the program name and process identification number.

This subprogram is useful for identifying the target program associated with a specific target variable when used in conjunction with the `action` call-back routine in `list` operations as described in “Scanning Target Programs for Variables” on page 2-39.

Ada Declarations

```
procedure info_program (
   program           : in program_descriptor :=
                           current_program
   program_name      : out string ;
   program_name_last : out natural ;
```

```
   program_pid       : out integer) ;

function info_program (
   program : in program_descriptor := current_program)
   return string ;
```

Parameters

program refers to a program descriptor that has been returned on a previous call to `open_program` and has not yet been closed (see page 2-8 for an explanation of this subprogram)

program_name upon return, is set to the path name that was specified on the `open_program` call corresponding to *program*

program_name_last upon return, is set to the last element of *program_name* modified by this call

program_pid upon return, is set to the process identification number of the process corresponding to *program*

Return Values

The function form returns the path name as previously specified on the call to `open_program` corresponding to *program*.

Error Conditions

When an error is detected, the exception `real_time_monitoring_error` is raised. Possible error conditions include the following:

- *Program* is not a valid, open program descriptor
- The size of *program_name* is insufficient to hold the path name corresponding to *program*.

Set_Interest_Threshold – Setting the Interest Threshold

An interest threshold refers to an integer value which controls the visibility of target variables. The default value for this setting is **`0`**, unless explicitly set via the *interest_threshold* parameter to the `open_program` subprogram. All eligible variables have an interest value which is set by their compiler. By default, all eligible variables have an interest value of zero. The Ada compiler allows users to change the interest value of selected variables via the implementation-defined pragma INTERESTING. (See Annex M of the *MAXAda Reference Manual* (0890516) for more information on pragma INTERESTING). The interest threshold controls whether an otherwise eligible variable is visible to the subprograms in the `Real_Time_Data_Monitoring` package. If the interest value of a variable is below the interest threshold, it is as if the variable did not exist. Once set, the interest threshold remains associated with the specified target program until reset by a subsequent `set_interest_threshold` call.

Note that subsequent changes to the interest threshold have no effect on internal descriptors already obtained by previous `get_descriptor` calls.

Ada Declaration

```
procedure set_interest_threshold (
   interest_threshold : in integer ;
   program            : in program_descriptor :=
                           current_program);
```

Parameters

interest_threshold refers to an integer value which will be the new interest threshold for the target program corresponding to *program*

program refers to a program descriptor that has been returned on a previous call to `open_program` and has not yet been closed (see page 2-8 for an explanation of this subprogram)

Error Conditions

When an error is detected, the exception `real_time_monitoring_error` is raised. Possible error conditions include the following:

- *Program* is not a valid, open program descriptor

Set_Variant_Handling – Setting Ada Record Variant Sensitivity

The `set_variant_handling` routine defines the mode in which Ada record variants are handled. By default, the *active_variants_only* mode is set to `false`; thus look-up and `list` subprograms within the `Real_Time_Data_Monitoring` package are not sensitive to a record variant's governing discriminant, inasmuch as all variants are considered active at all times. Setting the *active_variants_only* mode to `true` will cause look-up and `list` subprograms within this package to determine the value of an enclosing record variant's governing discriminant when considering components within the record (see section 3.8.1(2-21) of the *Ada 95 Reference Manual* for more information on Ada record variants). In general, this sensitivity requires that the target program be executing, because the value of discriminants must be obtained from the target process. If *active_variants_only* mode is `true` and a component of a record is contained in an inactive variant, it is as if the component did not exist. The *active_variants_only* mode has no effect on C or FORTRAN variables.

If this mode is set to `true` and subsequent calls to subprograms within this package require the value of discriminants from the target program and those values are in memory and the target program is not executing, those subprogram calls will fail as described subsequently in this chapter. The setting of the *active_variants_only* mode is associated with the specified target program and remains in effect until a subsequent call to `set_variant_handling`.

Note that subsequent changes to the *active_variants_only* mode have no effect on internal descriptors which have already been obtained via a previous `get_descriptor` call.

Ada Declaration

```
procedure set_variant_handling (
   active_variants_only : in boolean ;
```

```
program              : in program_descriptor :=
                         current_program)
```

Parameters

active_variants_only — refers to a boolean value which controls the handling of variants for Ada records for the target program corresponding to *program.* Setting the value to `true` will cause sensitivity to record variant's governing discriminants as described above. Setting the value to `false` causes all variants to be considered active.

program — refers to a program descriptor that has been returned on a previous call to `open_program` and has not yet been closed (see page 2-8 for an explanation of this subprogram)

Error Conditions

When an error is detected, the exception `real_time_monitoring_error` is raised. Possible error conditions include the following:

- *Program* is not a valid, open program descriptor

Set_Class_Interpretation – Interpreting Class-Wide Types

The `set_class_interpretation` routine sets the *interpret_classes* mode for the specified target program. This mode controls the interpretation of values of variables of Ada class-wide types. By default, the *interpret_classes* mode is `false`. Thus values of variables of class-wide types are interpreted using the specific type of the root of the class-wide type (see section 3.4.1(3-5) of the *Ada 95 Reference Manual* for more information on Ada class-wide types). If the mode is set to `true`, then values of variables of class-wide types are interpreted using the specific type associated with the actual value of the variable. In general, setting the *interpret_classes* mode to `true` requires that the target program be executing, because the value of the variable's *tag* (see section 3.9 of the *Ada 95 Reference Manual* for more information on *tags* and *type extensions*) is required to find the specific type covered by the root of the class-wide type.

Consider the following example:

```
package p is
   type t is
      record
         x : integer ;
      end record ;
   type e is new t with
      record
         y : integer ;
      end record ;
   object_t : t'class := t'(x => 4) ;
   object_e : t'class := e'(x => 1, y => 2) ;
end p ;
```

In the table below, the first column represents the string passed to look-up subprograms such as `get_descriptor` and `get_value`. The second and third columns represent whether such calls would succeed, based on the specified setting of the *interpret_classes* mode:

String Descriptor	interpret_classes mode	
	false	**true**
`"p.object_t.x"`	succeed	succeed
`"p.object_t.y"`	fail	fail
`"p.object_e.x"`	succeed	succeed
`"p.object_e.y"`	fail	succeed

Of course the example in the second row, `"p.object_t.y"`, isn't very interesting since the value of that class-wide variable really is of type `"t"` and therefore doesn't have a component named `"y"`. However, the example in the fourth row, `"p.object_e.y"` demonstrates the point of the *interpret_classes* mode; since the value of that class-wide actually is of type `"e"`, a type extended from the specific type of the root of the class-wide type, it does contain a component called `"y"`.

Ada Declaration

```
procedure set_class_interpretation (
   interpret_classes : in boolean ;
   program           : in program_descriptor :=
                          current_program);
```

Parameters

interpret_classes refers to a boolean value which controls the interpretation of values of variables of Ada class-wide types for the target program corresponding to *program.* Setting the value to `true` will cause the specific type of the value of the variable to be based on the actual value of the variable. Setting the value to `false` will cause the specific type of the value of the variable to be obtained directly from the specific type of the root of the class-wide type.

program refers to a program descriptor that has been returned on a previous call to `open_program` and has not yet been closed (see page 2-8 for an explanation of this subprogram)

Error Conditions

When an error is detected, the exception `real_time_monitoring_error` is raised. Possible error conditions include the following:

- *Program* is not a valid, open program descriptor

Obtaining Internal Descriptors for Variables

To obtain the value of a target variable or to modify a target variable, information about the variable must be located from the target program file. Such information includes the variable's type, size, shape, and address. This information is collected and stored in an internal descriptor. Part of the process of obtaining an internal descriptor involves creating a memory mapping between the target variable and the monitoring process's virtual address space; memory mapping makes subsequent access to target variables from the monitoring process extremely efficient. After the internal descriptor for a variable has been defined, `get_value` and `set_value` operations can occur (see pages 2-22 and 2-25, respectively, for explanations of these subprograms).

The `Real_Time_Data_Monitoring` package provides several forms of the `get_value` and `set_value` operations. For ease of use, all of these forms allow you to specify the target variable in one of the following ways:

- By specifying a string describing the *expanded name* of the target variable

  or

- By specifying an internal descriptor that has been obtained from a previous call to `get_descriptor` on which you have supplied a string describing the *expanded name* of the target variable (see page 2-18 for an explanation of this subprogram)

In the first case, the routines first obtain an internal descriptor via a hidden call to `get_descriptor`. After the `get_value` or `set_value` operation, that internal descriptor is discarded (no storage space is lost). In the second case, the operation is completed more quickly because you have already obtained the internal descriptor.

Another advantage of explicitly obtaining an internal descriptor is that the lifetime of the descriptor exceeds that of its corresponding program descriptor; that is, the program descriptor associated with the program containing the target variable may be closed (thereby freeing significant memory associated with target program symbol tables), but the internal descriptors remain valid.

Note that when you obtain an internal descriptor for a variable, its size, shape, type, and address are frozen— for example, if the variable involves pointer indirection (`ptr.all`), the value of the `ptr` at the time of the call to `get_descriptor` is used to determine the final address of the `ptr.all`. Subsequent calls to `get_value` or `set_value` with the

resultant internal descriptor will refer to the address calculated during the `get_descriptor` call, regardless of the current value of the `ptr`. If you wish to re-evaluate the address of the `ptr.all` considering the current value of `ptr`, then call `get_descriptor` again, or call `get_value` and `set_value` with an explicit variable name (that is, `"ptr.all"`) rather than an internal descriptor. This applies not only to variables involving pointer indirection, but records whose size and shape can change as the target process executes, as well as variables of class-wide types.

Get_Descriptor – Obtaining an Internal Descriptor

This subprogram is invoked to obtain an internal descriptor for a specified variable. The amount of time required to obtain the descriptor may be significant for applications with stringent performance constraints.

Ada Declarations

```
function get_descriptor (
   string_descriptor : in string ;
   no_addr_translate : in boolean := false ;
   program           : program_descriptor := current_program)
   return internal_descriptor ;

procedure get_descriptor (
   string_descriptor : in string ;
   descriptor        : out internal_descriptor ;
   no_addr_translate : in boolean := false ;
   program           : program_descriptor := current_program)

function get_descriptor (
   address_descriptor : in system.address ;
   code               : in codes ;
   atomic_type        : in atomic_types ;
   bit_size           : in natural ;
   bit_offset         : in natural ;
   no_addr_translate  : in boolean := false ;
   program            : program_descriptor:=current_program)
   return internal_descriptor ;

procedure get_descriptor (
   address_descriptor : in system.address ;
   code               : in codes ;
   atomic_type        : in atomic_types ;
   bit_size           : in natural ;
   bit_offset         : in natural ;
   descriptor         : out internal_descriptor ;
   no_addr_translate  : in boolean := false ;
   program            : program_descriptor:=current_program);
```

Parameters

string_descriptor refers to a string that contains the *expanded name* of the target variable for which you wish to obtain the internal descriptor

descriptor refers to the internal descriptor returned by the subprogram. The function forms of this subprogram supply *descriptor* as the return value.

no_addr_translate refers to a boolean flag that indicates whether or not address translation (mapping) is to occur. The default value for this flag is `false`. When the flag is set to `false`, the monitoring process's virtual address space is to be mapped to the target variable. Set the flag to `true` if the target variable is already accessible at the same virtual address in the monitoring process as in the target process (for example, a variable in a shared memory segment attached at a common address). This flag is ignored if the *is_same_address_space* parameter to the `open_program` call corresponding to *program* was set true; thus no address translation occurs.

program refers to a valid program descriptor that has been returned from a previous call to `open_program` (see page 2-8 for an explanation of this subprogram). If this parameter is not specified, the `current_program` is supplied.

address_descriptor refers to an address in the target process that is to be interpreted as a target variable with the attributes specified by the *code, atomic_type, bit_size,* and *bit_offset* parameters. This parameter allows you to obtain and modify anonymous memory locations in the target process.

code identifies the memory location(s) associated with *address_descriptor*. Examples of the values that you may specify are `code_float`, `code_integer`, and `code_record`. For additional information, refer to *"Enumerations"* (page 2-7).

atomic_type identifies the atomic type of memory location(s) associated with *address_descriptor*. Examples of the values you may specify are `discrete_1byte_signed` and `discrete_4byte_unsigned`. For additional information, refer to *"Enumerations"* (page 2-7).

bit_size identifies the bits composing the anonymous target variable starting at *address_descriptor* + *bit_offset*.

bit_offset identifies the first bit of the anonymous target variable by specifying the bit offset from the byte specified by *address_descriptor*. Bit offsets are numbered from zero to seven, where zero is the most significant bit within a byte.

Error Conditions

When an error is detected, the exception `real_time_monitoring_error` is raised. Possible error conditions include the following:

- *Program* does not refer to a valid, open program descriptor.
- *String_descriptor* does not refer to an eligible variable.
- *Descriptor* is not a valid internal descriptor.

- The specified variable could not be found in the target program's symbol tables (perhaps the user forgot to compile with the debug (`-g`) option).
- *String_descriptor* contains invalid *expanded name* syntax.
- The target program is not executing and *string_descriptor* refers to a variable with a dynamic size, shape, address or requires a value from the target process due to modes *active_variants_only* or *interpret_classes* (see pages 2-14 and 2-15).
- The target variable could not be mapped into the monitoring process's address space; an `errno` value associated with the offending **`usermap(3rt)`** call is included in the text of the message associated with a subsequent call to `get_real_time_monitoring_error`.

Invalidate_Descriptor – Invalidating an Internal Descriptor

This subprogram is provided as a convenience. It is invoked to invalidate a specified internal descriptor. After an internal descriptor has been invalidated, subsequent use of it will cause an error.

Ada Declaration

```
procedure invalidate_descriptor
   (descriptor : in out internal_descriptor) ;
```

Parameters

descriptor Refers to an internal descriptor that you wish to invalidate

Error Conditions

This subprogram does not have any error conditions.

Is_Valid_Descriptor – Checking Internal Descriptor Validity

This subprogram is provided as a convenience. It is invoked to determine whether or not a specified internal descriptor is valid. An internal descriptor is valid if it has been obtained via a call to `get_descriptor` (see page 2-18 for an explanation of this subprogram) and has not been invalidated via a subsequent call to `invalidate_descriptor`.

Ada Declaration

```
function is_valid_descriptor
   (descriptor : in internal_descriptor) return boolean ;
```

Parameters

descriptor refers to an internal descriptor whose validity you wish to check

Return Values

The value `true` is returned if *descriptor* corresponds to a valid internal descriptor; otherwise, the value `false` is returned.

Error Conditions

This subprogram does not have any error conditions.

Is_Active_Component – Active Variant Checking

This function is provided as a convenience; it is invoked to determine if a specific component is nominally contained within a specific record variable and, if contained within a variant, that the variant is active. The preferred method is to initially call the `set_variant_handling` subprogram (page 2-14) to set the *active_variants_only* mode to `true` such that look-up and `list` operations on records will disregard components in inactive variants outright.

Ada Declaration

```
function is_active_component (
   string_descriptor : string ;
   program           : program_descriptor := current_program)
   return boolean ;
```

Parameters

string_descriptor refers to a string that contains the *expanded name* of a component of a target variable (for example, *package_p.record_item.component*)

program refers to a valid program descriptor that has been returned from a previous call to `open_program` (see page 2-8 for an explanation of this subprogram). If this parameter is not specified, the `current_program` is supplied.

Return Value

This function returns `true` if the specified component exists in the record; which implies that it is not contained in an inactive variant; otherwise, this function return `false`. The current setting of the *active_variants_only* mode (see page 2-14) has no actual effect on this function. Regardless of the setting of that mode, the value of the governing discriminants of any variants within the record will be obtained in order to determine if the specified component is active. If the value of any governing discriminant of the enclosing record is in memory, use of this function requires the target program to be executing.

Error Conditions

When an error is detected, the exception `real_time_monitoring_error` is raised. Possible error conditions include the following:

- *Program* does not refer to a valid, open program descriptor.
- *Program* was omitted, and there is no valid, open current program descriptor.
- *String_descriptor* contains invalid *expanded name* syntax.

- *Program* does not specify an executing process and *string_descriptor* refers to a variable with a dynamic size, shape, address or requires a value from the target process due to *interpret_classes* mode (see page 2-15).
- Governing discriminants exist for the enclosing record and their values are in memory and *program* does not specify an executing process.

Obtaining or Modifying Target Variables

This section describes the subprograms that allow you to obtain or modify the values of target variables. As explained in "Get_Descriptor – Obtaining an Internal Descriptor" on page 2-18, most of these subprograms accept the specification of the target variable in one of the following ways:

- By specifying a string describing the *expanded name* of the target variable

  or

- By specifying an internal descriptor that has been obtained from a previous call to `get_descriptor` on which you have supplied a string describing the *expanded name* of the target variable (see page 2-18 for an explanation of this subprogram)

`Get_value` allows you to obtain the value of a variable. `Set_value` (page 2-25) allows you modify the value of a variable. `Validate_value` (page 2-27) allows you to verify that a user-supplied ASCII representation of the value of a variable is appropriate for that variable. The `io` package (page 2-28) allows you to read and modify the values of complex variables.

Get_Value – Obtaining the Value of Variables

This subprogram is invoked to obtain the value of a target variable.

The default ASCII representation used by `get_value` depends upon the type of the variable:

signed integer	the C `printf` "**`%d`**" conversion format
unsigned integer, pointers	the C `printf` "**`16#%-08.8x#`**" conversion format
floating point	the C `printf` "**`%g`**" conversion format
fixed point (Ada)	the C `printf` "**`%g`**" conversion format
enumeration (Ada)	the enumeration image in lower case unless the `enumeration_case_image` variable in the Real_Time_Data_Monitoring package is set to `upper_case`.

Ada Declaration

```
function get_value (
   string_descriptor : string ;
   no_addr_translate : boolean := false ;
   program           : program_descriptor := current_program)
   return string ;

procedure get_value (
   string_descriptor : in string ;
   object_value      : out string ;
   object_last       : out natural ;
   no_addr_translate : in boolean := false ;
   program           : program_descriptor := current_program)

procedure get_value (
   string_descriptor : in string ;
   address_to_store  : in system.address ;
   bytes_at_address  : in natural ;
   no_addr_translate : in boolean := false ;
   program           : program_descriptor := current_program)

function get_value (
   object_descriptor : in internal_descriptor)
   return string ;

procedure get_value (
   object_descriptor : in internal_descriptor ;
   object_value      : out string ;
   object_last       : out natural) ;

procedure get_value (
   object_descriptor : in internal_descriptor ;
   address_to_store  : in system.address ;
   bytes_at_address  : in natural) ;
```

Parameters

string_descriptor refers to a string that contains the *expanded name* of the target variable (for example, *package_p.data_item*) for which you wish to obtain the value. You may specify this parameter or the *object_descriptor* parameter.

object_descriptor refers to an internal descriptor associated with the target variable for which you wish to obtain the value. You can obtain this descriptor by making a call to `get_descriptor` (see page 2-18 for an explanation of this subprogram). You may specify this parameter or the *string_descriptor* parameter.

no_addr_translate refers to a boolean flag that indicates whether or not address translation (mapping) is to occur. The default value for this flag is `false`. When the flag is set to `false`, the monitoring process's address space is to be mapped to the target variable. Set the flag to `true` only if the target variable is already accessible at the same virtual address in the monitoring process as in the target process. This

parameter can be specified only for subprograms that require a *string_descriptor*. This flag is ignored if the *is_same_address_space* parameter to the `open_program` call corresponding to *program* was set true; thus no address translation occurs.

program refers to a valid program descriptor that has been returned from a previous call to `open_program` (see page 2-8 for an explanation of this subprogram). If this parameter is not specified, the `current_program` is supplied. This parameter can be specified only for subprograms that require a *string_descriptor*.

address_to_store refers to an address within the monitoring process's address space at which the subprogram is to place the raw value of the target variable. This value will be right justified in the memory range *address_to_store .. address_to_store* + *bytes_at_address* -1.

bytes_at_address refers to the number of bytes of space that you have reserved to hold the raw value of the target variable. The raw value of the target variable will be right justified in the memory range *address_to_store .. address_to_store* + *bytes_at_address* -1.

object_value upon return, contains the ASCII representation of the value of the specified target variable

object_last upon return, identifies the last string element that has been set in the *object_value* parameter

Return Values

The function forms of this subprogram return a string that contains the ASCII representation of the value of the specified target variable.

Error Conditions

When an error is detected, the exception `real_time_monitoring_error` is raised. Possible error conditions include the following:

- A *string_descriptor* was specified and *program* does not refer to a valid, open program descriptor.
- A *string_descriptor* was specified; *program* was omitted; and there is no valid, open current program descriptor.
- *String_descriptor* does not refer to an eligible variable.
- A *string_descriptor* was specified and the target variable it references could not be found in the target program's symbol tables (perhaps the user forgot to compile with the debug (`-g`) option).
- *String_descriptor* contains invalid *expanded name* syntax.
- The target program is not executing and *string_descriptor* refers to a variable with a dynamic size, shape, address or requires a value from the target process due to modes *active_variants_only* or *interpret_classes* (see pages 2-14 and 2-15).

- The target variable could not be mapped into the monitoring process's address space; an `errno` value associated with the offending **`usermap(3rt)`** call is included in the text of the message associated with a subsequent call to `get_real_time_monitoring_error`.
- *Object_descriptor* is not a valid internal descriptor.
- The type of the target variable represented by *string_descriptor* or *object_descriptor* is a composite type (array, record, or structure). The generic `io` package may be used for obtaining the value of such variables.
- The type of the target variable represented by *string_descriptor* or *object_descriptor* is unknown (for example, `code_unknown`).

Set_Value – Setting the Value of Variables

This subprogram is invoked to modify the value of a target variable.

The default ASCII representation expected by `set_value` depends upon the type of the variable:

signed integer	the C `sscanf` "**%d**" conversion format
unsigned integers, pointers	the C `sscanf` "**%d**" conversion format
floating point	the C `sscanf` "**%g**" conversion format
fixed point (Ada)	the C `sscanf` "**%g**" conversion format
enumeration (Ada)	the enumeration image in upper or lower case

Ada Declarations

```
procedure set_value (
   string_descriptor : in string ;
   value_in_ascii    : in string ;
   no_addr_translate : in boolean := false ;
   program           : program_descriptor := current_program)

procedure set_value (
   string_descriptor : in string ;
   address_of_value  : in system.address ;
   bytes_at_address  : in positive ;
   no_addr_translate : in boolean := false ;
   program           : program_descriptor := current_program)

procedure set_value (
   object_descriptor : in internal_descriptor ;
   value_in_ascii    : in string) ;

procedure set_value (
   object_descriptor : in internal_descriptor ;
   address_of_value  : in system.address ;
```

```
   bytes_at_address  : in positive) ;
```

Parameters

string_descriptor refers to a string that contains the *expanded name* of the target variable (for example, *package_p.data_item*) whose value you wish to modify. You may specify this parameter or the *object_descriptor* parameter.

object_descriptor refers to an internal descriptor associated with the target variable whose value you wish to modify. You can obtain this descriptor by making a call to `get_descriptor` (see page 2-18 for an explanation of this subprogram). You may specify this parameter or the *string_descriptor* parameter.

no_addr_translate refers to a boolean flag that indicates whether or not address translation (mapping) is to occur. The default value for this flag is `false`. When the flag is set to `false`, the monitoring process's address space is to be mapped to the target variable. Set the flag to `true` only if the target variable is already accessible at the same virtual address in the monitoring process as in the target process. This parameter can be specified only for subprograms that require a *string_descriptor*. This flag is ignored if the *is_same_address_space* parameter to the `open_program` call corresponding to *program* was set true; thus no address translation occurs.

program refers to a valid program descriptor that has been returned from a previous call to `open_program` (see page 2-8 for an explanation of this subprogram). If this parameter is not specified, the `current_program` is supplied. This parameter can be specified only for subprograms that require a *string_descriptor*.

value_in_ascii refers to a string that contains the ASCII representation of the new value of the target variable as specified by *string_descriptor* or *object_descriptor*. The value must be expressed in a form that is consistent with the type of the target variable (for example, an integer literal for an integer type, a floating point literal for a floating point type, and so on). The value must be within the range of the type of the target variable. You may specify this parameter or the *address_of_value* parameter.

address_of_value refers to a variable that specifies the address of the first byte of the set of storage locations that holds the raw value that will be used to modify the target variable. The address specified must be in the monitoring process's virtual address space. The value must be right justified in the memory range *address_of_value .. address_of_value* + *bytes_at_address* -1. You may specify this parameter or the *value_in_ascii* parameter.

bytes_at_address refers to a variable that contains an integer value indicating the number of bytes that compose the raw value starting at the address specified by *address_of_value*. This parameter may be specified only by subprograms that require the *address_of_value* parameter.

Error Conditions

When an error is detected, the exception `real_time_monitoring_error` is raised. Possible error conditions include the following:

- A *string_descriptor* was specified and *program* does not refer to a valid, open program descriptor.
- A *string_descriptor* was specified; *program* was omitted; and there is no valid, open current program descriptor.
- *String_descriptor* does not refer to an eligible variable.
- A *string_descriptor* was specified and the target variable it references could not be found in the target program's symbol tables (perhaps the user forgot to compile with the debug (**`-g`**) option).
- *String_descriptor* contains invalid *expanded name* syntax.
- The target program is not executing and *string_descriptor* refers to a variable with a dynamic size, shape, address or requires a value from the target process due to modes *active_variants_only* or *interpret_classes* (see pages 2-14 and 2-15).
- The target variable could not be mapped into the monitoring process's address space; an `errno` value associated with the offending **`usermap(3rt)`** call is included in the text of the message associated with a subsequent call to `get_real_time_monitoring_error`.
- *Object_descriptor* is not a valid internal descriptor.
- The type of the target variable represented by *string_descriptor* or *object_descriptor* is a composite type (array, record, or structure). The generic `io` package may be used for modifying such variables.
- The type of the target variable represented by *string_descriptor* or *object_descriptor* is unknown (for example, `code_unknown`).
- The value as specified by *value_in_ascii* has an inappropriate form for the type of the target variable.
- The value as specified by *value_in_ascii* is out of range for the type of the target variable.

Validate_Value – Verifying an ASCII Representation

This subprogram is invoked to verify that a user-supplied ASCII representation of the value of a variable is of an appropriate form for the variable's type.

The default ASCII representation used by `validate_value` depends upon the type of the variable:

signed integer	the C `sscanf` "**%d**" conversion format
unsigned integer, pointers	the C `sscanf` "16#**%d**" conversion format
floating point	the C `sscanf` "**%g**" conversion format

fixed point (Ada)	the C `sscanf` "**%g**" conversion format
enumeration (Ada)	the enumeration image in upper or lower case

Use of this subprogram is optional. You may wish to use it to ensure that a subsequent call to `set_value` in a time-critical section does not incur the overhead of exception handling for errors resulting from specifying an inappropriate ASCII representation (see page 2-25 for an explanation of the `set_value` subprogram).

Ada Declaration

```
procedure validate_value (
   object_descriptor  : in internal_descriptor ;
   value_in_ascii     : in string ;
   is_valid           : out boolean) ;
```

Parameters

object_descriptor	refers to an internal descriptor that is associated with the target variable whose proposed value (that is, *value_in_ascii*) you wish to validate
value_in_ascii	refers to a string that contains the ASCII representation of the value that you wish to validate
is_valid	upon return, contains a boolean value that indicates whether or not *value_in_ascii* is of the correct form and range for the variable's type. The value of this parameter is set to `false` if the value of the variable is out of range or is of the wrong form; otherwise, it is set to `true`.

Error Conditions

An invalid value specified by *value_in_ascii* is not an error condition; however, information obtained on a subsequent call to `get_real_time_monitoring_error` will indicate why the value is invalid. When an error condition is detected, the exception `real_time_monitoring_error` is raised. Possible error conditions include the following:

- *Object_descriptor* is not a valid internal descriptor.

IO Package – Generic Read and Write of Variables

The `io` package is nested within the `Real_Time_Data_Monitoring` package. It contains subprograms that read and write arbitrarily complex target variables based on internal descriptors.

This package allows you to read or write composite variables on a single invocation of a subprogram. Note that the `get_value` and `set_value` subprograms cannot operate on target variables with composite types (see pages 2-22 and page 2-25, respectively, for explanations of these subprograms).

Ada Declaration

```
generic

   type variable_type is private ;

package io is

   procedure read (
      address     : in internal_descriptor ;
      value       : out variable_type ;
      byte_offset : in natural := 0) ;

   procedure write (
      address     : in internal_descriptor ;
      value       : in variable_type ;
      byte_offset : in natural := 0) ;

end io ;
```

Parameters

variable_type refers to a user-defined type that is supplied during an instantiation of the generic package. It is meant to represent the type of target variables whose values are to be read or written via subsequent calls to `read` or `write` subprograms within the instantiation being defined.

address refers to an internal descriptor that specifies the target variable of interest

value refers to a user-defined variable or expression of type *variable_type* in the monitoring process's address space. For `read` calls, the value of the target variable will be placed in *value* upon return. For `write` calls, the target variable will be updated with the supplied *value*.

byte_offset refers to a non-negative integer value that is added to the virtual base address found in the internal descriptor before the read or write operation begins. This offset must not exceed the address range of the variable as defined by the *address*. The value of *byte_offset* defaults to zero.

Error Conditions

When an error is detected, the exception `real_time_monitoring_error` is raised. Possible error conditions include the following:

- *Address* is not a valid internal descriptor.
- *Byte_offset*, when added to the base address of the variable defined by *address*, will exceed the bit size of the variable as defined by *address*.

Obtaining Information about Variables

This section presents the subprograms that may be invoked to obtain information about a specified target variable. The information that can be obtained includes the virtual address, atomic type, code, bit size, bit offset, array shape and component information, type name, and constraints.

`Get_info` and `info_only` allow you to obtain such information as the following: the virtual address of the variable in the monitoring process's address space and in the target process's address space; the atomic type of the variable; the bit size and bit offset. `Get_array_info` (page 2-32) allows you to obtain information about an array variable. `Get_type_name` (page 2-33) allows you to obtain information about the type of a target variable. `Get_constraints` (page 2-38) allows you to obtain constraint information about a target variable.

Get_Info and Info_Only – Obtaining Information about Variables

Ada Declarations

```
procedure get_info (
   string_descriptor : in string ;
   virtual_address   : out system.address ;
   target_address    : out system.address ;
   atomic_type       : out atomic_types ;
   bit_size          : out natural ;
   bit_offset        : out natural ;
   code              : out codes ;
   program          : program_descriptor := current_program)

procedure get_info (
   object_descriptor : in internal_descriptor ;
   virtual_address   : out system.address ;
   target_address    : out system.address ;
   atomic_type       : out atomic_types ;
   bit_size          : out natural ;
   bit_offset        : out natural ;
   code              : out codes) ;

procedure info_only (
   string_descriptor : in string ;
   target_address    : out system.address ;
   atomic_type       : out atomic_types ;
   bit_size          : out natural ;
   bit_offset        : out natural ;
   code              : out codes ;
   program          : program_descriptor := current_program)
```

Parameters

string_descriptor refers to a string that contains the *expanded name* of the target variable (for example, *package_p.data_item*) for which you wish to obtain information. You may specify this parameter or the *object_descriptor* parameter.

object_descriptor refers to an internal descriptor associated with the target variable for which you wish to obtain information. You may specify this parameter or the *string_descriptor* parameter.

program refers to a valid program descriptor that has been returned from a previous call to `open_program` (see page 2-8 for an explanation of this subprogram). If this parameter is not specified, the `current_program` is supplied. This parameter can be specified only for subprograms that require a *string_descriptor*.

virtual_address upon return, contains the address of the first byte of the contiguous memory locations that hold the target variable in the monitoring process's address space. Note that normally, the address returned is not the location of the variable in the target process's address space. This parameter is not available on the `info_only` subprogram; the `info_only` subprogram does not create a mapping between the monitoring process and the target process (therefore it is generally not necessary for the target program to be executing).

target_address upon return, contains the address of the first byte of the contiguous memory locations that hold the target variable in the target process's address space. Note that normally, the address returned is not the location of the variable in the monitoring process's address space.

atomic_type upon return, contains the enumeration value that indicates the atomic type of the specified target variable

bit_size upon return, contains the size in bits of the specified target variable

bit_offset upon return, contains the bit offset from the first byte that is returned in the *virtual_address* parameter

Error Conditions

When an error is detected, the exception `real_time_monitoring_error` is raised. Possible error conditions include the following:

- A *string_descriptor* was specified and *program* does not refer to a valid, open program descriptor.
- A *string_descriptor* was specified; *program* was omitted; and there is no valid, open current program descriptor.
- *String_descriptor* does not refer to an eligible variable.
- A *string_descriptor* was specified and the target variable it references could not be found in the target program's symbol tables (perhaps the user forgot to compile with the debug (**`-g`**) option).
- *String_descriptor* contains invalid *expanded name* syntax.

- The target program is not executing and *string_descriptor* refers to a variable with a dynamic size, shape, address or requires a value from the target process due to modes *active_variants_only* or *interpret_classes* (see pages 2-14 and 2-15).
- The target variable could not be mapped into the monitoring process's address space; an `errno` value associated with the offending **`usermap(3rt)`** call is included in the text of the message associated with a subsequent call to `get_real_time_monitoring_error`.
- *Object_descriptor* is not a valid internal descriptor.

Get_Array_Info – Obtaining Array Bounds and Component Info

Ada Declarations

```
type indicies is
   record
      lower_bound : integer ;
      upper_bound : integer ;
   end record ;
type indicies_list is array (1..10) of indicies ;

procedure get_array_info (
   object_descriptor   : in internal_descriptor ;
   component_bit_size  : out natural ;
   component_code      : out codes ;
   component_signed    : out boolean ;
   indicies            : out indicies_list ;
   dimensions          : out positive) ;
```

Parameters

object_descriptor refers to an internal descriptor associated with the target variable for which you wish to obtain information

component_bit_size upon return, contains the size in bits of the component type of the array specified by *object_descriptor*

component_code upon return, contains the `code` associated with the component type of the array specified by *object_descriptor*

component_signed upon return, contains the value `true` if the component type of the array specified by *object_descriptor* has a signed representation; otherwise, it contains the value `false`.

indicies upon return, contains integer values that represent the lower and upper bounds of each dimension of the array variable specified by *object_descriptor*. Components of *indicies* that correspond to dimensions not present in the array variable specified by *object_descriptor* are left undefined. If *object_descriptor* refers to an array that has more than 10 dimensions, the lower and upper bounds of only the first 10 dimensions are returned.

dimensions upon return, contains the number of dimensions of the array specified by *object_descriptor*

Error Conditions

When an error is detected, the exception `real_time_monitoring_error` is raised. Possible error conditions include the following:

- *Object_descriptor* is not a valid internal descriptor.
- *Object_descriptor* does not refer to an array.

Get_Type_Name – Obtaining Variable Type Names

Ada Declaration

```
function get_type_name (
   string_descriptor : string ;
   program           : program_descriptor := current_program;
   expanded_name     : boolean := false ;
   interpret_classes : boolean := false) return string ;
```

Parameters

string_descriptor refers to a string that contains the *expanded name* of the target variable (for example, *package_p.data_item*) whose type name you wish to obtain

program refers to a valid program descriptor that has been returned from a previous call to `open_program` (see page 2-8 for an explanation of this subprogram). If this parameter is not specified, the `current_program` is supplied.

expanded_name refers to a boolean flag which controls whether the name of the type associated with the variable identified by *string_descriptor* is expressed in Ada's *expanded name* notation. The default value for this flag is `false`. When `true`, type names are preceded by the *expanded name* of their enclosing scope (e.g. "pkg.type_t"); whereas the direct name of the type is used when the flag is `false` (e.g. "type_t"). This parameter has no effect for C or FORTRAN variables.

interpret_classes refers to a value which controls the interpretation of the type of values of variables of Ada class-wide types. The default value for this setting is `false`. When `false`, the type name is obtained using the name of the specific type (suffixed by 'class) of the root of the class-wide type of the variable specified by *string_descriptor*. When `true`, the type is chosen using the specific type associated with the value of the variable specified by string_descriptor. When *interpret_classes* is set to `true`, the target program must be executing. The setting of *interpret_classes* on this subprogram call overrides the *interpret_classes* mode which is set via a call to `set_class_interpretation` (see page 2-15). For example,

using the code fragment from the example of `set_class_interpretation` on page 2-15, a call such as `get_type_name("pkg.object_e")` would return "t'class", whereas a call such as `get_type_name("pkg.object_e",interpret_classes=>true)` would return "e".

Return Value

This subprogram returns a string that describes the type of the target variable specified by *string_descriptor*. For Ada variables, this string consists of the direct name of the type of the target variable; this name may be a user-defined type name or a language-defined type name. For C and FORTRAN variables, a name that represents the type of the variable is returned. Examples are as follows:

int * var_1	get_type_name returns "int *"
void (*var_2)()	get_type_name returns "void (*)()"
typedef int xxx; xxx var_3	get_type_name returns "xxx"
struct {...} var_4 ;	get_type_name returns "<struct>"
integer*4 fortran_variable	get_type_name returns "integer*4"

Error Conditions

When an error is detected, the exception `real_time_monitoring_error` is raised. Possible error conditions include the following:

- *Program* does not refer to a valid, open program descriptor.
- *Program* was omitted and there is no valid, open current program descriptor.
- *String_descriptor* does not refer to an eligible variable.
- The target variable referenced by *string_descriptor* could not be found in the target program's symbol tables (perhaps the user forgot to compile with the debug (**`-g`**) option).
- *String_descriptor* contains invalid *expanded name* syntax.
- The target program is not executing and *string_descriptor* refers to a variable with a dynamic size, shape, address or requires a value from the target process due to modes *active_variants_only* (see page 2-14) or the *interpret_classes* parameter.

Get_Enum_Image – Obtaining Images of Enumeration Constants

The `get_enum_image` subprogram is invoked to obtain the image of the enumeration literal that corresponds to a specified position within the enumerated type associated with a variable in a target program.

Ada Declaration

```
function get_enum_image (
   string_descriptor : string ;
   enum_position     : natural ;
   program           : program_descriptor := current_program)
   return string ;
```

Parameters

string_descriptor refers to a string that contains the *expanded name* of the target variable (for example, *package_p.data_item*) whose type is the enumerated type of interest. The specified variable is required only to identify its type; the value of the variable is not used (unless portions of the variable's value are required to satisfy *active_variants_only* or *interpret_classes* modes; see pages 2-14 and 2-15).

enum_position refers to a variable that contains a non-negative integer value that identifies the position of interest in the enumerated type associated with the variable specified by *string_descriptor*. A value of zero indicates the first position in the enumerated type.

The position and value of a literal of an enumerated type are typically the same unless an explicit enumeration representation clause has been specified for the type. For example:

```
type colors is (red, white, blue);
type more_colors is (x, y, z) ;
for more_colors use (x => 5,y => 10, z => 20) ;
```

The position and value of the literal `white` are both 1, whereas the position and value of the literal `y` are 1 and 10, respectively.

The `get_enum_image` service expects a position, not a value.

program refers to a valid program descriptor that has been returned from a previous call to `open_program` (see page 2-8 for an explanation of this subprogram). If this parameter is not specified, the `current_program` is supplied.

Return Values

The image of the enumeration literal corresponding to *enum_position* for the enumerated type associated with the specified target variable is returned.

Error Conditions

When an error is detected, the exception `real_time_monitoring_error` is raised. Possible error conditions include the following:

- A *string_descriptor* was specified and *program* does not refer to a valid, open program descriptor.
- A *string_descriptor* was specified; *program* was omitted; and there is no valid, open current program descriptor.
- *String_descriptor* does not refer to an eligible variable.

- A *string_descriptor* was specified and the target variable it references could not be found in the target program's symbol tables (perhaps the user forgot to compile with the debug (**-g**) option).
- *String_descriptor* contains invalid *expanded name* syntax.
- The target program is not executing and *string_descriptor* refers to a variable with a dynamic size, shape, address or requires a value from the target process due to modes *active_variants_only* or *interpret_classes* (see pages 2-14 and 2-15).
- The target variable could not be mapped into the monitoring process's address space; an `errno` value associated with the offending **`usermap(3rt)`** call is included in the text of the message associated with a subsequent call to `get_real_time_monitoring_error`.
- The type of the variable specified by *string_descriptor* is not an enumerated type.
- The position specified by *enum_position* is illegal for the enumerated type; perhaps a value was supplied instead of a position.

Get_Enum_Val – Obtaining Values of Enumeration Constants

The `get_enum_val` subprogram is invoked to obtain the value, as opposed to the image, of the enumeration literal that corresponds to a specified position within the enumerated type associated with a variable in a target program.

Ada Declaration

```
function get_enum_val (
   string_descriptor : string ;
   enum_position     : natural ;
   program           : program_descriptor := current_program)
   return integer ;
```

Parameters

string_descriptor refers to a string that contains the *expanded name* of the target variable (for example, *package_p.data_item*) whose type is the enumerated type of interest. The specified variable is required only to identify its type; the value of the variable is not used (unless portions of the variable's value are required to satisfy *active_variants_only* or *interpret_classes* modes; see pages 2-14 and 2-15).

enum_position refers to a variable that contains a non-negative integer value that identifies the position of interest in the enumerated type associated with the variable specified by *string_descriptor*. A value of zero indicates the first position in the enumerated type.

The position and value of a literal of an enumerated type are typically the same unless an explicit enumeration representation clause has been specified for the type. For example:

```
type colors is (red, white, blue);
type more_colors is (x, y, z) ;
for more_colors use (x => 5,y => 10, z => 20) ;
```

The position and value of the literal `white` are both 1, whereas the position and value of the literal `y` are 1 and 10, respectively.

The `get_enum_val` service expects a position, not a value.

program refers to a valid program descriptor that has been returned from a previous call to `open_program` (see page 2-8 for an explanation of this subprogram). If this parameter is not specified, the `current_program` is supplied.

Return Values

The value of the enumeration literal corresponding to *enum_position* for the enumerated type associated with the specified target variable is returned.

Error Conditions

When an error is detected, the exception `real_time_monitoring_error` is raised. Possible error conditions include the following:

- A *string_descriptor* was specified and *program* does not refer to a valid, open program descriptor.
- A *string_descriptor* was specified; *program* was omitted; and there is no valid, open current program descriptor.
- *String_descriptor* does not refer to an eligible variable.
- A *string_descriptor* was specified and the target variable it references could not be found in the target program's symbol tables (perhaps the user forgot to compile with the debug (**`-g`**) option).
- *String_descriptor* contains invalid *expanded name* syntax.
- The target program is not executing and *string_descriptor* refers to a variable with a dynamic size, shape, address or requires a value from the target process due to modes *active_variants_only* or *interpret_classes* (see pages 2-14 and 2-15).
- The target variable could not be mapped into the monitoring process's address space; an `errno` value associated with the offending **`usermap(3rt)`** call is included in the text of the message associated with a subsequent call to `get_real_time_monitoring_error`.
- The type of the variable specified by *string_descriptor* is not an enumerated type.
- The position specified by *enum_position* is illegal for the enumerated type; perhaps a value was supplied instead of a position.

Get_Constraints – Obtaining Constraints of Scalar Variables

The `get_constraints` subprogram is invoked to obtain constraint information about a variable specified by a *string_descriptor* or *object_descriptor.*

Ada Declarations

```
procedure get_constraints (
   string_descriptor : in string ;
   lower_bound       : out long_float ;
   upper_bound       : out long_float ;
   program          : program_descriptor := current_program)

procedure get_constraints (
   (object_descriptor : in internal_descriptor ;
    lower_bound       : out long_float ;
    upper_bound       : out long_float) ;
```

Parameters

string_descriptor — refers to a string that contains the *expanded name* of the target variable (for example, *package_p.data_item*) for which you wish to obtain information. You may specify this parameter or the *object_descriptor* parameter.

object_descriptor — refers to an internal descriptor associated with the target variable for which you wish to obtain information. You may specify this parameter or the *string_descriptor* parameter.

program — refers to a valid program descriptor that has been returned from a previous call to `open_program` (see page 2-8 for an explanation of this subprogram). If this parameter is not specified, the `current_program` is supplied. This parameter can be specified only for subprograms that require a *string_descriptor.*

lower — upon return, holds the lower bound of the constraints of the variable specified by *string_descriptor* or *object_descriptor.* The lower bound is expressed as a floating point number. For variables with enumerated types, the value represents the `pos` of the base type (that is, it is always zero). For variables whose type is not scalar, this value is undefined.

upper — upon return, holds the upper bound of the constraints of the variable specified by *string_descriptor* or *object_descriptor.* The upper bound is expressed as a floating point number. For variables with enumerated types, the value represents the `pos` of the base type. For variables whose type is not scalar, this value is undefined.

Error Conditions

When an error is detected, the exception `real_time_monitoring_error` is raised. Possible error conditions include the following:

- A *string_descriptor* was specified and *program* does not refer to a valid, open program descriptor.

- A *string_descriptor* was specified; *program* was omitted; and there is no valid, open current program descriptor.
- *String_descriptor* does not refer to an eligible variable.
- A *string_descriptor* was specified and the target variable it references could not be found in the target program's symbol tables (perhaps the user forgot to compile with the debug (**-g**) option).
- *String_descriptor* contains invalid *expanded name* syntax.
- The target program is not executing and *string_descriptor* refers to a variable with a dynamic size, shape, address or requires a value from the target process due to modes *active_variants_only* or *interpret_classes* (see pages 2-14 and 2-15).
- The target variable could not be mapped into the monitoring process's address space; an `errno` value associated with the offending **`usermap(3rt)`** call is included in the text of the message associated with a subsequent call to `get_real_time_monitoring_error`.
- *Object_descriptor* is not a valid internal descriptor.

Scanning Target Programs for Variables

The generic `lists` package provides subprograms that traverse the internal symbol tables of target program files and call a user-specified procedure for each item in a list. The list is formed by examining the symbol tables in relation to a set of requirements that has been defined by parameters specified on each call to a subprogram within an instantiation of the `lists` package.

Generic Package Lists – Listing Scopes, Variables, and Components

Ada Declarations

```
type list_position is private ;

type list_mode is (list_scopes,
                   list_variables,
                   list_components) ;

function list_packages return list_mode renames list_scopes ;

generic
   with procedure action (item      : in string ;
                          program   : in program_descriptor ;
                          position  : in out list_position ;
                          quit      : in out boolean) ;
package lists is
   procedure list
```

```
      (mode        : list_mode ;
       qualifier   : string := "" ;
       restriction : string := "" ;
       components  : boolean := false ;
       program     : program_descriptor := current_program);

   procedure global_list
      (mode        : list_mode ;
       qualifier   : string := "" ;
       restriction : string := "" ;
       components  : boolean := false) ;
end lists ;
```

Dynamic Semantics

The procedures `list` and `global_list` differ in only one respect: `global_list` searches all currently open program descriptors while `list` searches only the specified (or `current_program`) program descriptor.

The list of items is formed by examining the symbol tables of target programs in relation to the requirements specified by the *qualifier*, list *mode*, and optional regular expression *restriction* parameters to the `list` and `global_list` subprograms.

If the program associated with a list item candidate is not currently executing, then list item candidates with dynamic addresses, sizes, or shapes may fail to qualify for the list and may be excluded from it.

For each item in the list, a call is made to the user-defined *action* procedure.

The list *mode* defines the class of objects being considered during the search:

`list_scopes`	defines the class of objects to be scopes. Examples are Ada packages, C subprograms that contain static data, and FORTRAN subprograms.
`list_variables`	defines the class of objects to be variables
`list_components`	defines the class of objects to be components of composite variables

Parameters to List and Global_list

mode refers to a value of type `list_mode`

qualifier refers to a string that is interpreted in accordance with the specified *mode*. By default, *qualifier* is a null string. *Qualifier* is interpreted as follows:

`list_scopes` *Qualifier* should specify the name of a scope or a null string. If a null string is specified, all scopes are considered; otherwise, only the scopes that are immediately contained within the scope specified by *qualifier* are considered. Note that an Ada child package is considered to be a global scope with an *expanded name* such

as “parent.child”; it is not considered to be a scope within “parent”.

`list_variables` Q*ualifier* should specify the name of a package or other scope. If a null string is specified, all scopes are considered.

`list_components` *Qualifier* should specify the *expanded name* of a composite variable.

restriction refers to a string that forms a valid regular expression as defined by **`regexpr(3G)`** on PowerMAX OS and **`regexec(5)`** on RedHawk Linux. It is used to restrict the list elements. The regular expression is applied to the *expanded name* of the list item as it will be passed to the user-defined *action* procedure. The *restriction* is applied as the last step in forming the elements of the list. By default, *restriction* is null, which indicates there is no restriction.

components refers to a boolean flag that indicates whether or not components of a variable are to be listed in *list_variables* mode. This flag is ignored for all other list modes. If `true`, components of composite variables are included in the list; otherwise, they are not. Note that the list of components formed is significantly affected by the settings of the *active_variants_only* and *interpret_classes* modes as described on pages 2-14 and 2-15.

program refers to a valid program descriptor that has been returned from a previous call to `open_program` (see page 2-8 for an explanation of this subprogram). If this parameter is not specified, the `current_program` is supplied. This parameter can be specified only for the `list` procedure.

Error Conditions for List and Global_list

When an error is detected, the exception `real_time_monitoring_error` is raised. Possible error conditions include the following:

- The specified *program* is not a valid, open program descriptor, or it was omitted and there are no valid, open program descriptors.
- The specified *restriction* is not null and is an invalid regular expression as defined by **`regexpr(3G)`**.
- An exception is propagated from the call to the user-defined *action* procedure.

Note that it is not an error to specify parameters that result in the formation of an empty list—that is, `list` and `global_list` return without calling the user-defined *action* procedure.

Parameters to the User-Defined Action Procedure

item refers to a string describing the *expanded name* of the item. *Item* is a scope name (for example, Ada package, FORTRAN subprogram), a variable, or a component of a variable.

program refers to the program descriptor associated with *item*

position refers to a value of a private type that describes the current position in the list. The *action* routine may store into this in-out parameter a previous list position value that resets the specified list position (that is, the next call to *action* will pass the item associated with the changed value of *position*).

quit refers to a boolean flag that indicates whether or not list processing should continue; this value is always set to `false` on entry to the *action* procedure. If you set this in-out parameter to `true`, list processing will stop upon return from the current *action* call; otherwise, list processing continues.

3
C Interface

The Data Monitoring library, **`/usr/lib/libdatamon.a`**, contains C interfaces that allow you to monitor variables in executing processes. These interfaces allow you to specify executable programs that contain Ada, C, or FORTRAN variables to be monitored; obtain lists of eligible variables that can be monitored; obtain and modify the values of selected variables; and obtain such information about the variables as their virtual addresses, types, and sizes. Interfaces that allow you to obtain and modify values are of two types: those that accept and return values expressed in symbolic formats that are appropriate for the respective variables and those that accept and return values without symbolic formatting.

Organization

This chapter provides all of the information that you need to use the C Data Monitoring interfaces. *"Types and Objects"* (page 3-1) describes type and variable declarations that are used by the C interfaces to Data Monitoring. *"Error Processing"* (page 3-4) presents the enumerations and subprograms which describe error conditions. The remaining sections explain the procedures for using each of the C routines in the Data Monitoring library. Examples using the C interface and instructions on *"C Compilation and Linking Instructions"* are found in Appendix B (page B-1).

Types and Objects

This section describes type and object declarations that are used by the C interfaces to Data Monitoring. *"Descriptors"* presents the types of descriptors that are used. *"Enumerations"* presents the enumerated types that are used.

Descriptors

The header file **`<datamon.h>`** declares two types of descriptors that are used by the C interfaces to Data Monitoring: a *program descriptor*, which is used to represent a specific target program or process, and an *object descriptor*, which is used to represent a specific target variable associated with a target program or process.

The program descriptor is declared as follows:

```
typedef int program_descriptor_t;
```

A descriptor of this type is created by the dm_open_program routine and destroyed by the dm_close_program routine (see pages 3-8 and 3-9, respectively, for explanations of these routines). It is also used by the dm_get_descriptor and dm_list routines (see pages 3-14 and 3-27, respectively, for explanations of these routines).

The object descriptor is declared as follows:

```
typedef struct object_descriptor {
   int       od_valid;           /* Flag: true if valid */
   int       od_atomic_type;     /* Internal data field */
   dm_codes  od_code;            /* Object code */
   void     *od_target_address;  /* Virt.addr in target program */
   void     *od_virtual_address; /* Virt.addr in this! process */
   int       od_bit_size;        /* Size in bits of object */
   int       od_bit_offset;      /* Bit offset from virt. addr */
   int       od_signed;          /* 1 if signed representation */
   int       od_extra_info1;     /* delta, image_database, n/a */
   int       od_extra_info2;     /* n/a,   val2pos_database, n/a */
   double    od_lower_bound;     /* Lower bound for scalar types */
   double    od_upper_bound;     /* Upper bound for scalar types */
   int       od_language;        /* DWARF DW_LANG_ see dwarf.h */
   dm_codes  od_component_code;  /* Valid iff od_code is array */
   int       od_component_bit_size; /* Valid iff od_code is array */
   int       od_component_signed; /* Valid iff od_code is array */
   int       od_number_dims;      /* Num of dimensions for arrays */
   int       od_lower_dims[MAX_DIMENSIONS];  /* Low bounds */
   int       od_upper_dims[MAX_DIMENSIONS];  /* Upper bounds */
} object_descriptor_t;
```

A descriptor of this type is created by the dm_get_descriptor routine (see page 3-14 for an explanation of this routine). It contains type, size, and address information about the target variable. It holds sufficient information to make subsequent modification or reference of the associated target variable very efficient. The object descriptor is used by the dm_peek, dm_poke, dm_get_value, and dm_set_value routines (see pages 3-16, 3-17, 3-18, and 3-19, respectively, for explanations of these routines).

Enumerations

The header file **<datamon.h>** also declares two enumerated types that are used by the C interfaces to Data Monitoring: dm_codes, which identifies the categories of language-defined types for a variable, and dm_list_modes, which identifies the class of objects to be considered when using the dm_list routine to scan a target program for variables (see page 3-27 for an explanation of this routine).

The dm_codes enumerated type is declared as follows:

```
typedef enum dm_codes {
   code_enumeration,
   code_float,
   code_fixed,
```

```
   code_integer,
   code_record,
   code_array,
   code_char,
   code_pointer,
   code_complex,
   code_common,
   code_unknown
} dm_codes ;
```

The `dm_codes` values are explained as follows:

`code_enumeration`	Ada or C enumerated types
`code_float`	floating point types
`code_fixed`	Ada fixed point types
`code_integer`	integer types
`code_record`	Ada record or C structure types
`code_array`	array types
`code_char`	Ada character, C `char`, and FORTRAN `character`
`code_pointer`	Ada access types, C pointer types
`code_complex`	FORTRAN complex types
`code_common`	FORTRAN common blocks
`code_unknown`	reserved for unrecognized types

A variable's `code` aids in interpreting the bits associated with the variable. The **`<datamon.h>`** header file also includes a `dm_code_images[]` array that maps the enumeration values to their corresponding enumeration images. Note that in order for this image array to be visible, the C Data Monitoring program must be compiled with the **`-Ddatamon_images`** option. See *"C Compilation and Linking Instructions"* in Appendix B (page B-1) for more information.

The `dm_list_modes` enumerated type is declared as follows:

```
typedef enum dm_list_modes {
   list_scopes,
   list_variables,
   list_components
} dm_list_modes;
```

The `dm_list_modes` values are explained as follows:

`list_scopes`	defines the class of objects to be scopes. Examples are Ada packages, C subprograms that contain static data, and FORTRAN subprograms

`list_variables`	defines the class of objects to be variables
`list_components`	defines the class of objects to be components of composite variables

Error Processing

When a call to one of the Data Monitoring subprograms fails, the following steps are typically performed:

- The error code for the last failure associated with the current subprogram call is recorded.

  When available, a description of the error is also recorded. This description may include a system call, an `errno` value, or other information that is specific to the parameters supplied on the subprogram call.

- A value of **`-1`** is returned from the subprogram.

Both the error code and the description of the error can be retrieved as shown below by the declarations related to error processing. These declarations, which are provided in the file **`<datamon.h>`**, are as follows:

```
typedef enum dm_error_codes {
   DM_NOMEM,     /* Insufficient program memory for operation */
   DM_EXCEPT,    /* Exception raised during operation */
   DM_BADENUM,   /* Illegal or unexpected enumeration literal/value */
   DM_SYNTAX,    /* Illegal char. in expanded var_name/expression */
   DM_NODWARF,   /* Insufficient debug information (DWARF) available */
   DM_NOTVAR,    /* Specified name is not a variable or constant */
   DM_DYNAMIC,   /* Object has dynamic size, shape, or address */
   DM_NOTRECORD, /* Object is not a record, structure, or common blk */
   DM_NOTARRAY,  /* Object is not an array */
   DM_NOTFOUND,  /* Could not find package/module/variable/component */
   DM_RANGE,     /* Specified value/subscript is out-of-range */
   DM_BADDIM,    /* Wrong number of subscripts specified for array */
   DM_NOELF,     /* Unrecognized/Illegal ELF object file format */
   DM_BADPID,    /* Invalid (or missing) pid for file w/ shared libs */
   DM_USRMAP,    /* usermap(3C) failed to map process; bad pid? */
   DM_SYMBOLS,   /* Insufficient sym table information for operation */
   DM_BADDWARF,  /* illegal/missing debug (DWARF) information */
   DM_AMBIG,     /* Specified identifier is ambiguous */
   DM_SERVICE,   /* System/library service call failed */
   DM_NAME2BIG,  /* Expanded name too long */
   DM_NOTOPEN,   /* dm_open_program call skipped or was unsuccessful */
   DM_NOFILE,    /* Could not open specified program file */
   DM_BADPROG,   /* Bad program descriptor specified */
   DM_BADDESC,   /* Bad object descriptor specified */
   DM_UNSUP,     /* Unsupported (or unsupported type for) operation */
   DM_COMPOSIT,  /* Composite type/object not allowed for operation */
   DM_BUF2SMALL, /* User-specified buffer too small */
   DM_NOBITS,    /* Operation requires byte-aligned types */
   DM_BADREG     /* Illegal regular expression */
} dm_error_codes ;

#ifdef datamon_mappings
static char * dm_error_code_images[] = {
...
#endif

extern
dm_error_codes
dm_get_error_code (void) ;

extern
char *
dm_get_error_string (void) ;
```

Invoke the `dm_get_error_code` function to obtain an enumeration value that indicates the type of error that has occurred. Invoke the `dm_get_error_string` function to obtain a string that more fully describes the error that has occurred. Note that the array `dm_error_code_images` maps enumeration values to their corresponding image; it is only provided when the **`-Ddatamon_images`** compilation option to the C compiler is used. See *"C Compilation and Linking Instructions"* in Appendix B (page B-1) for more information.

Routines

In the sections that follow, all of the C Data Monitoring routines contained in the **libdatamon** library are grouped and presented according to function. The following information is provided for each routine:

- The C declaration of the routine
- Detailed descriptions of each parameter
- The return value

Figure 3-1 illustrates the approximate order in which you might call the routines from an application program.

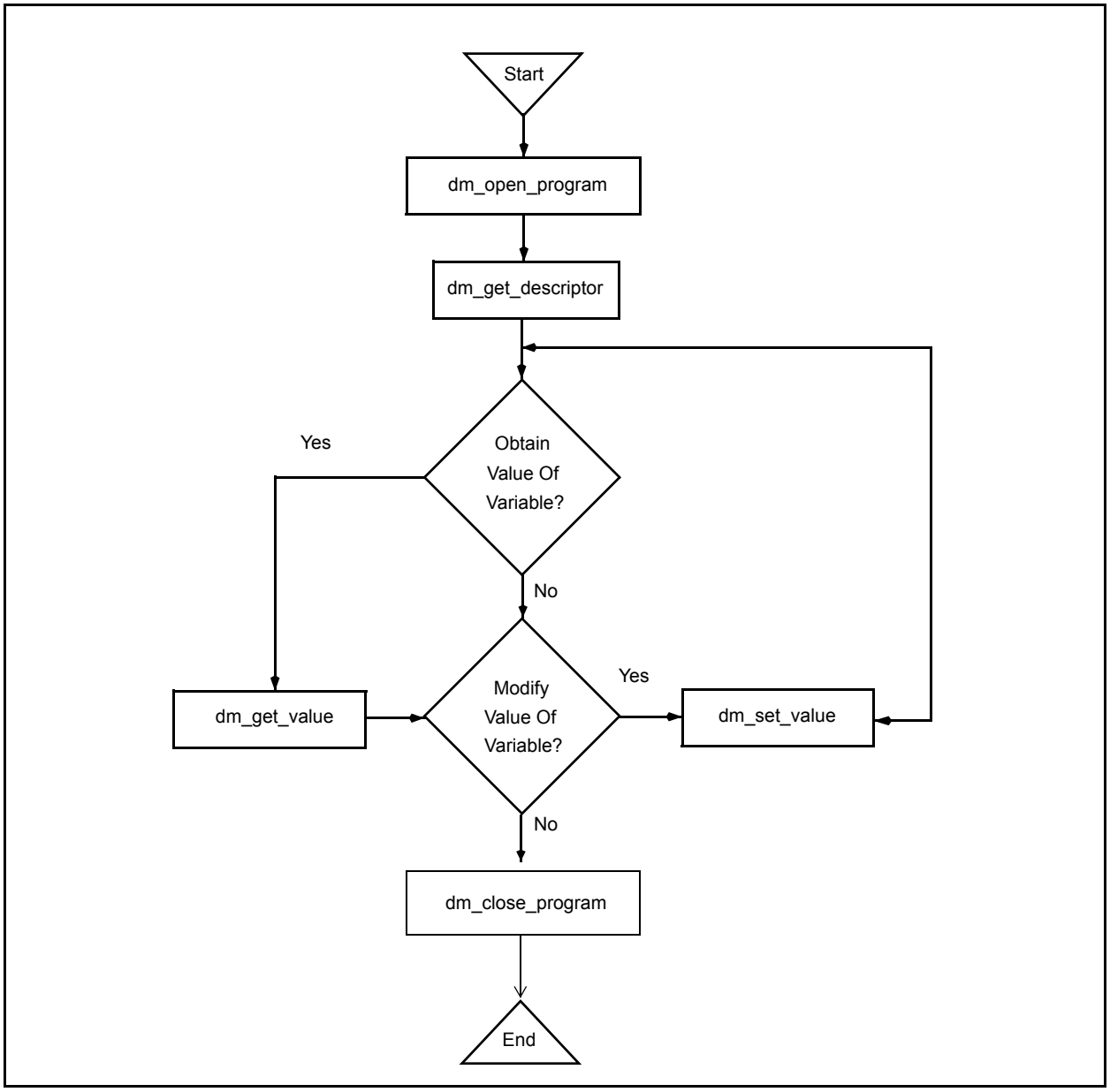

Figure 3-1. C Data Monitoring Call Sequence

With the sequence illustrated by Figure 3-1, you first obtain the object descriptors for the target variables whose values you wish to obtain or modify; subsequently, you specify an object descriptor on each call to `dm_get_value` or `dm_set_value`. Obtaining the object descriptors involves symbol table searches; it may require a significant amount of time for time-critical applications. For such applications, it is recommended that you invoke `dm_get_descriptor` during application initialization and then use the resultant descriptor(s) to invoke `dm_get_value` and `dm_set_value` during the time-critical sections of your monitoring application.

Target Program Selection and Identification

This section presents the subprograms that allow you to (1) specify the target program for Data Monitoring, (2) obtain and close a program descriptor, (3) obtain and change the current program descriptor, and (4) obtain information about a program descriptor.

Dm_Open_Program – Obtaining Program Descriptors

This routine is invoked to specify the target program for Data Monitoring. You must invoke `dm_open_program` prior to invoking any other routine in the Data Monitoring library. Subsequent calls to `dm_get_descriptor` to obtain an object descriptor for a target variable require an open program descriptor. Object descriptors that you have obtained following a previous `dm_open_program` call continue to be valid; you may use them to obtain or modify the values of the target variables with which they are associated.

The `dm_open_program` call requires that portions of the target program file be read from disk into memory and that an internal symbol table be built. These procedures can use significant amounts of memory; the amounts used depend upon the size of the target program and the number of variables that can be monitored. You are advised not to invoke `dm_open_program` from time-critical sections of your application. The memory utilized by `dm_open_program` can be reclaimed by a subsequent call to `dm_close_program`.

Declaration

```
#include <datamon.h>
extern
int
dm_open_program (char                  * pgm_name,
                 int                     pid,
                 program_descriptor_t  * pgm_desc);
```

Parameters

pgm_name points to a string that contains a standard UNIX path name identifying the program to be monitored. Note that a full or relative path name of up to 1024 characters can be specified.

pid refers to a variable that contains an integer value representing the process identification number of the target executable program specified by the *pgm_name* parameter

If the value of *pid* is 0, then `dm_open_program` will attempt to locate a process that is executing on the system with the specified path name. If successful, the corresponding process identification number of that process is used; otherwise, it is as if an invalid value for *pid* has been specified.

Under specific conditions, the value of *pid* may be specified as -1. In this case, the target program does not need to be executing. These conditions are as follows: 1) the target program is statically linked (that is, it does not

contain any shared libraries); 2) the variables of interest have static addresses, sizes, and shapes; and 3) subsequent use of Data Monitoring subprograms is confined to one or more of the following:

- `dm_get_type_name, dm_get_type_name_long`
- `dm_list`
- `dm_get_error_code`
- `dm_get_error_string`
- `dm_open_program`
- `dm_close_program`

pgm_desc points to a location to which the program descriptor is to be returned

Return Value

A return value of **`0`** indicates that the call has been successful. A return value of **`-1`** indicates that an error has occurred. Invoke `dm_get_error_code` or `dm_get_error_string` for a description of the error. Possible error conditions include the following:

- The file associated with *pgm_name* could not be located or opened for read.
- The specified *pid* was a value other than -1 and did not identify an executing process.
- The specified *pid* was -1 but the target program associated with *pgm_name* requires shared libraries.
- The specified *pid* was 0 but no target process associated with *pgm_name* could be located.
- The file associated with *pgm_name* is not a valid ELF executable file.
- The file associated with *pgm_name* contains no symbolic information.

Dm_Close_Program – Closing Program Descriptors

This routine is used to free internal storage that is being used to hold symbolic information associated with the specified program descriptor. After invoking this routine, you may not call any other routines with the specified program descriptor. Object descriptors for target variables that have already been obtained by calls to `dm_get_descriptor` (see page 3-14), however, are still valid; for example, `dm_get_value, dm_set_value`, `dm_peek`, and `dm_poke` operations can still occur.

Declaration

```
#include <datamon.h>
extern
int
dm_close_program (program_descriptor_t pgm_desc);
```

Parameters

pgm_desc refers to a variable that contains a valid program descriptor that has been obtained from a previous call to `dm_open_program` (see page 3-8 for an explanation of this routine)

Return Value

A return value of **`0`** indicates that the call has been successful. A return value of **`-1`** indicates that an error has occurred. Invoke `dm_get_error_code` or `dm_get_error_string` for a description of the error. Possible error conditions include the following:

- *pgm_desc* is not a valid, open program descriptor

Dm_Set_Interest_Threshold – Setting the Interest Threshold

An interest threshold refers to an integer value which controls the visibility of target variables. The default value for this setting is **`0`**, unless explicitly set via the *interest_threshold* parameter to the `dm_open_program` subprogram. All eligible variables have an interest value which is set by their compiler. By default, all eligible variables have an interest value of zero. The Ada compiler allows users to change the interest value of selected variables via the implementation-defined pragma INTERESTING. (See Annex M of the *MAXAda Reference Manual* (0890516) for more information on pragma INTERESTING). The interest threshold controls whether an otherwise eligible variable is visible to the subprograms in the Data Monitoring library. If the interest value of a variable is below the interest threshold, it is as if the variable did not exist. Once set, the interest threshold remains associated with the specified target program until reset by a subsequent `dm_set_interest_threshold` call.

Note that subsequent changes to the interest threshold have no effect on object descriptors already obtained by previous `dm_get_descriptor` calls.

Declaration

```
#include <datamon.h>
extern
int
dm_set_interest_threshold
   (int                  threshold,
    program_descriptor_t pgm_desc);
```

Parameters

threshold refers to an integer value which will be the new interest threshold for the target program corresponding to *pgm_desc.*

pgm_desc refers to a valid program descriptor that has been returned from a previous call to `dm_open_program` (see page 3-8 for an explanation of this routine)

Return Value

A return value of **0** indicates that the call has been successful. A return value of **-1** indicates that an error has occurred. Invoke `dm_get_error_code` or `dm_get_error_string` for a description of the error. Possible error conditions include the following:

- *Pgm_desc* is not a valid, open program descriptor

Dm_Set_Variant_Handling – Setting Ada Record Variant Sensitivity

The `dm_set_variant_handling` routine defines the mode in which Ada record variants are handled. By default, the *active_variants_only* mode is set to `false`; thus look-up and `dm_list` subprograms within the Data Monitoring library are not sensitive to a record variant's governing discriminant, inasmuch as all variants are considered active at all times. Setting the *active_variants_only* mode to `true` will cause look-up and `dm_list` subprograms within this package to determine the value of an enclosing record variant's governing discriminant when considering components within the record (see section 3.8.1(2-21) of the *Ada 95 Reference Manual* for more information on Ada record variants). In general, this sensitivity requires that the target program be executing, because the value of discriminants must be obtained from the target process. If *active_variants_only* mode is `true` and a component of a record is contained in an inactive variant, it is as if the component did not exist. The *active_variants_only* mode has no effect on C or FORTRAN variables.

If this mode is set to `true` and subsequent calls to subprograms within this package require the value of discriminants from the target program and those values are in memory and the target program is not executing, those subprogram calls will fail as described subsequently in this chapter. The setting of the *active_variants_only* mode is associated with the specified target program and remains in effect until a subsequent call to `dm_set_variant_handling`.

Note that subsequent changes to the *active_variants_only* mode have no effect on object descriptors which have already been obtained via a previous `dm_get_descriptor` call.

Declaration

```
#include <datamon.h>
extern
int
dm_set_variant_handling (int                    handling,
                         program_descriptor_t   pgm_desc);
```

Parameters

handling refers to an integer value which controls the handling of variants for Ada records for the target program corresponding to *pgm_desc.* Setting the value to `1` will cause sensitivity to record variant's governing discriminants as described above. Setting the value to `0` causes all variants to be considered active.

pgm_desc refers to a program descriptor that has been obtained via a previous call to `dm_open_program` and has not yet been closed (see page 3-8 for an explanation of this subprogram)

Return Value

A return value of **0** indicates that the call has been successful. A return value of **-1** indicates that an error has occurred. Invoke `dm_get_error_code` or `dm_get_error_string` for a description of the error. Possible error conditions include the following:

- *Pgm_desc* is not a valid, open program descriptor

Dm_Set_Class_Interpretation – Interpreting Class-Wide Types

The `dm_set_class_interpretation` routine sets the *interpret_classes* mode for the specified target program. This mode controls the interpretation of values of variables of Ada class-wide types. By default, the *interpret_classes* mode is `false`. Thus values of variables of class-wide types are interpreted using the specific type of the root of the class-wide type (see section 3.4.1(3-5) of the *Ada 95 Reference Manual* for more information on Ada class-wide types). If the mode is set to `true`, then values of variables of class-wide types are interpreted using the specific type associated with the actual value of the variable. In general, setting the *interpret_classes* mode to `true` requires that the target program be executing, because the value of the variable's *tag* (see section 3.9 of the *Ada 95 Reference Manual* for more information on *tags* and *type extensions*) is required to find the specific type covered by the root of the class-wide type.

Consider the following Ada example:

```
package p is
   type t is
      record
         x : integer ;
      end record ;
   type e is new t with
      record
         y : integer ;
      end record ;
   object_t : t'class := t'(x => 4) ;
   object_e : t'class := e'(x => 1, y => 2) ;
end p ;
```

In the table below, the first column represents the string passed to look-up subprograms such as `dm_get_descriptor` and `dm_get_value`. The second and third columns represent whether such calls would succeed, based on the specified setting of the *interpret_classes* mode:

String Descriptor	interpret_classes mode	
	0	**1**
"p.object_t.x"	succeed	succeed

String Descriptor	interpret_classes mode	
	0	1
"p.object_t.y"	fail	fail
"p.object_e.x"	succeed	succeed
"p.object_e.y"	fail	succeed

Of course the example in the second row, "p.object_t.y", isn't very interesting since the value of that class-wide variable really is of type "t" and therefore doesn't have a component named "y". However, the example in the fourth row, "p.object_e.y" demonstrates the point of the *interpret_classes* mode; since the value of that class-wide actually is of type "e", a type extended from the specific type of the root of the class-wide type, it does contain a component called "y".

Declaration

```
#include <datamon.h>
extern
int
dm_set_class_interpretation
   (int                    interpret,
    program_descriptor_t   pgm_desc);
```

Parameters

interpret — refers to a boolean value which controls the interpretation of values of variables of Ada class-wide types for the target program corresponding to *pgm_desc*. Setting the value to `1` will cause the specific type of the value of the variable to be based on the actual value of the variable. Setting the value to `0` will cause the specific type of the value of the variable to be obtained directly from the specific type of the root of the class-wide type.

pgm_desc — refers to a program descriptor that has been obtained via a previous call to `dm_open_program` and has not yet been closed (see page 3-8 for an explanation of this subprogram)

Return Value

A return value of **`0`** indicates that the call has been successful. A return value of **`-1`** indicates that an error has occurred. Invoke `dm_get_error_code` or `dm_get_error_string` for a description of the error. Possible error conditions include the following:

- *Pgm_desc* is not a valid, open program descriptor

Obtaining Object Descriptors for Variables

To obtain the value of a target variable or to modify a target variable, information about the variable must be located from the target program file. Such information includes the variable's type, size, shape, and address. This information is collected and stored in an internal descriptor. Part of the process of obtaining an internal descriptor involves creating a memory mapping between the target variable and the monitoring process's virtual address space; memory mapping makes subsequent access to target variables from the monitoring process extremely efficient. After the internal descriptor for a variable has been defined, `dm_get_value` and `dm_set_value` operations can occur (see pages 3-18 and 3-19, respectively, for explanations of these subprograms).

The amount of time required to obtain the descriptor may be significant for applications with stringent performance constraints.

The lifetime of an object descriptor exceeds the lifetime of its corresponding program descriptor; that is, the program descriptor associated with the program containing the target variable may be closed (thereby freeing significant memory associated with target program symbol tables), but the object descriptors remain valid.

Note that when you obtain an object descriptor for a variable, its size, shape, type, and address are frozen— for example, if the variable involves pointer indirection (`ptr.all`), the value of the `ptr` at the time of the call to `dm_get_descriptor` is used to determine the final address of the `ptr.all`. Subsequent calls to `dm_get_value` or `dm_set_value` with the resultant object descriptor will refer to the address calculated during the `dm_get_descriptor` call, regardless of the current value of the `ptr`. If you wish to re-evaluate the address of the `ptr.all` considering the current value of `ptr`, then call `dm_get_descriptor` again. This applies not only to variables involving pointer indirection, but records whose size and shape can change as the target process executes, as well as variables of class-wide types.

Part of the process of obtaining an object descriptor involves creating a memory mapping between the target variable and the monitoring process's virtual address space; memory mapping makes subsequent access to target variables from the monitoring process extremely efficient. After the object descriptor for a variable has been defined, `dm_get_value`, `dm_set_value`, `dm_peek`, and `dm_poke` operations can occur (see pages 3-18, 3-19, 3-16, and 3-17 respectively, for explanations of these routines).

Dm_Get_Descriptor – Obtaining an Object Descriptor

This routine is invoked to obtain an object descriptor for a specified variable.

Declaration

```
#include <datamon.h>
extern
int
dm_get_descriptor (char                  * item,
                   int                     no_map,
                   program_descriptor_t    pgm_desc,
                   object_descriptor_t   * obj_desc) ;
```

Parameters

item points to a string that contains the *expanded name* of the target variable for which you wish to obtain the object descriptor

no_map refers to a flag that contains an integer value that indicates whether or not address translation (mapping) is to occur. Specify a value of 0 if the monitoring process's virtual address space is to be mapped to the target variable. Specify a nonzero value under one of the following circumstances:

(1) If the target program is executing and the target variable is already accessible at the same virtual address in the monitoring process as in the target process (in this case, mapping is not necessary)

(2) If the target program is not executing and you simply wish to obtain information about the target variable (its type, size, virtual address, and so on)

If the target program is not executing and you set *no_map* to zero, the call to `dm_get_descriptor` will fail.

pgm_desc refers to a valid program descriptor that has been returned from a previous call to `dm_open_program` (see page 3-8 for an explanation of this routine).

obj_desc points to a location to which the object descriptor for the variable specified by *item* is to be returned

Return Value

A return value of **`0`** indicates that the call has been successful. A return value of **`-1`** indicates that an error has occurred. Invoke `dm_get_error_code` or `dm_get_error_string` for a description of the error. Possible error conditions include the following:

- *Pgm_desc* does not refer to a valid, open program descriptor.
- *Item* does not refer to an eligible variable.
- The specified variable could not be found in the target program's symbol tables (perhaps the user forgot to compile with the debug (**`-g`**) option).
- *Item* contains invalid expanded-notation syntax.
- The target program is not executing and *item* refers to a variable with a dynamic size, shape, address or requires a value from the target process due to modes *active_variants_only* or *interpret_classes* (see pages 3-11 and 3-12).
- The target variable could not be mapped into the monitoring process's address space; an `errno` value associated with the offending **`usermap(3rt)`** call is included in the text of the message associated with a subsequent call to `dm_get_error_string`.

Obtaining or Modifying Target Variables

This section describes the subprograms that allow you to obtain or modify the values of target variables. As explained in "Dm_Get_Descriptor – Obtaining an Object Descriptor" on page 3-14, these subprograms require the specification of the target variable via an `object_descriptor`.

`Dm_peek` and `dm_poke` (pages 3-16 and 3-17) allow you to respectively obtain and modify the value of variables directly. `Dm_get_value` and `dm_set_value` (pages 3-18 and 3-19) allow you to respectively obtain and modify the value of variables using an ASCII representation of the value.

Dm_Peek – Peeking at Variables

This routine is invoked to read the value of a variable in the target process without conversion.

Declaration

```
#include <datamon.h>
extern
int
dm_peek (object_descriptor_t * from_target,
         void                * to_addr,
         int                   bytes) ;
```

Parameters

from_target points to an `object_descriptor_t` structure that contains an object descriptor that is associated with the target variable whose value you wish to read. This descriptor is obtained from a previous call to `dm_get_descriptor` (see page 3-14 for an explanation of this routine).

to_addr points to a buffer in the monitoring process's address space to which the raw value of the target variable specified by *from_target* is to be copied

bytes refers to a variable that contains an integer value indicating the number of consecutive bytes that compose the buffer specified by *to_addr*.

For composite types (arrays, records and structures), the transfer of data occurs as if a bit-stream copy were issued using the lowest bit-address of the object specified by *from_target* as the source and the lowest bit-address of the buffer specified by *to_addr* as the destination. The number of bits copied from the source to the destination depends upon the number of bits required by *from_target*.

For noncomposite types, the value will be right justified in the buffer specified by *to_addr* (sign and zero extension for unused bits placed in the first word). No other bit-pattern conversion takes place.

The transfer of data from the source to the destination is effected via the

most appropriate machine instruction available (for example, a short value will be stored via a single instruction that transfers two bytes).

Return Value

A return value of **0** indicates that the call has been successful. A return value of **-1** indicates that an error has occurred. Invoke `dm_get_error_code` or `dm_get_error_string` for a description of the error. Possible error conditions include the following:

- *From_target* is not a valid object descriptor.
- The address range specified by *to_addr .. to_addr+bytes-1* are not valid addresses in the monitoring processes address space.

Dm_Poke – Poking at Variables

This routine is invoked to modify the value of a variable in the target process without conversion.

Declaration

```
#include <datamon.h>
extern
int
dm_poke (object_descriptor_t * to_target,
         void                * from_addr,
         int                   bytes);
```

Parameters

to_target points to an `object_descriptor_t` structure that contains an object descriptor that is associated with the target variable whose value you wish to modify. This descriptor is obtained from a previous call to `dm_get_descriptor` (see page 3-14 for an explanation of this routine).

from_addr points to a buffer in the monitoring process's address space that contains the raw value that is to be copied to the target variable specified by *to_target*

bytes refers to a variable that contains an integer value indicating the number of consecutive bytes that compose the buffer specified by *from_addr*. Note that *bytes* must be at least as large as the number of bytes required by the variable specified by *to_target*.

For composite types (arrays, records and structures), the transfer of data occurs as if a bit-stream copy were issued using the lowest bit-address of the variable specified by *from_target* as the source and the lowest bit-address of the buffer specified by *to_target* as the destination. The number of bits transferred depends on the number of bits required by *to_target*.

The bit pattern of the value in the buffer specified by *from_addr* is not modified. For noncomposite types, the required number of bits is assumed to

be right justified in the buffer.

The transfer of data to the variable specified by *to_target* is effected via the most appropriate machine instruction available (for example, a short value will be stored via a single instruction that transfers two bytes).

Return Value

A return value of **0** indicates that the call has been successful. A return value of **-1** indicates that an error has occurred. Invoke `dm_get_error_code` or `dm_get_error_string` for a description of the error. Possible error conditions include the following:

- *To_target* is not a valid object descriptor.
- The address range specified by *from_addr .. from_addr+bytes-1* are not valid addresses in the monitoring processes address space.

Dm_Get_Value – Obtaining the Value of Variables

This routine is invoked to obtain the ASCII representation of the value of a variable in the target program. The default ASCII representation used by `dm_get_value` depends upon the type of the variable:

signed integer	the `printf "%d"` conversion format
unsigned integer, pointers	the `printf "%x"` conversion format
floating point	the `printf "%g"` conversion format
fixed point (Ada)	the `printf "%g"` conversion format
enumeration (Ada)	the enumeration image in lower case

Declaration

```
#include <datamon.h>
extern
int
dm_get_value (object_descriptor_t * from_target,
              char                * value,
              int                   bytes);
```

Parameters

from_target points to an `object_descriptor_t` structure that contains an object descriptor that is associated with the target variable for which you wish to obtain the value. The descriptor is obtained from a call to `dm_get_descriptor` (see page 3-14 for an explanation of this rou-

tine). Note that if the variable to which *from_target* refers is of a composite type, an error will occur.

value points to a string to which `dm_get_value` will return the default ASCII representation of the value of the target variable specified by *from_target*

bytes refers to a variable that contains an integer value indicating the number of bytes in the string pointed to by *value*. Note that if the ASCII representation of the value of the target variable exceeds the space specified by *bytes*, an error will occur.

Return Value

A return value of **`0`** indicates that the call has been successful. A return value of **`-1`** indicates that an error has occurred. Invoke `dm_get_error_code` or `dm_get_error_string` for a description of the error. Possible error conditions include the following:

- *From_target* is not a valid object descriptor.
- The type of the target variable represented by *from_target* is a composite type (array, record, or structure). The `dm_peek` subprogram may be used for obtaining the value of such variables.
- The type of the target variable represented by *from_target* is unknown (for example, `code_unknown`).
- The size of the string referred to by *value* and *bytes* is too small to hold the ASCII representation of the value of the variable denoted by *from_target*.

Dm_Set_Value – Setting the Value of Variables

This routine is invoked to modify the value of a variable in the target process. It allows you to use ASCII representation to specify the new value to which the variable is to be set. The default ASCII representation expected by `dm_set_value` depends upon the type of the variable:

signed integer	the `sscanf "%d"` conversion format
unsigned integer, pointers	the `sscanf` "`%x`" conversion format
floating point	the `sscanf "%g"` conversion format
fixed point (Ada)	the `sscanf "%g"` conversion format
enumeration (Ada)	the enumeration image in upper or lower case

Declaration

```
#include <datamon.h>
extern
int
dm_set_value (object_descriptor_t * to_target,
```

```
                char                * value);
```

Parameters

to_target points to an `object_descriptor_t` structure that contains an object descriptor that is associated with the target variable whose value you wish to modify. This descriptor is obtained from a previous call to `dm_get_descriptor` (see page 3-14 for an explanation of this routine). Note that if the variable to which *to_target* refers is of a composite type, an error will occur.

value points to a valid ASCII representation of the new value to which the target variable specified by *to_target* is to be set. Note that this value must be expressed in a form that is consistent with the type of the target variable (for example, an integer literal for an integer type, a floating point literal for a floating point type, and so on). The value must be within the range of the type of the target variable.

Return Value

A return value of **`0`** indicates that the call has been successful. A return value of **`-1`** indicates that an error has occurred. Invoke `dm_get_error_code` or `dm_get_error_string` for a description of the error. Possible error conditions include the following:

- *To_target* is not a valid object descriptor.
- The type of the target variable represented by *to_target* is a composite type (array, record, or structure). The `dm_poke` subprogram may be used for setting the value of such variables.
- The type of the target variable represented by *to_target* is unknown (for example, `code_unknown`).
- The ASCII representation of the new value for the variable specified by *to_target* is inappropriate for the type of that variable.

Obtaining Information about Variables

This section presents the subprograms that may be invoked to additional information about a specified target variable that isn't readily available in an object descriptor.

Dm_Get_Type_Name – Obtaining Type Names

This routine is invoked to obtain the symbolic type name associated with a specified variable in a target program.

Declaration

```
#include <datamon.h>
extern
int
dm_get_type_name (char                  * item,
                  program_descriptor_t    pgm_desc,
                  char                  * type_name,
                  int                     bytes);
```

Parameters

item points to a string that specifies the *expanded name* of the target variable for which you wish to obtain the symbolic type name

pgm_desc refers to a variable that contains a valid program descriptor that has been obtained via a previous call to `dm_open_program` (see page 3-8 for an explanation of this routine)

type_name points to a character array to which `dm_get_type_name` will return the symbolic type name of the target variable specified by *item*.

bytes refers to a variable that contains an integer value indicating the size in bytes of the array pointed to by *type_name*. If the symbolic type name associated with *item* exceeds the amount of space specified by *bytes*, an error will occur.

Return Value

A return value of **`0`** indicates that the call has been successful. A return value of **`-1`** indicates that an error has occurred. Invoke `dm_get_error_code` or `dm_get_error_string` for a description of the error. Possible error conditions include the following:

- *Pgm_desc* does not refer to a valid, open program descriptor.
- *Item* does not refer to an eligible variable.
- The target variable referenced by *item* could not be found in the target program's symbol tables (perhaps the user forgot to compile with the debug (**`-g`**) option).
- *Item* contains invalid *expanded name* syntax.
- The target program is not executing and *item* refers to a variable with a dynamic size, shape, address or requires a value from the target process due to modes *active_variants_only* or *interpret_classes* (see pages 3-11 and 3-12).
- The size of the string referred to by *type_name* and *bytes* is too small to hold the name of the type of the variable specified by *item*.

Dm_Get_Type_Name_Long – Obtaining Long Type Names

This routine is invoked to obtain the symbolic type name associated with a specified variable in a target program.

Declaration

```
#include <datamon.h>
extern
int
dm_get_type_name_long
   (char                 * item,
    int                    expanded_notation,
    int                    interpret_classes,
    program_descriptor_t   pgm_desc,
    char                 * type_name,
    int                    bytes) ;
```

Parameters

item — points to a string that specifies the *expanded name* of the target variable for which you wish to obtain the symbolic type name

expanded_notation — refers to a integer value which controls whether the name of the type associated with the variable identified by *item* is expressed in Ada's *expanded name* notation. If the value specified is `1`, type names for Ada variables are preceded by the *expanded name* of their enclosing scope (e.g. "pkg.type_t"); whereas the direct name of the type is used when the flag is `0` (e.g. "type_t"). This parameter has no effect for C or FORTRAN variables.

interpret_classes — refers to a value which controls the interpretation of the type of values of variables of Ada class-wide types. When this value is `0`, the type name is obtained using the name of the specific type (suffixed by 'class) of the root of the class-wide type of the variable specified by *item*. When `1`, the type is chosen using the specific type associated with the value of the variable specified by *item*. When *interpret_classes* is set to `true`, the target program must be executing. The setting of *interpret_classes* on this subprogram call overrides the *interpret_classes* mode which is set via a call to `dm_set_class_interpretation` (see page 3-12). For example, using the code fragment from the example of `dm_set_class_interpretation` on page 3-12, a call such as `get_type_name("pkg.object_e")` would return "t'class", whereas a call such as `get_type_name_long("pkg.object_e",interpret_classes=>true)` would return "e".

pgm_desc — refers to a variable that contains a valid program descriptor that has been retuned on a previous call to `dm_open_program` (see page 3-8 for an explanation of this routine)

type_name — points to a character array to which `dm_get_type_name_long` will return the symbolic type name of the target variable specified by *item*.

bytes refers to a variable that contains an integer value indicating the size in bytes of the array pointed to by *type_name*. If the symbolic type name associated with *item* exceeds the amount of space specified by *bytes*, an error will occur.

Return Value

A return value of **0** indicates that the call has been successful. A return value of **-1** indicates that an error has occurred. Invoke `dm_get_error_code` or `dm_get_error_string` for a description of the error. Possible error conditions include the following:

- *Pgm_desc* does not refer to a valid, open program descriptor.
- *Item* does not refer to an eligible variable.
- The target variable referenced by *item* could not be found in the target program's symbol tables (perhaps the user forgot to compile with the debug (**-g**) option).
- *Item* contains invalid *expanded name* syntax.
- The target program is not executing and *item* refers to a variable with a dynamic size, shape, address or requires a value from the target process due to modes *active_variants_only* (see page 3-11) or the *interpret_classes* parameter.
- The size of the string referred to by *type_name* and *bytes* is too small to hold the name of the type of the variable specified by *item*.

Dm_Get_Enum_Image – Obtaining Enumeration Constant Images

This routine is invoked to obtain the image of the enumeration literal that corresponds to a specified position within the enumerated type associated with a variable in a target program.

Declaration

```
#include <datamon.h>
extern
int
dm_get_enum_image (char                  * item,
                   int                     position,
                   program_descriptor_t    pgm_desc,
                   char                  * image,
                   int                     bytes);
```

Parameters

item refers to a string that contains the *expanded name* of the target variable (for example, *package_p.data_item*) whose type is the enumerated type of interest. The specified variable is required only to identify its type; the value of the variable is not used (unless portions of the variable's value are required

to satisfy *active_variants_only* or *interpret_classes* modes; see pages 3-11 and 3-12).

position refers to a variable that contains a non-negative integer value that identifies the position of interest in the enumerated type associated with the variable specified by *item*. A value of zero indicates the first position in the enumerated type.

The position and value of a literal of an enumerated type are typically the same unless an explicit enumeration representation clause has been specified for the type. For example:

```
type colors is (red, white, blue);
type more_colors is (x, y, z) ;
for more_colors use (x => 5,y => 10, z => 20) ;
```

The position and value of the literal `white` are both 1, whereas the position and value of the literal `y` are 1 and 10, respectively.

The `dm_get_enum_image` service expects a position, not a value. You may use the predefined language attributes 'pos and 'val, respectively, to convert from value to position and from position to value.

pgm_desc refers to a variable that contains a valid program descriptor that has been retuned on a previous call to `dm_open_program` (see page 3-8 for an explanation of this routine)

image points to a character array to which `dm_get_enum_image` will return the image of the enumeration literal corresponding to *position* in the enumerated type associated with *item*

bytes refers to a variable that contains an integer value indicating the size in bytes of the array pointed to by *image*. If the image of the enumeration literal exceeds the amount of space specified by *bytes*, an error will occur.

Return Value

A return value of **`0`** indicates that the call has been successful. A return value of **`-1`** indicates that an error has occurred. Invoke `dm_get_error_code` or `dm_get_error_string` for a description of the error. Possible error conditions include the following:

- *Pgm_desc* does not refer to a valid, open program descriptor.
- *Item* does not refer to an eligible variable.
- The target variable referenced by *item* could not be found in the target program's symbol tables (perhaps the user forgot to compile with the debug (**`-g`**) option).
- *Item* contains invalid *expanded name* syntax.
- The target program is not executing and *item* refers to a variable with a dynamic size, shape, address or requires a value from the target process due to modes *active_variants_only* or *interpret_classes* (see pages 3-11 and 3-12).

- The target variable could not be mapped into the monitoring process's address space; an `errno` value associated with the offending **`usermap(3rt)`** call is included in the text of the message associated with a subsequent call to `dm_get_error_string`.
- The type of the variable specified by *item* is not an enumerated type.
- The position specified by *position* is illegal for the enumerated type; perhaps a value was supplied instead of a position.
- The size of the string referred to by *image* and *bytes* is too small to hold the image of the enumeration constant specified by *item* and *position*.
- The address range specified by *image..image+bytes-1* is not a valid address range in the monitoring process.

Dm_Get_Enum_Val – Obtaining Enumeration Constant Values

This routine is invoked to obtain the value of the enumeration literal that corresponds to a specified position within the enumerated type associated with a variable in a target program.

Declaration

```
#include <datamon.h>
extern
int
dm_get_enum_val (char                  * item,
                 int                     position,
                 int                   * value,
                 program_descriptor_t    pgm_desc);
```

Parameters

item refers to a string that contains the *expanded name* of the target variable (for example, *package_p.data_item*) whose type is the enumerated type of interest. The specified variable is required only to identify its type; the value of the variable is not used (unless portions of the variable's value are required to satisfy *active_variants_only* or *interpret_classes* modes; see pages 3-11 and 3-12).

position refers to a variable that contains a non-negative integer value that identifies the position of interest in the enumerated type associated with the variable specified by *item*. A value of zero indicates the first position in the enumerated type.

The position and value of a literal of an enumerated type are typically the same unless an explicit enumeration representation clause has been specified for the type. For example:

```
type colors is (red, white, blue);
type more_colors is (x, y, z) ;
for more_colors use (x => 5,y => 10, z => 20) ;
```

The position and value of the literal `white` are both 1, whereas the position and value of the literal `y` are 1 and 10, respectively.

The `dm_get_enum_val` service expects a position, not a value. You may use the predefined language attributes 'pos and 'val, respectively, to convert from value to position and from position to value.

value points to an integer variable to which `dm_get_enum_val` will return the value of the enumeration literal corresponding to *position* in the enumerated type associated with *item*.

pgm_desc refers to a variable that contains a valid program descriptor that has been retuned on a previous call to `dm_open_program` (see page 3-8 for an explanation of this routine)

Return Value

A return value of **`0`** indicates that the call has been successful. A return value of **`-1`** indicates that an error has occurred. Invoke `dm_get_error_code` or `dm_get_error_string` for a description of the error. Possible error conditions include the following:

- *Pgm_desc* does not refer to a valid, open program descriptor.
- *Item* does not refer to an eligible variable.
- The target variable referenced by *item* could not be found in the target program's symbol tables (perhaps the user forgot to compile with the debug (**`-g`**) option).
- *Item* contains invalid *expanded name* syntax.
- The target program is not executing and *item* refers to a variable with a dynamic size, shape, address or requires a value from the target process due to modes *active_variants_only* or *interpret_classes* (see pages 3-11 and 3-12).
- The target variable could not be mapped into the monitoring process's address space; an `errno` value associated with the offending **`usermap(3rt)`** call is included in the text of the message associated with a subsequent call to `dm_get_error_string`.
- The type of the variable specified by *item* is not an enumerated type.
- The position specified by *position* is illegal for the enumerated type; perhaps a value was supplied instead of a position.
- The address specified by *value* is not a valid address in the monitoring process.

Scanning Target Programs for Variables

The generic `dm_list` routine traverse the internal symbol tables of target program files and call a user-specified procedure for each item in a list. The list is formed by examining

the symbol tables in relation to a set of requirements that has been defined by parameters specified on each call to `dm_list`.

Dm_List – Scanning Target Programs for Variables

This routine is invoked to scan target programs for variables. It traverses the internal symbol tables of target program files and calls a user-specified function for each item in a list. The list is formed by examining the symbol tables in relation to a set of requirements that has been defined by parameters specified on each call to `dm_list`.

You can use this routine to search for all named scopes, all eligible variables, or all components of an eligible variable of a composite type (array, structure, or record).

Declaration

```
#include <datamon.h>
extern
int
dm_list (dm_list_modes          mode,
         char                 * qualifier,
         char                 * restriction,
         int                    do_components,
         program_descriptor_t   pgm_desc,
         void                (* action)());
```

Parameters

mode refers to a variable that contains an enumeration constant indicating the list mode that is to be used to form the list. These constants are defined in <**datamon.h**> as follows: **list_scopes**, **list_variables**, and **list_components**.

qualifier points to a location that contains a string whose interpretation depends upon the value specified by *mode*

If the value of *mode* is set to **list_scopes**, *qualifie*r should contain a null string or the name of a scope. If *qualifier* contains a null string, all scopes are listed; otherwise, the only scopes that are listed are those contained immediately within the scope identified by *qualifier.*

If the value of *mode* is set to **list_variables**, *qualifier* should contain a null string or the name of a global scope (for example, routine). If *qualifier* contains a null string, all global scopes are considered.

If the value of *mode* is set to **list_components**, *qualifier* should contain the *expanded name* of a composite variable (array, structure, or record).

restriction points to a location that contains a null string or a valid regular expression as specified by **regexpr(3G)** for PowerMAX OS and **regexec(5)** for RedHawk Linux. The regular expression is applied to the fully *expanded name* of the list item as it would be

passed to the user-specified function pointed to by *action*.

If *restriction* contains a null string, no restriction is applied.

do_components refers to a variable that contains an integer value indicating whether or not components of a composite variable are to be listed in **list_variables** mode. A nonzero value indicates that components of a composite variable are to be included in the list. If the variable listed is not a composite type, this parameter has no effect.

If the value of mode is set to **list_scopes** or **list_components**, the *do_components* parameter is ignored.

pgm_desc refers to a variable that contains a valid program descriptor that has been obtained from a previous call to `dm_open_program` (see page 3-8 for an explanation of this routine).

action refers to a variable that contains the address of a user function that is to be called for each item in the list. The *action* function will be called as if it had the following declaration:

```
void action (char *item,
             program_descriptor_t pgm_desc,
             int *quit);
```

item points to a string that contains the *expanded name* of the item

pgm_desc refers to the program descriptor that is associated with *item*

quit points to an integer whose value indicates whether or not list processing should continue. The value of this integer is always set to zero on entry to the *action* function. If you set the value of this integer to nonzero, list processing will stop upon return from the current *action* call; otherwise, list processing continues.

Return Value

A return value of **0** indicates that the call has been successful. A return value of **-1** indicates that an error has occurred. Invoke `dm_get_error_code` or `dm_get_error_string` for a description of the error. Possible error conditions include the following:

- *Pgm_desc* is not a valid, open program descriptor, or it was omitted and there are no valid, open program descriptors.
- *Restriction* is not null and is an invalid regular expression as defined by **regexpr(3G)** for PowerMAX OS and **regexec(5)** for RedHawk Linux.
- An exception is propagated from the call to the user-defined *action* procedure.

Note that it is not an error to specify parameters that result in the formation of an empty list; that is, `dm_list` returns with a value of zero without calling the user-defined *action* procedure.

4
FORTRAN Interface

The Data Monitoring library, **/usr/lib/libdatamon.a**,. contains FORTRAN interfaces that allow you to monitor variables in executing processes. These interfaces allow you to specify executable programs that contain Ada, C, or FORTRAN variables to be monitored; obtain and modify the values of selected variables; and obtain such information about the variables as their virtual addresses, types, and sizes. Interfaces that allow you to obtain and modify values are of two types: those that accept and return values expressed in symbolic formats that are appropriate for the respective variables and those that accept and return values without symbolic formatting

Organization

This chapter provides all of the information that you need to use the FORTRAN Data Monitoring interfaces. *"Types and Objects"* (page 4-1) describes type and object declarations that are used by the C interfaces to Data Monitoring. *"Error Processing"* (page 4-4) presents the enumerations and subprograms which describe error conditions. The remaining sections explain the procedures for using each of the FORTRAN routines in the Data Monitoring library. Examples using the FORTRAN interface and instructions on *"FORTRAN Compilation and Linking Instructions"* are found in Appendix C (page C-1).

Types and Objects

This section describes type and object declarations that are used by the FORTRAN interfaces to Data Monitoring. *"Descriptors"* presents the types of descriptors that are used. *"Enumerations"* presents predefined names that will assist you in determining the attributes of a variable.

Descriptors

Two types of descriptors are used by the FORTRAN interfaces to Data Monitoring: a *program descriptor*, which is used to represent a particular target program or process, and an *object descriptor*, which is used to represent a particular target variable associated with a target program or process. The header file "**/usr/include/datamon_.h**" contains predefined names that will assist you in declaring these descriptors.

The program descriptor is declared as follows:

```
INTEGER*4   pgm_desc
```

A descriptor of this type is created by the `dm_open_program` function and destroyed by the `dm_close_program` function (see pages 4-7 and 4-8, respectively, for explanations of these functions). It is used by the `dm_get_descriptor` function (see page 4-13 for an explanation of this function).

The object descriptor is declared as follows:

```
INTEGER*4   obj_desc(DM_descriptor_size)
```

`DM_descriptor_size` is declared in the header file.

The elements in the `obj_desc` array correspond to the components of the C structure of type `object_descriptor_t` that is presented in the description of the C interface to Data Monitoring on page 3-1. The following names, which are of integer type and are declared in the header file, will assist you in accessing appropriate elements in the array.

```
parameter ( DM_valid              =  1 )
parameter ( DM_private            =  2 )
parameter ( DM_code               =  3 )
parameter ( DM_target_address     =  4 )
parameter ( DM_virtual_address    =  5 )
parameter ( DM_bit_size           =  6 )
parameter ( DM_bit_offset         =  7 )
parameter ( DM_signed             =  8 )
parameter ( DM_extra_info1        =  9 )
parameter ( DM_extra_info2        = 10 )
parameter ( DM_lower_bound        = 11 ) ! real*8 aligned
parameter ( DM_upper_bound        = 13 ) ! real*8 aligned
parameter ( DM_language           = 15 )
parameter ( DM_component_code     = 16 )
parameter ( DM_component_bit_size = 17 )
parameter ( DM_component_signed   = 18 )
parameter ( DM_num_dimensions     = 19 )
parameter ( DM_lower_dimension    = 20 ) ! array[10]
parameter ( DM_upper_dimension    = 30 ) ! array[10]
```

An object descriptor is created by the `dm_get_descriptor` function (see page 4-13 for an explanation of this function). It contains type, size, and address information about the target variable. It holds sufficient information to make subsequent modification or reference of the associated target variable very efficient. The object descriptor is used by the `dm_peek`, `dm_poke`, `dm_get_value`, and `dm_set_value` functions (see pages 4-15, 4-16, 4-17, and 4-18, respectively, for explanations of these functions).

Note that the parameters `DM_lower_bound` and `DM_upper_bound` specify locations in the `obj_desc` array which actually contain `real*8` values; utilize equivalence statements to obtain the information from these components.

Note that the `DM_lower_dimension` and `DM_upper_dimension` parameters specify locations in the obj_desc array which are arrays themselves (each of length 10). The first element in each of the arrays corresponds to the bound of the first dimension, the second element to the second dimension, etc.

Enumerations

The header file "**/usr/include/datamon_.h**" also contains predefined names that will assist you in determining the attributes of a variable as described by the components of an object descriptor.

The following names, which are of integer type and are declared in the header file, are valid values for the `obj_desc(DM_language)` element in the `obj_desc` array described in the preceding section.

```
parameter ( DM_lang_C89          = 1 )
parameter ( DM_lang_C            = 2 )
parameter ( DM_lang_Ada83        = 3 )
parameter ( DM_lang_C_plus_plus  = 4 )
parameter ( DM_lang_Cobol74      = 5 )
parameter ( DM_lang_Cobol85      = 6 )
parameter ( DM_lang_Fortran77    = 7 )
parameter ( DM_lang_Fortran90    = 8 )
parameter ( DM_lang_Pascal83     = 9 )
parameter ( DM_lang_Modula2      = 10 )
parameter ( DM_lang_Ada95        = 11 )
```

The following names, which are of integer type and are declared in the header file, are valid values for the `obj_desc(DM_code)` and the `obj_desc(DM_component_code)` elements in the `obj_desc` array.

```
parameter ( DM_enumeration_code = 0 )
parameter ( DM_float_code       = 1 )
parameter ( DM_fixed_code       = 2 )
parameter ( DM_integer_code     = 3 )
parameter ( DM_record_code      = 4 )
parameter ( DM_array_code       = 5 )
parameter ( DM_char_code        = 7 )
parameter ( DM_pointer_code     = 8 )
parameter ( DM_complex_code     = 9 )
parameter ( DM_common_code      = 10 )
parameter ( DM_unknown_code     = 11 )
```

These names are explained as follows:

`DM_enumeration_code`	Ada or C enumerated types
`DM_float_code`	floating point types
`DM_fixed_code`	Ada fixed point types
`DM_integer_code`	integer types
`DM_record_code`	Ada record or C structure types
`DM_array_code`	array types
`DM_char_code`	Ada character, C `char`, and FORTRAN `character`

`DM_pointer_code`	Ada access types, C pointer types
`DM_complex_code`	FORTRAN complex types
`DM_common_code`	FORTRAN common blocks
`DM_unknown_code`	reserved for unrecognized types

A variable's `code` is required for interpreting the bits associated with the variable. The "**`/usr/include/datamon_tables_.h`**" header file includes a `code_names` array that maps these names to their corresponding enumeration images as well as a `language_names` array that maps the parameters describing languages (as shown above) to their corresponding enumeration images.

Error Processing

When a call to one of the Data Monitoring subprograms fails, the following steps are typically performed:

- The error code for the last failure associated with the current subprogram call is recorded.

  When available, a description of the error is also recorded. This description may include a system call, an `errno` value, or other information that is specific to the parameters supplied on the subprogram call.

- A value of **`-1`** is returned from the subprogram**`.`**

Both the error code and the description of the error can be retrieved as shown below by the declarations related to error processing. These declarations, which are provided in the file **`/usr/include/datamon_.h`**, are as follows:

```
integer DM_NOMEM     ! Insufficient program memory for operation
integer DM_EXCEPT    ! Exception raised during operation
integer DM_BADENUM   ! Illegal or unexpected enum literal/value
integer DM_SYNTAX    ! Illegal char. in expanded var_name/expression
integer DM_NODWARF   ! Insufficient debug info (DWARF) available
integer DM_NOTVAR    ! Specified name is not a variable or constant
integer DM_DYNAMIC   ! Object has dynamic size, shape, or address
integer DM_NOTRECORD ! Object not a record, structure, or common blk
integer DM_NOTARRAY  ! Object is not an array
integer DM_NOTFOUND  ! Could not find pkg/variable/component
integer DM_RANGE     ! Specified value/subscript out-of-range
integer DM_BADDIM    ! Insufficient/extra subscripts for array
integer DM_NOELF     ! Unrecognized/Illegal ELF object file format
integer DM_BADPID    ! Invalid (or missing) pid for; shared libs
integer DM_USRMAP    ! usermap(3C) failed to map process; bad pid?
integer DM_SYMBOLS   ! Insufficient symbol table info for operation
integer DM_BADDWARF  ! Unexpected/illegal/missing debug information
integer DM_AMBIG     ! Specified identifier is ambiguous
integer DM_SERVICE   ! System/library service call failed
integer DM_NAME2BIG  ! Expanded name too long
integer DM_NOTOPEN   ! dm_open_program call skipped/unsuccessful
```

```
integer DM_NOFILE    ! Could not open specified program file
integer DM_BADPROG   ! Bad program descriptor specified
integer DM_BADDESC   ! Bad object descriptor specified
integer DM_UNSUP     ! Unsupported operation or type
integer DM_COMPOSIT  ! Composite type/object ! allowed for operation
integer DM_BUF2SMALL ! User-specified buffer too small
integer DM_NOBITS    ! Operation requires byte-aligned types
integer DM_BADREG    ! Illegal regular expression

parameter ( DM_NOMEM      = 0 )
parameter ( DM_EXCEPT     = 1 )
parameter ( DM_BADENUM    = 2 )
parameter ( DM_SYNTAX     = 3 )
parameter ( DM_NODWARF    = 4 )
parameter ( DM_NOTVAR     = 5 )
parameter ( DM_DYNAMIC    = 6 )
parameter ( DM_NOTRECORD  = 7 )
parameter ( DM_NOTARRAY   = 8 )
parameter ( DM_NOTFOUND   = 9 )
parameter ( DM_RANGE      = 10 )
parameter ( DM_BADDIM     = 11 )
parameter ( DM_NOELF      = 12 )
parameter ( DM_BADPID     = 13 )
parameter ( DM_USRMAP     = 14 )
parameter ( DM_SYMBOLS    = 15 )
parameter ( DM_BADDWARF   = 16 )
parameter ( DM_AMBIG      = 17 )
parameter ( DM_SERVICE    = 18 )
parameter ( DM_NAME2BIG   = 19 )
parameter ( DM_NOTOPEN    = 20 )
parameter ( DM_NOFILE     = 21 )
parameter ( DM_BADPROG    = 22 )
parameter ( DM_BADDESC    = 23 )
parameter ( DM_UNSUP      = 24 )
parameter ( DM_COMPOSIT   = 25 )
parameter ( DM_BUF2SMALL  = 26 )
parameter ( DM_NOBITS     = 27 )
parameter ( DM_BADREG     = 28 )

integer function dm_get_error_code
character *(*) function dm_get_error_string
```

The header file **`/usr/include/datamon_tables_.h`** includes an `error_code_names` array that maps the parameters describing error codes (as shown above) to their corresponding enumeration images.

Functions

In the sections that follow, all of the FORTRAN Data Monitoring functions contained in the Data Monitoring library are grouped and presented according to function. The following information is provided for each function:

- A description of the function

- Detailed descriptions of each parameter
- The return value

Figure 4-1 illustrates the approximate order in which you might call the functions from an application program.

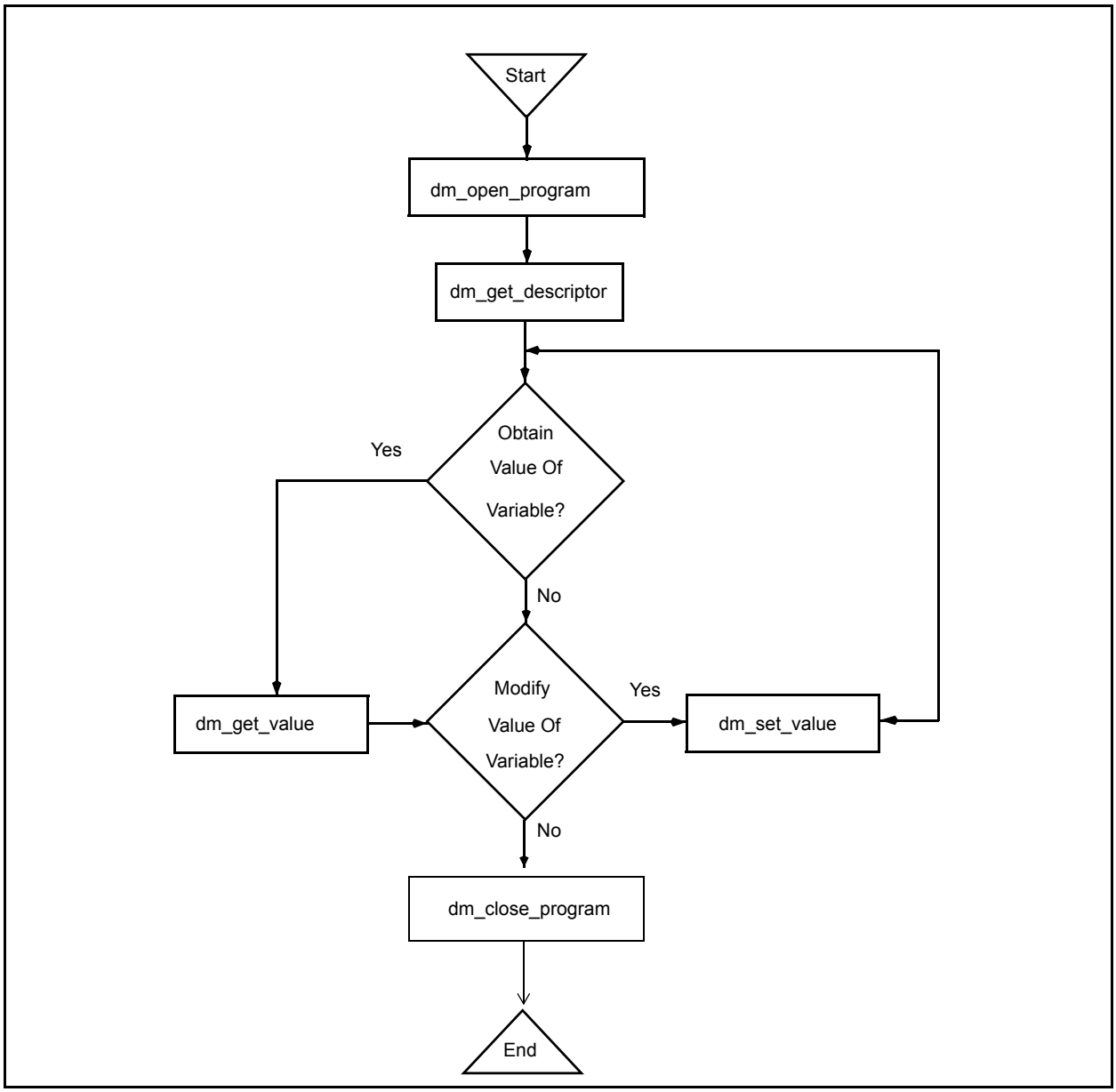

Figure 4-1. FORTRAN Data Monitoring Call Sequence

With the sequence illustrated by Figure 4-1, you first obtain the object descriptors for the target variables whose values you wish to obtain or modify; subsequently, you specify an object descriptor on each call to `dm_get_value` or `dm_set_value`. Obtaining the object descriptors involves symbol table searches; it may require a significant amount of time for time-critical applications. For such applications, it is recommended that you

invoke `dm_get_descriptor` during application initialization and then use the resultant descriptor(s) on `dm_get_value` and `dm_set_value` calls during the time-critical sections of the monitoring application.

Target Program Selection and Identification

This section presents the subprograms that allow you to (1) specify the target program for Data Monitoring, (2) obtain and close a program descriptor, (3) obtain and change the current program descriptor, and (4) obtain information about a program descriptor.

Dm_Open_Program – Obtaining Program Descriptors

This function is invoked to specify the target program for Data Monitoring. You must invoke `dm_open_program` prior to invoking any other function in the Data Monitoring library. Subsequent calls to `dm_get_descriptor` to obtain an object descriptor for a target variable require an open program descriptor. Object descriptors that you have obtained following a previous `dm_open_program` call continue to be valid; you may use them to obtain or modify the values of the target variables with which they are associated.

The `dm_open_program` call requires that portions of the target program file be read from disk into memory and that an internal symbol table be built. These procedures can use significant amounts of memory; the amounts used depend upon the size of the target program and the number of variables that can be monitored. You are advised not to invoke `dm_open_program` from time-critical sections of your application. The memory associated with a program descriptor can be reclaimed with a call to `dm_close_program`.

Function Definition

```
integer function dm_open_program (pgm_name,
                                  pid,
                                  pgm_desc)
   character*(*)  pgm_name
   integer*4      pid
   integer*4      pgm_desc
```

Parameters

pgm_name refers to a character string that contains a standard UNIX path name identifying the program to be monitored. Note that a full or relative path name of up to 1024 characters can be specified.

pid refers to a variable that contains an integer value representing the process identification number of the target executable program specified by the *pgm_name* parameter.

If the value of *pid* is 0, then `dm_open_program` will attempt to locate a process that is executing on the system with the specified path name. If successful, the corresponding process identification number of that process

is used; otherwise, it is as if an invalid value for *pid* has been specified.

Under specific conditions, the value of *pid* may be specified as **-1**. In this case, the target program does not need to be executing. These conditions are as follows: 1) the target program is statically linked (that is, it does not contain any shared libraries); 2) the variables of interest have static addresses, sizes, and shapes; and 3) subsequent use of Data Monitoring subprograms is confined to one or more of the following:

- `dm_get_type_name, dm_get_type_name_long`
- `dm_get_error_code`
- `dm_get_error_string`
- `dm_open_program`
- `dm_close_program`

pgm_desc refers to a variable to which `dm_open_program` will return the program descriptor

Return Value

A return value of **0** indicates that the call has been successful. A return value of **-1** indicates that an error has occurred. Invoke `dm_get_error_code` or `dm_get_error_string` for a description of the error. Possible error conditions include the following:

- The file associated with *pgm_name* could not be located or opened for read.
- The specified *pid* was a value other than -1 and did not identify an executing process.
- The specified *pid* was -1 but the target program associated with *pgm_name* requires shared libraries.
- The specified *pid* was 0 but no target process associated with *pgm_name* could be located.
- The file associated with *pgm_name* is not a valid ELF executable file.

Dm_Close_Program – Closing Program Descriptors

This function is used to free internal storage that is being used to hold symbolic information associated with the specified program descriptor. After invoking this function, you may not call any other functions with the specified program descriptor. Object descriptors for target variables that have already been obtained by calls to `dm_get_descriptor` (see page 4-13), however, are still valid; for example `dm_get_value`, `dm_set_value`, `dm_peek`, and `dm_poke` operations can still occur.

Function Definition

```
integer function dm_close_program (pgm_desc)
   integer*4   pgm_desc
```

Parameters

pgm_desc refers to a variable that contains a valid program descriptor that has been obtained from a previous call to `dm_open_program` (see page 4-7 for an explanation of this function)

Return Value

A return value of **`0`** indicates that the call has been successful. A return value of **`-1`** indicates that an error has occurred. Invoke `dm_get_error_code` or `dm_get_error_string` for a description of the error. Possible error conditions include the following:

- *Pgm_desc* is not a valid, open program descriptor

Dm_Set_Interest_Threshold – Setting the Interest Threshold

An interest threshold refers to an integer value which controls the visibility of target variables. The default value for this setting is **`0`**, unless explicitly set via the *interest_threshold* parameter to the `dm_open_program` subprogram. All eligible variables have an interest value which is set by their compiler. By default, all eligible variables have an interest value of zero. The Ada compiler allows users to change the interest value of selected variables via the implementation-defined pragma INTERESTING. (See Annex M of the *MAXAda Reference Manual* (0890516) for more information on pragma INTERESTING). The interest threshold controls whether an otherwise eligible variable is visible to the subprograms in the Data Monitoring library. If the interest value of a variable is below the interest threshold, it is as if the variable did not exist. Once set, the interest threshold remains associated with the specified target program until reset by a subsequent `dm_set_interest_threshold` call.

Note that subsequent changes to the interest threshold have no effect on object descriptors already obtained by previous `dm_get_descriptor` calls.

Function Definition

```
integer function dm_set_interest_threshold (threshold,
                                            pgm_desc)
   integer*4   threshold
   integer*4   pgm_desc
```

Parameters

threshold refers to an integer value which will be the new interest threshold for the target program corresponding to *pgm_desc.*

pgm_desc refers to a valid program descriptor that has been returned from a previous call to `dm_open_program` (see page 4-7 for an explanation of this routine)

Return Value

A return value of **`0`** indicates that the call has been successful. A return value of **`-1`** indicates that an error has occurred. Invoke `dm_get_error_code` or

dm_get_error_string for a description of the error. Possible error conditions include the following:

- *Pgm_desc* is not a valid, open program descriptor

Dm_Set_Variant_Handling – Setting Ada Record Variant Sensitivity

The dm_set_variant_handling routine defines the mode in which Ada record variants are handled. By default, the *active_variants_only* mode is set to false; thus look-up subprograms within the Data Monitoring library are not sensitive to a record variant's governing discriminant, inasmuch as all variants are considered active at all times. Setting the *active_variants_only* mode to true will cause look-up subprograms within this package to determine the value of an enclosing record variant's governing discriminant when considering components within the record (see section 3.8.1(2-21) of the *Ada 95 Reference Manual* for more information on Ada record variants). In general, this sensitivity requires that the target program be executing, because the value of discriminants must be obtained from the target process. If *active_variants_only* mode is true and a component of a record is contained in an inactive variant, it is as if the component did not exist. The *active_variants_only* mode has no effect on C or FORTRAN variables.

If this mode is set to true and subsequent calls to subprograms within this package require the value of discriminants from the target program and those values are in memory and the target program is not executing, those subprogram calls will fail as described subsequently in this chapter. The setting of the *active_variants_only* mode is associated with the specified target program and remains in effect until a subsequent call to dm_set_variant_handling.

Note that subsequent changes to the *active_variants_only* mode have no effect on object descriptors which have already been obtained via a previous dm_get_descriptor call.

Function Definition

```
integer function dm_set_variant_handling (handling,
                                          pgm_desc)
   integer*4   handling
   integer*4   pgm_desc
```

Parameters

handling refers to an integer value which controls the handling of variants for Ada records for the target program corresponding to *pgm_desc*. Setting the value to 1 will cause sensitivity to record variant's governing discriminants as described above. Setting the value to 0 causes all variants to be considered active.

pgm_desc refers to a program descriptor obtained via a previous call to dm_open_program and has not yet been closed (see page 4-7 for an explanation of this subprogram)

Return Value

A return value of **0** indicates that the call has been successful. A return value of **-1** indicates that an error has occurred. Invoke dm_get_error_code or

`dm_get_error_string` for a description of the error. Possible error conditions include the following:

- *Pgm_desc* is not a valid, open program descriptor

Dm_Set_Class_Interpretation – Interpreting Class-Wide Types

The `dm_set_class_interpretation` routine sets the *interpret_classes* mode for the specified target program. This mode controls the interpretation of values of variables of Ada class-wide types. By default, the *interpret_classes* mode is `false`. Thus values of variables of class-wide types are interpreted using the specific type of the root of the class-wide type (see section 3.4.1(3-5) of the *Ada 95 Reference Manual* for more information on Ada class-wide types). If the mode is set to `true`, then values of variables of class-wide types are interpreted using the specific type associated with the actual value of the variable. In general, setting the *interpret_classes* mode to `true` requires that the target program be executing, because the value of the variable's *tag* (see section 3.9 of the *Ada 95 Reference Manual* for more information on *tags* and *type extensions*) is required to find the specific type covered by the root of the class-wide type.

Consider the following Ada example:

```
package p is
   type t is
      record
         x : integer ;
      end record ;
   type e is new t with
      record
         y : integer ;
      end record ;
   object_t : t'class := t'(x => 4) ;
   object_e : t'class := e'(x => 1, y => 2) ;
end p ;
```

In the table below, the first column represents the string passed to look-up subprograms such as `dm_get_descriptor` and `dm_get_value`. The second and third columns represent whether such calls would succeed, based on the specified setting of the *interpret_classes* mode:

String Descriptor	interpret_classes mode	
	0	1
"p.object_t.x"	succeed	succeed
"p.object_t.y"	fail	fail
"p.object_e.x"	succeed	succeed
"p.object_e.y"	fail	succeed

Of course the example in the second row, "p.object_t.y", isn't very interesting since the value of that class-wide variable really is of type "t" and therefore doesn't have a component named "y". However, the example in the fourth row, "p.object_e.y" demonstrates the point of the *interpret_classes* mode; since the value of that class-wide actually is of type "e", a type extended from the specific type of the root of the class-wide type, it does contain a component called "y".

Function Definition

```
integer function dm_set_class_interpretation (interpret,
                                                pgm_desc)
   integer*4   interpret
   integer*4   pgm_desc
```

Parameters

interpret refers to an integer value which controls the interpretation of values of variables of Ada class-wide types for the target program corresponding to *pgm_desc*. Setting the value to `1` will cause the specific type of the value of the variable to be based on the actual value of the variable. Setting the value to `0` will cause the specific type of the value of the variable to be obtained directly from the specific type of the root of the class-wide type.

pgm_desc refers to a program descriptor obtained via a previous call to `dm_open_program` and has not yet been closed (see page 4-7 for an explanation of this subprogram)

Return Value

A return value of **`0`** indicates that the call has been successful. A return value of **`-1`** indicates that an error has occurred. Invoke `dm_get_error_code` or `dm_get_error_string` for a description of the error. Possible error conditions include the following:

- *Pgm_desc* is not a valid, open program descriptor

Obtaining Object Descriptors for Variables

To obtain the value of a target variable or to modify a target variable, information about the variable must be located from the target program file. Such information includes the variable's type, size, shape, and address. This information is collected and stored in an internal descriptor. Part of the process of obtaining an internal descriptor involves creating a memory mapping between the target variable and the monitoring process's virtual address space; memory mapping makes subsequent access to target variables from the monitoring process extremely efficient. After the internal descriptor for a variable has been defined, `dm_get_value` and `dm_set_value` operations can occur (see pages 4-17 and 4-18, respectively, for explanations of these subprograms).

The amount of time required to obtain the descriptor may be significant for applications with stringent performance constraints.

The lifetime of an object descriptor exceeds the lifetime of its corresponding program descriptor; that is, the program descriptor associated with the program containing the target variable may be closed (thereby freeing significant memory associated with target program symbol tables), but the object descriptors remain valid.

Note that when you obtain an object descriptor for a variable, its size, shape, type, and address are frozen— for example, if the variable involves pointer indirection (`ptr.all`), the value of the `ptr` at the time of the call to `dm_get_descriptor` is used to determine the final address of the `ptr.all`. Subsequent calls to `dm_get_value` or `dm_set_value` with the resultant object descriptor will refer to the address calculated during the `dm_get_descriptor` call, regardless of the current value of the `ptr`. If you wish to re-evaluate the address of the `ptr.all` considering the current value of `ptr`, then call `dm_get_descriptor` again. This applies not only to variables involving pointer indirection, but records whose size and shape can change as the target process executes, as well as variables of class-wide types.

Part of the process of obtaining an object descriptor involves creating a memory mapping between the target variable and the monitoring process's virtual address space; memory mapping makes subsequent access to target variables from the monitoring process extremely efficient. After the object descriptor for a variable has been defined, `dm_get_value`, `dm_set_value`, `dm_peek`, and `dm_poke` operations can occur (see pages 4-17, 4-18, 4-15, and 4-16 respectively, for explanations of these routines).

Dm_Get_Descriptor – Obtaining Object Descriptors

This function is invoked to obtain an object descriptor for a specified variable.

Function Definition

```
integer function dm_get_descriptor (item,
                                    no_map,
                                    pgm_desc,
                                    obj_desc)
   character*(*)   item
   integer*4       no_map
   integer*4       pgm_desc
   integer*4       obj_desc(DM_descriptor_size)
```

Parameters

item refers to a character string that contains the *expanded name* of the target variable for which you wish to obtain the object descriptor

no_map refers to a flag that contains an integer value that indicates whether or not address translation (mapping) is to occur. Specify a zero value if the monitoring process's virtual address space is to be mapped to the target variable. Specify a nonzero value under one of the following circumstances:

(1) If the target program is executing and the target variable is already accessible at the same virtual address in the monitoring process as in the target process (in this case, mapping is not necessary)

(2) If the target program is not executing and you simply wish to obtain

information about the target variable (its type, size, virtual address, and so on)

If the target program is not executing and you set *no_map* to zero, the call to `dm_get_descriptor` will fail.

pgm_desc refers to a valid program descriptor that has been returned from a previous call to `dm_open_program` (see page 4-7 for an explanation of this function).

obj_desc refers to an array to which `dm_get_descriptor` will return the object descriptor for the variable specified by *item*. The size of the array is specified by the integer constant `DM_descriptor_size`, which is defined in the "**`/usr/include/datamon_.h`**." header file. Each component of the array corresponds to a different piece of information about the variable; see *"Descriptors"* (page 4-1) for more information.

Return Value

A return value of **`0`** indicates that the call has been successful. A return value of **`-1`** indicates that an error has occurred. Invoke `dm_get_error_code` or `dm_get_error_string` for a description of the error. Possible error conditions include the following:

- *Pgm_desc* does not refer to a valid, open program descriptor.
- *Item* does not refer to an eligible variable.
- The specified variable could not be found in the target program's symbol tables (perhaps the user forgot to compile with the debug (**`-g`**) option).
- *Item* contains invalid expanded-notation syntax.
- The target program is not executing and *item* refers to a variable with dynamic size, shape, address or requires a value from the target process due to modes *active_variants_only* or *interpret_classes* (see pages 4-10 and 4-11).
- The target variable could not be mapped into the monitoring process's address space; an `errno` value associated with the offending **`usermap(3rt)`** call is included in the text of the message associated with a subsequent call to `dm_get_error_string`.

Obtaining or Modifying Target Variables

This section describes the subprograms that allow you to obtain or modify the values of target variables. As explained in "Dm_Get_Descriptor – Obtaining Object Descriptors" on page 4-13, these subprograms require the specification of the target variable via an `object_descriptor`.

`Dm_peek` and `dm_poke` (pages 4-15 and 4-16) allow you to respectively obtain and modify the value of variables directly. `Dm_get_value` and `dm_set_value` (pages 4-17

and 4-18) allow you to respectively obtain and modify the value of variables using an ASCII representation of the value.

Dm_Peek – Peeking at Variables

This function is invoked to read the value of a variable in the target process without conversion.

Function Definition

```
integer function dm_peek (from_target,
                          to_buffer,
                          bytes)
   integer*4   from_target(DM_descriptor_size)
   integer*1   to_buffer(*)
   integer*4   bytes
```

Parameters

from_target refers to an array that contains an object descriptor that is associated with the target variable whose value you wish to read. This descriptor is obtained from a previous call to `dm_get_descriptor` (see page 4-13 for an explanation of this function). The size of the array is specified by the integer constant `DM_descriptor_size`, which is defined in the "**`/usr/include/datamon_.h`**" header file.

to_buffer refers to a byte array in the monitoring process's address space to which the raw value of the target variable specified by *from_target* is to be copied

bytes refers to a variable that contains an integer value indicating the number of consecutive bytes that compose the array specified by *to_buffer*. For composite types (arrays, records and structures), the transfer of data occurs as if a bit-stream copy were issued using the lowest bit-address of the variable specified by *from_target* as the source and the lowest bit-address of the array specified by *to_buffer* as the destination. The number of bits copied from the source to the destination depends upon the number of bits required by *from_target*.

For noncomposite types, the value will be right justified in the array specified by *to_buffer* (sign and zero extension for unused bits placed in the first word). No other bit-pattern conversion takes place.

The transfer of data from the source to the destination is effected via the most appropriate machine instruction available (for example, a short value will be stored via a single instruction that transfers two bytes).

Return Value

A return value of **`0`** indicates that the call has been successful. A return value of **`-1`** indicates that an error has occurred. Invoke `dm_get_error_code` or `dm_get_error_string` for a description of the error. Possible error conditions include the following:

- *From_target* is not a valid object descriptor.
- The address range specified by *to_buffer .. to_buffer+bytes-1* are not valid addresses in the monitoring processes address space.

Dm_Poke – Poking at Variables

This function is invoked to modify the value of a variable in the target process without conversion.

Function Definition

```
integer function dm_poke (to_target,
                          from_buffer,
                          bytes)
   integer*4   to_target(DM_descriptor_size)
   integer*1   to_buffer(*)
   integer*4   bytes
```

Parameters

to_target refers to an array that contains an object descriptor that is associated with the target variable whose value you wish to modify. This descriptor is obtained from a previous call to `dm_get_descriptor` (see page 4-13 for an explanation of this function). The size of the array is specified by the integer constant `DM_descriptor_size`, which is defined in the "**`/usr/include/datamon_.h`**" header file.

from_buffer refers to a byte array in the monitoring process's address space that contains the raw value that is to be copied to the target variable specified by *to_target*

bytes refers to a variable that contains an integer value indicating the number of consecutive bytes that compose the array specified by *from_buffer*. Note that *bytes* must be at least as large as the number of bytes required by the variable specified by *to_target*.

For composite types (arrays, records and structures), the transfer of data occurs as if a bit-stream copy were issued using the lowest bit-address of the variable specified by *from_target* as the source and the lowest bit-address of the array specified by *to_target* as the destination. The number of bits transferred depends on the number of bits required by *to_target*.

The bit pattern of the value in the array specified by *from_buffer* is not modified. For noncomposite types, the required number of bits is assumed to be right justified in the array.

The transfer of data to the variable specified by *to_target* is effected via the most appropriate machine instruction available (for example, a short value will be stored via a single instruction that transfers two bytes).

Return Value

A return value of **0** indicates that the call has been successful. A return value of **-1** indicates that an error has occurred. Invoke `dm_get_error_code` or `dm_get_error_string` for a description of the error. Possible error conditions include the following:

- *To_target* is not a valid object descriptor.
- The address range specified by *from_buffer .. from_buffer+bytes-1* are not valid addresses in the monitoring processes address space.

Dm_Get_Value – Obtaining the Value of Variables

This function is invoked to obtain the ASCII representation of the value of a variable in the target program.

The default ASCII representation used by `dm_get_value` depends upon the type of the variable:

signed integer	the C `printf "%d"` conversion format
unsigned integer, pointers	the C `printf "%x"` conversion format
floating point	the C `printf "%g"` conversion format
fixed point (Ada)	the C `printf "%g"` conversion format
enumeration (Ada)	the enumeration image in lower case

Function Definition

```
integer function dm_get_value (value, from_target)
   character*(*)   value
   integer*4       from_target(DM_descriptor_size)
```

Parameters

value refers to a character variable to which `dm_get_value` will return the default ASCII representation of the value of the target variable specified by *from_target.*

If the ASCII representation of the value being returned is smaller than the length of the character variable specified by *value*, the value will be terminated with zero bytes. If the ASCII representation of the value exceeds the length of the character variable specified by *value*, an error will occur.

from_target refers to an array that contains an object descriptor that is associated with the target variable for which you wish to obtain the value. The descriptor is obtained from a call to `dm_get_descriptor` (see page 4-13 for an explanation of this function). The size of the array is specified by the integer constant `DM_descriptor_size`, which is defined in the "**`/usr/include/datamon_.h`**" header file.

Note that if the variable to which *from_target* refers is of a composite type, an error will occur.

Return Value

A return value of **0** indicates that the call has been successful. A return value of **-1** indicates that an error has occurred. Invoke `dm_get_error_code` or `dm_get_error_string` for a description of the error. Possible error conditions include the following:

- *From_target* is not a valid object descriptor.
- The type of the target variable represented by *from_target* is a composite type (array, record, or structure). The `dm_peek` subprogram may be used for obtaining the value of such variables.
- The type of the target variable represented by *from_target* is unknown (for example, `code_unknown`).
- The size of *value* is too small to hold the ASCII representation of the value of the variable denoted by *from_target*.

Dm_Set_Value – Setting the Value of Variables

This function is invoked to modify the value of a variable in the target process. It allows you to use ASCII representation to specify the new value to which the variable is to be set. The default ASCII representation used by `dm_set_value` depends upon the type of the variable:

signed integer	the C `sscanf "%d"` conversion format
unsigned integer,pointers	the C `sscanf "%d"` conversion format
floating point	the C `sscanf "%g"` conversion format
fixed point (Ada)	the C `sscanf "%g"` conversion format
enumeration (Ada)	the enumeration image in upper or lower case

Function Definition

```
integer function dm_set_value (value, to_target)
   character*(*)   value
   integer*4       from_target(DM_descriptor_size)
```

Parameters

value refers to a character string that contains a valid ASCII representation of the new value to which the target variable specified by *to_target* is to be set. Note that this value must be expressed in a form that is consistent with the type of the target variable (for example, an integer literal for an integer

type, a floating point literal for a floating point type, and so on). The value must be within the range of the type of the target variable.

to_target refers to an array that contains an object descriptor that is associated with the target variable whose value you wish to modify. This descriptor is obtained from a previous call to `dm_get_descriptor` (see page 4-13 for an explanation of this function). The size of the array is specified by the integer constant `DM_descriptor_size`, which is defined in the "**`/usr/include/datamon_.h`**" file.

Note that if the variable to which *to_target* refers is of a composite type, an error will occur.

Return Value

A return value of **`0`** indicates that the call has been successful. A return value of **`-1`** indicates that an error has occurred. Invoke `dm_get_error_code` or `dm_get_error_string` for a description of the error. Possible error conditions include the following:

- *To_target* is not a valid object descriptor.
- The type of the target variable represented by *to_target* is a composite type (array, record, or structure). The `dm_poke` subprogram may be used for setting the value of such variables.
- The type of the target variable represented by *to_target* is unknown (for example, `code_unknown`).
- The ASCII representation of the new value for the variable specified by *to_target* is inappropriate for the type of that variable.

Obtaining Information about Variables

This section presents the subprograms that may be invoked to additional information about a specified target variable that isn't readily available in an object descriptor.

Dm_Get_Type_Name – Obtaining Type Names

This function is invoked to obtain the ASCII representation of the type of a specified variable in a target program.

Function Definition

```
integer function dm_get_type_name (type_name,
                                   item,
                                   pgm_desc)
   character*(*)   type_name
   character*(*)   item
   integer*4       pgm_desc
```

Parameters

type_name refers to a character array to which `dm_get_type_name` will return the symbolic type name of the target variable specified by *item*

If the ASCII representation of the type being returned is smaller than the length of the character variable specified by *type_name*, the value will be terminated with zero bytes. If the ASCII representation of the type exceeds the length of the character variable specified by *type_name*, an error will occur.

item refers to a character string that contains the *expanded name* of the target variable for which you wish to obtain the type name

pgm_desc refers to a variable that contains a valid program descriptor that has been retuned on a previous call to `dm_open_program` (see page 4-7 for an explanation of this function)

Return Value

A return value of **`0`** indicates that the call has been successful. A return value of **`-1`** indicates that an error has occurred. Invoke `dm_get_error_code` or `dm_get_error_string` for a description of the error. Possible error conditions include the following:

- *Pgm_desc* does not refer to a valid, open program descriptor.
- *Item* does not refer to an eligible variable.
- The target variable referenced by *item* could not be found in the target program's symbol tables (perhaps the user forgot to compile with the debug (**`-g`**) option).
- *Item* contains invalid *expanded name* syntax.
- The target program is not executing and *item* refers to a variable with dynamic size, shape, address or requires a value from the target process due to modes *active_variants_only* or *interpret_classes* (see pages 4-10 and 4-10).
- The size of the character variable referred to by *type_name* is too small to hold the name of the type of the variable specified by *item*.

Dm_Get_Type_Name_Long – Obtaining Long Type Names

This routine is invoked to obtain the symbolic type name associated with a specified variable in a target program.

Function Definition

```
integer function dm_get_type_name (type_name,
                                   item,
                                   expanded_notation,
                                   interpret_classes,
```

```
                                        pgm_desc)
   character*(*)   type_name
   character*(*)   item
   integer*4       expanded_notation
   integer*4       interpret_classes
   integer*4       pgm_desc
```

Parameters

type_name — refers to a character array to which `dm_get_type_name_long` will return the symbolic type name of the target variable specified by *item*.

item — refers to a character string that specifies the *expanded name* of the target variable for which you wish to obtain the symbolic type name.

expanded_notation — refers to a integer value which controls whether the name of the type associated with the variable identified by *item* is expressed in Ada's *expanded name* notation. If the value specified is `1`, type names for Ada variables are preceded by the *expanded name* of their enclosing scope (e.g. "pkg.type_t"); whereas the direct name of the type is used when the flag is `0` (e.g. "type_t"). This parameter has no effect for C or FORTRAN variables.

interpret_classes — refers to a value which controls the interpretation of the type of values of variables of Ada class-wide types. When this value is `0`, the type name is obtained using the name of the specific type (suffixed by 'class) of the root of the class-wide type of the variable specified by *item*. When `1`, the type is chosen using the specific type associated with the value of the variable specified by *item*. When *interpret_classes* is set to `true`, the target program must be executing. The setting of *interpret_classes* on this subprogram call overrides the *interpret_classes* mode which is set via a call to `dm_set_class_interpretation` (see page 4-10). For example, using the code fragment from the example of `dm_set_class_interpretation` on page 4-10, a call such as `get_type_name("pkg.object_e")` would return "t'class", whereas a call such as `get_type_name_long("pkg.object_e",interpret_classes=>true)` would return "e".

pgm_desc — refers to a variable that contains a valid program descriptor that has been retuned on a previous call to `dm_open_program` (see page 4-7 for an explanation of this routine)

Return Value

A return value of **`0`** indicates that the call has been successful. A return value of **`-1`** indicates that an error has occurred. Invoke `dm_get_error_code` or

`dm_get_error_string` for a description of the error. Possible error conditions include the following:

- *Pgm_desc* does not refer to a valid, open program descriptor.
- *Item* does not refer to an eligible variable.
- The target variable referenced by *item* could not be found in the target program's symbol tables (perhaps the user forgot to compile with the debug (**-g**) option).
- *Item* contains invalid *expanded name* syntax.
- The target program is not executing and *item* refers to a variable with dynamic size, shape, address or requires a value from the target process due to modes *active_variants_only* (see pages 4-10 and 4-10) or the *interpret_classes* parameter.
- The size of the character variable referred to by *value* is too small to hold the name of the type of the variable specified by *item*.

Dm_Get_Enum_Image – Obtaining Enumeration Constants Images

This function is invoked to obtain the image of the enumeration literal that corresponds to a specified position within the enumerated type associated with a variable in a target program.

Function Definition

```
integer function dm_get_enum_image (image,
                                    item,
                                    position,
                                    pgm_desc)
   character*(*)   image
   character*(*)   item
   integer*4       position
   integer*4       pgm_desc
```

Parameters

image refers to a character variable to which `dm_get_enum_image` will return the image of the enumeration literal corresponding to *position* in the enumerated type associated with *item*

item refers to a character string that contains the *expanded name* of the target variable whose type is the enumerated type of interest. The specified variable is required only to identify its type; the value of the variable is not used (unless portions of the variable's value are required to satisfy *active_variants_only* or *interpret_classes* modes; see pages 4-10 and 4-11).

position refers to a variable that contains a non-negative integer value that identifies the position of interest in the enumerated type associated with the variable specified by *item*. A value of zero indicates the first position in the enumer-

ated type.

The position and value of a literal of an enumerated type are typically the same unless an explicit enumeration representation clause has been specified for the type. For example:

```
type colors is (red, white, blue);
type more_colors is (x, y, z) ;
for more_colors use (x => 5,y => 10, z => 20) ;
```

The position and value of the literal `white` are both 1, whereas the position and value of the literal `y` are 1 and 10, respectively.

The `dm_get_enum_image` service expects a position, not a value.

pgm_desc refers to a variable that contains a valid program descriptor obtained via a previous call to `dm_open_program` (see page 4-7 for an explanation of this function)

Return Value

A return value of **`0`** indicates that the call has been successful. A return value of **`-1`** indicates that an error has occurred. Invoke `dm_get_error_code` or `dm_get_error_string` for a description of the error. Possible error conditions include the following:

- *Pgm_desc* does not refer to a valid, open program descriptor.
- *Item* does not refer to an eligible variable.
- The target variable referenced by *item* could not be found in the target program's symbol tables (perhaps the user forgot to compile with the debug (**`-g`**) option).
- *Item* contains invalid *expanded name* syntax.
- The target program is not executing and *item* refers to a variable with dynamic size, shape, address or requires a value from the target process due to modes *active_variants_only* or *interpret_classes* (see pages 4-10 and 4-10).
- The target variable could not be mapped into the monitoring process's address space; an `errno` value associated with the offending **`usermap(3rt)`** call is included in the text of the message associated with a subsequent call to `dm_get_error_string`.
- The type of the variable specified by *item* is not an enumerated type.
- The position specified by *position* is illegal for the enumerated type; perhaps a value was supplied instead of a position.
- The size of the character variable referred to by *image* is too small to hold the image of the enumeration constant specified by *item* and *position*.

Dm_Get_Enum_Val – Obtaining Enumeration Constant Values

This routine is invoked to obtain the value of the enumeration literal that corresponds to a specified position within the enumerated type associated with a variable in a target program.

Function Definition

```
integer function get_enum_val (item,
                               position,
                               value,
                               pgm_desc)
   character*(*) item
   integer*4     position
   integer*4     value
   integer*4     pgm_desc
```

Parameters

item refers to a string that contains the *expanded name* of the target variable (for example, *package_p.data_item*) whose type is the enumerated type of interest. The specified variable is required only to identify its type; the value of the variable is not used (unless portions of the variable's value are required to satisfy *active_variants_only* or *interpret_classes* modes; see pages 4-10 and 4-11).

position refers to a variable that contains a non-negative integer value that identifies the position of interest in the enumerated type associated with the variable specified by *item*. A value of zero indicates the first position in the enumerated type.

The position and value of a literal of an enumerated type are typically the same unless an explicit enumeration representation clause has been specified for the type. For example:

```
type colors is (red, white, blue);
type more_colors is (x, y, z) ;
for more_colors use (x => 5,y => 10, z => 20) ;
```

The position and value of the literal `white` are both 1, whereas the position and value of the literal `y` are 1 and 10, respectively.

The `dm_get_enum_val` service expects a position, not a value.

value refers to an integer variable to which `dm_get_enum_val` will return the value of the enumeration literal corresponding to *position* in the enumerated type associated with *item*

Return Value

A return value of **`0`** indicates that the call has been successful. A return value of **`-1`** indicates that an error has occurred. Invoke `dm_get_error_code` or `dm_get_error_string` for a description of the error. Possible error conditions include the following:

- *Pgm_desc* does not refer to a valid, open program descriptor.
- *Item* does not refer to an eligible variable.
- The target variable referenced by *item* could not be found in the target program's symbol tables (perhaps the user forgot to compile with the debug (**-g**) option).
- *Item* contains invalid *expanded name* syntax.
- The target program is not executing and *item* refers to a variable with dynamic size, shape, address or requires a value from the target process due to modes *active_variants_only* or *interpret_classes* (see pages 4-10 and 4-10).
- The target variable could not be mapped into the monitoring process's address space; an `errno` value associated with the offending **`usermap(3rt)`** call is included in the text of the message associated with a subsequent call to `dm_get_error_string`.
- The type of the variable specified by *item* is not an enumerated type.
- The position specified by *position* is illegal for the enumerated type; perhaps a value was supplied instead of a position.

A
MAXAda Examples

This appendix provides instructions for compilation and linking Ada programs that use the `Real_Time_Data_Monitoring` package as well as example programs.

Compilation and Linking Instructions

The following commands will create a compilation environment, add visibility to the `Real_Time_Data_Monitoring` package, introduce user source files into the environment, define a program, and build it. See Chapter 2 - "Using MAXAda" in the *MAXAda Reference Manual* for more information on MAXAda. All of the following commands require that **`/usr/ada/bin`** is in the user's **`PATH`** environment variable.

`a.mkenv -g`

The above command creates a compilation environment; a compilation environment is required for all subsequent MAXAda commands. The environment consists of a file, **`.Ada`**, and a directory, **`.ada`**, created in the user's current working directory. The **`-g`** option to a.mkenv sets the default compilation option for all Ada units to include debug information. Debug information isn't specifically required for use of subprograms within the `Real_Time_Data_Monitoring` package, however, *target programs* must be built with debug information.

`a.path -v -a rtdm`

The `Real_Time_Data_Monitoring` package is provided in a pre-compiled MAXAda environment called **`rtdm`**. Access to this environment is provided by placing **`rtdm`** in the environment search path for the user's environment.

`a.intro -v user_source_file.ada`

Before compilation and linking can occur, the user's Ada source files must be introduced into the environment. The files don't have to be in their final form, but they must either be empty or contain a reasonable facsimile of an Ada compilation unit (the syntax of the file must be close enough to valid Ada for a.intro to determine its basic structure).

`a.partition -create active main_subprogram_name`

This command defines a program to be build; at a minimum, it requires that the user specify the name of the main subprogram. If no other parameters are supplied, the name of the program file produced will be that of the specified main_subprogram_name.

`a.build -v main_subprogram_name`

This command compiles and links the program.

Examples

Two example programs are provided: **peek**, an extremely simple program utilizing just three of the subprograms from the `Real_Time_Data_Monitoring` package, and **scanner**, a complete program which scans executable programs and provides information on all eligible variables within them.

Example 1 — Peek

```
> a.mkenv -g

> a.path -v -a rtdm
Environment search path:
        /usr/ada/.../predefined
        /usr/ada/.../rtdm

> cat peek.ada

package global is
   data : integer := 45;
end global;

with real_time_data_monitoring;
with ada.text_io;
with ada.command_line;
with global;

procedure peek is
   package rtdm renames real_time_data_monitoring;
   package acl  renames ada.command_line;
   package atio renames ada.text_io;
begin
   rtdm.open_program (acl.argument(1));
   atio.put_line ("The value of """ &
                  acl.argument(2) &
                  """ is """ &
                  rtdm.get_value(acl.argument(2)) &
                  """");
   rtdm.close_program;
end peek;

> a.intro -v peek.ada
   introducing: peek.ada

> a.partition -create active peek

> a.build -v peek
   compiling: package spec global
   compiling: subprogram body peek
   linking: peek
```

```
> ./peek peek global.data
The value of "global.data" is " 45"
```

The example program above utilizes just three subprograms from the `Real_Time_Data_Monitoring` package: `open_program`, `get_value`, and `close_program`. The example program is extremely simple, yet quite powerful.

It requires two arguments: the name of a target executable program and the name of an *eligible variable* in *expanded notation*. It prints the current value of the specified variable from the specified *target program* (which must be executing).

For simplicity in the example, we specified the example program itself as the target program and the variable `data` in the package `global` as our variable. In fact, the only reason that the package `global` was included in the example was so that we could use the example program as our target program (i.e.; we needed an eligible variable to peek at).

Example 2 — Scanner

```
> a.mkenv -g

> a.path -v -a rtdm
Environment search path:
        /usr/ada/.../predefined
        /usr/ada/.../rtdm

> cat scanner.ada

with ada.command_line;
procedure scanner is
   passive : boolean;
   procedure scan (program_name         : in string;
                   fetch                : in boolean;
                   active_variants_only : in boolean;
                   interpret_classes    : in boolean;
                   indirect_pointers    : in boolean;
                   type_names_interpret : in boolean;
                   type_names_expanded  : in boolean;
                   max_array_components : in natural;
                   interest_threshold   : in integer) is separate;
begin
   passive := boolean'value(ada.command_line.argument(2)) = false;
   scan (program_name         => ada.command_line.argument(1),
         fetch                => not passive,
         active_variants_only => not passive,
         interpret_classes    => not passive,
         type_names_interpret => not passive,
         type_names_expanded  => false,
         max_array_components => 1,
         interest_threshold   => 0,
         indirect_pointers    => not passive);
end scanner;

with ada.text_io;
with ada.unchecked_conversion;
with real_time_data_monitoring;
with system;
```

```
separate (scanner)
procedure scan (program_name         : in string;
                fetch                : in boolean;
                active_variants_only : in boolean;
                interpret_classes    : in boolean;
                indirect_pointers    : in boolean;
                type_names_interpret : in boolean;
                type_names_expanded  : in boolean;
                max_array_components : in natural;
                interest_threshold   : in integer) is
--
   package rtm renames real_time_data_monitoring;

   type chic is mod 2**32;

   function too_chic is new ada.unchecked_conversion (system.address, chic);
   function too_chic is new ada.unchecked_conversion (integer,        chic);

   package iio is new ada.text_io.integer_io(integer);
   package mio is new ada.text_io.modular_io(chic);

   subtype stack_frames is natural range 0..100;
   type stack_frame is
      record
         count : natural;
         max   : natural;
      end record;

   -- Misc variables
   dummy_position       : rtm.list_position;
   dummy_quit           : boolean := false;
   stack                : array (stack_frames) of stack_frame;
   stack_top            : stack_frames;
   indirection_active   : boolean := false;

   -- Instantiations
   procedure variable_action (item     : in string;
                              program  : in rtm.program_descriptor;
                              position : in out rtm.list_position;
                              quit     : in out boolean);
   procedure scope_action (item     : in string;
                           program  : in rtm.program_descriptor;
                           position : in out rtm.list_position;
                           quit     : in out boolean);
   package list_variables is new rtm.lists (variable_action);
   package list_scopes    is new rtm.lists (scope_action);

   -- Subprograms
   procedure variable_action (item     : in string;
                              program  : in rtm.program_descriptor;
                              position : in out rtm.list_position;
                              quit     : in out boolean) is separate;
   procedure scope_action (item     : in string;
                           program  : in rtm.program_descriptor;
                           position : in out rtm.list_position;
                           quit     : in out boolean) is separate;
--
begin
--
   rtm.open_program (program_name       => program_name,
                     interest_threshold => interest_threshold);

   if active_variants_only then
      rtm.set_variant_handling (active_variants_only);
   end if;
```

```
   if interpret_classes then
      rtm.set_class_interpretation (interpret_classes);
   end if;


   stack_top := stack_frames'first;
   stack(stack_top).count := 0;
   stack(stack_top).max   := natural'last;

   scan.list_scopes.list (mode => rtm.list_scopes, components => false);

   rtm.close_program;
--
exception
when rtm.real_time_monitoring_error =>
   ada.text_io.put_line (
      rtm.error_codes'image(rtm.get_real_time_monitoring_error_code) & ": "
&
      rtm.get_real_time_monitoring_error);
end scan;

with system;
with unchecked_conversion;
separate (scanner.scan)
procedure variable_action (item     : in string;
                           program  : in rtm.program_descriptor;
                           position : in out rtm.list_position;
                           quit     : in out boolean) is
--
   use rtm;
   use ada.text_io;
   use iio;

   virtual     : system.address;
   target      : system.address;
   atomic      : atomic_types;
   size        : natural;
   offset      : natural;
   code        : codes;
   descriptor  : internal_descriptor;
   signed      : boolean;
   indicies    : indicies_list;
   dimensions  : integer;
   pointer     : integer := 0;
--
begin
--
   stack(stack_top).count := stack(stack_top).count + 1;
   if stack(stack_top).count > stack(stack_top).max then
      quit := true;
      return;
   end if;

   set_col (count((stack_top)*3)+1);
   put (item);
   put (" (");
   put
(get_type_name(item,program,type_names_expanded,type_names_interpret));

   get_descriptor (item, descriptor, not fetch, program);
   get_info (descriptor, virtual, target, atomic, size, offset, code);

   put (", ");
   mio.put (too_chic(target), width => 12, base=>16);
   put (", ");
```

```
iio.put (size, width => 0);
put (", ");
iio.put (offset, width => 0);
put (", ");

case code is
when code_array =>
   get_array_info (descriptor, size, code, signed, indicies, dimensions);
   put ("array [");
   for d in 1..dimensions loop
      if d /= 1 then
         put (",");
      end if;
      iio.put (indicies(d).lower_bound, width=>0);
      put ("..");
      iio.put (indicies(d).upper_bound, width=>0);
   end loop;
   put ("] of ");
   put (codes'image(code));
   put_line (")");
   stack_top := stack_top + 1;
   stack(stack_top).count := 0;
   stack(stack_top).max   := max_array_components;
   scan.list_variables.list (mode => list_components, qualifier => item);
   stack_top := stack_top - 1;
   return;
when code_record | code_common =>
   if code = code_record then
      put_line (" record)");
   else
      put_line (" common)");
   end if;
   stack_top := stack_top + 1;
   stack(stack_top).count := 0;
   stack(stack_top).max   := natural'last;
   scan.list_variables.list (mode => list_components, qualifier => item);
   stack_top := stack_top - 1;
   return;
when code_integer =>
   put (codes'image(code));
   case atomic is
   when discrete_1byte_signed |
        discrete_2byte_signed |
        discrete_4byte_signed =>
      put (", signed");
   when others =>
      put (", unsigned");
   end case;
when others =>
   put (codes'image(code));
end case;

if code = code_pointer then
   if fetch then
      put (", ");
      get_value (descriptor, pointer'address, pointer'size/8);
      mio.put (too_chic(pointer), width=>12, base=>16);
   end if;
   put_line (")");
   if indirect_pointers          and then
      indirection_active = false     then
      if pointer /= 0 then
         indirection_active := true;
         variable_action (item & ".all", program, position, quit);
         indirection_active := false;
```

```
         end if;
      end if;
   else
      if fetch then
         put (", ");
         put (get_value(descriptor));
      end if;
      put_line (")");
   end if;
--
exception
when real_time_monitoring_error =>
   set_col (count((stack_top)*3)+1);
   put_line (error_codes'image(get_real_time_monitoring_error_code) & ": " &
             get_real_time_monitoring_error);
--
end variable_action;

separate (scanner.scan)
procedure scope_action (item     : in string;
                        program  : in rtm.program_descriptor;
                        position : in out rtm.list_position;
                        quit     : in out boolean) is
--
   use rtm;
   use ada.text_io;
--
begin
--
   set_col (count((stack_top)*3)+1);

   put_line ("scope: " & item);

   stack_top := stack_top + 1;
   stack(stack_top).count := 0;
   stack(stack_top).max   := natural'last;

   scan.list_variables.list (mode       => rtm.list_variables,
                             qualifier  => item,
                             components => false,
                             program    => program);

   stack_top := stack_top - 1;
--
end scope_action;
```

```
> a.intro -v scanner.ada
   introducing: scanner.ada

> a.partition -create active scanner


> a.build scanner

> scanner scanner true > out

> fgrep ada.command_line out

scope: ada.command_line
scope: ada.command_line.local_bindings
       ada.command_line.local_bindings.u_mainp
```

```
        (a_environment_frame, 16#300CC230#, 32, 0, CODE_POINTER,
            16#300CBB10#)
      ada.command_line.local_bindings.u_mainp.all
       (environment_frame_t, 16#300CBB10#, 96, 0,  record)
      ada.command_line.local_bindings.u_mainp.all.argc
       (integer, 16#300CBB10#, 32, 0, CODE_INTEGER, signed,  3)
      ada.command_line.local_bindings.u_mainp.all.arg_list
       (a_address_list, 16#300CBB14#, 32, 0, CODE_POINTER, 16#2FF7D314#)
      ada.command_line.local_bindings.u_mainp.all.env_list
       (a_address_list, 16#300CBB18#, 32, 0, CODE_POINTER, 16#2FF7D324#)
```

```
> scanner peek false > out
> fgrep global out

scope: global
   global.data (integer, 16#3009C534#, 32, 0, CODE_INTEGER, signed)
```

The example above provides source code and build instructions for a **scanner** program which scans a user-specified *target program* for scopes and describes the variables in those scopes. The description includes:

- The variable's name in *expanded notation*
- The variable's type name
- The variable's address in the target program
- The variable's size in bits
- The variable's bit offset from its address
- The variable's `Real_Time_Data_Monitoring` code
- For record variables, a description of all its components
- For array variables, a description of the dimensions and bounds of the array
- For array variables, a description of the first component of the array

Additional information is supplied when the **scanner** program is run in non-passive mode; defined by the second parameter to the program (false => passive, true => non-passive). When run in non-passive mode, the target program must be executing and the description output by **scanner** further includes the following:

- The value of the variable
- Class-wide type interpretation is activated
- Sensitivity to Ada record variants is activated
- Pointer variables are indirected (once)

The output of the scanner program is rather lengthy, even for small Ada programs, since it includes descriptions of variables in support packages contained in most all Ada programs. The **fgrep** commands above are used to show some of the output from the scanner invocations (the output underwent minor formatting changes for inclusion in this manual).

The first invocation shown above specifies that the program to scan is the **`scanner`** program itself; the second argument of `true` indicates that the scan is to be done in non-passive mode. The second invocation specifies the program from the **`peek`** example describes in this appendix; since that program mostly likely isn't executing, we run the scan in passive mode as indicated by the second argument of `false`.

B
C Examples

This appendix provides instructions for compilation and linking C programs that use the Data Monitoring library as well as example programs.

C Compilation and Linking Instructions

For PowerMAX OS:

```
> ec -g main.c -ldatamon -lud
```

For RedHawk Linux:

```
> gcc -g main.c -ldatamon -lccur_rt -lccur_fbsched
```

The command above invokes the C compiler on the source file main.c. The **-g** option specifies that debug information should be generated; this isn't specifically required for use of subprograms in the data monitoring library, however, the *target programs* must be built with debug information. The **-ldatamon** link option specifies that the Data Monitoring library, **/usr/lib/libdatamon.a**, should be used when linking the program.

Examples

Two example programs are provided: **peek**, an extremely simple program utilizing just four of the subprograms from the Data Monitoring library, and **scanner**, a complete program which scans executable programs and provides information on all eligible variables within them.

Example 1 — Peek

```
> cat peek.c
#include <datamon.h>

int global_data = 45 ;

main (int argc, char * argv[])
{
     program_descriptor_t pgm_desc ;
     object_descriptor_t  obj_desc ;
     char                 image[1024] ;
```

```
    dm_open_program (argv[1], 0, &pgm_desc) ;
    dm_get_descriptor (argv[2], 0, pgm_desc, &obj_desc) ;
    dm_get_value (&obj_desc, image, sizeof(image)) ;
    printf ("The value of \"%s\" is \"%s\"\n", argv[2], image) ;
    dm_close_program (pgm_desc) ;
}
```

```
> cc -g peek.c -o peek -ldm
```

```
> peek peek global_data
The value of "global_data" is " 45"
```

The example program above utilizes just four subprograms from the Data Monitoring library: `dm_open_program`, `dm_get_descriptor`, `dm_get_value`, and `dm_close_program`. The example program is extremely simple, yet quite powerful.

It requires two arguments: the name of a target executable program and the name of an *eligible variable* in *expanded notation*. It prints the current value of the specified variable from the specified program.

For simplicity in the example, we specified the example program itself as the executable program and the variable `global_data` . In fact, the only reason that the variable `global_data` was included in the example was so that we could use the example program as our target program (i.e.; we needed an eligible variable to peek at).

Example 2 — Scanner

```
> cat scanner.c
```

```
#define datamon_mappings

#include "datamon.h"

static int stack ;
static int count[10000] ;
static int max[10000] ;
static int fetch ;


static
void
assert (int status, char * service)
{
   if (status != 0) {
      printf ("\n(ASSERTION FAILURE: %s: (%s) %s)\n",
              service,
              dm_error_code_images[dm_get_error_code()],
              dm_get_error_string()) ;
   }
}

static int indent = 0 ;
static int indirection_active = 0 ;
int * gratuitous_pointer = &indent ;

static
void
```

```
item_action (char * item, program_descriptor_t pgm, int * quit)
{
   static char               * example ;
   auto   object_descriptor_t   obj ;
   auto   int                   status ;
   auto   int                   d ;
   auto   int                   i ;
   auto   char                  type_name[80] ;
   auto   char                  buffer[80] ;
   auto   char                  indirected_item[1024] ;

   if (++count[stack] >= max[stack]) {
      *quit = 1 ;
   }

   example = item ;

   indent += 3 ;
   for (i=0; i<indent; ++i) {
      printf (" ") ;
   }
   printf ("%s", item) ;

   status = dm_get_type_name (item, pgm, type_name, sizeof(type_name)) ;
   assert (status, "dm_get_type_name") ;
   if (status == 0) {
      printf (" (%s", type_name) ;
   }

   status = dm_get_descriptor (item, !fetch, pgm, &obj) ;
   assert (status, "dm_get_descriptor") ;
   if (status==0) {
      printf (", 0x%-8.8x, %s, %s, %d, %d",
              obj.od_target_address,
              dm_code_images[obj.od_code],
              (obj.od_signed ? "signed" : "unsigned"),
              obj.od_bit_size,
              obj.od_bit_offset) ;
      if (obj.od_code == code_integer) {
         printf (" [ 0x%8-8x..0x%-8.8x ]", (int)obj.od_lower_bound,
                                 (int)obj.od_upper_bound) ;
      }

      if (obj.od_code == code_array) {
         printf (", dims=%d", obj.od_number_dims) ;
         for (d=0; d<obj.od_number_dims; ++d) {
            printf (" [%d..%d]", obj.od_lower_dims[d], obj.od_upper_dims[d])
;
         }
         printf (" (%s, %s, %d)",
                 dm_code_images[obj.od_component_code],
                 (obj.od_component_signed ? "signed" : "unsigned"),
                 obj.od_component_bit_size) ;
      }

      if (obj.od_code == code_array ||
          obj.od_code == code_record ||
          obj.od_code == code_common) {
         printf (")\n") ;
         count[++stack] = 0 ;
         if (obj.od_code == code_array) {
            max[stack] = 1 ;
         } else {
            max[stack] = 2000000000 ;
         }
```

```
            status = dm_list (list_components, item, 0, 1, pgm, &item_action) ;
            --stack ;
            assert (status, "dm_list") ;
         } else if (fetch) {
            status = dm_get_value (&obj, buffer, sizeof(buffer)) ;
            assert (status, "dm_get_value") ;
            if (status == 0) {
               printf (", %s", buffer) ;
            }
            printf (")\n") ;
         } else {
            printf (")\n") ;
         }

         if (fetch && obj.od_code == code_pointer && !indirection_active) {
            ++indirection_active ;
            strcpy (indirected_item, item) ;
            strcat (indirected_item, ".all") ;
            item_action (indirected_item, pgm, quit) ;
            --indirection_active ;
         }

      }

      indent -= 3 ;
   }

   static
   void
   scope_action (char * scope, program_descriptor_t pgm, int * quit)
   {
      static int status ;

      printf ("scope = %s\n", (*scope ? scope : "<global>")) ;
      status = dm_list (list_variables,
                        scope,
                        "",
                        0,
                        pgm,
                        &item_action) ;
      assert (status, "dm_list") ;
   }

   int
   main (int argc, char * argv[])
   {
      auto program_descriptor_t  pgm ;
      auto int                   status ;
      auto char                * program ;
      auto int                   dummy ;

      if (argc < 2) {
         printf ("Usage: scanner program_name [fetch [variable_to_scan]]\n") ;
         exit (1) ;
      }
      program = argv[1] ;
      fetch   = argc > 2 && strcmp(argv[2],"fetch")==0 ;

      stack  = 0 ;
      max[0] = 2000000000 ;
      count[0] = 0 ;

      status = dm_open_program (program, 0, &pgm) ;
      assert (status, "dm_open_program") ;
```

```
   if (fetch) {
      status = dm_set_variant_handling (1, pgm) ;
      assert (status, "dm_set_variant_handling") ;
      status = dm_set_class_interpretation (1, pgm) ;
      assert (status, "dm_set_class_interpretation") ;
   }

   if (argc > 3) {
      item_action (argv[3], pgm, &dummy) ;
   } else {
      status = dm_list (list_scopes,
                        "",
                        "",
                        0,
                        pgm,
                        &scope_action) ;
      assert (status, "dm_list") ;
   }
}
```

For PowerMAX OS:

```
> ec -g scanner.c -o scanner -ldatamon -lud
```

For RedHawk Linux:

```
> gcc -g scanner.c -o scanner -ldatamon -lccur_rt -
lccur_fbsched
```

```
> scanner scanner fetch > out
```

```
> egrep -e 'fetch|ind|grat' out
   "scanner.c".fetch (int, 0x30081078, integer, signed, 32, 0
      [ 0x80000000..0x7fffffff ],  1)
   "scanner.c".indent (int, 0x3005c484, integer, signed, 32, 0
      [ 0x80000000..0x7fffffff ],  3)
   "scanner.c".indirection_active (int, 0x3005c488, integer, signed, 32, 0
      [ 0x80000000..0x7fffffff ],  0)
   "scanner.c".gratuitous_pointer (int *, 0x3005c48c, pointer, unsigned,
      32, 0, 3005c484)
      "scanner.c".gratuitous_pointer.all (int, 0x3005c484, integer,
         signed, 32, 0 [ 0x80000000..0x7fffffff ],  6)
```

```
> scanner peek
```

```
scope = main
scope = "peek.c"
   "peek.c".global_data (int, 0x3005c1b0, integer, signed, 32, 0
      [ 0x80000000..0x7fffffff ])
```

The example above provides source code and build instructions for a **`scanner`** program which scans a user-specified target executable program for scopes and describes the variables in those scopes. The description includes:

- The variable's name in *expanded notation*
- The variable's type name or type description
- The variable's address in the target program
- The variable's data monitoring `code`

- The variable's size in bits
- The variable's bit offset from its address
- The variable's constraints (if scalar)
- For record variables, a description of all a components of the record
- For array variables, a description of the dimensions and bounds of the array
- For array variables, a description of the first component of the array

Additional information is supplied when the **`scanner`** program is run in non-passive mode; defined by the second parameter to the program (false => passive, true => non-passive). When run in non-passive mode, the target program must be executing and the description output by **`scanner`** further includes the following:

- The value of the variable
- Class-wide type interpretation is activated
- Sensitivity to Ada record variants is activated
- Pointer variables are indirected (once)

The output of the **`scanner`** program can be rather lengthy since it describes all eligible variables in the target program. The **`egrep`** command was used above to show some of the output from the scanner invocations (the output underwent minor formatting changes for inclusion in this manual).

The first invocation shown above specifies that the program to scan is the **`scanner`** program itself; the second argument of `fetch` indicates that the scan is to be done in non-passive mode. The second invocation specifies the program from the **`peek`** example describes in this appendix; since that program mostly likely isn't executing, we run the scan in passive mode as indicated by the second argument is omitted.

Note that the scanner program utilizes the `dm_error_code_images` and `dm_code_images` arrays from **`/usr/include/datamon.h`**; these arrays are only available if the **`-Ddatamon_mappings`** compilation option is used or a `#define` of datamon_mappings is specified within the source code before the inclusion of **`/usr/include/datamon.h`**.

C
FORTRAN Examples

This appendix provides instructions for compilation and linking FORTRAN programs that use the Data Monitoring library as well as an example program.

Compilation and Linking Instructions

For PowerMAX OS:

```
> f77 -g source_file.f -ldatamon -lud
```

For RedHawk Linux:

```
> g77 -g source_file.f -ldatamon -lccur_rt -lccur_fbsched
```

The command above invokes the FORTRAN compiler on the source file **source_file.f**. The **-g** option specifies that debug information should be generated; this isn't specifically required for use of subprograms in the data monitoring library, however, the *target programs* must be built with debug information. The **-ldatamon** link option specifies that the Data Monitoring library, **/usr/lib/libdatamon.a**, should be used when linking the program

Example 1 — Peek

```
> cat peek.f

      program peek

      include "/usr/include/datamon_.h"
      include "/usr/include/datamon_tables_.h

      integer*4 pgm_desc
      integer*4 status
      integer*4 obj_desc(DM_descriptor_size)
      integer*4 value
      integer*4 i
      integer*4 low
      integer*4 high
      integer*4 bn
      integer*4 pn
      integer*4 vn
```

```
 common // obj_desc

 real*8 lower_bound
 real*8 upper_bound

 character*80 buffer
 character*80 program_name
 character*80 variable_name

 equivalence (obj_desc(DM_lower_bound),lower_bound)
 equivalence (obj_desc(DM_upper_bound),upper_bound)

 external zip
 external check_status

 call zip(program_name)
 call zip(variable_name)
 call zip(buffer)
 call getarg(1,program_name)

 pn=indx(program_name,' ')

 call getarg(2,variable_name)
 vn=indx(variable_name,' ')

 status =
1   dm_open_program(program_name(1:pn-1), 0, pgm_desc)
 call check_status(status,"dm_open_program")

 call getarg(2,variable_name)
 write(6,*)variable_name(1:vn-1),":"

 status = dm_get_descriptor(variable_name(1:vn-1),
1                           .false.,
2                           pgm_desc,
3                           obj_desc)
 call check_status(status,"dm_get_descriptor")

 status =
1   dm_get_type_name(buffer,variable_name,pgm_desc)
 call check_status(status,"dm_get_type_name")

 bn = indx(buffer,'@')

 write(6,*)" type_name = ",buffer(1:bn-1)
 write(6,*)" size = ",obj_desc(DM_bit_size)
 write(6,*)" address = ",obj_desc(DM_target_address)
 write(6,*)" code = ",code_names(obj_desc(DM_code))

 if (obj_desc(DM_code).eq.DM_array_code) then
    do 10 i=1,obj_desc(DM_num_dimensions)
       write(6,*) " dimension = ",
1         obj_desc(DM_lower_dimension+i-1),
2         " .. ",
```

```
     3         obj_desc(DM_upper_dimension+i-1)
10       continue
      elseif (obj_desc(DM_code).eq.DM_enumeration_code) then
         low = int(lower_bound)
         high = int(upper_bound)
         write(6,*)" enum_info = "
         do 20 i=low,high
            call zip(buffer)
            status = dm_get_enum_image(buffer,
     1                                 variable_name(1:vn-1),
     2                                 i,
     3                                 pgm_desc)
            call check_status(status,"dm_get_enum_image")
            bn = indx(buffer,'@')
            status = dm_get_enum_val(variable_name(1:vn-1),
     1                               i,
     2                               value,
     3                               pgm_desc)
            call check_status(status,"dm_get_enum_val")
            write(6,*)" ",buffer(1:bn-1)," => ",value
20       continue
         call zip(buffer)
         status = dm_get_value(buffer,obj_desc)
         call check_status(status,"dm_get_value")
         bn = indx(buffer,'@')
         write(6,*)" value = ",buffer(1:bn-1)
      else
         call zip(buffer)
         status = dm_get_value(buffer,obj_desc)
         call check_status(status,"dm_get_value")
         bn = indx(buffer,'@')
         write(6,*)" value = ",buffer(1:bn-1)
      end if
      status = dm_close_program (pgm_desc)
      if(status .ne. 0) then
         write(6,*) "error from dm_close_program",
     1   dm_get_error_code(),dm_get_error_string()
         call exit( -1 )
      end if
      end

      subroutine zip (buf)
      character*(*) buf
      integer*4 i
      do 10 i=1,len(buf)
         buf(i:i) = '@'
10    continue
      end

      subroutine check_status (status, service)
      include "/usr/include/datamon_.h"
      include "/usr/include/datamon_tables_.h"
      integer*4 status
      character*(*) service
```

```
      integer*4 n
      if(status .ne. 0) then
         n = indx(error_code_names(dm_get_error_code()),' ')
         write(6,*) service, ": ",
     1      error_code_names(dm_get_error_code())(1:n-1),
     2      ": ",dm_get_error_string()
         write(6,*)server," failed with error",
     3      dm_get_error_code()
         call exit( -1 )
      end if
      end

      function indx (string, char)
      character*(*) string
      character char
      do 30 i=1,len(string)
         if (string(i:i) .eq. char) then
            indx = i
            return
         end if
30    continue
      indx = 0
      end
```

For PowerMAX OS:

```
> f77 -g peek.f -o peek -ldatamon -lud

> peek peek "peek.obj_desc"
 peek.obj_desc:
    type_name = integer*4 []
    size      =  1280
    address   =  805850216
    code      = array
    dimension =  1  ..  40

> peek peek "peek.obj_desc(6)"
 peek.obj_desc(6):
    type_name = integer*4
    size      =  32
    address   =  805850236
    code      = integer
    value     =  32
```

For RedHawk Linux:

```
> g77 -g peek.f -o peek -ldatamon -lccur_rt -lccur_fbsched

> peek peek "MAIN__.obj_desc__"
 MAIN__.obj_desc:
    type_name = integer []
```

```
    size      =  1280
    address   =  805850216
    code      = array
    dimension =  1  ..  40

> peek peek "MAIN__.obj_desc__(6)"
 MAIN__.obj_desc(6):
    type_name = integer
    size      =  32
    address   =  805850236
    code      = integer
    value     =  32
```

The example above provides source code and build instructions for a **peek** program which peeks into an executing user-specified target program and obtains the value of a user-specified variable and information about that variable. The description includes:

- The variable's name in *expanded notation*
- The variable's type name or type description
- The variable's size in bits
- The variable's address in the target program
- The variable's data monitoring `code`
- For array variables, a description of the dimensions and bounds of the array
- A description of the enumeration constants of enumeration variables
- The value of the variable (for non-composite variables)

Note that the **peek** program makes use of the `error_code_names` and `code_names` arrays which are defined in the include file, **/usr/include/datamon_tables_.h**.

Note:

On RedHawk Linux, if the test program is built with GNU Fortran, note that the names of the program and variables are mangled. This is due to a mangled description of those items in the debug information that the GNU Fortran compiler produces. The name of the main program is MAIN__, regardles of any supplied name. The name of the varible obj_desc is manged to obj_desc__.

Index

O

P

R

S

T

V

W

Spine for 1.0” Binder

Product Name: 0.5" from top of spine, Helvetica, 36 pt, Bold

Volume Number (if any): Helvetica, 24 pt, Bold

Volume Name (if any): Helvetica, 18 pt, Bold

Manual Title(s): Helvetica, 10 pt, Bold, centered vertically within space above bar, double space between each title

Bar: 1" x 1/8" beginning 1/4" in from either side

Part Number: Helvetica, 6 pt, centered, 1/8" up

Data Monitoring

Reference Manual

0890493