Mica Working Design Document
Application Run-Time Utility Services

Revision 0.5
24–March–1988

Sections 1 + 2

Issued by:
Al Simons

digital™

TABLE OF CONTENTS

TABLES

Revision History

Date	Revision Number	Author	Summary of Changes
19-November-1987	0.1	All	First draft
27-December-1987	0.2	Connors	Incorporated review comments. Split original chapter entitled "Applications Run-Time Library" into two chapters: "Application Run-Time Utility Services" and "Miscellaneous Run-Time Library Routines."
8-February-1988	0.3	Simons, Nogrady	Prepare sections on memory management and date/time manipulation for primary review.
4-March-1988	0.4	Simons	Incorporate primary review comments on the above sections.
14-March-1988	0.5	Simons, Nogrady	Prepare sections on condition handling and conversions for primary review.

CHAPTER 1

APPLICATION RUN-TIME UTILITY SERVICES

1.1 Overview

This chapter describes the interfaces to the Application Run-Time Utility Services (ARUS) library. This library will be implemented on Mica and on PRISM ULTRIX in time for each product's release. It will also be implemented on future releases of VAX/VMS and VAX/ULTRIX.

The ARUS library contains routines that provide the application program interface to Mica on Glacier, and provides that same interface on the other operating systems on which it is implemented, thereby easing portability of applications across DIGITAL operating systems. These routines are designed to adhere to the emerging Application Integration Architecture (AIA). The definition and development of ARUS on Mica is the result of a cooperative effort between DECwest and SDT. The major part of the implementation of ARUS is performed by SDT.

There are several discrete groups of routines contained in ARUS. Each of these groups is discussed in turn starting with Section 1.1.2.1, which describes the ARUS routines used to allocate and deallocate virtual memory.

The Mica applications run-time library also contains other application program interface routines that complement the capabilities provided by the routines described in this chapter. These additional routines are described in Chapter 57, Miscellaneous Run-Time Library Routines.

1.1.1 Goals and Requirements

ARUS shares many of the goals and requirements of the AIA program. Requirements include:

- ARUS routine interface implementations must be feasible on all Glacier client systems.
- ARUS routine definitions must allow for implementations with good performance.
- ARUS routine implementations must be compatible with other non-Mica implementations of the routines.

Goals include:

- To provide as complete a program interface as possible to contemporary DIGITAL-supplied operating systems such as Mica, VAX/VMS, and ULTRIX without including nonportable concepts or constructs.
- To provide a set of routines that are architected in such a fashion as to allow efficient library routine code implementations on all such contemporary DIGITAL operating systems.

Nongoals include:

- The code for ARUS routines must be inherently portable. (The AIA architecture requires that only the *interfaces* to AIA routines be portable.)
- ARUS routines provide interfaces to every underlying operating system capability or architecture-specific hardware feature.

- The performance of ARUS routines must on average exceed that of similar, non-AIA operating-system or architecture-specific routines. \There is a cost for portability.\

1.1.2 ARUS Routines

Although the ultimate version of ARUS will include a wide range of routines, the FRS offering is necessarily limited in scope. The FRS version of ARUS comprises those routines needed to support the FRS layered products and bundled utilities. This section discusses only the utility RTL capabilities for those areas in which there are FRS requirements.[1]

ARUS is composed of two conceptually different types of routines: generic operating system services and general purpose utility routines.

The generic operating system services provide, in an operating-system- and architecture-independent manner, those services normally associated with an operating system, such as virtual memory allocation. These routines are described starting at Section 1.1.2.1.

The general purpose utility routines provide access to common capabilities generally identified with run-time libraries, such as various data conversion routines. These routines are described starting at Section 1.1.2.8.

1.1.2.1 User Mode Virtual Memory Allocation/Deallocation Routines

ARUS contains user-level memory allocation and deallocation routines similar to the VAX/VMS LIB$VM routines. Unlike the LIB$VM routines, the ARUS routine interfaces do not use hardware-specific allocation units, such as pages. All quantities are expressed in terms of bytes.

\It is interesting to note that in a measurement made of the VMS RTL, the memory management routines were the most frequently used of any RTL routines by a factor of 10. The performance of these routines is critical, especially of *arus$get_memory*.[2]

User mode virtual memory allocation/deallocation routines include:

- *arus$get_memory*—mandatory for FRS
- *arus$free_memory*—mandatory for FRS
- *arus$create_memory_zone*—mandatory for FRS
- *arus$delete_memory_zone*—mandatory for FRS
- *arus$reset_memory_zone*

[1] The document "Overview of a New Utility RTL" by Al Simons (contained in the "AIA Strawman") contains descriptions of capabilities for the eventual ARUS library that are not represented in this chapter. All such omissions indicate that the capability described is not a realistic FRS deliverable.

[2] The spelling of all ARUS routine name prefixes, is TBD. The final routine names will have prefixes that serve to reinforce the logical grouping of the routines.

1.1.2.2 Condition Handling Routines

The ARUS condition handling routines provide an AIA-compatible interface to the Mica condition handling system. They allow the user to raise, modify, handle, and obtain information about conditions in an operating-system-independent manner.

The condition handling routines implement a dynamic condition dispatching environment whose semantics are based on the order of procedure invocation. This style of condition handling is identical to that present on VAX/VMS, Mica, and PRISM ULTRIX. The implementation of these routines utilizes the underlying operating-system-specific condition handling features. Note, however, that these routines do not operate with the traditional UNIX™ static signal handling capabilities;[3] however, the two condition handling systems do coexist, and their use can be intermixed.

The ARUS condition handling routines allow for access to the information in a condition record in an operating-system-independent manner. The routines do not provide access to the mechanism record except in a controlled way, for example, to replace the return value registers contained therein.

Note that these routines do not provide the capability of VAX/VMS routines LIB$ESTABLISH and LIB$REVERT. Those routines depend very heavily on peculiarities of the VAX architecture, and are not portable. Compilers are expected to catch references to those routines, and do "the right thing." What "the right thing" is depends on the operating system and hardware for which the code is being compiled.

Condition handling routines include:

- *arus$raise_condition*—mandatory for FRS (FORTRAN, Pascal)
- *arus$replace_condition*
- *arus$add_primary_condition*
- *arus$add_secondary_condition*
- *arus$examine_condition*—mandatory for FRS (for applications not coded in Pillar)
- *arus$unwind*—mandatory for FRS (FORTRAN, Pascal)
- *arus$unwind_to_exit*—mandatory for FRS (FORTRAN, Pascal)
- *arus$store_return_value*—mandatory for FRS (FORTRAN, Pascal)
- *arus$examine_return_value*
- *arus$test_for_success*
- *arus$compare_status*
- *arus$add_primary_handler*—not mandatory if DEBUG goes straight to the system as expected
- *arus$add_last_chance_handler*—mandatory for FRS (FORTRAN, Pascal)
- *arus$delete_primary_handler*
- *arus$delete_last_chance_handler*

\It has not been decided whether there will be routines to map conditions from the underlying system's condition facility into common AIA conditions, or whether there will be routines to provide the means to obtain the condition name in a system-independent manner.

The question is: how does an application test for a condition such as end-of-file when the language does not provide that mapping? Will an ARUS routine map SS$_ENDOFFILE to the equivalent PRISM ULTRIX and Mica condition names or is that the responsibility of the application?

™ UNIX is a trademark of AT&T

[3] That is, the condition handling routines available in UNIX whose actions are determined by the contents of a program's "signal vector." For more information about these incompatible condition handling routines, please see Chapter 2 of the UNIX documentation.

How thoroughly can we isolate the user from the underlying condition handling system?

We recognize that this is a desireable capability, but it is currently an unknown technical problem, and we are not sure if it can be understood and implemented in time. We believe that the PRISM systems are viable without this capability.\

1.1.2.3 Date and Time Conversion Routines

The date and time conversion routines are used to convert internal format time into text, text into internal format time, and to obtain and manipulate internal format time values. They allow flexibility of natural language and format in both directions of conversion. These routines recognize and process the DIGITAL standard internal time format, as specified in standard EL-EN112-00, "Representation of Time for Information Exchange." On ULTRIX, there are additional routines to convert between the UNIX standard time format and the DIGITAL standard format.

Date and time conversion routines include:

- *arus$get_system_time*—mandatory for FRS
- *arus$format_date_time*—mandatory for FRS
- *arus$format_rel_time*—mandatory for FRS
- *arus$convert_date_string*—mandatory for FRS
- *arus$convert_rel_time_string*—mandatory for FRS
- *arus$free_date_time_context*—mandatory for FRS
- *arus$get_date_format*—mandatory for FRS
- *arus$get_max_date_length*—mandatory for FRS
- *arus$cvt_to_numeric_rel_time*—mandatory for FRS
- *arus$cvt_to_numeric_abs_time*—mandatory for FRS
- *arus$cvt_from_numeric_rel_time*—mandatory for FRS
- *arus$cvt_from_numeric_abs_time*—mandatory for FRS
- *arus$cvt_to_binary_rel_time*—mandatory for FRS
- *arus$cvtf_to_binary_rel_time*—mandatory for FRS
- *arus$cvt_from_binary_rel_time*—mandatory for FRS
- *arus$cvtf_from_binary_rel_time*—mandatory for FRS
- *arus$cvt_from_binary_abs_time*—mandatory for FRS
- *arus$init_date_time_context*—mandatory for FRS
- *arus$add_mixed_times*
- *arus$add_relative_times*
- *arus$subtract_absolute_times*
- *arus$subtract_relative_times*
- *arus$subtract_mixed_times*
- *arus$compare_relative_times*
- *arus$compare_absolute_times*

1.1.2.4 Environment Attribute Routines

The environment attribute routines provide the ability to look up an attribute defined in the user's environment, and return the string which is the value of that attribute. Since attributes are also strings, attribute lookup can be nested. The complete architecture for these routines provides for a capability similar to that available with VAX/VMS logical names, including the ability to have secure attributes.

The FRS offering of environment attribute routines is more modest. At a minimum level of capability for FRS, these routines provide a uniform access to the underlying operating system string mapping capability (logical names on VAX/VMS and Mica, environment variables on ULTRIX systems). This FRS support includes the ability to map a string to a single string, but without any protection from user modification of the mapping.

Environment attribute routines include:

- *arus$create_environment_attribute*—mandatory for FRS
- *arus$get_attribute_value*—mandatory for FRS
- *arus$delete_environment_attribute*—mandatory for FRS
- *arus$create_attribute_table*
- *arus$delete_attribute_table*

At first release, these routines will not interact with extended environments such as the DECnet name server. Whether or not they will in the future is not yet determined. In tightly bound processes such as Mica bound processes, these routines will recognize the client's environment.

1.1.2.5 Internationalization Aids

The ARUS library provides several routines to aid in the internationalization of applications. They include support for specifying different collating sequences, obtaining the user's natural language, formatting numeric values, and so on. Some of these routines are tightly integrated with routines discussed in other areas and are described with those routines. Routines that exist solely for internationalization are described here.

- *arus$get_language*—mandatory for FRS
- *arus$radix_point*
- *arus$digit_separator*
- *arus$format_currency*
- A string-collating package similar to the VAX/VMS NCS$ routines provided in VAX/VMS Version 5.0
- A string case conversion utility

1.1.2.6 Process Information Routines

Pascal has a requirement to obtain the amount of CPU time consumed by the process. That is the only currently known requirement for process information routines.

1.1.2.7 Command Language Interpreter Interface Routines

The command language interpreter (CLI) interface routines are used to provide a portable method for applications to receive and parse simple command lines. The format of the command lines is operating system specific and these routines only enforce the concepts of command verb, command parameter, command qualifier, and so on, without resorting to describing the lexical representation of these entities. The method for describing commands, parameters, and qualifiers is <TBS>.

The CLI interface routines also provide for obtaining the unparsed command line. Additionally, a routine is provided to meet the requirement of the FORTRAN RTL to be able to pause program execution and return control to the CLI.

1.1.2.8 Data Conversion Routines

Virtually all of the capabilities present in the VAX/VMS OTS$ data type conversion routines are required at FRS to support FORTRAN. Please see the documented OTS$ definitions.

1.1.2.9 Text String and Message Formatting Routines

The capability needed for text string and message formatting is similar to the $FAO system service on VAX/VMS, and the *printf* statement in the C language. Like those facilities, the Mica text string and formatting routines are driven by a control string. Unlike those facilities, they include inherent support for internationalization.

Text string and message formatting routines include:

- *arus$format_string*—mandatory for FRS
- *arus$format_message*

1.1.2.10 String Routines

The string routines handle string allocation, copying, and deallocation. They closely resemble the current VAX/VMS STR$ routines that provide these capabilities. Please refer to the VAX/VMS documentation.

1.1.2.11 Table-Driven Parsing Routines

FORTRAN NAMELIST I/O currently utilizes the VAX/VMS routine named LIB$TPARSE to perform the parsing actions required. This general capability should be provided eventually in ARUS; if it is not available at FRS, the FORTRAN RTL will have to provide its own parsing routines.

1.1.2.12 Math Routines

Math support routines exist at two levels on Mica:

- A set of low-level routines designed for use by language run-time libraries and other callers where absolute performance is paramount. The interfaces to these routines are compatible with the VAX/VMS implementations of the routines. The low-level routines are described in Chapter 57, Miscellaneous Run-Time Library Routines.
- A set of high-level math routines with AIA-conformant interfaces. These routines are used where absolute performance is secondary to portability. The high-level routines are described in this chapter. Table 1–1 lists the entry points for these routines.

Table 1–1: High-Level Math Routines

math$tbs

1.1.3 Open Issues

- How to provide transportable condition handling is the area that is currently least understood. We believe that the routines described in Section 1.1.2.2 are necessary and feasible. Our current model may, however, change over the next several months as we learn more in this area.
- The most pressing issue in the area of the math routines is the lack of a definition of AIA-conformant math routine interfaces. This is delayed by the lack of a precise definition of the phrase "AIA-conformant."
- The concept of seamlessness between Glacier and its clients suggests that ARUS needs to be implemented at or near FRS on all possible Glacier client systems. This increases the overall effort and is potentially problematic under the current manpower constraints.

1.2 ARUS Routine Design Philosophy

The primary goal of the Application Integration Architecture is to provide interfaces to commonly used library routines which are operating system and hardware independent. This goal requires that the ARUS routines, which are AIA conformant, be written on a higher level of abstraction than regular system routines. For example, some type definitions may need to be defined differently in an ARUS routine than they would be in an ordinary PRISM or VAX system service.

In order to maintain hardware independence, it is occasionally necessary to duplicate the capabilities provided by a PRISM system service. That is, the ARUS routine may simply map the arguments it receives onto those of a PRISM service, then call that service directly.

\A note on ARUS routine design documentation: In a similar fashion, this chapter serves multiple purposes. The first is to be a part of the design of the Mica operating system. The second is to serve as the definition of the first set of routine interfaces of the AIA RTL routines which will be provided on multiple operating systems.

This merging of two sets of goals with two somewhat different audiences into one manuscript runs the risk of satisfying neither. I hope that I have succeeded in satisfying both.\

1.3 Memory Allocation and Deallocation Routines

The software described in this section provides a user-mode memory manager. By managing a pool (or heap) of memory in user mode with only infrequent allocation requests to the operating system, overall system performance is improved.

Throughout this section, the term *allocation* refers to allocating memory from the memory manager's pool for use by the application code. Occasionally, we discuss allocation from the operating system. These cases are clearly specified in the the accompanying text. Similarly, the term *deallocation* refers to returning memory to the memory manager when the application code is through with it. This section very rarely discusses deallocating memory from a process's address space through an operating system service. However, when such a deallocation is mentioned, it is clearly specified.

Throughout this chapter, we will use the term *octet* to refer to a unit of memory containing exactly eight bits; an octet is the same as a VAX architecture byte. The term "byte" is not used in this chapter, because it is not well defined; there are eight-bit bytes and nine-bit bytes.

The routines providing user mode memory allocation and deallocation have the following design goals:

- They must be fast.

- They must provide the user with the ability to allocate memory using different algorithms to fit several common cases.
- They must not block.
- They must work in a multithreaded and multiprocessing environment.

The routines must support the following operations:

- Allocate memory.
- Deallocate memory.
- Create a *memory zone*. (A memory zone is the way of defining the allocation and deallocation algorithms to be used, along with other desired characteristics of the allocated memory. See Section 1.3.1.)
- Delete a memory zone, returning to the pool the memory allocated to the user from the zone, as well as the memory associated with the zone's control structures.
- Reset a memory zone, returning to the pool the memory allocated to the user from the zone, but leaving the zone structure intact, ready for reuse.

1.3.1 Memory Zone Characteristics

Applications frequently need to allocate memory with certain characteristics, such as fixed size, 1024 octet blocks of memory, or memory aligned on 8 octet boundaries. Also, application writers can frequently make performance improvements by selecting a particular allocation algorithm based on their knowledge of the application's memory use. The application programmer defines different *memory zones* to allow the application to tailor its memory management.

Conceptually, a memory zone is a region of memory available for allocation to the application program. A memory zone has associated with it an allocation and deallocation algorithm pair and some number of memory characteristics. The following sections describe the allocation algorithms and memory characteristics associated with memory zones.

1.3.1.1 Allocation and Deallocation Algorithms

Table 1–2 lists the allocation and deallocation algorithms defined for memory zones.

Table 1–2: Allocation and Deallocation Algorithms

Algorithm	Symbol	Description
First fit	*arus$c_first_fit*	Allocate memory from the first available block that is at least as large as the request.
Quick fit	*arus$c_quick_fit*	Allocate memory from a lookaside list of appropriate size.
Frequent sizes	*arus$c_frequent_sizes*	Allocate memory from a lookaside list of appropriate size.
Fixed size blocks	*arus$c_fixed_size*	Allocate blocks of one fixed size only.
C compatibility	*arus$c_c_compatibility*	Allocate memory in a way that is compatible with the C memory management routines *calloc*, *malloc*, *realloc* and *free*.

Both the *arus$c_quick_fit* and *arus$c_frequent_sizes* algorithms allocate memory from a *lookaside list* when possible. The difference is that for the *arus$c_quick_fit* algorithm, the application writer is in control of the sizing of the lookaside lists, whereas for the *arus$c_frequent_sizes* algorithm, the

memory manager determines the sizing of the lookaside lists based on the actual values of memory returned through calls to *arus$free_memory*.

1.3.1.2 Alignment

A memory zone also has an alignment attribute associated with it. This controls the alignment of the low address of every block of memory allocated from the zone.

1.3.1.3 Allocation Sizes

A memory zone has two allocation sizes associated with it. These sizes do not in any way affect the amount of memory that can be allocated in one call to routine *arus$get_memory*. One value is the initial size; this is the amount of memory that is initially allocated to the memory zone when the zone is created. The other is the extend size; this represents a minimum amount of memory that will be requested from the operating system when the free memory under the control of the memory manager is insufficient to satisfy an allocation request. It is maximized with the size of the actual request to determine the size of the request to the operating system.

1.3.2 Functional Interface and Description

The following sections describe the various routines associated with memory allocation and deallocation, and with memory zone creation and management.

1.3.2.1 Types Used

The following types are used in the interface to the memory manager routines.

\WDD readers: remember that this chapter is both a part of the MICA WDD, and a general AIA interface spec, to be used on other operating systems. For this reason, all of the types used in these interfaces, including statuses and untyped pointers must be abstractly specified.\

```
TYPE
    arus$untyped_pointer : POINTER anytype;

    !
    ! The following is a pointer to the control block for memory zones.
    ! That control structure's definition is never made public.
    ! The pointer is typecast to the appropriate type by the
    ! memory management routines.
    !
    arus$memory_zone : POINTER anytype;

    arus$status : STATUS;

    arus$memory_algorithm_type : (
        arus$c_first_fit,
        arus$c_quick_fit,
        arus$c_frequent_sizes,

        arus$c_fixed_size,
        arus$c_c_compatibility

        );

    arus$memory_algorithm : SET [ arus$memory_algorithm_type ];

    arus$memory_zone_options_type : (

        arus$c_zero_on_allocation,

        arus$c_boundary_tags
        );

    arus$memory_zone_options : SET [ arus$memory_zone_options_type ];
```

1.3.2.2 Allocation

There are two routines used to allocate memory, *arus$get_memory* and *arus$reallocate_memory*.

1.3.2.2.1 The *arus$get_memory* Routine

The routine *arus$get_memory* is used to obtain memory. It is called with the number of octets desired and a designation of the zone to be used for allocation, and returns the starting address of the allocated memory via an OUT parameter. It also returns the status as the function result. The characteristics of the memory segment allocated are determined by the zone from which it was allocated. For more information, refer to Section 1.3.1.

```
PROCEDURE arus$get_memory (
                IN number_of_octets : integer;
                OUT starting_address : arus$untyped_pointer;
                IN memory_zone : arus$memory_zone OPTIONAL;
                ) RETURNS arus$status
                  LINKAGE
                      REFERENCE (
                          number_of_octets,
                          starting_address,
                          memory_zone
                          );
```

Parameters:

number_of_octets	The number of octets of memory that are to be allocated to the program.
starting_address	The low address of the allocated memory segment.
memory_zone	The zone from which the memory is to be allocated.

Routine *arus$get_memory* returns the unsuccessful status values listed in Table 1–3.

Table 1–3: Status Values Returned from Routine *arus$get_memory*

Status Value	Description
arus$_memory_limit	This indicates that the limit on memory size for the process was reached, and a complete allocation was not possible. Partial allocations are not made.
arus$_positive_size_required	This indicates that the *number_of_octets* argument was either zero or a negative value.
arus$_nonexistent_zone	This indicates that the zone specified either was never created, or existed at one time and has been deleted.

Routine *arus$get_memory* raises no conditions.

1.3.2.2.2 The *arus$reallocate_memory* Routine

The *arus$reallocate_memory* routine is used to conceptually extend or contract a segment of memory. It corresponds to the *realloc* library routine available on ULTRIX. When extending, if the memory passed to the routine can be extended contiguously to the desired size, then it is. If not, a new block of the desired size is allocated, the contents of the old block are copied to the new block, and then the old block is deallocated. When contracting, the memory after the end of the new size is deallocated.

This routine may only be used on zones whose allocation algorithm is *arus$c_c_compatibility*.

```
PROCEDURE arus$reallocate_memory (
                IN number_of_octets : integer;
                IN OUT starting_address : arus$untyped_pointer;
                IN memory_zone : arus$memory_zone OPTIONAL;
                ) RETURNS arus$status
                  LINKAGE
                      REFERENCE (
                          number_of_octets,
                          starting_address,
                          memory_zone
                          );
```

Parameters:

number_of_octets	The number of octets of memory that are to be allocated to the program.
starting_address	The low address of the allocated memory segment.
memory_zone	The zone from which the memory is to be allocated. If omitted, this defaults to *arus$c_default_memory_zone*.

Routine *arus$reallocate_memory* returns the unsuccessful status values listed in Table 1–4.

Table 1–4: Status Values Returned from Routine *arus$reallocate_memory*

Status Value	Description
arus$_memory_limit	This indicates that the limit on memory size for the process was reached, and a complete allocation was not possible. Partial allocations are not made.
arus$_positive_size_required	This indicates that the *number_of_octets* argument was either zero or a negative value.
arus$_nonexistent_zone	This indicates that the zone specified either was never created, or existed at one time and has been deleted.
arus$_not_c_compatibility_zone	This indicates that *arus$reallocate_memory* was attempted on a zone whose algorithm is not *arus$c_c_compatibility*. This is not allowed.

Routine *arus$reallocate_memory* raises no conditions.

1.3.2.3 Deallocation

There is only one memory deallocation routine, *arus$free_memory*.

1.3.2.3.1 The *arus$free_memory* Routine

The routine *arus$free_memory* is used to deallocate memory, thus freeing it for reuse by the application through subsequent calls to *arus$get_memory*. The procedure is invoked with the starting address of the memory to be returned, the zone from which it was allocated and, optionally, the number of octets which are being returned. The number of octets is mandatory for most allocation algorithms and ignored for the *arus$c_fixed_sizes* and *arus$c_c_compatibility* algorithms.

Some allocation algorithms allow for the partial return of memory. For instance, if 1000 octets are allocated in one call to *arus$get_memory*, some allocation algorithms allow the return of fewer than 1000 octets with *arus$free_memory*. Returning memory not obtained through *arus$get_memory* is not allowed, nor is returning memory obtained through more than one call to *arus$get_memory* (merging memory blocks).

Memory which is deallocated by *arus$free_memory* is returned to the memory manager used by *arus$get_memory* and *arus$free_memory*. It is not necessarily removed from the process's address space. Therefore, incorrectly coded programs might be able to reference deallocated memory without incurring an access violation or similar fault.

The behavior of an application is undefined if memory which has been freed by a call to *arus$free_memory* is referenced.

```
PROCEDURE arus$free_memory (
              IN starting_address : arus$untyped_pointer;
              IN number_of_octets : integer OPTIONAL;

              IN memory_zone : arus$memory_zone OPTIONAL;

              ) RETURNS arus$status
                LINKAGE
                    REFERENCE (
                        starting_address,
                        number_of_octets,
                        memory_zone
                        );
```

Parameters:

starting_address	The low address of the allocated memory segment.
number_of_octets	The number of octets of memory that are to be deallocated from the program.
memory_zone	The zone from which the memory is to be allocated. If omitted, this defaults to *arus$c_default_memory_zone*.

Routine *arus$free_memory* returns the unsuccessful status values listed in Table 1–5.

Table 1–5: Status Values Returned from Routine *arus$free_memory*

Status Value	Description
arus$_block_alignment	This indicates that the returned block of memory was not aligned on a boundary at least as great as that required by the zone.
arus$_partial_block	This indicates that a partial free was attempted in a zone whose allocation algorithm does not permit it.
arus$_not_allocated	This indicates that the memory to be freed was not allocated by routine *arus$get_memory*. This condition is also returned in the event that a merged return is attempted.
arus$_positive_size_required	This indicates that the *number_of_octets* argument was either zero or a negative value.
arus$_nonexistent_zone	This indicates that the zone specified either was never created, or existed at one time and has been deleted.

Routine *arus$free_memory* raises no conditions.

1.3.2.4 Memory Zone Creation

There is only one routine involved with memory zone creation, *arus$create_memory_zone*.

1.3.2.4.1 The *arus$create_memory_zone* Routine

Routine *arus$create_memory_zone* is used to create a memory zone with characteristics that differ from those of the default zone, which is described in Section 1.3.2.4.2. It does not, of itself, make any memory available to the application; however, it may allocate memory from the operating system. This routine must be called before *arus$get_memory* is called.

```
PROCEDURE arus$create_memory_zone (
                OUT memory_zone : arus$memory_zone;

                IN algorithm : arus$memory_algorithm OPTIONAL;
                IN algorithm_argument : integer OPTIONAL;
                IN options : arus$memory_zone_options OPTIONAL;
                IN extend_size : integer OPTIONAL;
                IN initial_size : integer OPTIONAL;
                IN block_size : integer OPTIONAL;
                IN alignment : integer OPTIONAL;
                IN first_quick_fit_list : integer OPTIONAL;
                IN allocation_routine : arus$allocation_routine OPTIONAL;
                IN deallocation_routine : arus$deallocation_routine OPTIONAL;

                ) RETURNS arus$status
                  LINKAGE
                      REFERENCE (
                          memory_zone,
                          algorithm,
                          algorithm_argument,
                          options,
                          extend_size,
                          initial_size,
                          block_size,
                          alignment,
                          first_quick_fit_list,
                          allocation_routine,
                          deallocation_routine
                          );
```

Parameters:

memory_zone — The identifier of the created memory zone. It is this value that must be passed to any other routines requiring a memory zone.

algorithm — The algorithm to be used in allocating memory from this zone. Section 1.3.1 discusses the different algorithms. If omitted, this parameter defaults to *arus$c_first_fit*.

algorithm_argument — An algorithm-specific argument, as specified below.

Algorithm	Argument Interpretation
First fit, C compatibility	Not used, argument ignored.
Quick fit	The number of lookaside lists to be used.
Frequent sizes	The number of lookaside lists to be used.
Fixed size blocks	The number of octets in each block to be allocated from this zone.

options — Controls various optional behaviors of the zone, as specified below.

Option Value	Effect on Allocation and Deallocation
arus$c_zero_on_allocate	Zero the memory block returned by *arus$get_memory*, and the memory returned by *arus$reallocate_memory* that is past the end of the original block's size.
arus$c_boundary_tags	Use boundary tags. Boundary tags are markers that are placed immediately before and immediately after the memory that is returned to the user. They allow memory to be freed faster and provide enhanced error checking. Boundary tags add a minimum of eight octets to each block allocated.

If omitted, this parameter defaults to no options.

extend_size — The minimum number of octets to be added to the allocation area when a memory request is made via *arus$get_memory* for more memory than is currently available. If omitted, this parameter defaults to the value *arus$c_default_extend_memory*.

initial_size — The number of octets initially in the allocation area when the memory zone is created. This may be specified as zero, in which case *extend_size* octets are added at the time of the first memory request. If omitted, this parameter defaults to the value *arus$c_default_initial_memory*.

block_size — For memory zones not using fixed size allocations, the minimum size block that can be allocated, and the basic unit of allocation for larger allocations. In other words, all allocation requests are rounded up to a multiple of this value. If omitted, this parameter defaults to the value *arus$c_default_block_size*.

alignment — The alignment of memory allocated from this zone. The alignment must be a power of two and greater than or equal to four. The maximum value is implementation defined to allow alignment on a protection boundary. Each implementation of this routine provides the symbolic constant *arus$c_protection_alignment* for applications which need such alignment. If omitted, this parameter defaults to the value *arus$c_default_memory_alignment*.

first_quick_fit_list — For memory zones utilizing the quick fit algorithm, this parameter indicates the block size to be stored on the first lookaside list, the list with the smallest blocks of memory. If omitted or if smaller then *block_size*, this value defaults to the value of parameter *block_size*.

allocation_routine — The routine to be used when memory must be allocated to the zone from the operating system. If not specified, an implementation-specific routine is used.

deallocation_routine — The routine to be used when memory must be deallocated from the zone back to the operating system. If not specified, an implementation-specific routine is used.

The statuses returned by *arus$create_memory_zone* are described in Table 1–6.

Table 1–6: Status Values Returned from Routine *arus$create_memory_zone*

Status Value	Description
arus$_insufficient_memory	There was not sufficient remaining memory to allocate space for the zone control structures or the initial memory for the allocation area.
arus$_invalid_algorithm	The *algorithm* argument did not specify a legal algorithm.
arus$_invalid_algorithm_argument	The *algorithm_argument* argument was not legal for the specified algorithm.
arus$_invalid_options	The *options* argument was illegal.
arus$_invalid_extend_size	The *extend_size* argument specified a negative or zero size.

Table 1–6 (Cont.): Status Values Returned from Routine *arus$create_memory_zone*

Status Value	Description
arus$_invalid_initial_size	The *initial_size* argument specified a negative or zero size.
arus$_invalid_block_size	The *block_size* argument specified a negative or zero size.
arus$_invalid_alignment	The *alignment* argument specified a size that was either not a power of two, or was less than four, or was greater than *arus$c_protection_alignment*.

Routine *arus$create_memory_zone* does not raise any conditions.

1.3.2.4.2 The Default Memory Zone

There is a default memory zone which the user does not need to explicitly create in order to use. While the actual mechanism by which this zone is created is implementation specific, the semantics of the zone are that it is created during activation of the image containing routine *arus$get_memory*. In other words, it already exists at the time of the user's first call to *arus$get_memory*. To use the default zone, the application references the symbol *arus$c_default_memory_zone* in the call to *arus$get_memory* or *arus$free_memory*, or omits the parameter altogether. The default zone's characteristics are listed in Table 1–7.

The default memory zone may not be deleted.

Table 1–7: Characteristics of the Default Memory Zone

Algorithm	First fit
Algorithm argument	Not applicable
Options	None
Extend size	Implementation defined, based on performance modelling and/or instrumentation during development. Each implementation provides the symbolic constant *arus$c_default_extend_memory*, so that applications may take advantage of this modelling and/or instrumentation for explicitly created memory zones when appropriate.
Initial size	Implementation defined. Each implementation provides the symbolic constant *arus$c_default_initial_memory*.
Alignment	Implementation defined. The minimum value needed for the largest of the alignments required for: • Efficient referencing of cells containing addresses • Efficient referencing of cells containing the largest numeric data type supported by the system • Use of interlocked instructions provided by the hardware. Each implementation provides the symbolic constant *arus$c_default_memory_alignment* which specifies this value.
Allocation routine	Implementation defined. An internal routine for allocating user mode memory. This is also the routine that is used if a call to *arus$create_memory_zone* does not specify a procedure in the *allocation_routine* parameter.
Deallocation routine	Implementation defined. An internal routine for deallocating user mode memory. This is also the routine that is used if a call to *arus$create_memory_zone* does not specify a procedure in the *deallocation_routine* parameter.

1.3.2.5 Memory Zone Reset

There is one routine used to reset a memory zone, *arus$reset_memory_zone*.

1.3.2.5.1 The *arus$reset_memory_zone* Routine

Memory initially allocated to a memory zone and returned to the memory manager through calls to *arus$free_memory* is available for reuse only within the zone in which it was initially allocated. If an application is through with a memory zone and wishes to make the memory contained within it available for use in other zones, then the application must either reset the zone or delete the zone. Resetting a memory zone is discussed in this section. Deleting a zone is discussed in the next section.

An application calls routine *arus$reset_memory_zone* to reset a zone. This frees all the memory contained in the zone for reuse, but retains the zone for future use. The effect of calling this routine with a memory zone argument is the same as calling *arus$free_memory* for all memory allocated from the zone with the exception that the memory is no longer reserved for use by this zone, but is available for use by any zone.

```
PROCEDURE arus$reset_memory_zone (
              IN memory_zone : arus$memory_zone;
              ) RETURNS arus$status
                LINKAGE
                    REFERENCE (
                        memory_zone
                        );
```

Parameters:

memory_zone The identifier of the memory zone to be reset.

Routine *arus$reset_memory_zone* returns the unsuccessful status values listed in Table 1–8.

Table 1–8: Status Values Returned from Routine *arus$reset_memory_zone*

Status Value	Description
arus$_nonexistent_zone	This indicates that the zone specified either was never created, or existed at one time and has been deleted.

Routine *arus$reset_memory_zone* raises no conditions.

1.3.2.6 Memory Zone Deletion

There is one routine used to delete a memory zone, *arus$delete_memory_zone*.

1.3.2.6.1 The *arus$delete_memory_zone* Routine

An application calls routine *arus$delete_memory_zone* to delete a zone. This frees all the memory contained in the zone for reuse and destroys the zone control structures. The effect of calling this routine with a memory zone argument is the same as calling *arus$reset_memory_zone*, with the exception that the zone is not available for future use by the application.

```
PROCEDURE arus$delete_memory_zone (
              IN OUT memory_zone : arus$memory_zone;
              ) RETURNS arus$status
                LINKAGE
                    REFERENCE (
                        memory_zone
                        );
```

Parameters:

memory_zone The identifier of the memory zone to be deleted. As an aid to debugging common problems, this parameter is set to NIL on completion.

Routine *arus$delete_memory_zone* returns the unsuccessful status values listed in Table 1–9.

Table 1–9: Status Values Returned from Routine *arus$delete_memory_zone*

Status Value	Description
arus$_nonexistent_zone	This indicates that the zone specified either was never created, or existed at one time and has been deleted.
arus$_default_zone	This indicates that the specified zone is the default zone, which may not be deleted.

Routine *arus$delete_memory_zone* raises no conditions.

1.3.3 Debugging Aids

One of the most difficult debugging tasks is finding improper references to heap storage. The most common examples of improper references are not initializing newly allocated memory and referencing memory after it has been deallocated. To aid in debugging such problems, the memory allocation and deallocation routines incorporate a pool poisoner which can be enabled or disabled without recoding or relinking the application that is being debugged.

Each implementation of routines *arus$get_memory* and *arus$free_memory* provides a way to detect at run time that poisoning is desired on allocation or deallocation or both, and what the desired patterns are. This preference is registered once per application invocation, and results in negligible overhead (one test and branch is the target) at the time of allocation or deallocation unless poisoning is requested. If requested, poisoning applies to all memory zones. If the arus$c_zero_on_allocate) option was requested for a zone, then that choice overrides poisoning on allocation. This is because poisoning on allocation is used to detect cells that were not initialized, and setting cells to zero provides that initialization.

If poisoning is requested, then the specified pattern is written to the memory being allocated or deallocated, repeating the pattern to fill the entire block.

One possible implementation of this feature on a VAX/VMS or Mica system is through the use of logical names. For example,

```
$ DEFINE arus$allocate_fill "%X5A5A5A5A"
$ DEFINE arus$deallocate_fill "%XA5A5A5A5"
```

1.3.4 VMS Compatibility

To aid in the porting of VMS applications to new systems, but especially to VAX/ULTRIX, PRISM ULTRIX and Mica, the following routines will be provided as jackets or aliases to routines described above:

LIB$CREATE_VM_ZONE	LIB$RESET_VM_ZONE	LIB$DELETE_VM_ZONE
LIB$GET_VM	LIB$FREE_VM	

These entry points are provided only to increase the number of applications that will run without modification. It is undetermined whether these routines will be undocumented, or whether they will be documented as compatibility routines which are not to be used for new program development.

1.4 International Date and Time Routines

The software described in this section provides the capability to obtain, format, and manipulate absolute and relative times.

The routines use the corporate standard binary format for times, which is in turn based on ISO standards. For further information about this time format, refer to DIGITAL standard EL-EN112-00, "Representation of Time for Information Exchange."

The routines that allow formatting of dates and times into text, and conversion of text into internal time stamps have inherent support for internationalization. They allow each user to declare his or her desired text format for dates and times, and if there are alphabetic parts in that format (for instance, the name of the month), those parts are output in the user's native language.

1.4.1 Treatment of Time Zones

The binary format used for absolute times contains the time value in Universal Coordinated Time (UTC), along with a separate field containing the time zone of the node, expressed as an offset from UTC in minutes of time.

All routines that convert to or from the internal absolute time format allow the application writer to choose the interpretation of time zone information. The possible interpretations follow:

- Ignore the time zone information altogether, expressing the time in UTC.
- Use the time zone information contained in the binary time stamp, expressing the time in the time zone of the node.
- Substitute time zone information representing the user's time zone which may or may not be the same as that of the node.

It must be noted (and must be documented in user manuals) that when an application writer uses the user's time zone information, the security of the time information is lost; because a nonprivileged user can set the definition of his time zone, he or she can also (intentionally or accidentally) set it incorrectly.

The means by which the node's time zone information is set and obtained is implementation specific. However, every implementation of these routines must ensure that a nonprivileged user cannot tamper with the node's time zone information.

1.4.2 Functional Interface and Description

This section describes the user-visible interface to the date/time-manipulation routines.

1.4.2.1 Types Used

The following types are used in the interface to the date/time routines.

```
TYPE
    !
    ! The following types and structures are defined by the corporate
    ! time representation standard.
    !
    arus$timevalue : large_integer SIZE (QUADWORD);

    arus$inaccuracy : large_integer[0..2**48-1] SIZE (BYTE,6);

    arus$time_diff_factor : integer[-720..780] SIZE (BIT,12);

    arus$version : integer SIZE (BIT,4);
```

```
arus$binary_absolute_time :
    RECORD
        time  : arus$timevalue;
        inacc : arus$inaccuracy;
        tdf   : arus$time_diff_factor;
        vers  : arus$version = 1;
        LAYOUT
            time;
            inacc;
            tdf;
            vers;       ! must be 1
        END LAYOUT
    END RECORD;

arus$binary_relative_time :
    RECORD
        time     : arus$timevalue;
        inacc    : arus$inaccuracy;
        reserved : arus$time_diff_factor = 0;
        vers     : arus$version = 1;
        LAYOUT
            time;
            inacc;
            reserved;     ! must be 0
            vers;         ! must be 1
        END LAYOUT
    END RECORD;
!
! End of types and structures defined by the corporate time
! representation standard.
!
arus$status : STATUS;

!
! The following type is the public view of the context block
! used by the date and time formatting routines. It is typecast
! to the appropriate type pointer by the routines. The actual
! format of the context block is never made public.
!
arus$dt_context : POINTER anytype;

arus$numeric_absolute_time :
    RECORD
        year,
        month,
        day,
        hour,
        minute,
        tdf      : integer;
        seconds,
        inacc    : real;

 timezone : integer;

        LAYOUT
            year;
            month;
            day;
            hour;
            minute;
            seconds;
            inacc;
            tdf;

            timezone;
```

```
        END LAYOUT
    END RECORD;

arus$numeric_relative_time :
    RECORD
        day,
        hour,
        minute  : integer;
        seconds,
        inacc   : real;
        LAYOUT
            day;
            hour;
            minute;
            seconds;
            inacc;
        END LAYOUT
    END RECORD;
arus$dt_format_type : (
      arus$c_date_fields,
      arus$c_time_fields
);

arus$dt_format : SET [arus$dt_format_type];

arus$timezone_options : (
arus$dt_utc,
arus$dt_nodes_timezone,
arus$dt_users_timezone
);

arus$dt_component : (
      arus$c_month_name,
      arus$c_month_name_abb,
      arus$c_format_mnemonics,
      arus$c_weekday_name,
      arus$c_weekday_name_abb,
      arus$c_relative_day_name,
      arus$c_meridiem_indicator,
      arus$c_output_format,
      arus$c_input_format
);

arus$dt_default_field_type : (
      arus$c_year,
      arus$c_month,
      arus$c_day,
      arus$c_hour,
      arus$c_minute,
      arus$c_second,
      arus$c_inacc,
      arus$c_tdf
);
arus$dt_default_field : SET [arus$dt_default_field_type];

arus$dt_truncation : (

      arus$c_truncate_hour,
      arus$c_truncate_minute,
      arus$c_truncate_second,
      arus$c_truncate_frac_second
);
```

```
arus$dt_compare_type : (
      arus$c_less,
      arus$c_equal,
      arus$c_greater
);

arus$dt_relative_operation : (
      arus$c_relative_weeks,
      arus$c_relative_days,
      arus$c_relative_hours,
      arus$c_relative_minutes,
      arus$c_relative_seconds
);

arus$dt_absolute_operation : (
      arus$c_month_of_year,
      arus$c_day_of_year,
      arus$c_hour_of_year,
      arus$c_minute_of_year,
      arus$c_second_of_year,
      arus$c_day_of_month,
      arus$c_hour_of_month,
      arus$c_minute_of_month,
      arus$c_second_of_month,
      arus$c_day_of_week,
      arus$c_hour_of_week,
      arus$c_minute_of_week,
      arus$c_second_of_week,
      arus$c_hour_of_day,
      arus$c_minute_of_day,
      arus$c_second_of_day,
      arus$c_minute_of_hour,
      arus$c_second_of_hour,
      arus$c_second_of_minute,
      arus$c_julian_date
);
```

1.4.2.2 Obtaining the System Time

There is one routine used to obtain the current system time, *arus$get_system_time*.

1.4.2.2.1 The *arus$get_system_time* Routine

The *arus$get_system_time* routine is used to obtain the current date and time in binary format.

```
PROCEDURE arus$get_system_time (
              OUT system_time : arus$binary_absolute_time;
              ) RETURNS arus$status
                LINKAGE
                    REFERENCE (
                        system_time
                        );
```

Parameters:

system_time Receives the current system date and time.

Routine *arus$get_system_time* returns no unsuccessful status values.

Routine *arus$get_system_time* raises no conditions.

1.4.2.3 Arithmetic Operations on Time Stamps

The binary representation of a date/time has several fields that must be manipulated when performing arithmetic on it.

There are seven routines that perform arithmetic operations on time stamps. They are *arus$subtract_absolute_times*, *arus$subtract_relative_times*, *arus$subtract_mixed_times*, *arus$add_relative_times*, *arus$add_mixed_times*, *arus$multiply_relative_time* and *arus$multiplyf_relative_time*.

1.4.2.3.1 The *arus$subtract_absolute_times* Routine

Routine *arus$subtract_absolute_times* allows the application to compute the interval between two absolute times.

```
PROCEDURE arus$subtract_absolute_times (
            IN time1 : arus$binary_absolute_time;
            IN time2 : arus$binary_absolute_time;
            OUT resultant_time : arus$binary_relative_time;
            ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      time1,
                      time2,
                      resultant_time
                      );
```

Parameters:

time1	Absolute time, from which the second time is subtracted.
time2	Absolute time, which is subtracted from the first time.
resultant_time	The result, a relative time, of subtracting *time2* from *time1*.

Routine *arus$subtract_absolute_times* returns the unsuccessful status values listed in Table 1–10.

Table 1–10: Status Values Returned from Routine *arus$subtract_absolute_times*

Status Value	Description
arus$_invalid_time_computed	Invalid time computed.
arus$_invalid_time_argument	Invalid time passed to the routine.

Routine *arus$subtract_absolute_times* raises no conditions.

1.4.2.3.2 The *arus$subtract_relative_times* Routine

Routine *arus$subtract_relative_times* allows the application to compute the difference of two time intervals.

```
PROCEDURE arus$subtract_relative_times (
            IN time1 : arus$binary_relative_time;
            IN time2 : arus$binary_relative_time;
            OUT resultant_time : arus$binary_relative_time;
            ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      time1,
                      time2,
                      resultant_time
                      );
```

Parameters:

time1	Relative time, from which the second time is subtracted.
time2	Relative time, which is subtracted from the first time.
resultant_time	The result, a relative time, of subtracting *time2* from *time1*.

Routine *arus$subtract_relative_times* returns the unsuccessful status values listed in Table 1–11.

Table 1–11: Status Values Returned from Routine *arus$subtract_relative_times*

Status Value	Description
arus$_invalid_time_computed	Invalid time computed.
arus$_invalid_time_argument	Invalid time passed to the routine.

Routine *arus$subtract_relative_times* raises no conditions.

1.4.2.3.3 The *arus$subtract_mixed_times* Routine

Routine *arus$subtract_mixed_times* allows an application to compute the absolute time that is separated from another absolute time by a given interval. The interval is subtracted from the given absolute time to compute the new absolute time. For instance, this routine might be used to compute the time 30 minutes before midnight.

\It is tempting to say that the computed time is "before" the given time, but that is not accurate, because the binary format used allows negative relative times. Therefore, the computed time may end up being after the starting time.\

```
PROCEDURE arus$subtract_mixed_times (
            IN time1 : arus$binary_absolute_time;
            IN time2 : arus$binary_relative_time;
            OUT resultant_time : arus$binary_absolute_time;
            ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      time1,
                      time2,
                      resultant_time
                      );
```

Parameters:

time1	Absolute time, from which the second time is subtracted.
time2	Relative time, which is subtracted from the first time.
resultant_time	The result, an absolute time, of subtracting *time2* from *time1*.

Routine *arus$subtract_mixed_times* returns the unsuccessful status values listed in Table 1–12.

Table 1–12: Status Values Returned from Routine *arus$subtract_mixed_times*

Status Value	Description
arus$_invalid_time_computed	Invalid time computed.
arus$_invalid_time_argument	Invalid time passed to the routine.

Routine *arus$subtract_mixed_times* raises no conditions.

1.4.2.3.4 The *arus$add_relative_times* Routine

Routine *arus$add_relative_times* allows the application to add two time intervals together.

```
PROCEDURE arus$add_relative_times (
            IN time1 : arus$binary_relative_time;
            IN time2 : arus$binary_relative_time;
            OUT resultant_time : arus$binary_relative_time;
            ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      time1,
                      time2,
                      resultant_time
                      );
```

Parameters:

time1	The first, relative, time which is added to the second time.
time2	The second, relative, time which is added to the first time.
resultant_time	The result, a relative time, of adding *time1* to *time2*.

Routine *arus$add_relative_times* returns the unsuccessful status values listed in Table 1–13.

Table 1–13: Status Values Returned from Routine *arus$add_relative_times*

Status Value	Description
arus$_invalid_time_computed	Invalid time computed.
arus$_invalid_time_argument	Invalid time passed to the routine.

Routine *arus$add_relative_times* raises no conditions.

1.4.2.3.5 The *arus$add_mixed_times* Routine

Routine *arus$add_mixed_times* allows the application to compute an absolute time that is separated from another absolute time by a given interval. The interval is added to the given absolute time to compute the resultant absolute time.

\It is tempting to say that the computed time is "after" the given time, but that is not accurate, because the binary format used allows negative relative times. Therefore, the computed time may end up being before the starting time.\

```
PROCEDURE arus$add_mixed_times (
            IN time1 : arus$binary_absolute_time;
            IN time2 : arus$binary_relative_time;
            OUT resultant_time : arus$binary_absolute_time;
            ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      time1,
                      time2,
                      resultant_time
                      );
```

Parameters:

time1	The first, absolute, time which is added to the second time.
time2	The second, relative, time which is added to the first time.
resultant_time	The result, an absolute time, of adding *time1* to *time2*.

Routine *arus$add_mixed_times* returns the unsuccessful status values listed in Table 1–14.

Table 1–14: Status Values Returned from Routine *arus$add_mixed_times*

Status Value	Description
arus$_invalid_time_computed	Invalid time computed.
arus$_invalid_time_argument	Invalid time passed to the routine.

Routine *arus$add_mixed_times* raises no conditions.

1.4.2.3.6 The *arus$multiply_relative_time* Routine

Routine *arus$multiply_relative_time* allows the application to multiply a time interval by an integer value.

```
PROCEDURE arus$multiply_relative_time (
            IN time : arus$binary_relative_time;
            IN multiplier : integer;
            OUT resultant_time : arus$binary_relative_time;
            ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      time,
                      multiplier,
                      resultant_time
                      );
```

Parameters:

time	The relative time which is to be multiplied.
multiplier	The multiplier by which the time is to be multiplied.
resultant_time	The result, a relative time, of multiplying *time* by *multiplier*.

Routine *arus$multiply_relative_time* returns the unsuccessful status values listed in Table 1–15.

Table 1–15: Status Values Returned from Routine *arus$multiply_relative_time*

Status Value	Description
arus$_invalid_time_computed	Invalid time computed.
arus$_invalid_time_argument	Invalid time passed to the routine.

Routine *arus$multiply_relative_time* raises no conditions.

1.4.2.3.7 The *arus$multiplyf_relative_time* Routine

Routine *arus$multiplyf_relative_time* allows the application to multiply a time interval by a floating point value. It is otherwise identical to routine *arus$multiply_relative_time*.

```
PROCEDURE arus$multiplyf_relative_time (
            IN time : arus$binary_relative_time;
            IN multiplier : integer;
            OUT resultant_time : arus$binary_relative_time;
            ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      time,
                      multiplier,
                      resultant_time
                      );
```

Parameters:

time	The relative time which is to be multiplied.
multiplier	The multiplier by which the time is to be multiplied.
resultant_time	The result, a relative time, of multiplying *time* by *multiplier*.

Routine *arus$multiplyf_relative_time* returns the unsuccessful status values listed in Table 1–16.

Table 1–16: Status Values Returned from Routine *arus$multiplyf_relative_time*

Status Value	Description
arus$_invalid_time_computed	Invalid time computed.
arus$_invalid_time_argument	Invalid time passed to the routine.

Routine *arus$multiplyf_relative_time* raises no conditions.

1.4.2.4 Conversion to and from Numeric Time Structures

In general, the binary representations for relative and absolute times are clumsy to work with. For instance, the basis for an absolute time is the number of tens of microseconds since an arbitrary base time. The next four routines, *arus$cvt_to_numeric_rel_time*, *arus$cvt_to_numeric_abs_time*, *arus$cvt_from_numeric_rel_time* and *arus$cvt_from_numeric_abs_time*, convert between binary times and structures containing the separated numeric values that make up the time.

These routines are similar in intent to the VAX/VMS system service SYS$NUMTIM and ULTRIX routines *localtime* and *gmtime*.

1.4.2.4.1 The *arus$cvt_to_numeric_rel_time* Routine

Routine *arus$cvt_to_numeric_rel_time* takes a binary relative time and unpacks it into the numeric fields day, hour, minute, and so on.

```
PROCEDURE arus$cvt_to_numeric_rel_time (
            IN time : arus$binary_relative_time;
            OUT numeric_time : arus$numeric_relative_time;
            ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      time,
                      numeric_time
                      );
```

Parameters:

time	The relative time to be converted.
numeric_time	The numeric time structure into which the converted relative time is written.

Routine *arus$cvt_to_numeric_rel_time* returns the unsuccessful status values listed in Table 1–17.

Table 1–17: Status Values Returned from Routine *arus$cvt_to_numeric_rel_time*

Status Value	Description
arus$_invalid_time_argument	Invalid time passed to the routine.

Routine *arus$cvt_to_numeric_rel_time* raises no conditions.

1.4.2.4.2 The *arus$cvt_to_numeric_abs_time* Routine

Routine *arus$cvt_to_numeric_abs_time* takes a binary absolute time and unpacks it into the numeric fields year, month, day, hour, minute, and so on.

```
PROCEDURE arus$cvt_to_numeric_abs_time (
            IN time : arus$binary_absolute_time OPTIONAL;
            OUT numeric_time : arus$numeric_absolute_time;

            IN timezone_options : arus$timezone_options OPTIONAL;

            ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      time,

                      numeric_time,
                      timezone_options
                      );
```

Parameters:

time	The absolute time to be converted. If not specified, the current system time is used.
numeric_time	The numeric time structure into which the converted absolute time is written.
timezone_options	Selects whether UTC, the node's time zone, or the user's time zone is desired. If omitted, defaults to the node's time zone.

Routine *arus$cvt_to_numeric_abs_time* returns the unsuccessful status values listed in Table 1–18.

Table 1–18: Status Values Returned from Routine *arus$cvt_to_numeric_abs_time*

Status Value	Description
arus$_invalid_time_argument	Invalid time passed to the routine.

Routine *arus$cvt_to_numeric_abs_time* raises no conditions.

1.4.2.4.3 The *arus$cvt_from_numeric_rel_time* Routine

Routine *arus$cvt_from_numeric_rel_time* converts a separated relative time into a binary relative time.

```
PROCEDURE arus$cvt_from_numeric_rel_time (
            IN numeric_time : arus$numeric_relative_time;
            OUT resultant_time : arus$binary_relative_time;
            ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      numeric_time,
                      resultant_time
                      );
```

Parameters:

numeric_time	The numeric time which is to be converted.
resultant_time	The resulting, relative, time.

Routine *arus$cvt_from_numeric_rel_time* returns the unsuccessful status values listed in Table 1–19.

Table 1–19: Status Values Returned from Routine *arus$cvt_from_numeric_rel_time*

Status Value	Description
arus$_invalid_time_argument	Invalid time passed to the routine.

Routine *arus$cvt_from_numeric_rel_time* raises no conditions.

1.4.2.4.4 The *arus$cvt_from_numeric_abs_time* Routine

Routine *arus$cvt_from_numeric_abs_time* converts a separated absolute time into a binary absolute time.

```
PROCEDURE arus$cvt_from_numeric_abs_time (
            IN numeric_time : arus$numeric_absolute_time;
            OUT resultant_time : arus$binary_absolute_time;

            IN timezone_options : arus$timezone_options OPTIONAL;

            ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      numeric_time,

                      resultant_time,
                      timezone_options
                      );
```

Parameters:

numeric_time The numeric time which is to be converted.

resultant_time The resulting, absolute, time.

timezone_options Selects whether UTC, the node's time zone, or the user's time zone is desired. If omitted, defaults to the node's time zone.

Routine *arus$cvt_from_numeric_abs_time* returns the unsuccessful status values listed in Table 1–20.

Table 1–20: Status Values Returned from Routine *arus$cvt_from_numeric_abs_time*

Status Value	Description
arus$_invalid_time_computed	Invalid time computed.
arus$_invalid_time_argument	Invalid time passed to the routine.

Routine *arus$cvt_from_numeric_abs_time* raises no conditions.

1.4.2.5 Time Comparison

Comparison of binary times is not simple for at least two reasons: a binary time is a structure consisting of several fields, and there is an inaccuracy associated with each time. Therefore, we provide routines for comparing binary times.

If the two times being compared have a smaller difference than the sum of their inaccuracies, it becomes impossible to tell their relative ordering. In this case, these routines return the value that indicates equality.

1.4.2.5.1 The *arus$compare_relative_times* Routine

Routine *arus$compare_relative_times* compares two relative times.

```
PROCEDURE arus$compare_relative_times (
            IN time1    : arus$binary_relative_time;
            IN time2    : arus$binary_relative_time;
            OUT relation : arus$dt_compare_type;
            ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      time1,
                      time2,
                      relation
                      );
```

Parameters:

time1 Relative time which is compared to *time2*.

time2 Relative time which is compared to *time1*.

relation Receives the relation of *time1* to *time2* in the comparison *time1* <RELATION> *time2*, where <RELATION> is equal to, less than, or greater than.

Routine *arus$compare_relative_times* returns the unsuccessful status values listed in Table 1–21.

Table 1–21: Status Values Returned from Routine *arus$compare_relative_times*

Status Value	Description
arus$_invalid_time_argument	Invalid time passed to the routine.

Routine *arus$compare_relative_times* raises no conditions.

1.4.2.5.2 The *arus$compare_absolute_times* Routine

Routine *arus$compare_absolute_times* compares two absolute times.

```
PROCEDURE arus$compare_absolute_times (
            IN time1 : arus$binary_absolute_time;
            IN time2 : arus$binary_absolute_time;
            OUT relation: arus$dt_compare_type;
            ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      time1,
                      time2,
                      relation
                      );
```

Parameters:

time1 Absolute time which is compared to *time2*.

time2 Absolute time which is compared to *time1*.

relation Receives the relation of *time1* to *time2* in the comparison *time1* <RELATION> *time2*, where <RELATION> is equal to, less than, or greater than.

Routine *arus$compare_absolute_times* returns the unsuccessful status values listed in Table 1–22.

Table 1–22: Status Values Returned from Routine *arus$compare_absolute_times*

Status Value	Description
arus$_invalid_time_argument	Invalid time passed to the routine.

Routine *arus$compare_absolute_times* raises no conditions.

1.4.2.6 Conversion to Arbitrary Units of Time

People think of time intervals in terms of weeks, days, hours, and so on. Binary format times are expressed in units of tenths of microseconds. The routines discussed in this section, *arus$cvt_to_binary_rel_time*, *arus$cvtf_to_binary_rel_time*, *arus$cvt_from_binary_rel_time*, *arus$cvtf_from_binary_rel_time* and *arus$cvt_from_binary_abs_time*, allow users to convert between binary formats and a variety of more easily understood expressions of the time value.

1.4.2.6.1 The *arus$cvt_to_binary_rel_time* Routine

Routine *arus$cvt_to_binary_rel_time* allows the easy conversion of such concepts to binary format relative times. The procedure takes an encoding of the unit used, the number of those units in the interval, and performs the conversion returning the binary relative time; for instance, it can convert the time expression "three weeks" to a binary format relative time.

```
PROCEDURE arus$cvt_to_binary_rel_time (
            IN operation : arus$dt_relative_operation;
            IN input_time : integer;
            OUT resultant_time : arus$binary_relative_time;
            ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      operation,
                      input_time,
                      resultant_time
                      );
```

Parameters:

operation	The conversion to be performed.
input_time	Time interval to be converted.
resultant_time	Receives the relative time that results from the conversion.

Routine *arus$cvt_to_binary_rel_time* returns the unsuccessful status values listed in Table 1–23.

Table 1–23: Status Values Returned from Routine *arus$cvt_to_binary_rel_time*

Status Value	Description
arus$_invalid_time_computed	Invalid time computed.
arus$_invalid_time_argument	Invalid time passed to the routine.
arus$_invalid_operation	Invalid operation specified.

Routine *arus$cvt_to_binary_rel_time* raises no conditions.

1.4.2.6.2 The *arus$cvtf_to_binary_rel_time* Routine

The *arus$cvtf_to_binary_rel_time* routine is similar to the *arus$cvt_to_binary_rel_time* routine, except that it takes a real value instead of an integer value.

```
PROCEDURE arus$cvtf_to_binary_rel_time (
            IN operation : arus$dt_relative_operation;
            IN input_time : real;
            OUT resultant_time : arus$binary_relative_time;
            ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      operation,
                      input_time,
                      resultant_time
                      );
```

Parameters:

operation	The conversion to be performed.
input_time	Time interval to be converted.
resultant_time	Receives the relative time that results from the conversion.

Routine *arus$cvtf_to_binary_rel_time* returns the unsuccessful status values listed in Table 1–24.

Table 1–24: Status Values Returned from Routine *arus$cvtf_to_binary_rel_time*

Status Value	Description
arus$_invalid_time_computed	Invalid time computed.
arus$_invalid_time_argument	Invalid time passed to the routine.
arus$_invalid_operation	Invalid operation specified.

Routine *arus$cvtf_to_binary_rel_time* raises no conditions.

1.4.2.6.3 The *arus$cvt_from_binary_rel_time* Routine

Routine *arus$cvt_from_binary_rel_time* answers the question, "How many things are in this time interval?", where the "things" are either weeks, days, hours, minutes, or seconds.

```
PROCEDURE arus$cvt_from_binary_rel_time (
            IN operation : arus$dt_relative_operation;
            IN input_time : arus$binary_relative_time;
            OUT resultant_time : integer;
            ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      operation,
                      input_time,
                      resultant_time
                      );
```

Parameters:

operation	The conversion to be performed.
input_time	The relative time to be converted.
resultant_time	Receives the time interval that results from the conversion.

Routine *arus$cvt_from_binary_rel_time* returns the unsuccessful status values listed in Table 1–25.

Table 1–25: Status Values Returned from Routine *arus$cvt_from_binary_rel_time*

Status Value	Description
arus$_invalid_time_computed	Invalid time computed.
arus$_invalid_time_argument	Invalid time passed to the routine.
arus$_invalid_operation	Invalid operation specified.

Routine *arus$cvt_from_binary_rel_time* raises no conditions.

1.4.2.6.4 The *arus$cvtf_from_binary_rel_time* Routine

Routine *arus$cvtf_from_binary_rel_time* is similar to routine *arus$cvt_from_binary_rel_time*, except that it returns a real value instead of an integer value.

```
PROCEDURE arus$cvtf_from_binary_rel_time (
            IN operation : arus$dt_relative_operation;
            IN input_time : arus$binary_relative_time;
            OUT resultant_time : real;
            ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      operation,
                      input_time,
                      resultant_time
                      );
```

Parameters:

operation	The conversion to be performed.
input_time	The relative time to be converted.
resultant_time	Receives the f-float time interval that results from the conversion.

Routine *arus$cvtf_from_binary_rel_time* returns the unsuccessful status values listed in Table 1–26.

Table 1–26: Status Values Returned from Routine *arus$cvtf_from_binary_rel_time*

Status Value	Description
arus$_invalid_time_computed	Invalid time computed.
arus$_invalid_time_argument	Invalid time passed to the routine.
arus$_invalid_operation	Invalid operation specified.

Routine *arus$cvtf_from_binary_rel_time* raises no conditions.

1.4.2.6.5 The *arus$cvt_from_binary_abs_time* Routine

Routine *arus$cvt_from_binary_abs_time* answers the question "Which *x* of the *y* is this?". For instance, it can determine which day of the year is represented by the input time, or day of the week, or hour of the year, and so on. (See the declaration of type *arus$dt_absolute_operation* for a complete enumeration of the legal operations.)

This routine is also capable of calculating the Julian date associated with a particular date.

```
PROCEDURE arus$cvt_from_binary_abs_time (
            IN operation : arus$dt_absolute_operation;
            OUT resultant_time : integer;
            IN input_time : arus$binary_absolute_time OPTIONAL;

            IN timezone_options : arus$timezone_options OPTIONAL;

            ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      operation,
                      resultant_time,

                      input_time,
                      timezone_options
                      );
```

Parameters:

operation	The conversion to be performed.
resultant_time	Receives the time interval that results from the conversion.
input_time	The absolute time to be converted. If not specified, the current system time is used.
timezone_options	Selects whether UTC, the node's time zone, or the user's time zone is desired. If omitted, defaults to the node's time zone.

Routine *arus$cvt_from_binary_abs_time* returns the unsuccessful status values listed in Table 1–27.

Table 1–27: Status Values Returned from Routine *arus$cvt_from_binary_abs_time*

Status Value	Description
arus$_invalid_time_computed	Invalid time computed.
arus$_invalid_time_argument	Invalid time passed to the routine.
arus$_invalid_operation	Invalid operation specified.

Routine *arus$cvt_from_binary_abs_time* raises no conditions.

1.4.2.7 Flexible Date and Time Formatting

This section describes a suite of routines with the ability to convert binary times to text in the user's natural language and desired format, and to convert such text into binary times. There are routines to support both absolute and relative times; they are *arus$format_date_time*, *arus$convert_date_string*, *arus$format_rel_time*, *arus$convert_rel_time_string*, *arus$get_date_format*, *arus$get_rel_time_format*, *arus$get_max_date_length*, *arus$get_max_rel_time_length*, *arus$free_date_time_context* and *arus$init_date_time_context*.

1.4.2.7.1 The *arus$format_date_time* Routine

Routine *arus$format_date_time* formats an absolute time into a date/time text string in the user's desired format. If there are any alphabetic portions to the string, those portions are spelled in the user's natural language.

While the user can normally indicate whether he or she wants just the date fields, just the time fields, or both to be included in the string, the routine has a means to override the user's choice in this area. For instance, an application designer might know that in a certain context the time fields would be meaningless. Therefore, the application designer is allowed to either suppress a field, or force one to be included even if the user did not express an interest in it.

Similarly, the amount of accuracy included in the text is normally at the discretion of the user, but there is a way for the application designer to force truncation of some of the lower-order time fields.

```
PROCEDURE arus$format_date_time (
            IN OUT context : arus$dt_context;
            OUT date_string : string (*);
            IN date_time : arus$binary_absolute_time OPTIONAL;
            IN field_option : arus$dt_format OPTIONAL;
            IN desired_truncation : arus$dt_truncation OPTIONAL;

            IN timezone_options : arus$timezone_options OPTIONAL;

            OUT date_length : integer OPTIONAL;
            ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      context,
                      date_time,
                      field_option,
                      desired_truncation,

                      timezone_options,

                      date_length
                      )
                  DESCRIPTOR (
                      date_string
                      );
```

Parameters:

context — Context variable that retains the translation context over multiple calls to the date/time formatting routines. This variable is initialized to NIL by the caller before the first call to any date/time routine. After that, it is maintained by the routines. Any date/time routine that uses context allocates a new context area if it receives a *context* parameter that is NIL.

date_string — Receives the requested date, time, or both, that has been formatted for output according to the currently selected format and language.

date_time — The absolute time to be formatted for output. If omitted, the current system date and time is used.

field_option — Allows the application designer to specify whether the date, time, or both are to be formatted for output. If omitted, it formats the fields requested by the user.

desired_truncation — Allows the application designer to truncate the output string at a selected field.

timezone_options — Selects whether UTC, the node's time zone, or the user's time zone is desired. If omitted, defaults to the node's time zone.

date_length — Number of octets of text written to the *date_string* argument.

Routine *arus$format_date_time* returns the unsuccessful status values listed in Table 1–28.

Table 1–28: Status Values Returned from Routine *arus$format_date_time*

Status Value	Description
arus$_default_format_used	Default format used; unable to determine desired format.
arus$_string_truncated	The output string designated by *date_string* was too short to contain the entire formatted date/time. The string was truncated to fit.
arus$_invalid_string_desc	Invalid string descriptor.
arus$_reentrancy	Reentrancy detected within a thread. In this case it returns with an error, because waiting would result in a deadlock.
arus$_english_used	English used by default; unable to determine user's choice of language.
arus$_unrecognized_format_code	Unrecognized format code. This normally indicates a corrupt format.
arus$_invalid_time_argument	Invalid time passed to the routine.

Routine *arus$format_date_time* raises no conditions.

1.4.2.7.2 The *arus$convert_date_string* Routine

Routine *arus$convert_date_string* can be viewed as the inverse operation of *arus$format_date_time*. This routine takes a text string in the user's natural language and format, and attempts to convert it into a binary absolute time.

The application designer is able to indicate which fields are legal for the user to omit, and what default values are to be used if the fields are omitted. The application is informed which fields were actually omitted.

```
PROCEDURE arus$convert_date_string (
            IN date_string : string (*);
            OUT date_time  : arus$binary_absolute_time;
            IN OUT context : arus$dt_context;
            IN defaultable : arus$dt_field OPTIONAL;
            IN defaults    : arus$numeric_absolute_time OPTIONAL;
            IN timezone_options : arus$timezone_options OPTIONAL;
            OUT defaulted_fields : arus$dt_field OPTIONAL;
            ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      date_time,
                      context,
                      defaultable,
                      defaults,
                      timezone_options,
                      defaulted_fields
                      )
                  DESCRIPTOR (
                      date_string
                      );
```

Parameters:

date_string	Date string that specifies the absolute time to be converted to a binary absolute time.
date_time	Receives the converted time.
context	Context variable that retains the translation context over multiple calls to the date/time formatting routines.
defaultable	Specifies which date or time fields of the *date_string* argument might be omitted so that default values apply.
defaults	Supplies the defaults to be used for omitted fields.
timezone_options	Selects whether UTC, the node's time zone, or the user's time zone is desired. If omitted, defaults to the node's time zone.
defaulted_fields	Indicates which date or time fields have been defaulted.

Routine *arus$convert_date_string* returns the unsuccessful status values listed in Table 1–29.

Table 1–29: Status Values Returned from Routine *arus$convert_date_string*

Status Value	Description
arus$_default_format_used	Default format used; unable to determine desired format.
arus$_invalid_string_desc	Invalid string descriptor.
arus$_reentrancy	Reentrancy detected within a thread. In this case it returns with an error, because waiting would result in a deadlock.
arus$_english_used	English used by default; unable to determine user's language.
arus$_unrecognized_format_code	Unrecognized format code.
arus$_invalid_time_argument	Invalid time passed to the routine.
arus$_ambiguous_dattim	Ambiguous date/time.
arus$_incomplete_dattim	Incomplete date/time; missing fields with no defaults.
arus$_illegal_format	Illegal format string; too many or not enough fields.
arus$_invalid_argument	Invalid argument; a required argument was not specified.

Routine *arus$convert_date_string* raises no conditions.

1.4.2.7.3 The *arus$get_date_format* Routine

An application using the date/time routines can select any number of input formats for the parsing of date/time strings. Therefore, when a user enters a time string that cannot be parsed by the date/time routines, it is frequently because the user did not use the format that he or she had previously specified. Due to the many variations of input formats, the application cannot output a fixed informational message to assist the user. The application must use routine *arus$get_date_format* to obtain a string representation of the user-supplied format to output to the user as a prompt or reminder.

```
PROCEDURE arus$get_date_format (
            OUT format_string : string (*);
            IN OUT context : arus$dt_context;
            ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      context
                      )
                  DESCRIPTOR (
                      format_string
                      );
```

Parameters:

format_string Receives a text string indicating the user's preferred format for date and time input.

context Context variable that retains the translation context over multiple calls to the date/time formatting routines.

Routine *arus$get_date_format* returns the unsuccessful status values listed in Table 1–30.

Table 1–30: Status Values Returned from Routine *arus$get_date_format*

Status Value	Description
arus$_default_format_used	Default format used; unable to determine desired format.
arus$_string_truncated	String truncated.
arus$_invalid_string_desc	Invalid string descriptor.
arus$_reentrancy	Reentrancy detected within a thread. In this case it returns with an error, because waiting would result in a deadlock.
arus$_english_used	English used by default; unable to determine user's language.
arus$_invalid_argument	Invalid argument; a required argument was not specified.
arus$_illegal_format	Illegal format string.
arus$_unrecognized_format_code	Unrecognized format code.

Routine *arus$get_date_format* raises no conditions.

1.4.2.7.4 The *arus$get_max_date_length* Routine

This procedure is used by applications that must know the length of the longest possible date string that could be returned by *arus$format_date_time*, for instance, when the dates are displayed in a column in a table.

```
PROCEDURE arus$get_max_date_length (
            OUT date_length : integer;
            IN OUT context  : arus$dt_context;
            IN field_option : arus$dt_format OPTIONAL;
            IN desired_truncation : arus$dt_truncation OPTIONAL;
            ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      date_length,
                      context,
                      field_option,
                      desired_truncation
                      );
```

Parameters:

date_length Receives the maximum possible length of the *date_string* argument returned to *arus$format_date_time*.

context Context variable that retains the translation context over multiple calls to the date/time formatting routines.

field_option Mask that allows the user to specify whether the date, time, or both are to be included in the calculation of the maximum date length.

desired_truncation Allows the application designer to truncate the output string at a selected field.

Routine *arus$get_max_date_length* returns the unsuccessful status values listed in Table 1–31.

Table 1–31: Status Values Returned from Routine *arus$get_max_date_length*

Status Value	Description
arus$_english_used	English used by default; unable to determine user's language.
arus$_default_format_used	Default format used; unable to determine desired format.
arus$_unrecognized_format_code	Unrecognized format code.
arus$_string_truncated	String truncated.
arus$_reentrancy	Reentrancy detected within a thread. In this case it returns with an error, because waiting would result in a deadlock.

Routine *arus$get_max_date_length* raises no conditions.

1.4.2.7.5 The *arus$format_rel_time* Routine

Routine *arus$format_rel_time* is a close parallel of routine *arus$format_date_time*, except that it is used for relative times.

```
PROCEDURE arus$format_rel_time (
            IN time : arus$binary_relative_time;
            OUT time_string : string (*);
            IN OUT context : arus$dt_context;
            IN desired_truncation : arus$dt_truncation OPTIONAL;
            OUT time_length : integer OPTIONAL;
            ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      time,
                      context,
                      desired_truncation,
                      time_length
                      )
                  DESCRIPTOR (
                      time_string
                      );
```

Parameters:

time	The relative time to be formatted for output.
time_string	Receives the requested relative time that has been formatted for output according to the currently selected format and language.
context	Context variable that retains the translation context over multiple calls to the date/time formatting routines.
desired_truncation	Allows the application writer to truncate the output string at a selected field.
time_length	Number of octets of text written to the *time_string* argument.

Routine *arus$format_rel_time* returns the unsuccessful status values listed in Table 1–32.

Table 1–32: Status Values Returned from Routine *arus$format_rel_time*

Status Value	Description
arus$_default_format_used	Default format used; unable to determine desired format.
arus$_string_truncated	String truncated.
arus$_invalid_string_desc	Invalid string descriptor.

Table 1–32 (Cont.): Status Values Returned from Routine *arus$format_rel_time*

Status Value	Description
arus$_reentrancy	Reentrancy detected within a thread. In this case it returns with an error, because waiting would result in a deadlock.
arus$_unrecognized_format_code	Unrecognized format code.
arus$_invalid_time_argument	Invalid time passed to the routine.

Routine *arus$format_rel_time* raises no conditions.

1.4.2.7.6 The *arus$convert_rel_time_string* Routine

The *arus$convert_rel_time_string* routine is a close parallel of routine *arus$convert_date_string*, except that it is used for relative times.

```
PROCEDURE arus$convert_rel_time_string (
            IN time_string : string (*);
            OUT time : arus$binary_relative_time;
            IN OUT context : arus$dt_context;
            ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      time,
                      context
                      )
                  DESCRIPTOR (
                      time_string
                      );
```

Parameters:

time_string	Date string that specifies the relative time to be converted.
time	Receives the converted time.
context	Context variable that retains the translation context over multiple calls to the date/time formatting routines.

Routine *arus$convert_rel_time_string* returns the unsuccessful status values listed in Table 1–33.

Table 1–33: Status Values Returned from Routine *arus$convert_rel_time_string*

Status Value	Description
arus$_default_format_used	Default format used; unable to determine desired format.
arus$_invalid_string_desc	Invalid string descriptor.
arus$_reentrancy	Reentrancy detected within a thread. In this case it returns with an error, because waiting would result in a deadlock.
arus$_unrecognized_format_code	Unrecognized format code.
arus$_invalid_time_argument	Invalid time passed to the routine.
arus$_ambiguous_dattim	Ambiguous date/time.
arus$_illegal_format	Illegal format string; too many or not enough fields.
arus$_invalid_argument	Invalid argument; a required argument was not specified.

Routine *arus$convert_rel_time_string* raises no conditions.

1.4.2.7.7 The *arus$get_rel_time_format* Routine

An application using the date/time routines can select any number of input formats for the parsing of relative time strings. Therefore, when a user enters a time string that cannot be parsed by routine *arus$convert_rel_time_string*, it is frequently because the user did not use the format that he or she had previously specified. Due to the many variations of input formats, the application cannot output a fixed informational message to assist the user. The application must use routine *arus$get_rel_time_format* to obtain a string representation of the user-supplied format to output to the user as a prompt or reminder.

This routine is very similar to *arus$get_date_format*, except that it is used to obtain the user's relative time input format, rather than the user's (absolute) date and time format.

```
PROCEDURE arus$get_rel_time_format (
            OUT format_string : string (*);
            IN OUT context : arus$dt_context;
            ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      context
                      )
                  DESCRIPTOR (
                      format_string
                      );
```

Parameters:

format_string Receives a text string indicating the user's preferred format for inputting dates and times.

context Context variable that retains the translation context over multiple calls to the date/time formatting routines.

Routine *arus$get_rel_time_format* returns the unsuccessful status values listed in Table 1–34.

Table 1–34: Status Values Returned from Routine *arus$get_rel_time_format*

Status Value	Description
arus$_default_format_used	Default format used; unable to determine desired format.
arus$_string_truncated	String truncated.
arus$_invalid_string_desc	Invalid string descriptor.
arus$_reentrancy	Reentrancy detected within a thread. In this case it returns with an error, because waiting would result in a deadlock.
arus$_english_used	English used by default; unable to determine user's language.
arus$_invalid_argument	Invalid argument; a required argument was not specified.
arus$_illegal_format	Illegal format string.
arus$_unrecognized_format_code	Unrecognized format code.

Routine *arus$get_rel_time_format* raises no conditions.

1.4.2.7.8 The *arus$get_max_rel_time_length* Routine

This procedure is used by applications that must know the length of the longest possible relative time string that could be returned by *arus$format_rel_time*, for instance, when the times are displayed in a column in a table.

This routine is very similar to *arus$get_max_date_length*, except that it is used when formatting relative time output, rather than absolute dates and times.

```
PROCEDURE arus$get_max_rel_time_length (
            OUT date_length : integer;
            IN OUT context  : arus$dt_context;
            IN field_option : arus$dt_format OPTIONAL;
            IN desired_truncation : arus$dt_truncation OPTIONAL;
            ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      date_length,
                      context,
                      field_option,
                      desired_truncation
                      );
```

Parameters:

date_length — Receives the maximum possible length of the *date_string* argument returned to *arus$format_date_time*.

context — Context variable that retains the translation context over multiple calls to the date/time formatting routines.

field_option — Mask that allows the user to specify whether the date, time, or both are to be included in the calculation of the maximum date length.

desired_truncation — Allows the application designer to truncate the output string at a selected field.

Routine *arus$get_max_rel_time_length* returns the unsuccessful status values listed in Table 1–35.

Table 1–35: Status Values Returned from Routine *arus$get_max_rel_time_length*

Status Value	Description
arus$_english_used	English used by default; unable to determine user's language.
arus$_default_format_used	Default format used; unable to determine desired format.
arus$_unrecognized_format_code	Unrecognized format code.
arus$_string_truncated	String truncated.
arus$_reentrancy	Reentrancy detected within a thread. In this case it returns with an error, because waiting would result in a deadlock.

Routine *arus$get_max_rel_time_length* raises no conditions.

1.4.2.7.9 The *arus$free_date_time_context* Routine

All of the routines associated with the formatting of dates and times use a context area to speed execution. If an application desires, it can free the memory associated with this context after it is through formatting dates and times. It uses routine *arus$free_date_time_context* to do so.

```
PROCEDURE arus$free_date_time_context (
            IN OUT context : arus$dt_context;
            ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      context
                      );
```

Parameters:

context	Context variable that retained the translation context over multiple calls to the date/time formatting routines.

Routine *arus$free_date_time_context* returns the unsuccessful status values listed in Table 1–36.

Table 1–36: Status Values Returned from Routine *arus$free_date_time_context*

Status Value	Description
arus$_reentrancy	Reentrancy detected within a thread. In this case it returns with an error, because waiting would result in a deadlock.

Routine *arus$free_date_time_context* raises no conditions.

1.4.2.7.10 The *arus$init_date_time_context* Routine

The normal use of the date/time formatting routines described above is to format dates and times for eventual presentation to people. However, the routines are a very powerful set of generalized formatting routines, and can be equally well-used for formatting dates and times for input to other computer applications where a binary time is not appropriate for some reason. In this case, the application designer needs to hard code the desired formats, rather than let the user select them at run time.

Routine *arus$init_date_time_context* allows the application designer to preinitialize the context area, thus causing the formats to be taken from the code, and not from the user.

```
PROCEDURE arus$init_date_time_context (
            IN component : arus$dt_component;
            IN init_string : string (*);
            IN OUT context : arus$dt_context;
            ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      component,
                      context
                      )
                  DESCRIPTOR (
                      init_string
                      );
```

Parameters:

component	The component of the context that is being initialized.
init_string	The characters which are to be used in formatting dates and times for input or output.
context	Context variable that retains the translation context over multiple calls to the date/time formatting routines.

Routine *arus$init_date_time_context* returns the unsuccessful status values listed in Table 1–37.

Table 1–37: Status Values Returned from Routine *arus$init_date_time_context*

Status Value	Description
arus$_numelements	Incorrect number of elements for the component.
arus$_illegal_init_string	Illegally formed *init_string*.
arus$_unrecognized_format_code	Unrecognized format code.

Table 1–37 (Cont.): Status Values Returned from Routine *arus$init_date_time_context*

Status Value	Description
arus$_illegal_component	Illegal value for the component.
arus$_invalid_string_desc	Invalid string descriptor.
arus$_reentrancy	Reentrancy detected within a thread. In this case it returns with an error, because waiting would result in a deadlock.

Routine *arus$init_date_time_context* raises no conditions.

1.4.3 VMS Compatibility

To aid in the porting of VMS applications to new systems, but especially to VAX/ULTRIX, PRISM ULTRIX and to Mica, the following routines are provided as jackets or aliases to routines described above:

LIB$ADD_TIMES
LIB$CONVERT_DATE_STRING
LIB$CVT_FROM_INTERNAL_TIME
LIB$CVTF_FROM_INTERNAL_TIME
LIB$CVT_TO_INTERNAL_TIME
LIB$CVT_VECTIM
LIB$DATE_TIME
LIB$DAY
LIB$DAY_OF_WEEK
LIB$FORMAT_DATE_TIME
LIB$FREE_DATE_TIME_CONTEXT
LIB$GET_DATE_FORMAT
LIB$GET_MAXIMUM_DATE_LENGTH
LIB$GET_USERS_LANGUAGE
LIB$INIT_DATE_TIME_CONTEXT
LIB$SUB_TIMES

These entry points are provided only to increase the number of applications that will run without modification. It is undetermined whether these routines will be undocumented, or will be documented as compatibility routines which are not to be used for new program development.

1.5 General Internationalization Aids

The software described in this section allows an application to more easily be international, or at least, internationalizable. This section includes those routines that do not fit tidily into another major section. For instance, the date and time formatting routines are not described here, but rather with the other date and time routines in Section 1.4.

1.5.1 Functional Interface and Description

The following sections describe the individual internationalization routines and their programming interfaces.

1.5.1.1 Determining the User's Natural Language

In the entire run-time system, there is exactly one routine which other routines should use to determine the user's natural language, *arus$get_language*.

1.5.1.1.1 The *arus$get_language* Routine

Routine *arus$get_language* is the routine that allows language-sensitive routines to determine the user's choice of natural language. This routine returns a text string that is the English spelling of the user's natural language.

The reasons for returning a string, rather than an enumeration or other encoded value, is not readily apparent. There are two reasons:

- This strategy most easily allows extensions in the field without having to worry about conflicts. Any other system requires a registry; although possible, this is always hard to coordinate once outside our doors.
- Much of the rest of the system support for multiple natural languages requires strings anyway. The most obvious example of this is in constructing file names and directory specifications.

```
PROCEDURE arus$get_language (
            OUT language : string (*);
            ) RETURNS arus$status
              LINKAGE
                  DESCRIPTOR (
                      language
                      );
```

Parameters:

language Receives the name of the user's language.

Routine *arus$get_language* returns the unsuccessful status values listed in Table 1–38.

Table 1–38: Status Values Returned from Routine *arus$get_language*

Status Value	Description
arus$_english_used	English used by default; unable to determine user's language.

Routine *arus$get_language* raises no conditions.

1.6 Condition Handling Routines

Whenever possible, portable applications should use only those condition handling capabilities inherent in the implementation language being used. There are languages, however, that do not have condition handling capabilities, or have capabilities that are too limited for the problem at hand.

Condition handling is, by its nature, very system specific; there is, however, a common subset of capabilities that are found in all DIGITAL operating systems.

The routines described in this section of the chapter allow portable applications which use AIA interfaces to initiate a condition and perform basic condition handling.

1.6.1 The ARUS Condition Handling Model

The ARUS model of condition handling is based on routine invocations. Every routine that is activated can establish a *condition handler*, which can handle any conditions caused by that routine or its descendants. In addition to these routine-invocation-based condition handlers, the ARUS routines also support *primary condition handlers* and *last-chance condition handlers*. Primary condition handlers are handlers to be invoked before the routine-invocation hierarchy of handlers is searched. Last-chance condition handlers are handlers to be invoked after all other handlers established in the routine-invocation hierarchy have been invoked.

In summary, the ARUS condition handlers are invoked in the following order:

1. Primary condition handlers are invoked in order from the first-established condition handler to the last-established condition handler.
2. After the primary condition handlers are invoked, the routine-invocation-based condition handlers are invoked in order from the most-recently-established condition handler to the least-recently-established condition handler.
3. Finally, the last-chance condition handlers are invoked, beginning with the most-recently-established condition handler.

This mode of condition handling works alongside normal ULTRIX signal handling on ULTRIX implementations. In such an implementation, ULTRIX signal handlers are invoked after the last-chance condition handlers.

Finally, an assumed part of the ARUS condition handling model is that for systems that support multithreading, condition handling occurs on a per-thread basis. However, since the ARUS routines are under the control of the operating system condition dispatching routines, there is nothing that the ARUS routines can do to enforce this assumption.

1.6.2 ARUS Condition Handlers

Condition handlers supported by the ARUS routines *arus$add_primary_handler*, *arus$delete_primary_handler*, *arus$add_last_chance_handler* and *arus$delete_last_chance_handler* are not the same as condition handlers supported by the base system on which the ARUS routines are implemented, because those condition handlers are system specific. ARUS condition handlers are procedures of type *arus$condition_handler*; they take one parameter, the primary condition name of the condition currently being handled.

ARUS condition handlers return with a value of type *arus$continue_code*. If they return *arus$c_continue* and the condition is a continuable condition, then the condition is dismissed and program execution is continued at the point immediately following that at which the condition was raised. If the condition is not continuable and the condition handler returns *arus$c_continue*, a new, noncontinuable condition is raised indicating the fact that an attempt was made to continue from a noncontinuable condition.

If the condition handler returns *arus$c_reraise*, the next condition handler is invoked.

REVIEWERS

\I am currently defining an ARUS condition handling routine which has a single parameter: the primary condition name (type *arus$status*) of the condition it is handling. This seems fairly limiting, but portable.

By defining a handler with zero parameters, we would unify handlers maintained by the ARUS primary and last-chance handler managers, and those established by the language RTLs. Both types would assume that they had no parameters, and use routine *arus$examine_condition* to get the information about what condition it was handling.

The other possibility I see is simply stating that the procedure type of a condition handler is implementation defined; condition handlers are not transportable. I am against this model, but am open to other ideas and suggestions. It seems illogical to provide a portable way of referencing nonportable code. Comments are invited.

1.6.3 Functional Interface and Description

The following sections describe the interface to the ARUS condition routines.

1.6.3.1 Types Used

There are only a few structures used by the condition handling routines, but they bear more than a cursory explanation. In general, since the underlying operating system condition raising and handling code is fundamentally incompatible, the user interface structures must contain the union of all the information the different systems require for dealing with conditions. To simplify the structure, we also leave out all those fields that are not set by the user.

The structure *arus$condition_record* fully describes a condition, or series of related conditions. It is similar in function to a *signal vector* on VAX/VMS.

NOTE

The logical contents of each of these structures are required to be the same for all implementations of these routines. However, the actual physical materialization of each of the structures listed below are for the Mica implementation, and designed to give a feel for the structures.

\The condition record described below is the main interface structure for these routines, and is going to give some languages and unsophisticated users grief. For the class of users that are likely to be using condition handling outside the scope of their language, will that be a problem? Will unsophisticated users be using these routines? How much pain is acceptable?

An alternate interface would not give the user the capability of raising multiple conditions simultaneously; they would be limited to one. The interface would involve an atomic condition name, and three lists of parameters, one for the pointer to each argument, one for the length of each argument, and one for the datatype of each argument. Not as nicely encapsulated, but perhaps easier for average users. Comments?\

```
TYPE
    arus$status : status;

    !
    ! The possible values for field argument_datatype are to be
    ! determined later, in conjunction with the PRISM calling standard.
    !
    arus$condition_argument : RECORD
        argument_datatype : arus$condition_arg_type;
        argument_extent : longword;
        argument : POINTER anytype;
    END RECORD;

    arus$condition_record (number_of_arguments : integer [0..]) : RECORD
        CAPTURE number_of_arguments;
        condition_value : arus$status;
        condition_list : POINTER arus$condition_record;
        condition_arguments : ARRAY [1..argument_number] arus$condition_argument;
        LAYOUT
            condition_value;
            condition_list;
            number_of_arguments;
            condition_arguments;
        END LAYOUT;
    END RECORD;
```

```
arus$continue_code : (
    arus$c_continue,
    arus$c_reraise);

arus$condition_handler : PROCEDURE (
    IN condition_name : arus$status;
    ) RETURNS arus$continue_code
      LINKAGE
          REFERENCE (
              condition_name
              );

arus$handler_id : longword;
```

1.6.3.2 Condition Raising Routines

When software detects an erroneous situation, it may choose to invoke code designed to correct the situation, report the situation, or both. It does so by raising a condition. There are two routines used to raise conditions, *arus$raise_condition* and *arus$raise_stop_condition*.

1.6.3.2.1 The *arus$raise_condition* Routine

The *arus$raise_condition* procedure is used to initiate a condition.

```
PROCEDURE arus$raise_condition (
    IN condition_record : arus$condition_record;
    ) RETURNS arus$status;
      LINKAGE
          REFERENCE (
              condition_record);
```

Parameters:

condition_record The condition record completely describing the condition to be raised.

If the condition handling routines invoked due to this condition cause execution to continue, routine *arus$raise_condition* returns the value that is left in the system-defined return register by the condition handling routines invoked. This value may be set by *arus$store_return_value*, described in Section 1.6.3.4.3. Otherwise, the call to routine *arus$raise_condition* never returns.

Routine *arus$raise_condition* raises no conditions other than the one(s) specified by parameter *condition_record*.

1.6.3.2.2 The *arus$raise_stop_condition* Routine

The *arus$raise_stop_condition* procedure is used to initiate a condition that is noncontinuable. With that one exception, it is identical to *arus$raise_condition*.

```
PROCEDURE arus$raise_stop_condition (
    IN condition_record : arus$condition_record;
    ) LINKAGE
          REFERENCE (condition_record);
```

Parameters:

condition_record The condition record completely describing the condition to be raised.

Routine *arus$raise_stop_condition* never returns; therefore, it cannot return a value.

Routine *arus$raise_stop_condition* raises no conditions other than the one(s) specified by parameter *condition_record*.

1.6.3.3 Condition Modification Routines

The condition modification routines are designed to be called from condition handler routines. They allow the handler routines to update the information contained in the original condition with additional or more accurate information.

An implementation of these routines may raise a new condition with the updated information; therefore, the original faulting address and processor state are not guaranteed to be preserved if these routines are used.

It is an error to call any of the routines in this section when not in the process of handling a condition.

1.6.3.3.1 The *arus$replace_condition* Routine

Routine *arus$replace_condition* is used to completely replace one condition with another. Because information is undoubtedly lost by doing this, the use of this routine should be carefully considered. Other routines in this section may be more appropriate.

```
PROCEDURE arus$replace_condition : (
    IN condition_record : arus$condition_record;
    ) RETURNS arus$status
      LINKAGE
          REFERENCE (
              condition_record
              );
```

Parameters:

condition_record The condition record completely describing the condition to be raised.

If routine *arus$replace_condition* executes successfully, and the condition handling routines invoked due to this condition cause execution to continue, routine *arus$replace_condition* returns the value that is left in the system-defined return register by the condition handling routines invoked. This value may be set by *arus$store_return_value*, described in Section 1.6.3.4.3. If the condition handling routines do not cause execution to continue, the call to routine *arus$replace_condition* never returns.

In addition to the return values just described, it is also possible for the routine to not execute successfully. In that case, routine *arus$replace_condition* returns the unsuccessful status values listed in Table 1–39.

Table 1–39: Status Values Returned from Routine *arus$replace_condition*

Status Value	Description
arus$_no_condition_active	The routine was called while not in the process of handling a condition.

Routine *arus$replace_condition* raises no conditions other than the one(s) specified by parameter *condition_record*.

1.6.3.3.2 The *arus$add_primary_condition* Routine

Routine *arus$add_primary_condition* is used to "add" a *primary condition* to an existing condition or conditions. The effect of this addition is that the condition that was formerly the primary condition becomes a *secondary condition*.

While principally intended to add a single primary condition, if the *condition_record* argument is actually a list of condition records, then the old condition list is appended to the new list, effectively adding a new primary and one or more superior secondary conditions.

If the former primary condition was flagged as not continuable, then the new primary condition is similarly flagged.

Routine *arus$add_primary_condition* provides outer layers of code the ability to express the condition in terms suitable to that layer, without losing any detailed information supplied by inner layers. For instance, if a Pascal *readln* call failed, the system could return an error stating that the file was not accessed, RMS could add on a condition stating that the file was not open and give the file name, and the Pascal run-time library code could add a condition stating the source line of the *readln* call.

```
PROCEDURE arus$add_primary_condition : (
    IN condition_record : arus$condition_record;
    ) RETURNS arus$status
      LINKAGE
          REFERENCE (
              condition_record
              );
```

Parameters:

condition_record The condition record completely describing the condition(s) to be added.

If routine *arus$add_primary_condition* executes successfully, and the condition handling routines invoked due to this condition cause execution to continue, routine *arus$add_primary_condition* returns the value that is left in the system-defined return register by the condition handling routines invoked. This value may be set by *arus$store_return_value*, described in Section 1.6.3.4.3. If the condition handling routines do not cause execution to continue, the call to routine *arus$add_primary_condition* never returns.

In addition to the return values just described, it is also possible for the routine to not execute successfully. In that case, routine *arus$add_primary_condition* returns the unsuccessful status values listed in Table 1–40.

Table 1–40: Status Values Returned from Routine *arus$add_primary_condition*

Status Value	Description
arus$_no_condition_active	The routine was called while not in the process of handling a condition.

Routine *arus$add_primary_condition* raises no conditions other than the one(s) specified by the new combined condition record.

1.6.3.3.3 The *arus$add_secondary_condition* Routine

Routine *arus$add_secondary_condition* is similar to *arus$add_primary_condition*, except that the newly added condition is placed at the end of the list of conditions, rather than at the head of the list. It is also used by outer layers that wish to add information to the reported condition.

While principally intended to add a single secondary condition, if the *condition_record* argument is actually a list of condition records, then the entire new list is appended to the old condition list; effectively adding two or more inferior secondary conditions.

```
PROCEDURE arus$add_secondary_condition : (
    IN condition_record : arus$condition_record;
    ) RETURNS arus$status
      LINKAGE
         REFERENCE (
             condition_record
             );
```

Parameters:

condition_record The condition record completely describing the condition(s) to be added.

If routine *arus$add_secondary_condition* executes successfully, and the condition handling routines invoked due to this condition cause execution to continue, routine *arus$add_secondary_condition* returns the value that is left in the system-defined return register by the condition handling routines invoked. This value may be set by *arus$store_return_value*, described in Section 1.6.3.4.3. If the condition handling routines do not cause execution to continue, the call to routine *arus$add_secondary_condition* never returns.

In addition to the return values just described, it is also possible for the routine to not execute successfully. In that case, routine *arus$add_secondary_condition* returns the unsuccessful status values listed in Table 1–41.

Table 1–41: Status Values Returned from Routine *arus$add_secondary_condition*

Status Value	Description
arus$_no_condition_active	The routine was called while not in the process of handling a condition.

Routine *arus$add_secondary_condition* raises no conditions other than the one(s) specified by the new combined condition record.

1.6.3.4 Condition Information Routines

Since the structures used to represent conditions vary from system to system, we provide a way to obtain and modify selected information contained within those structures. The routines used for this purpose are *arus$examine_condition*, *arus$examine_return_value* and *arus$store_return_value*.

As with the condition manipulation routines, it is an error to call these routines while not in the process of handling a condition.

1.6.3.4.1 The *arus$examine_condition* Routine

The *arus$examine_condition* routine is used to extract the condition "name" from the current primary condition record. This value, once removed from the context of a condition record, is the same as a status value.

```
PROCEDURE arus$examine_condition : (
    OUT condition_name : arus$status;
    ) RETURNS arus$status
      LINKAGE
         REFERENCE (
             condition_name
             );
```

Parameters:

condition_name The variable into which the current condition name is written.

Routine *arus$examine_condition* returns the unsuccessful status values listed in Table 1–42.

Table 1–42: Status Values Returned from Routine *arus$examine_condition*

Status Value	Description
arus$_no_condition_active	The routine was called while not in the process of handling a condition.

This routine does not raise any conditions.

1.6.3.4.2 The *arus$examine_return_value* Routine

In addition to the condition record, there is a secondary channel of information available to users of the condition handling routines. This second channel is the *return value*. In the event of a condition caused by the *arus$raise_condition* routine, the return value is used as the completion status of the call to *arus$raise_condition* if execution is continued. If a condition causes an unwind to occur, the return value is used as the completion status of the final routine being unwound. By means of this secondary channel, it is possible for a handler to notify the raising routine of whether or not the condition was handled.

When a condition is initially raised, the return value is set to the condition name contained in the condition record. Therefore a routine can detect through normal language mechanisms if the return value has been modified when it receives control back from routine *arus$raise_condition*.

This routine, however, is used to examine the return value while still in the process of handling conditions.

REVIEWERS

\The VAX/VMS and PRISM calling standards both define 64 bits of return value for routine calls, and provide 64 bits of return value in the mechanism records. I do not know if there is a system-wide default amount of return value on VAX/ULTRIX. Is there? For portability now and in the future, I am limiting the return value accessible by these routines to 32 bits, the size of return value most frequently used on VAX/VMS. Is this an acceptable restriction?

```
PROCEDURE arus$examine_return_value : (
    OUT return_value : LONGWORD;
    ) RETURNS arus$status
      LINKAGE
          REFERENCE (
              return_value
              );
```

Parameters:

return_value The variable into which the return value is written.

Routine *arus$examine_return_value* returns the unsuccessful status values listed in Table 1–43.

Table 1–43: Status Values Returned from Routine *arus$examine_return_value*

Status Value	Description
arus$_no_condition_active	The routine was called while not in the process of handling a condition.

1.6.3.4.3 The *arus$store_return_value* Routine

The *arus$store_return_value* routine is the means by which a condition handler can modify the return value described in Section 1.6.3.4.2.

```
PROCEDURE arus$store_return_value : (
    IN return_value : LONGWORD;
    ) RETURNS arus$status
      LINKAGE
          REFERENCE (
              condition_name
              );
```

Parameters:

return_value The new return value.

Routine *arus$store_return_value* returns the unsuccessful status values listed in Table 1–44.

Table 1–44: Status Values Returned from Routine *arus$store_return_value*

Status Value	Description
arus$_no_condition_active	The routine was called while not in the process of handling a condition.

1.6.3.5 Status Value Routines

Status values are opaque, with formats that vary across implementations of these interfaces. Therefore, there must be routines to compare for equality, and to test for success. We provide two routines, *arus$test_for_success* and *arus$compare_status*.

1.6.3.5.1 The *arus$test_for_success* Routine

Routine *arus$test_for_success* allows an application to determine in a portable way whether or not a status value returned by a routine was successful or not.

REVIEWERS

\How committed are we to "low bit set equals success"? This is not the case on ULTRIX. I am trying to balance two differing goals, and would like input on where to strike the balance. It would be nice to have AIA fit in nicely with the underlying base system. Conditions could have differing severity field definitions on the different systems: low bit set on Mica and VAX/VMS, low bit clear for success on ULTRIX. This routine would be able to interpret the different schemes. However, the other side of the coin is that there is *a lot* of code that we will attempt to port that simply does an in-line low bit test. That code would need to be rewritten.

I am leaning toward having low bit set be success for all routines written at or above the AIA level. If that happens, this routine is probably not needed.

Comments?

```
PROCEDURE arus$test_for_success (
    IN status_value : arus$status;
    ) RETURNS boolean
      LINKAGE
          REFERENCE (
              status_value
              );
```

Parameters:

status_value The status value that is to be tested for success.

Routine *arus$test_for_success* returns no unsuccessful status values.

Routine *arus$test_for_success* raises no conditions.

1.6.3.5.2 The *arus$compare_status* Routine

Routine *arus$compare_status* is used to compare a condition for equality against an array of known conditions. If a match is found, the index into the array of the matching condition is returned in the OUT parameter *match_index*. If no match is found, a zero is returned in the OUT parameter, and the routine returns an unsuccessful status.

```
PROCEDURE arus$compare_status (
    IN status_to_match : arus$status;
    IN status_array : ARRAY [1..] arus$status;
    IN number_of_statuses : integer;
    OUT match_index : integer;
    ) RETURNS arus$status
      LINKAGE
          REFERENCE (
              status_to_match,
              status_array,
              number_of_statuses,
              match_index
              );
```

Parameters:

status_to_match	The unknown status, which is to be compared to the known statuses in the array.
status_array	The array of known statuses.
number_of_statuses	The count of known statuses contained in *status_array*.
match_index	If one of the known statuses in *status_array* matched the unknown status in *status_to_match*, then this contains the index of the matching status. If no match was found, this contains the value zero.

Routine *arus$compare_status* returns the unsuccessful status values listed in Table 1–45.

Table 1–45: Status Values Returned from Routine *arus$compare_status*

Status Value	Description
arus$_no_match	No match was found for the unknown status.

Routine *arus$compare_status* raises no conditions.

1.6.3.6 Unwind Routines

During the course of handling a condition, it is sometimes necessary to abort the current action, and return to a previous known state in the thread or to abort the thread. The means to do this are two unwind routines, *arus$unwind* and *arus$unwind_to_exit*.

1.6.3.6.1 The *arus$unwind* Routine

The *arus$unwind* routine is used to abort the processing that was in progress when the condition was raised, and return to a known point in the program. That known point is the instruction after the call to the routine that established the handler currently handling the condition. This, in effect, makes it appear as if that routine call had just completed.

During the process of unwinding, each routine with an associated condition handler is unwound only after its condition handler is invoked. This allows any necessary cleanup activities to occur.

The return value available after completion of the unwind can be set by the call to *arus$unwind*, and by any handler invoked during the course of the unwind operation through use of the *arus$store_return_value* routine.

It is an error to call routine *arus$unwind* while not in the process of handling a condition.

```
PROCEDURE arus$unwind (
    IN condition_record : arus$condition_record OPTIONAL;
    ) RETURNS arus$status
      LINKAGE
          REFERENCE (
              condition_record
              );
```

Parameters:

condition_record — The condition record completely describing the condition to be passed to each handler. If omitted, the current condition is used.

If it is called while processing a condition, routine *arus$unwind* never returns. Otherwise, it returns the unsuccessful status values listed in Table 1–46.

Table 1–46: Status Values Returned from Routine *arus$unwind*

Status Value	Description
arus$_no_condition_active	The routine was called while not in the process of handling a condition.

1.6.3.6.2 The *arus$unwind_to_exit* Routine

The other unwind operation supported by ARUS routines is a total unwind, resulting in a *modular thread termination*. It is termed modular because each routine gets a chance to perform any cleanup activities necessary, from most-recently-invoked routine to least-recently-invoked routine.

```
PROCEDURE arus$unwind_to_exit (
    IN condition_record : arus$condition_record OPTIONAL;
    ) RETURNS arus$status
      LINKAGE
          REFERENCE (
              condition_record
              );
```

Parameters:

condition_record — The condition record completely describing the condition to be passed to the handlers. If omitted, and *arus$unwind_to_exit* was called by a routine that is handling a condition, the current condition is used. Otherwise, a condition record containing the status value arus$_unwinding is used.

Routine *arus$unwind_to_exit* never returns.

1.6.3.7 Condition Handler Management Routines

As discussed in Section 1.6.1, there are three different types of handlers supported by the ARUS condition handling model. The ARUS routines do not support the establishment of stack-based handlers; there are no ARUS equivalents of the VAX/VMS routines LIB$ESTABLISH and LIB$REVERT, because on some architectures that operation requires the assistance of the compilers. The ARUS routines do support the establishment and removal of primary and last-chance condition handlers through routines *arus$add_primary_handler*, *arus$add_last_chance_handler*, *arus$delete_primary_handler* and *arus$delete_last_chance_handler*.

Primary and last-chance handlers are inherently nonmodular; their uses should be few and far between. Almost all problems are better solved by an appropriately placed, routine-invocation-based handler.

Systems that directly support primary and last-chance handler routines may interfere with the routines provided by ARUS if the system's capability is exploited directly. For instance, if the VAX/VMS system service that is used to establish a primary condition handler is used after ARUS routine *arus$add_primary_handler* is used, the routines declared by *arus$add_primary_handler* may no longer be known to the underlying system.

NOTE

Use these routines with care, and only when they are really required.

1.6.3.7.1 The *arus$add_primary_handler* Routine

Routine *arus$add_primary_handler* is used to add a primary handler to the end of the primary handler list. In other words, primary handlers are executed in the order in which they were established.

```
PROCEDURE arus$add_primary_handler : (
    IN condition_handler : arus$condition_handler;
    OUT handler_id : arus$handler_id OPTIONAL;
    ) RETURNS arus$status
      LINKAGE
          REFERENCE (
              condition_handler,
              handler_id
              );
```

Parameters:

condition_handler — The address of the condition handling routine.

handler_id — An identifier which can be passed to routine *arus$delete_primary_handler*, to remove this handler from the list of handlers.

Routine *arus$add_primary_handler* returns no unsuccessful status values.

Routine *arus$add_primary_handler* raises no conditions.

1.6.3.7.2 The *arus$add_last_chance_handler* Routine

Routine *arus$add_last_chance_handler* is used to add a last-chance handler to the beginning of the last-chance handler list. In other words, last-chance handlers are executed in the reverse of the order in which they were established.

```
PROCEDURE arus$add_last_chance_handler : (
    IN condition_handler : arus$condition_handler;
    OUT handler_id : arus$handler_id OPTIONAL;
    ) RETURNS arus$status
      LINKAGE
          REFERENCE (
              condition_handler,
              handler_id
              );
```

Parameters:

condition_handler — The address of the condition handling routine.

handler_id — An identifier which can be passed to routine *arus$delete_last_chance_handler*, to remove this handler from the list of handlers.

Routine *arus$add_last_chance_handler* returns no unsuccessful status values.

Routine *arus$add_last_chance_handler* raises no conditions.

1.6.3.7.3 The *arus$delete_primary_handler* Routine

Routine *arus$delete_primary_handler* is used to remove a declared primary handler from the primary handler list.

```
PROCEDURE arus$delete_primary_handler (
    IN handler_id : arus$handler_id;
    ) RETURNS arus$status
      LINKAGE
          REFERENCE (
              handler_id
              );
```

Parameters:

handler_id — The handler id value returned by routine *arus$add_primary_handler* when this handler was added to the primary handler list.

Routine *arus$delete_primary_handler* returns the unsuccessful status values listed in Table 1–47.

Table 1–47: Status Values Returned from Routine *arus$delete_primary_handler*

Status Value	Description
arus$_no_such_handler	The handler_id does not represent a handler currently on the primary handler list.

1.6.3.7.4 The *arus$delete_last_chance_handler* Routine

Routine *arus$delete_last_chance_handler* is used to remove a declared primary handler from the primary handler list.

```
PROCEDURE arus$delete_last_chance_handler (
    IN handler_id : arus$handler_id;
    ) RETURNS arus$status
      LINKAGE
          REFERENCE (
              handler_id
              );
```

Parameters:

handler_id — The handler id value returned by routine *arus$add_last_chance_handler* when this handler was added to the last-chance handler list.

Routine *arus$delete_last_chance_handler* returns the unsuccessful status values listed in Table 1–48.

Table 1–48: Status Values Returned from Routine *arus$delete_last_chance_handler*

Status Value	Description
arus$_no_such_handler	The handler_id does not represent a handler currently on the last-chance handler list.

1.7 Data Conversion Routines

ARUS provides several routines to convert between textual and binary forms of numeric data. In addition, there are a few routines that convert data between different binary formats. These routines attempt to solve the needs of two different classes of calling routines: those that require flexibility and performance, such as language support library routines, and those that require an easy-to-use interface, such as end-user, or high-level calling routines.

1.7.1 Functional Interface and Description

The following sections describe the various routines associated with data conversion.

1.7.1.1 Types Used

The following types are used in the interface to the data conversion routines.

```
TYPE

    arus$status : status;

    arus$radix : (
        arus$c_binary,
        arus$c_octal,
        arus$c_decimal,
        arus$c_hexidecimal
        );

    arus$int_text_flags : (
        arus$c_force_plus
        );

    arus$text_int_flags : (
        arus$c_skip_blanks,
        arus$c_skip_tabs
        );

    arus$text_float_flags : (
        arus$c_skip_blanks,
        arus$c_skip_tabs,
        arus$c_only_e,
        arus$c_err_underflow,
        arus$c_exp_letter_required,
        arus$c_truncate,
         arus$c_force_scale
        );

    arus$float_text_flags : (
        arus$c_force_exponential,
        arus$c_force_plus,
        arus$c_force_plus_exponent,
        arus$c_suppress_trailing_zeroes
        );

    arus$int_real_text_flags : (
        arus$c_force_exponential
        arus$c_force_plus
        arus$c_suppress_trailing_zeroes
        );

    arus$format_int_text_flags : (
        arus$c_force_plus
        );

    arus$format_text_int_flags : (
        arus$c_skip_blanks,
        arus$c_skip_tabs
        );

    arus$format_float_text_flags : (
        arus$c_force_exponential,
        arus$c_force_plus,
        arus$c_force_plus_exponent,
        arus$c_suppress_trailing_zeroes,
        arus$c_use_digit_separator
        );
```

```
arus$format_text_float_flags : (
    arus$c_skip_blanks,
    arus$c_skip_tabs,
    arus$c_only_e,
    arus$c_err_underflow,
    arus$c_exp_letter_required,
    arus$c_truncate,
    arus$c_force_scale
    );

arus$text_int_options : SET [arus$text_int_flags];

arus$int_text_options : SET [arus$int_text_flags];

arus$float_text_options : SET [arus$float_text_flags];

arus$text_float_options : SET [arus$text_float_flags];

arus$int_real_text_options : SET [arus$int_real_text_flags];

arus$intl_text_int_options : SET [arus$intl_text_int_flags];

arus$intl_int_text_options : SET [arus$intl_int_text_flags];

arus$intl_float_text_options : SET [arus$intl_float_text_flags];

arus$intl_text_float_options : SET [arus$intl_text_float_flags];

arus$context : POINTER anytype;
```

REVIEWERS

Do we want to implement Base 36 conversions?

1.7.1.2 Convert an Integer to a Text String

The integer-to-text routines convert either a signed integer to a decimal ASCII text string or an unsigned longword to an ASCII text string of a specified radix value. The valid radix values for unsigned integer to text conversions are binary, octal, decimal, and hexadecimal.

The syntax of the resultant string for the integer-to-text routines is *n*, where *n* expands as follows:

Symbol	Expansion
n	[+ | −] decimal_digit [decimal_digit...]

The sign ([+ | −]) is only applicable in the conversion of a signed integer value to a text string.

1.7.1.2.1 The *arus$cvt_longword_to_text* Routine

The routine *arus$cvt_longword_to_text* is used to convert an unsigned integer to a text representation, using the specified radix. This routine supports FORTRAN Ow, Ow.m, Zw, and Zm.w output conversion.

```
PROCEDURE arus$cvt_longword_to_text (
              IN  input_value : integer [0..];
              IN  radix : arus$radix;
              OUT resultant_string : string (*);
              IN  number_of_digits : integer OPTIONAL;
              ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      input_value,
                      radix,
                      number_of_digits
                      )
                  DESCRIPTOR (
                      resultant_string
                      );
```

Parameters:

input_value	The input value to be converted to text.
radix	The base used when converting the input value to a text representation.
resultant_string	The resulting text string.
number_of_digits	Minimum number of digits to be produced. If the actual number of significant digits is smaller, leading zeroes are produced. If *number_of_digits* is less than one, a blank field may result when the *input_value* equals zero. The default is 1.

Routine *arus$cvt_longword_to_text* returns the unsuccessful status values listed in Table 1–49.

Table 1–49: Status Values Returned from Routine *arus$cvt_longword_to_text*

Status Value	Description
arus$_invalid_radix	Invalid radix value passed.
arus$_output_conversion_error	Output conversion error. The result would have exceeded the size of the resultant string; *resultant_string* is filled with asterisks.

Routine *arus$cvt_longword_to_text* raises no conditions.

1.7.1.2.2 The *arus$cvt_integer_to_text* Routine

The routine *arus$cvt_integer_to_text* is used to convert a signed integer to a decimal text string. This routine supports FORTRAN Iw and Iw.m output conversion.

```
PROCEDURE arus$cvt_integer_to_text (
              IN  input_value : integer;
              OUT resultant_string : string (*);
              IN  number_of_digits : integer OPTIONAL;
              IN  options : arus$int_text_options OPTIONAL;
              ) RETURNS arus$status
              LINKAGE
                  REFERENCE  (
                      input_value,
                      number_of_digits,
                      options
                      )
                  DESCRIPTOR (
                      resultant_string
                      );
```

Parameters:

input_value	The input value to be converted to text.
resultant_string	The resulting decimal ASCII text string.
number_of_digits	Minimum number of digits to be produced. If the actual number of significant digits is smaller, leading zeroes are produced. If *number_of_digits* is less than one, a blank field may result when the *input_value* equals zero. The default is 1.
options	The following caller-supplied option is supported:

Option Value	Description
arus$c_force_plus	If set, a plus sign is forced for positive values.

Routine *arus$cvt_integer_to_text* returns the unsuccessful status values listed in Table 1–50.

Table 1–50: Status Values Returned from Routine *arus$cvt_integer_to_text*

Status Value	Description
arus$_output_conversion_error	Output conversion error. The result would have exceeded the size of the resultant string; *resultant_string* is filled with asterisks.

Routine *arus$cvt_integer_to_text* raises no conditions.

1.7.1.3 Convert a Text String to an Integer Value

The text-to-integer routines convert either a signed decimal text string, or an unsigned text string of a specified base, to an integer value. The unsigned text representation may be in binary, octal, decimal, or hexadecimal bases. The radix point is assumed at the right of the input string.

The syntax of an input string for the text-to-integer routines is *n* where *n* expands as follows:

Symbol	Expansion
n	[+ | −] decimal_digit [decimal_digit...]

The sign ([+ | −]) is only applicable in the conversion of a text string to a signed integer value. Blanks and tabs may appear at the beginning of the input string or they may be embedded in the string.

1.7.1.3.1 The *arus$cvt_text_to_longword* Routine

The routine *arus$cvt_text_to_longword* is used to convert an ASCII text string representation of an unsigned value to an unsigned integer value.

```
PROCEDURE arus$cvt_text_to_longword (
              IN  input_string : string (*);
              IN  radix : arus$radix;
              OUT resultant_value : integer [0..];
              IN  options : arus$text_int_options OPTIONAL;
              ) RETURNS arus$status
              LINKAGE
                  REFERENCE  (
                      radix,
                      resultant_value,
                      options
                      )
                  DESCRIPTOR (
                      input_string
                      );
```

Parameters:

input_string The input string to be converted.

radix The base in which the text string is represented.

resultant_value The resulting unsigned integer value. On error, this value is set to 0.

options The following caller-supplied options are supported:

Option Value	Description
arus$c_skip_blanks	If set, blanks are ignored. Otherwise, blanks are equivalent to zero.
arus$c_skip_tabs	If set, tabs are ignored. Otherwise, tab characters are illegal.

Routine *arus$cvt_text_to_longword* returns the unsuccessful status values listed in Table 1–51.

Table 1–51: Status Values Returned from Routine *arus$cvt_text_to_longword*

Status Value	Description
arus$_invalid_radix	Invalid radix value passed.
arus$_invalid_character	Invalid character in the input string.
arus$_overflow	Overflow detected; unable to do conversion.

Routine *arus$cvt_text_to_longword* raises no conditions.

1.7.1.3.2 The *arus$cvt_text_to_integer* Routine

The routine *arus$cvt_text_to_integer* is used to convert an ASCII text string representation of a decimal number to a signed integer value.

```
PROCEDURE arus$cvt_text_to_integer (
              IN  input_string : string (*);
              OUT resultant_value : integer;
              IN  options : arus$text_int_options OPTIONAL;
              ) RETURNS arus$status
              LINKAGE
                  REFERENCE  (
                      resultant_value,
                      options
                      )
                  DESCRIPTOR (
                      input_string
                      );
```

Parameters:

input_string The input string to be converted.

resultant_value The resulting signed integer value. On error, this value is set to 0.

options The following caller-supplied options are supported:

Option Value	Description
arus$c_skip_blanks	If set, blanks are ignored. Otherwise, blanks are equivalent to zero.
arus$c_skip_tabs	If set, tabs are ignored. Otherwise, tab characters are illegal.

Routine *arus$cvt_text_to_integer* returns the unsuccessful status values listed in Table 1–52.

Table 1–52: Status Values Returned from Routine *arus$cvt_text_to_integer*

Status Value	Description
arus$_invalid_character	Invalid character in the input string.
arus$_overflow	Overflow detected; unable to do conversion.

Routine *arus$cvt_text_to_integer* raises no conditions.

1.7.1.4 Convert a Numeric Text String to an F_floating or G_floating Value

The text-to-floating-point routines convert a text string representation of a numeric value to an F_floating or G_floating value.

The text-to-floating-point routines convert a string representing the mathematical formula $n\ 10^m$ into a binary floating point value. The syntax for the string is [n][m], where n and m expand as follows:

Symbol	Expansion
n	[blank...] [+ | −] [decimal_digit...] [.] [decimal_digit...]
m	[letter [blank...] [+ | −]] | [[+ | −] [decimal_digit...]]
letter	$[E\|e\|D\|d\|Q\|q]$

There is no difference in semantics among any of the six valid exponent letters.

1.7.1.4.1 The *arus$cvt_text_to_real* Routine

The routine *arus$cvt_text_to_real* converts a text string containing a representation of numeric value to an F_floating representation of that value. The routine supports FORTRAN F, E, D, and G input type conversion as well as similar types for other languages.

```
PROCEDURE arus$cvt_text_to_real (
              IN  input_string : string (*);
              OUT resultant_value : real;
              IN  digits_in_fraction : integer OPTIONAL;
              IN  scale_factor : integer OPTIONAL;
              IN  options : arus$text_float_options OPTIONAL;
              OUT extension_bits : integer; OPTIONAL;
              ) RETURNS arus$status
              LINKAGE
                  REFERENCE  (
                      resultant_value,
                      digits_in_fraction,
                      scale_factor,
                      options,
                      extension_bits
                      )
                  DESCRIPTOR (
                      input_string
                      );
```

Parameters:

input_string — The input string to be converted.

resultant_value — The resulting F_floating value.

digits_in_fraction — If no decimal point is present in the input string, this specifies how many digits are to be treated as being to the right of the decimal point. If omitted, 0 is the default.

scale_factor — Signed scale factor. If present, and exponent absent, the resulting value is divided by 10***scale_factor*. If the *arus$c_force_scale* option is set, the scale factor is always applied.

options — The following caller-supplied options are supported:

Option Value	Description
arus$c_skip_blanks	If set, blanks are ignored. Otherwise, blanks are equivalent to zero.
arus$c_skip_tabs	If set, tabs are ignored. Otherwise, tab characters are illegal.
arus$c_only_e	If set, only *E* or *e* exponents are allowed.
arus$c_err_underflow	If set, underflow is an error. Otherwise, return zero.
arus$c_truncate	If set, truncate the value.
arus$c_exp_letter_required	If set, the exponent must begin with a valid exponent letter. If clear, the exponent letter may be omitted.
arus$c_force_scale	If set, the scale factor is always applied. If clear, it is only applied if there is no exponent present in the string.

extension_bits — If present, the *resultant_value* is *not* rounded, regardless of the value of the *arus$c_truncate* option, and the first 8 bits after truncation are returned in this argument.

Routine *arus$cvt_text_to_real* returns the unsuccessful status values listed in Table 1–53.

Table 1–53: Status Values Returned from Routine *arus$cvt_text_to_real*

Status Value	Description
arus$_invalid_character	Invalid character encountered; unable to do conversion.
arus$_overflow	Overflow detected; unable to do conversion.
arus$_underflow	Underflow detected; unable to do conversion.

Routine *arus$cvt_text_to_real* raises no conditions.

1.7.1.4.2 The *arus$cvt_text_to_double* Routine

The routine *arus$cvt_text_to_double* converts a text string containing a representation of a numeric value to a G_floating representation of that value. The routine supports FORTRAN F, E, D, and G input type conversion as well as similar types for other languages.

```
PROCEDURE arus$cvt_text_to_double (
              IN  input_string : string (*);
              OUT resultant_value : double;
              IN  digits_in_fraction : integer OPTIONAL;
              IN  scale_factor : integer OPTIONAL;
              IN  options : arus$text_float_options OPTIONAL;
              OUT extension_bits : integer; OPTIONAL;
              ) RETURNS arus$status
              LINKAGE
                  REFERENCE  (
                      resultant_value,
                      digits_in_fraction,
                      scale_factor,
                      options,
                      extension_bits
                      )
                  DESCRIPTOR (
                      input_string
                      );
```

Parameters:

input_string — The input string to be converted.

resultant_value — The resulting G_floating value.

digits_in_fraction — If no decimal point is present in the input string, this specifies how many digits are to be treated as being to the right of the decimal point. If omitted, 0 is the default.

scale_factor — Signed scale factor. If present, and exponent absent, the resulting value is divided by 10***scale_factor*. If the *arus$c_force_scale* option is set, the scale factor is always applied.

options — The following caller-supplied options are supported:

Option Value	Description
arus$c_skip_blanks	If set, blanks are ignored. Otherwise, blanks are equivalent to zero.
arus$c_skip_tabs	If set, tabs are ignored. Otherwise, tab characters are illegal.
arus$c_only_e	If set, only *E* or *e* exponents are allowed.
arus$c_err_underflow	If set, underflow is an error. Otherwise, return zero.
arus$c_truncate	If set, truncate the value.
arus$c_exp_letter_required	If set, the exponent must begin with a valid exponent letter. If clear, the exponent letter may be omitted.
arus$c_force_scale	If set, the scale factor is always applied. If clear, it is only applied if there is no exponent present in the string.

extension_bits If present, the *resultant_value* is *not* rounded regardless of the value of the *arus$c_truncate* option, and the first 11 bits after truncation are returned in this argument.

Routine *arus$cvt_text_to_double* returns the unsuccessful status values listed in Table 1–54.

Table 1–54: Status Values Returned from Routine *arus$cvt_text_to_double*

Status Value	Description
arus$_invalid_character	Invalid character encountered; unable to do conversion.
arus$_overflow	Overflow detected; unable to do conversion.
arus$_underflow	Underflow detected; unable to do conversion.

Routine *arus$cvt_text_to_double* raises no conditions.

1.7.1.5 Convert an F_floating or G_floating Value to a Text String

The floating-point-to-text routines convert an F_floating or G_floating value to a character string. The output string can take one of two forms; the form is determined by the value of parameter *options*. One form is a fractional value, the other is a exponential (scientific notation).

The syntax of the exponential resultant string is nm, where n and m expand as follows:

Symbol	Expansion
n	[+ | −] [decimal_digit...] [.] [decimal_digit...]
m	E [+ | −] [decimal_digit...]

The syntax of the fractional resultant string is n, where n expands as follows:

Symbol	Expansion
n	[+ | −] [decimal_digit...] [.] [decimal_digit...]

1.7.1.5.1 The *arus$cvt_real_to_text* Routine

The routine *arus$cvt_real_to_text* converts an F_floating value to a numeric text representation.

```
PROCEDURE arus$cvt_real_to_text (
                IN  input_value : REAL;
                OUT resultant_string : STRING (*);
                IN  scale_factor : integer OPTIONAL;
                IN  digits_in_integer : integer OPTIONAL;
                IN  digits_in_fraction : integer OPTIONAL;
                IN  digits_in_exponent : integer OPTIONAL;
                IN  options : arus$float_text_options OPTIONAL;
                ) RETURNS arus$status
                LINKAGE
                    REFERENCE  (
                        input_value,
                        scale_factor,
                        digits_in_integer,
                        digits_in_fraction,
                        digits_in_exponent,
                        options
                        )
                    DESCRIPTOR (
                        resultant_string
                        );
```

Parameters:

input_value The F_floating value to be converted.

resultant_string The resulting ASCII text string.

scale_factor The scale factor.

digits_in_integer The number of digits in the integer part of an exponentially formatted value.

digits_in_fraction Maximum number of digits in the fraction portion.

digits_in_exponent Minimum number of digits in the exponent field.

options The following caller-supplied options are supported:

Option Value	Description
arus$c_force_exponential	If set, the text string is returned in exponential format. If clear, the text string is returned as a fraction.
arus$c_force_plus	If set, a plus sign is forced for positive values.
arus$c_force_plus_exponent	If set, a plus sign is forced for positive exponents when exponential form is being used. This flag is ignored if fractional format is being used.
arus$c_suppress_trailing_zeroes	If set, trailing zeroes are suppressed.

Routine *arus$cvt_real_to_text* returns the unsuccessful status values listed in Table 1–55.

Table 1–55: Status Values Returned from Routine *arus$cvt_real_to_text*

Status Value	Description
arus$_output_conversion_error	Output conversion error. The result would have exceeded the size of the resultant string; *resultant_string* is filled with asterisks.
arus$_reserved_operand	Reserved operand fault.

Routine *arus$cvt_real_to_text* raises no conditions.

1.7.1.5.2 The *arus$cvt_double_to_text* Routine

The routine *arus$cvt_double_to_text* converts a G_floating value to a numeric text representation.

```
PROCEDURE arus$cvt_double_to_text (
              IN  input_value : double;
              OUT resultant_string : string (*);
              IN  scale_factor : integer OPTIONAL;
              IN  digits_in_integer : integer OPTIONAL;
              IN  digits_in_fraction : integer OPTIONAL;
              IN  digits_in_exponent : integer OPTIONAL;
              IN  options : arus$float_text_options OPTIONAL;
              ) RETURNS arus$status
              LINKAGE
                  REFERENCE  (
                      input_value,
                      scale_factor,
                      digits_in_integer,
                      digits_in_fraction,
                      digits_in_exponent,
                      options
                      )
                  DESCRIPTOR (
                      resultant_string
                      );
```

Parameters:

input_value The G_floating value to be converted.

resultant_string The resulting ASCII text string.

scale_factor The scale factor.

digits_in_integer The number of digits in the integer part of an exponentially formatted value.

digits_in_fraction Maximum number of digits in the fraction portion.

digits_in_exponent Minimum number of digits in the exponent field.

options The following caller-supplied options are supported:

Option Value	Description
arus$c_force_exponential	If set, the text string is returned in exponential format. If clear, the text string is returned as a fraction.
arus$c_force_plus	If set, a plus sign is forced for positive values.
arus$c_force_plus_exponent	If set, a plus sign is forced for positive exponents when exponential form is being used. This flag is ignored if fractional format is being used.
arus$c_suppress_trailing_zeroes	If set, trailing zeroes are suppressed.

Routine *arus$cvt_double_to_text* returns the unsuccessful status values listed in Table 1–56.

Table 1–56: Status Values Returned from Routine *arus$cvt_double_to_text*

Status Value	Description
arus$_output_conversion_error	Output conversion error. The result would have exceeded the size of the resultant string; *resultant_string* is filled with asterisks.
arus$_reserved_operand	Reserved operand fault.

Routine *arus$cvt_double_to_text* raises no conditions.

1.7.1.6 Convert a D_floating or G_floating Value to a G_floating or D_floating Value

The routines in this section are equivalent to the current VAX Math Run-Time Library routines MTH$CVT_D_G and MTH$CVT_G_D. See the VAX/VMS documentation for more information.

1.7.1.6.1 The *arus$cvt_g_to_d* Routine

The routine *arus$cvt_g_to_d* is used to convert a G_floating value to a D_floating value.

```
PROCEDURE arus$cvt_g_to_d (
              IN  input_value : double;
              OUT resultant_value : quadword_data;
              ) RETURNS arus$status
              LINKAGE
                  REFERENCE  (
                      input_value,
                      resultant_value
                      );
```

Parameters:

input_value The G_floating value to be converted.

resultant_value The converted value in D_floating format.

Routine *arus$cvt_g_to_d* returns the unsuccessful status values listed in Table 1–57.

Table 1–57: Status Values Returned from Routine *arus$cvt_g_to_d*

Status Value	Description
arus$_overflow	Floating-point overflow.
arus$_underflow	Floating-point underflow.
arus$_reserved_operand	Reserved operand fault.

Routine *arus$cvt_g_to_d* raises no conditions.

1.7.1.6.2 The *arus$cvt_d_to_g* Routine

The routine *arus$cvt_d_to_g* is used to convert a D_floating value to a G_floating value. The resulting value is rounded.

```
PROCEDURE arus$cvt_d_to_g (
              IN  input_value : quadword_data;
              OUT resultant_value : double;
              ) RETURNS arus$status
              LINKAGE
                  REFERENCE  (
                      input_value,
                      resultant_value
                      );
```

Parameters:

input_value The D_floating value to be converted.

resultant_value The converted value in G_floating format.

Routine *arus$cvt_d_to_g* returns the unsuccessful status values listed in Table 1–58.

Table 1–58: Status Values Returned from Routine *arus$cvt_d_to_g*

Status Value	Description
arus$_reserved_operand	Reserved operand fault.

Routine *arus$cvt_d_to_g* raises no conditions.

1.7.1.7 Convert an F_floating or G_floating Value to ASCII Digits and Exponent Strings

The floating-point-to-scaled-string conversion routines convert an F_floating or G_floating value to a pair of character strings. They are intended to be used as core routines for formats not directly provided by other ARUS data conversion and formatting routines.

One output string (the *digits* string) contains the significant digits of the value, with no decimal point at all. The other string (the *exponent* string) contains the exponent associated with the value if the digits string is interpreted as a number between zero and one. There is also a third output from the routine, the scale factor to be applied to the digits string to obtain the real value. This is always the binary equivalent of the exponent string. See Table 1–59 for some examples.

Table 1–59: Examples of Routines *arus$cvt_real_to_scaled_text* and *arus$cvt_double_to_scaled_text*

Input Value	*digits_string*	*exponent_string*	*scale_factor*
2.34	"234"	"1"	1
0.00067891	"67891"	"-3"	-3
-320000	"-32"	"6"	6

1.7.1.7.1 The *arus$cvt_real_to_scaled_text* Routine

The routine *arus$cvt_real_to_scaled_text* is used to convert an F_floating value to a string of ASCII digits, an exponent string, and a scale factor.

```
PROCEDURE arus$cvt_real_to_scaled_text (
              IN  input_value : real;
              OUT digits_string : string (*);
              OUT exponent_string : string (*);
              OUT scale_factor : integer;
              IN  options : arus$float_scaled_text_options OPTIONAL;
              ) RETURNS arus$status
              LINKAGE
                  REFERENCE  (
                      input_value,
                      scale_factor,
                      options
                      )
                  DESCRIPTOR (
                      digits_string,
                      exponent_string
                      );
```

Parameters:

input_value	The F_floating value to be converted.
digits_string	Text string containing the converted significant digits.
exponent_string	Text string containing the converted exponent value.
scale_factor	The scale factor.
options	The following caller-supplied options are supported:

Option Value	Description
arus$c_force_plus	If set, a plus sign is forced for positive values.
arus$c_force_plus_exponent	If set, a plus sign is forced for positive exponents.

Routine *arus$cvt_real_to_scaled_text* returns the unsuccessful status values listed in Table 1–60.

Table 1–60: Status Values Returned from Routine *arus$cvt_real_to_scaled_text*

Status Value	Description
arus$_output_conversion_error	Output conversion error. The result would have exceeded the size of the resultant string; *resultant_string* is filled with asterisks.
arus$_reserved_operand	Reserved operand fault.

Routine *arus$cvt_real_to_scaled_text* raises no conditions.

1.7.1.7.2 The *arus$cvt_double_to_scaled_text* Routine

The routine *arus$cvt_double_to_scaled_text* is used to convert a G_floating value to a string of ASCII digits, an exponent string, and a scale factor.

```
PROCEDURE arus$cvt_double_to_scaled_text (
              IN  input_value : double;
              OUT digits_string : string (*);
              OUT exponent_string : string (*);
              OUT scale_factor : integer;
              IN  options : arus$float_scaled_text_options OPTIONAL;
              ) RETURNS arus$status
              LINKAGE
                  REFERENCE  (
                      input_value,
                      scale_factor,
                      options
                      )
                  DESCRIPTOR (
                      digits_string,
                      exponent_string
                      );
```

Parameters:

input_value	The G_floating value to be converted.
digits_string	Text string containing the converted significant digits.
exponent_string	Text string containing the converted exponent value.
scale_factor	The scale factor.
options	The following caller-supplied options are supported:

Option Value	Description
arus$c_force_plus	If set, a plus sign is forced for positive values.
arus$c_force_plus_exponent	If set, a plus sign is forced for positive exponents.

Routine *arus$cvt_double_to_scaled_text* returns the unsuccessful status values listed in Table 1–61.

Table 1–61: Status Values Returned from Routine *arus$cvt_double_to_scaled_text*

Status Value	Description
arus$_output_conversion_error	Output conversion error. The result would have exceeded the size of the resultant string; *resultant_string* is filled with asterisks.
arus$_reserved_operand	Reserved operand fault.

Routine *arus$cvt_double_to_scaled_text* raises no conditions.

1.7.1.8 Convert an Integer and Scale Factor to a Text String

The integer-and-scale-factor-to-text conversion routines convert an integer value and scale factor to a character string. The output string can take one of two forms; the form is determined by the value of parameter *options*. One form is a fractional value, the other is a exponential (scientific notation).

The syntax of the exponential resultant string is nm, where n and m expand as follows:

Symbol	Expansion
n	[+ | −] [decimal_digit...] [.] [decimal_digit...]
m	E [+ | −] [decimal_digit...]

The syntax of the fractional resultant string is n, where n expands as follows:

Symbol	Expansion
n	[+ | −] [decimal_digit...] [.] [decimal_digit...]

1.7.1.8.1 The *arus$cvt_integer_to_real_text* Routine

The *arus$cvt_integer_to_real_text* routine converts a scaled integer value to a character representation of a real value.

```
PROCEDURE arus$cvt_integer_to_real_text (
              IN  integer_value : integer;
              IN  scale_factor : integer;
              OUT resultant_string : string (*);
              IN  options : arus$int_real_text_options OPTIONAL;
              ) RETURNS arus$status
              LINKAGE
                  REFERENCE  (
                      integer_value,
                      scale_factor
                      )
                  DESCRIPTOR (
                      resultant_string
                      );
```

Parameters:

integer_value An integer value.

scale_factor The scale factor.

resultant_string The resulting text string.

options The following caller-supplied options are supported:

Option Value	Description
arus$c_force_exponential	If set, the text string is returned in exponential format. If clear, the text string is returned as a fraction.
arus$c_force_plus	If set, a plus sign is forced for positive values.
arus$c_suppress_trailing_zeroes	If set, trailing zeroes are suppressed.

Routine *arus$cvt_integer_to_real_text* returns the unsuccessful status values listed in Table 1–60.

Table 1–62: Status Values Returned from Routine *arus$cvt_integer_to_real_text*

Status Value	Description
arus$_output_conversion_error	Output conversion error. The result would have exceeded the size of the resultant string; *resultant_string* is filled with asterisks.

Routine *arus$cvt_integer_to_real_text* raises no conditions.

1.7.1.9 International Data Conversion and Formatting Routines

The following set of routines supports digit separators and radix point symbols other than the default, United States, symbols.

A context variable is used to store the information to avoid the overhead of translating the symbols every time. If the context variable is omitted or is NIL, the routine attempts to translate the symbols. The input and output, as well as the accepted values for the *options* arguments are unique for each international routine.

1.7.1.9.1 The *arus$cvt_integer_to_format_text* Routine

The *arus$cvt_integer_to_format_text* routine converts a signed integer to a decimal ASCII text string with optional, user-defined, digit-separator support. The default digit-separator symbol is a comma (,).

The syntax of the resultant string is *n*, where *n* expands as follows:

Symbol	Expansion
n	[+ | −] [decimal_digit...] [digit_separator] [decimal_digit...]

The declaration of routine *arus$cvt_integer_to_format_text* is as follows:

```
PROCEDURE arus$cvt_integer_to_format_text (
                IN  input_value : integer;
                OUT resultant_string : string (*);
                IN  number_of_digits : integer OPTIONAL;
                IN  options  : arus$int_format_text_options OPTIONAL;
                IN OUT context : arus$context OPTIONAL
                ) RETURNS arus$status
                LINKAGE
                    REFERENCE  (
                        input_value,
                        number_of_digits,
                        options,
                        context
                        )
                    DESCRIPTOR (
                        resultant_string
                        );
```

Parameters:

input_value — The input value to be converted to text.

resultant_string — The resulting decimal ASCII text string.

number_of_digits — Minimum number of digits to be produced. If the actual number of significant digits is smaller, leading zeroes are produced. If *number_of_digits* is zero, a blank field results. The default is 1.

options — The following caller-supplied option is supported:

Option Value	Description
arus$c_force_plus	If set, a plus sign is forced for positive values.

context — Context variable that retains the translation context over multiple calls to the conversion routines with international digit separator and radix support. This variable is initialized to NIL by the caller before the first call to the international conversion routines.

Routine *arus$cvt_integer_to_format_text* returns the unsuccessful status values listed in Table 1–50.

Table 1–63: Status Values Returned from Routine *arus$cvt_integer_to_format_text*

Status Value	Description
arus$_output_conversion_error	Output conversion error. The result would have exceeded the size of the resultant string; *resultant_string* is filled with asterisks.

Routine *arus$cvt_integer_to_format_text* raises no conditions.

1.7.1.9.2 The *arus$cvt_format_text_to_integer* Routine

The routine *arus$cvt_format_text_to_integer* is used to convert an ASCII text string representation of a decimal number to a signed integer value. The text string representation may contain digit separators; they are ignored. The user can select the character used for the digit separator; the default digit separator is a comma (,).

The syntax of a valid input string is *n*, where *n* expands as follows:

Symbol	Expansion
n	[+ | −] [decimal_digit...] [digit_separator] [decimal_digit...]

The declaration of routine *arus$cvt_format_text_to_integer* is as follows:

```
PROCEDURE arus$cvt_format_text_to_integer (
              IN  input_string : string (*);
              OUT resultant_value : integer;
              IN  options : arus$format_text_int_options OPTIONAL;
              IN OUT context : arus$context OPTIONAL;
              ) RETURNS arus$status
              LINKAGE
                  REFERENCE  (
                      resultant_value,
                      options,
                      context
                      )
                  DESCRIPTOR (
                      input_string
                      );
```

Parameters:

input_string The input string containing the string representation of a signed decimal value to be converted.

resultant_value The resulting signed integer value.

options The following caller-supplied options are supported:

Option Value	Description
arus$c_skip_blanks	If set, blanks are ignored. Otherwise, blanks are equivalent to zero.
arus$c_skip_tabs	If set, tab characters are ignored. Otherwise, tab characters are illegal.

context Context variable that retains the translation context over multiple calls to the conversion routines with international digit separator and radix support. This variable is initialized to NIL by the caller before the first call to the international conversion routines.

Routine *arus$cvt_format_text_to_integer* returns the unsuccessful status values listed in Table 1–64.

Table 1–64: Status Values Returned from Routine *arus$cvt_format_text_to_integer*

Status Value	Description
arus$_invalid_character	Invalid character encountered; unable to do conversion.
arus$_overflow	Overflow detected; unable to do conversion.

Routine *arus$cvt_format_text_to_integer* raises no conditions.

1.7.1.9.3 The *arus$cvt_real_to_format_text* Routine

The *arus$cvt_real_to_format_text* routine converts an F_floating value to a character string, and allows the user to specify a radix point other than the default United States radix point, the period (.). The output string can take one of two forms; the form is determined by the value of parameter *options*. One form is a fractional value, the other is a exponential (scientific notation).

The syntax of the exponential resultant string is nm, where n and m expand as follows:

Symbol	Expansion
n	[+ | −] [decimal_digit...] [.] [decimal_digit...]
m	E [+ | −] [decimal_digit...]

The syntax of the fractional resultant string is n, where n expands as follows:

Symbol	Expansion
n	[+ | −] [decimal_digit...] [.] [decimal_digit...]

The declaration of routine *arus$cvt_real_to_format_text* is as follows:

```
PROCEDURE arus$cvt_real_to_format_text (
              IN  input_value : real;
              OUT resultant_string : string (*);
              IN  scale_factor : integer OPTIONAL;
              IN  digits_in_fraction : integer OPTIONAL;
              IN  options : arus$real_format_text_options OPTIONAL;
              IN OUT context : arus$context OPTIONAL;
              ) RETURNS arus$status
              LINKAGE
                  REFERENCE  (
                      input_value,
                      scale_factor,
                      digits_in_fraction,
                      options,
                      context
                      )
                  DESCRIPTOR (
                      resultant_string
                      );
```

Parameters:

input_value The F_floating value to be converted.

resultant_string The resulting ASCII text string.

scale_factor The scale factor.

digits_in_fraction Minimum number of digits in the fraction portion.

options The following caller-supplied options are supported:

Option Value	Description
arus$c_force_exponential	If set, the text string is returned in exponential format. If clear, the text string is returned as a fraction.
arus$c_force_plus	If set, a plus sign is forced for positive values.
arus$c_force_plus_exponent	If set, a plus sign is forced for positive exponents.
arus$c_suppress_trailing_zeroes	If set, trailing zeroes are suppressed.

context Context variable that retains the translation context over multiple calls to the conversion routines with international digit separator and radix support. This variable is initialized to NIL by the caller before the first call to the international conversion routines.

Routine *arus$cvt_real_to_format_text* returns the unsuccessful status values listed in Table 1–65.

Table 1–65: Status Values Returned from Routine *arus$cvt_real_to_format_text*

Status Value	Description
arus$_output_conversion_error	Output conversion error. The result would have exceeded the size of the resultant string; *resultant_string* is filled with asterisks.
arus$_reserved_operand	Reserved operand fault.

Routine *arus$cvt_real_to_format_text* raises no conditions.

1.7.1.9.4 The *arus$cvt_double_to_format_text* Routine

The *arus$cvt_double_to_format_text* routine converts a G_floating value to a character string, and allows the user to specify a radix point other than the default United States radix point, the period (.). The output string can take one of two forms; the form is determined by the value of parameter *options*. One form is a fractional value, the other is a exponential (scientific notation).

The syntax of the exponential resultant string is nm, where n and m expand as follows:

Symbol	Expansion
n	[+ | −] [decimal_digit...] [.] [decimal_digit...]
m	E [+ | −] [decimal_digit...]

The syntax of the fractional resultant string is n, where n expands as follows:

Symbol	Expansion
n	[+ | −] [decimal_digit...] [.] [decimal_digit...]

The declaration of routine *arus$cvt_double_to_format_text* is as follows:

```
PROCEDURE arus$cvt_double_to_format_text (
              IN  input_value : double;
              OUT resultant_string : string (*);
              IN  scale_factor : integer OPTIONAL;
              IN  digits_in_fraction : integer OPTIONAL;
              IN  options : arus$real_format_text_options OPTIONAL;
              IN OUT context : arus$context OPTIONAL;
              ) RETURNS arus$status
              LINKAGE
                  REFERENCE  (
                      input_value,
                      scale_factor,
                      digits_in_fraction,
                      options,
                      context
                      )
                  DESCRIPTOR (
                      resultant_string
                      );
```

Parameters:

input_value The G_floating value to be converted.

resultant_string The resulting ASCII text string.

scale_factor The scale factor.

digits_in_fraction Minimum number of digits in the fraction portion.

options The following caller-supplied options are supported:

Option Value	Description
arus$c_force_exponential	If set, the text string is returned in exponential format. If clear, the text string is returned as a fraction.
arus$c_force_plus	If set, a plus sign is forced for positive values.
arus$c_force_plus_exponent	If set, a plus sign is forced for positive exponents.
arus$c_suppress_trailing_zeroes	If set, trailing zeroes are suppressed.

context Context variable that retains the translation context over multiple calls to the conversion routines with international digit separator and radix support. This variable is initialized to NIL by the caller before the first call to the international conversion routines.

Routine *arus$cvt_double_to_format_text* returns the unsuccessful status values listed in Table 1–66.

Table 1–66: Status Values Returned from Routine *arus$cvt_double_to_format_text*

Status Value	Description
arus$_output_conversion_error	Output conversion error. The result would have exceeded the size of the resultant string; *resultant_string* is filled with asterisks.
arus$_reserved_operand	Reserved operand fault.

Routine *arus$cvt_double_to_format_text* raises no conditions.

1.7.1.9.5 The *arus$cvt_format_text_to_real* Routine

The *arus$cvt_format_text_to_real* routine converts an ASCII text string representation of a numeric value to an F_floating value, and allows the user to specify a radix point other than the default United States radix point, the period (.).

This routine and the related routine, *arus$cvt_format_text_to_double*, convert a string representing the mathematical formula $n10^m$ into a binary floating-point value. The syntax for the string is [n][m], where n and m expand as follows:

Symbol	Expansion
n	[blank...] [+ | −] [decimal_digit...] [radix_point] [decimal_digit...]
m	[letter [blank...] [+ | −]] | [[+ | −] [decimal_digit...]]
letter	$[E\|e\|D\|d\|Q\|q]$

There is no difference in semantics among any of the six valid exponent letters.

The declaration of routine *arus$cvt_format_text_to_real* is as follows:

```
PROCEDURE arus$cvt_format_text_to_real (
                IN  input_string : string (*);
                OUT resultant_value : real;
                IN  digits_in_fraction : integer OPTIONAL;
                IN  scale_factor : integer OPTIONAL;
                IN  options : arus$format_text_real_options OPTIONAL;
                OUT extension_bits : integer; OPTIONAL;
                IN OUT context : arus$context OPTIONAL;
                ) RETURNS arus$status
                LINKAGE
                    REFERENCE  (
                        resultant_value,
                        digits_in_fraction,
                        scale_factor,
                        options,
                        extension_bits,
                        context
                        )
                    DESCRIPTOR (
                        input_string
                        );
```

Parameters:

input_string The input string containing the string representation of a numeric value to be converted.

resultant_value The resulting F_floating value.

digits_in_fraction Number of digits in the fraction if no decimal point is included in the input string.

scale_factor The scale factor.

options The following caller-supplied options are supported:

Option Value	Description
arus$c_skip_blanks	If set, blanks are ignored. Otherwise, blanks are equivalent to zero.
arus$c_skip_tabs	If set, tabs are ignored. Otherwise, tab characters are illegal.
arus$c_only_e	If set, only *E* or *e* exponents are allowed.
arus$c_err_underflow	If set, underflow is an error.
arus$c_truncate	If set, do not round value.
arus$c_exp_letter_required	If set, the exponent must begin with a valid exponent letter. If clear, the exponent letter may be omitted.
arus$c_force_scale	If set, the scale factor is always applied. If clear, it is only applied if there is no exponent present in the string.
arus$c_translate_symbols	If set, the routine does a translation of the digit separator and radix point regardless of the context value.

extension_bits If present, the *resultant_value* is *not* rounded regardless of the value of the *arus$c_truncate* option, and the first 8 bits after truncation are returned in this argument.

context Context variable that retains the translation context over multiple calls to the conversion routines with international digit separator and radix support. This variable is initialized to NIL by the caller before the first call to the international conversion routines.

Routine *arus$cvt_format_text_to_real* returns the unsuccessful status values listed in Table 1–67.

Table 1–67: Status Values Returned from Routine *arus$cvt_format_text_to_real*

Status Value	Description
arus$_invalid_character	Invalid character encountered; unable to do conversion.
arus$_overflow	Overflow detected; unable to do conversion.
arus$_underflow	Underflow detected; unable to do conversion.

Routine *arus$cvt_format_text_to_real* raises no conditions.

1.7.1.9.6 The *arus$cvt_format_text_to_double* Routine

The *arus$cvt_format_text_to_real* routine converts an ASCII text string representation of a numeric value to a G_floating value, and allows the user to specify a radix point other than the default United States radix point, the period (.).

This routine converts a string representing the mathematical formula $n10^m$ into a binary floating-point value. The syntax for the string is [n][m], where n and m expand as follows:

Symbol	Expansion
n	[blank...] [+ | −] [decimal_digit...] [radix_point] [decimal_digit...]
m	[letter [blank...] [+ | −]] | [[+ | −] [decimal_digit...]]
letter	[*E*|*e*|*D*|*d*|*Q*|*q*]

There is no difference in semantics among any of the six valid exponent letters.

The declaration of routine *arus$cvt_format_text_to_real* is as follows:

```
PROCEDURE arus$cvt_format_text_to_double (
              IN  input_string : string (*);
              OUT resultant_value : double;
              IN  digits_in_fraction : integer OPTIONAL;
              IN  scale_factor : integer OPTIONAL;
              IN  options : arus$format_text_real_options OPTIONAL;
              OUT extension_bits : integer; OPTIONAL;
              IN OUT context : arus$context OPTIONAL;
              ) RETURNS arus$status
              LINKAGE
                  REFERENCE  (
                      resultant_value,
                      digits_in_fraction,
                      scale_factor,
                      options,
                      extension_bits,
                      context
                      )
                  DESCRIPTOR (
                      input_string
                      );
```

Parameters:

input_string The input string containing the string representation of a numeric value to be converted.

resultant_value The resulting G_floating value.

digits_in_fraction Number of digits in the fraction if no decimal point is included in the input string.

scale_factor The scale factor.

options The following caller-supplied options are supported:

Option Value	Description
arus$c_skip_blanks	If set, blanks are ignored. Otherwise, blanks are equivalent to zero.
arus$c_skip_tabs	If set, tabs are ignored. Otherwise, tab characters are illegal.
arus$c_only_e	If set, only *E* or *e* exponents are allowed.
arus$c_err_underflow	If set, underflow is an error.
arus$c_truncate	If set, do not round value.
arus$c_exp_letter_required	If set, the exponent must begin with a valid exponent letter. If clear, the exponent letter may be omitted.
arus$c_force_scale	If set, the scale factor is always applied. If clear, it is only applied if there is no exponent present in the string.
arus$c_translate_symbols	If set, the routine does a translation of the digit separator and radix point regardless of the context value.

extension_bits — If present, the *resultant_value* is *not* rounded regardless of the value of the *arus$c_truncate* option, and the first 11 bits after truncation are returned in this argument.

context — Context variable that retains the translation context over multiple calls to the conversion routines with international digit separator and radix support. This variable is initialized to NIL by the caller before the first call to the international conversion routines.

Routine *arus$cvt_format_text_to_double* returns the unsuccessful status values listed in Table 1–68.

Table 1–68: Status Values Returned from Routine *arus$cvt_format_text_to_double*

Status Value	Description
arus$_invalid_character	Invalid character encountered; unable to do conversion.
arus$_overflow	Overflow detected; unable to do conversion.

Routine *arus$cvt_format_text_to_double* raises no conditions.

GLOSSARY

AIA: Application Integration Architecture

ARUS: Application Run-Time Utility Services

CMA: Common Multithread Architecture

PSM: Print System Model

INDEX

O

Mica Working Design Document
Application Run-Time Utility Services

Section 3 of 4

Revision 0.6
5–April–1988

Issued by:
Al Simons

digital™

1.9 Environment Attribute Routines

An *environment attribute* is a named entity with one or more string values associated with it. It serves the same purpose that a logical name does on VAX/VMS, and that an environment variable does on an ULTRIX system.

Environment attributes have a number of associated characteristics. Those characteristics are as follows:

- Environment attributes may be shared or private.
- Environment attributes may be secure or insecure. A user must have special privileges to read secure environment attributes, whereas an insecure environment attribute can be read by any user.

  An example of an insecure environment attribute on VAX/VMS is the logical name SYS$SYSTEM; anyone on the system can determine the string value associated with that environment attribute.
- Environment attributes may be trustworthy or untrustworthy. A user must have special privileges to modify a trustworthy environment attribute, whereas an untrustworthy environment attribute can be modified by any number of users.

  The EXEC mode translation of VAX/VMS logical SYS$SYSTEM, though insecure, is trustworthy; a user needs the SYSNAM privilege to modify it.
- Environment attributes may be temporary or permanent. They are permanent, by default.

  Temporary attributes may be created only in a private domain. (See the discussion of domains in Section 1.9.1.) Temporary attributes are automatically destroyed at the termination of the program. It is legal to create a temporary environment attribute in the same domain as a permanent environment attribute of the same name. In this case, the permanent attribute is hidden for the duration of the program, but it is not destroyed or modified in any way. It becomes accessible again when the temporary attribute is destroyed.

The ARUS routines described in this section provide a means to examine and manipulate environment attributes. The intent of these routines is to build on underlying system services when the operating system provides adequate support, such as that provided by VAX/VMS logical names. When necessary, however, the ARUS routines will be more extensive, providing the entire environment attribute facility when the underlying system's support is nonexistent or inadequate.

\We will eventually provide this entire interface on ULTRIX systems. However, because ULTRIX falls into the second category (inadequate underlying support), and because the time until FLINT FRS is so short, it is unlikely that full support will be available by FLINT FRS. The minimum we will provide is a direct mapping onto environment attributes. This, of course, provides no sharing of attributes or trustworthiness characteristics.\

\Future enhancements to these routines may allow for network-wide "environments," through use of the DECnet Name Server or other name services. For now, however, the scope of these routines is the system, for shared attributes, and the process, for private attributes. In a client and server environment, the environment will be the distributed environment of the client and server pair.\

\The concept of secure attributes poses several problems. We would like to have a single method for specifying the security characteristics, but that involves posing a single security model across several systems with differing underlying security mechanisms. I think that it is a needed feature, but I am not sure if it is even possible. I would appreciate any comments, but especially any dealing with these issues:

- Are secure attributes necessary at the ARUS level for FRS? Ever?
- Is attempting to build a secure package that has a uniform interface/model across several systems the proper venue of an RTL?

- Is it possible?

\

1.9.1 Environment Attribute Domains

Environment attributes are defined within an *environment attribute domain*. This allows a partitioning of the name space to avoid attribute name collisions; it also allows control of attribute characteristics, particularly sharing, security and trustworthiness. A domain is either private to the process (in the VAX/VMS sense) or it is shared. Shared domains can be accessed by other processes on the system, subject to the protection of the domain.

The model for environment attribute domains is logical name tables on VAX/VMS. We believe that they are very powerful and flexible, and suitable as the model to be used on other systems.

\We depart from this model a bit in the interests of simplicity and portability. In the VMS logical name table model, a logical name table specification can itself be a multiply-valued logical name, and each table is searched in turn. In these routines, we require the domain specification to really be a domain. Note that the application writer can still trivially do the multidomain lookup with repeated calls.

Is this a reasonable simplification, or should that capability be built into the ARUS routines on all systems?\

1.9.2 Functional Interface and Description

The following sections describe the various routines associated with environment attributes.

1.9.2.1 Types Used

```
TYPE
    arus$env_att_num_chars : (
        arus$c_number_of_values,
        arus$c_is_trustworthy
        );

    arus$env_att_string_chars : (
        arus$c_containing_domain
        );

    arus$env_att_req_chars_type : (
        arus$c_only_trustworthy
        );

    arus$env_att_required_chars :
        SET [arus$env_att_req_chars_type];

    arus$env_att_create_opt_type : (
        arus$c_trustworthy,
        arus$c_temporary
        );

    arus$env_att_create_options :
        SET [arus$env_att_create_opt_type];

    arus$cre_env_att_dom_opt_type : (
        arus$c_trustworthy_dom,
        arus$c_private_dom
        );

    arus$cre_env_att_dom_options :
        SET [arus$cre_env_att_dom_opt_type];
```

1.9.2.2 Obtaining Environment Attribute Values

The most common operation performed on environment attributes is determining the value of an attribute. On VAX/VMS, this is similar to translating a logical name; indeed, the routines in this section might be implemented on VAX/VMS by translating logical names.

Because environment attributes are multiply valued, there are several routines. Routine *arus$get_env_attribute_num_char* obtains various numeric characteristics of the specified attribute, among them the number of values associated with it; routine *arus$get_env_attribute_str_char* obtains string-valued characteristics, notably the domain in which the attribute is defined; routine *arus$get_env_attribute_value* returns one of the values associated with the environment attribute.

1.9.2.2.1 The *arus$get_env_attribute_num_char* Routine

This routine is used to inquire about one of the numeric characteristics of the specified environment attribute.

```
PROCEDURE arus$get_env_attribute_num_char : (
    IN attribute : string (*);
    IN desired_characteristic : arus$env_att_num_chars;
    OUT characteristic_value : integer;
    IN attribute_domain : string (*) OPTIONAL;
    ) RETURNS arus$status
    LINKAGE
        REFERENCE (
            desired_characteristic,
            characteristic_value
            )
        DESCRIPTOR (
            attribute,
            attribute_domain
            );
```

Parameters:

attribute The name of the attribute whose characteristics are to be examined.

desired_characteristic The characteristic whose value is to be returned. This parameter may take the following values:

Characteristic Value	Description
arus$c_number_of_values	Specifies that the number of values associated with the environment attribute is to be returned.
arus$c_is_trustworthy	Specifies that the routine should return a value indicating whether or not the environment attribute is trustworthy.

characteristic_value The value of the characteristic specified in parameter *desired_characteristic*. This value is interpreted based on *desired_characteristic* as follows:

Characteristic Value	Description
arus$c_number_of_values	The number of values associated with the environment attribute.
arus$c_trustworthy	The value -1 if the environment attribute is trustworthy, the value 0 if it is not.

attribute_domain The name of the attribute domain to be searched for the specified attribute. If omitted, this parameter defaults to an implementation-specific default domain.

Routine *arus$get_env_attribute_num_char* returns the unsuccessful status values listed in Table 1–1.

Table 1–1: Status Values Returned from Routine *arus$get_env_attribute_num_char*

Status Value	Description
arus$_no_such_domain	The attribute domain specified for the environment attribute lookup does not exist, or the domain is secure and the user does not have the right to look in the domain. Note that in such a case, the routine does not disclose the existence of the domain.
arus$_no_such_attribute	The attribute specified for the lookup does not exist, or the attribute is secure and the user does not have the right to look at it. Note that in such a case, the routine does not disclose the existence of the attribute.

1.9.2.2.2 The *arus$get_env_attribute_str_char* Routine

This routine is used to inquire about one of the string-valued characteristics of the specified environment attribute.

```
PROCEDURE arus$get_env_attribute_str_char : (
    IN attribute : string (*);
    IN desired_characteristic : arus$env_att_string_chars;
    OUT characteristic_value : string (*);
    IN attribute_domain : string (*) OPTIONAL;
    ) RETURNS arus$status
    LINKAGE
        REFERENCE (
            desired_characteristic,
            )
        DESCRIPTOR (
            attribute,
            characteristic_value,
            attribute_domain
            );
```

Parameters:

attribute
The name of the attribute whose characteristics are to be examined.

desired_characteristic
The characteristic whose value is to be returned. This parameter may take the following values:

Characteristic Value	Description
arus$c_containing_domain	Specifies that the name of the domain in which the environment attribute is found is to be returned.

characteristic_value
The value of the characteristic specified in parameter *desired_characteristic*. This value is interpreted based on *desired_characteristic* as follows:

Characteristic Value	Description
arus$c_containing_domain	The name of the domain in which the environment attribute is found.

attribute_domain
The name of the attribute domain to be searched for the specified attribute. If omitted, this parameter defaults to an implementation-specific default domain.

Routine *arus$get_env_attribute_str_char* returns the unsuccessful status values listed in Table 1–2.

Table 1–2: Status Values Returned from Routine *arus$get_env_attribute_str_char*

Status Value	Description
arus$_no_such_domain	The attribute domain specified for the environment attribute lookup does not exist, or the domain is secure and the user does not have the right to look in the domain. Note that in such a case, the routine does not disclose the existence of the domain.
arus$_no_such_attribute	The attribute specified for the lookup does not exist, or the attribute is secure and the user does not have the right to look at it. Note that in such a case, the routine does not disclose the existence of the attribute.

1.9.2.2.3 The *arus$get_env_attribute_value* routine

Routine *arus$get_env_attribute_value* is used to actually retrieve one of the possibly numerous values associated with an environment attribute.

```
PROCEDURE arus$get_env_attribute_value : (
    IN attribute : string (*);
    IN index : integer;
    OUT value : string (*);
    IN attribute_domain : string (*) OPTIONAL;
    IN required_characteristics : arus$env_att_required_chars OPTIONAL;
    ) RETURNS arus$status
    LINKAGE
        REFERENCE (
            index,
            required_characteristics
            )
        DESCRIPTOR (
            attribute,
            value,
            attribute_domain
            );
```

Parameters:

attribute The attribute whose value is to be returned.

index Selects which of potentially several values is to be returned. Environment attribute values are one based, that is, an index argument on 1 causes the first value in the list to be returned, 2 causes the second to be returned, and so on.

value The actual value returned. If the routine completed unsuccessfully, the null string is returned.

attribute_domain The name of the attribute domain to be searched for the specified attribute. If omitted, this parameter defaults to an implementation-specific default domain.

required_characteristics The characteristics that must be associated with an environment attribute for the lookup to be successful. The possible characteristics are:

Characteristic Value	Description
arus$c_only_trustworthy	The environment attribute must be trustworthy.

If this argument is omitted, the lookup is made without regard to characteristics.

Routine *arus$get_env_attribute_value* returns the unsuccessful status values listed in Table 1–3.

Table 1–3: Status Values Returned from Routine *arus$get_env_attribute_value*

Status Value	Description
arus$_no_such_domain	The attribute domain specified for the environment attribute lookup does not exist, or the domain is secure and the user does not have the right to look in the domain. Note that in such a case, the routine does not disclose the existence of the domain.
arus$_no_such_attribute	The attribute specified for the lookup does not exist, or it does not have the required characteristics, or the attribute is secure and the user does not have the right to look at it. Note that in such a case, the routine does not disclose the existence of the attribute.
arus$_no_such_index	The *index* argument is larger than the number of values associated with the attribute.

1.9.2.3 Creation of Environment Attributes

In order to get environment attribute values, one has to first create them. The routines *arus$create_env_attribute* and *arus$create_env_attribute_dom* provide the necessary capabilities.

1.9.2.3.1 The *arus$create_env_attribute* Routine

Routine *arus$create_env_attribute* allows a user to create an environment attribute. The attribute may be temporary or permanent, it may be trustworthy or untrustworthy, and it may have many values.

```
PROCEDURE arus$create_env_attribute : (
    IN attribute : string (*);
    IN value : LIST string (*);
    IN characteristics : arus$env_att_create_options OPTIONAL;
    IN attribute_domain : string (*) OPTIONAL;
    ) RETURNS arus$status
    LINKAGE
        REFERENCE (
            characteristics
            )
        DESCRIPTOR (
            attribute,
            value,
            attribute_domain
            );
```

Parameters:

attribute — The name of the attribute that is to be created.

value — A list of the values to be associated with the new attribute. The first value in the list is assigned index 1, the second index 2, and so on.

characteristics — The characteristics that are to be associated with the environment attribute. The possible characteristics are:

Characteristic Value	Description
arus$c_trustworthy	The environment attribute is trustworthy. The use of this characteristic requires an implementation-defined privilege.
arus$c_temporary	The environment exists only for the duration of the program that creates it.

If this argument is omitted, the attribute is created without special characteristics.

attribute_domain The name of the attribute domain to contain the new attribute. If omitted, this parameter defaults to an implementation-specific default domain.

Routine *arus$create_env_attribute* returns the unsuccessful status values listed in Table 1–4.

Table 1–4: Status Values Returned from Routine *arus$create_env_attribute*

Status Value	Description
arus$_insufficient_privilege	The program attempted to create a trustworthy environment attribute without the necessary implementation-defined privilege.
arus$_no_such_domain	The attribute domain specified for the environment attribute lookup does not exist, or the domain is secure and the user does not have the right to look in the domain. Note that in such a case, the routine does not disclose the existence of the domain.

1.9.2.3.2 The *arus$create_env_attribute_dom* Routine

Routine *arus$create_env_attribute_dom* gives the user the ability to create a new domain. At the time of creation, the domain is declared to be either private or shared, and if shared, its protection is specified.

\The method of specifying the protection is TBS.\

With the proper privilege, a shared domain can also be declared to be trustworthy. Only trustworthy environment attributes may be created within a trustworthy domain.

It is an error to attempt to create a domain that already exists.

```
PROCEDURE arus$create_env_attribute_dom (
     IN attribute_domain : string (*);
     IN characteristics : arus$cre_env_att_dom_options OPTIONAL;
     ) RETURNS arus$status
     LINKAGE
         REFERENCE (
             characteristics
             )
         DESCRIPTOR (
             attribute_domain
             );
```

Parameters:

attribute_domain The name of the domain that is to be created.

characteristics The characteristics that are to be associated with the newly created domain. This parameter may take the following values:

Characteristic Value	Description
arus$c_trustworthy_dom	The environment attribute domain, and all attributes it contains are trustworthy. The use of this characteristic requires an implementation-defined privilege.
arus$c_private_dom	The domain is private to this process.

If this argument is omitted, the domain is created without special characteristics.

\The parameters associated with security of the domain will be specified later, when the questions around the whole security issue are resolved.\

Routine *arus$create_env_attribute_dom* returns the unsuccessful status values listed in Table 1–5.

Table 1–5: Status Values Returned from Routine *arus$create_env_attribute_dom*

Status Value	Description
arus$_insufficient_privilege	The program attempted to create a trustworthy environment attribute domain without the necessary implementation-defined privilege.
arus$_domain_exists	Another environment attribute domain with the specified name already exists. A new domain was not created.

1.9.2.4 Destruction of Unneeded Environment Attributes

Temporary environment attributes are destroyed when the program terminates. However, permanent environment attributes survive until explicitly destroyed or until the system fails. Therefore, environment attribute destruction routines are needed to perform housekeeping.

1.9.2.4.1 The *arus$destroy_env_attribute* Routine

Routine *arus$destroy_env_attribute* destroys the specified environment attribute. The privileges needed to create the attribute are also required to destroy it. Specifically, the process attempting the destruction must have access to the domain, and if the domain or the attribute is trustworthy, the process must have the appropriate implementation-defined privilege.

```
PROCEDURE arus$destroy_env_attribute (
    IN attribute : string (*);
    IN attribute_domain : string (*) OPTIONAL;
    ) RETURNS arus$status
    LINKAGE
        DESCRIPTOR (
            attribute,
            attribute_domain
            );
```

Parameters:

attribute The name of the attribute that is to be destroyed.

attribute_domain The name of the attribute domain to be searched for the specified attribute. If omitted, this parameter defaults to an implementation-specific default domain.

Routine *arus$destroy_env_attribute* returns the unsuccessful status values listed in Table 1–6.

Table 1–6: Status Values Returned from Routine *arus$destroy_env_attribute*

Status Value	Description
arus$_insufficient_privilege	The program attempted to destory a trustworthy environment attribute without the necessary implementation-defined privilege.
arus$_no_such_domain	The attribute domain specified for the environment attribute lookup does not exist, or the domain is secure and the user does not have the right to look in the domain. Note that in such a case, the routine does not disclose the existence of the domain.
arus$_no_such_attribute	The attribute specified for destruction does not exist, or the attribute is secure and the user does not have the right to look at it. Note that in such a case, the routine does not disclose the existence of the attribute.
arus$_perm_attribute_revealed	The attribute that was destroyed was a temporary attribute that had been masking a permanent attribute with the same name. The permanent attribute is now revealed.

1.9.2.4.2 The *arus$destroy_env_attribute_dom* Routine

Routine *arus$destroy_env_attribute_dom* destroys the specified environment attribute domain. The privileges needed to create the domain are necessary to destroy it. Specifically, the process attempting the destruction must have access to the domain, and if the domain is trustworthy, the process must have the appropriate implementation-defined privilege.

All attributes contained within the domain are destroyed with the domain.

```
PROCEDURE arus$destroy_env_attribute_dom (
    IN attribute_domain : string (*);
    ) RETURNS arus$status
    LINKAGE
        DESCRIPTOR (
            attribute_domain
            );
```

Parameters:

attribute_domain The name of the attribute domain to be destroyed.

Routine *arus$destroy_env_attribute_dom* returns the unsuccessful status values listed in Table 1–7.

Table 1–7: Status Values Returned from Routine *arus$destroy_env_attribute_dom*

Status Value	Description
arus$_insufficient_privilege	The program attempted to destroy a trustworthy environment attribute domain without the necessary implementation-defined privilege.
arus$_no_such_attribute	The domain specified for destruction does not exist, or the domain is secure and the user does not have the right to look at it. Note that in such a case, the routine does not disclose the existence of the domain.

1.10 String Manipulation Routines

The string routines described in this section facilitate frequently performed operations on traditional 8-bit/character strings. These routines do not operate on any 16-bit (T2) or 32-bit (T4) strings.

While we strive to use the term octet consistently to describe eight bit units of storage, when speaking about strings it is frequently much more natural to speak of *characters*. The term character shall always refer to exactly one octet.

\The run-time libraries group has also been asked to provide routines that act on DDIS-encoded "general strings"; we are currently looking at the possible schedule and level of such support. When it is determined, those routines will also be documented in this section.\

1.10.1 Functional Interface and Description

The following sections describe the various routines associated with string manipulation.

1.10.1.1 Types Used

The following types are used in the interface to the string routines.

```
TYPE

    string$status : status;

    string$match_result :  (
        string$c_match,
        string$c_no_match
        );

    string$comparison_result :  (
        string$c_first_source_less,
        string$c_equal_with_pad,
        string$c_equal,
        string$c_first_source_greater
        );

    string$comparison_flags_type :  (
        string$c_case_blind
        );

    string$comparison_flags : SET [string$comparison_flags_type];

    string$address : POINTER string;
```

1.10.1.2 String Comparison Routine

The routine *string$compare* compares the contents of two strings and returns the result of the comparison in the *comparison_result* argument.

The routine *string$compare* is intended to be equivalent to the current VAX/VMS Run-time Library routines STR$COMPARE, STR$CASE_BLIND_COMPARE, and STR$COMPARE_EQL.

1.10.1.2.1 The *string$compare* Routine

The routine *string$compare* compares two strings for the same contents.

Unless otherwise specified by the *flags* argument, *string$compare* distinguishes between uppercase and lowercase alphabetic characters, uses the DEC Multinational Character Set, and, if the strings are unequal in length, considers the shorter string as blank filled to the length of the longer string.

```
PROCEDURE string$compare (
              IN  first_string : string (*);
              IN  second_string : string (*);
              OUT comparison_result : string$comparison_result;
              IN  flags : string$comparison_flags OPTIONAL;
              ) RETURNS string$status
              LINKAGE
                  REFERENCE  (
                      comparison_result
                      flags,
                      )
                  DESCRIPTOR (
                      first_string,
                      second_string
                      );
```

Parameters:

first_string — First source string.

second_string — Second source string.

comparison_result — The value representing the result of the string comparison. Possible values are as follows:

Return Value	Description
string$c_first_source_less	The first source string is less than the second source string.
string$c_first_source_greater	The first source string is greater than the second source string.
string$c_equal	The source strings are exactly the same, both in length and in contents.
string$c_equal_with_padding	The source strings are equal, but only if the shorter string is padded (blank filled).

flags — The flags used by *string$compare*. Possible values are as follows:

Value	Description
string$c_case_blind	Specifies that case distinctions should be ignored. The default is to distinguish between uppercase and lowercase letters.

Routine *string$compare* returns the unsuccessful status values listed in Table 1–8.

Table 1–8: Status Values Returned from Routine *string$compare*

Status Value	Description
string$_invalid_descriptor	Invalid string descriptor.

Routine *string$compare* raises no conditions.

1.10.1.3 Copy Routine

The routine *string$copy* allows you to copy a string passed by descriptor to another string.

1.10.1.3.1 The *string$copy* Routine

The routine *string$copy* copies a source string to a destination string, where both are passed by descriptor.

Depending on the characteristics of the destination string, the following actions occur:

String Characteristic	Action
Fixed length	Copy source string to destination string. If the source string is shorter than the space available in the destination string, the destination string is blank filled. If the source string is longer than the destination string, it is truncated on the right.
Varying length	Copy source string to destination string up to the maximum destination string length with no padding. If the source is longer than the maximum length of the destination string, it is truncated on the right. The current length field is set to the actual number of octets copied.
Dynamic length	If the area specified by the destination descriptor is large enough to contain the source string, copy the source string and set the new length in the destination descriptor. If the area specified is not large enough, return the previous space allocation (if any) and then dynamically allocate the amount of space needed. Copy the source string and set the new length and address in the destination descriptor.

```
PROCEDURE string$copy (
              IN  source_string : string (*);
              OUT  destination_string : string (*);
              OUT resultant_string_length : integer OPTIONAL;
              ) RETURNS string$status
              LINKAGE
                  REFERENCE (
                      resultant_string_length
                      )
                  DESCRIPTOR (
                      source_string,
                      destination_string
                      );
```

Parameters:

source_string — Source string that *string$copy* copies into the destination string.

destination_string — Destination string into which *string$copy* writes the source string.

resultant_string_length — Number of octets written into the output string, not counting padding in the case of a fixed-length string. If the input string is truncated to fit the output string, *resultant_string_length* is set to the size of the output string. Therefore, *resultant_string_length* can always be used by the calling program to access a valid substring of the output string.

Routine *string$copy* returns the unsuccessful status values listed in Table 1–9.

Table 1–9: Status Values Returned from Routine *string$copy*

Status Value	Description
string$_invalid_descriptor	Invalid string descriptor.
string$insufficient_memory	The routine needed to allocate memory, but was unable to do so.
string$_truncation	String truncation warning. The fixed-length or varying destination string could not contain all the characters.

Routine *string$copy* raises no conditions.

1.10.1.4 Allocate and Deallocate Routines

Routines *string$allocate* and *string$deallocate* allocate and deallocate a dynamic string. These routines are the only allowed method for allocating and deallocating a dynamic string. Simply filling in the length and pointer fields of a dynamic string descriptor can cause serious and unexpected problems with string management.

1.10.1.4.1 The *string$allocate* Routine

The routine *string$allocate* allocates a specified number of octets of dynamic virtual memory to a specified string descriptor. The descriptor must be dynamic.

If the string descriptor already has dynamic memory allocated to it, but the amount is less than *length_to_allocate*, *string$allocate* deallocates that space and allocates new space.

```
PROCEDURE string$allocate (
              IN  length_to_allocate : integer;
              IN OUT  character_string : string (*);
              ) RETURNS string$status
              LINKAGE
                  REFERENCE (
                      length_to_allocate
                      )
                  DESCRIPTOR (
                      character_string
                      );
```

Parameters:

length_to_allocate	Number of octets that *string$allocate* allocates.
character_string	Dynamic string to which *string$allocate* allocates the area. The descriptor is checked to verify it is dynamic.

Routine *string$allocate* returns the unsuccessful status values listed in Table 1–10.

Table 1–10: Status Values Returned from Routine *string$allocate*

Status Value	Description
string$_invalid_descriptor	Invalid string descriptor.
string$insufficient_memory	The routine needed to allocate memory, but was unable to do so.

Routine *string$allocate* raises no conditions.

1.10.1.4.2 The *string$deallocate* Routine

The routine *string$allocate* deallocates the described string space and flags the descriptor as describing no string at all (pointer and length are set to zero).

```
PROCEDURE string$deallocate (
              IN OUT  character_string : string (*);
              ) RETURNS string$status
              LINKAGE
                  DESCRIPTOR (
                      character_string
                      );
```

Parameters:

character_string Dynamic string that *string$deallocate* deallocates. The descriptor is checked to verify it is dynamic.

Routine *string$deallocate* returns the unsuccessful status values listed in Table 1–11.

Table 1–11: Status Values Returned from Routine *string$deallocate*

Status Value	Description
string$_invalid_descriptor	Invalid string descriptor.

Routine *string$deallocate* raises no conditions.

1.10.1.5 Concatenate Strings Routine

The routine *string$concatenate* allows you to concatenate two strings.

The routine *string$concatenate* is intended to be equivalent to the current VAX/VMS Run-time Library routines STR$APPEND, STR$PREFIX and STR$CONCAT.

1.10.1.5.1 The *string$concatenate* Routine

The routine *string$concatenate* concatenates two source strings into a single destination string. Any valid string descriptor may be used.

Two source strings are required as input; if only one string is desired then use *string$copy*. The concatenation procedure copies the first source string then the second source string to the destination string. The maximum length of the concatenated string is implementation specific.

A warning status is returned if one or more input characters were not copied to the destination string.

```
PROCEDURE string$concatenate (
              IN  first_source_string : string (*);
              IN  second_source_string : string (*);
              OUT  destination_string : string (*);
              OUT resultant_string_length : integer OPTIONAL;
              ) RETURNS string$status
              LINKAGE
                  REFERENCE (
                      resultant_string_length
                      )
                  DESCRIPTOR (
                      first_source_string,
                      second_source_string,
                      destination_string
                      );
```

Parameters:

first_source_string	First source string.
second_source_string	Second source string.
destination_string	Destination string into which *string$concatenate* concatenates specified source strings.
resultant_string_length	Number of octets written into the output string, not counting padding in the case of a fixed-length string. If the input string is truncated to fit the output string, *resultant_string_length* is set to the size of the output string. Therefore, *resultant_string_length* can always be used by the calling program to access a valid substring of the output string.

Routine *string$concatenate* returns the unsuccessful status values listed in Table 1–12.

Table 1–12: Status Values Returned from Routine *string$concatenate*

Status Value	Description
string$_invalid_descriptor	Invalid string descriptor.
string$insufficient_memory	The routine needed to allocate memory, but was unable to do so.
string$_truncation	String truncation warning. The fixed-length destination string could not contain all the characters.
string$_string_too_long	One or more source strings or the destination string exceeds the maximum allowable string length; the null string is returned.

Routine *string$concatenate* raises no conditions.

1.10.1.6 Search Routines

Routines *string$find_first_in_set*, *string$find_first_not_in_set*, and *string$find_substring* are routines that search a string.

1.10.1.6.1 The *string$find_first_in_set* Routine

The routine *string$find_first_in_set* compares each character in a string to every character in the specified set of characters. As soon as the first match is found, *string$find_first_in_set* returns the position in the string where the matching character was found in *string_position*. The first character of the *source_string* has position number one. If no match is found or if either *source_string* or *set_of_chars* is of zero length, a zero is returned in *string_position*.

```
PROCEDURE string$find_first_in_set (
              IN  source_string : string (*);
              IN  set_of_chars : string (*);
              OUT string_position : integer;
              ) RETURNS string$status
              LINKAGE
                  REFERENCE (
                      string_position
                      )
                  DESCRIPTOR (
                      source_string,
                      set_of_chars
                      );
```

Parameters:

source_string	String that *string$find_first_in_set* compares to the set of characters, looking for the first match.
set_of_chars	String that is interpreted as a set of characters that *string$find_first_in_set* is searching for in the source string.
string_position	Position in the *source_string* where the first match is found; zero if no match is found.

Routine *string$find_first_in_set* returns the unsuccessful status values listed in Table 1–13.

Table 1–13: Status Values Returned from Routine *string$find_first_in_set*

Status Value	Description
string$_invalid_descriptor	Invalid string descriptor.

Routine *string$find_first_in_set* raises no conditions.

1.10.1.6.2 The *string$find_first_not_in_set* Routine

The routine *string$find_first_not_in_set* searches a string, comparing each character to the characters in a specified set of characters. The string is searched character by character, from left to right. When *string$find_first_not_in_set* finds a character in *source_string* that is not in *set_of_chars*, it stops searching and returns the position of the nonmatching character in *string_position*. The first character of *source_string* has position number one. If all characters in the string match some character in the set of characters, a zero is returned in *string_position*. If *source_string* is of zero length, the value returned in *string_position* is one, because none of the elements in the set of characters will be found in the string. If there are no characters in the set of characters, zero is returned because "nothing" can always be found.

```
PROCEDURE string$find_first_not_in_set (
              IN  source_string : string (*);
              IN  set_of_chars : string (*);
              OUT string_position : integer;
              ) RETURNS string$status
              LINKAGE
                  REFERENCE (
                      string_position
                      )
                  DESCRIPTOR (
                      source_string,
                      set_of_chars
                      );
```

Parameters:

source_string	String that *string$find_first_not_in_set* searches.
set_of_chars	String that is interpreted as a set of characters that *string$find_first_not_in_set* is searching for in *source_string*.
string_position	Position in *source_string* where the first nonmatch is found.

Routine *string$find_first_not_in_set* returns the unsuccessful status values listed in Table 1–14.

Table 1–14: Status Values Returned from Routine *string$find_first_not_in_set*

Status Value	Description
string$_invalid_descriptor	Invalid string descriptor.

Routine *string$find_first_not_in_set* raises no conditions.

1.10.1.6.3 The *string$find_substring* Routine

The routine *string$find_substring* returns the relative position of the first occurrence of a substring in the source string. The value is returned in *string_position*. The relative character positions are numbered one, two, three, and so on. If *start_position* is omitted, the relative starting position used is one, the first character in the source string. Zero indicates that the substring was not found.

If the substring has a length of zero, *string$find_substring* returns the minimum of two values: *start_position* and the length of *source_string* plus one.

If the source string is shorter than the substring or the relative starting position is greater than the source string length, zero is returned, indicating that the substring was not found. If *start_position* is less than one, one is used and a warning is returned to the caller.

```
PROCEDURE string$find_substring (
              IN  source_string : string (*);
              IN  substring : string (*);
              OUT string_position : integer;
              IN  start_position : integer OPTIONAL;
              ) RETURNS string$status
              LINKAGE
                  REFERENCE (
                      string_position,
                      start_position
                      )
                  DESCRIPTOR (
                      source_string,
                      substring
                      );
```

Parameters:

source_string	String that *string$find_substring* searches.
substring	Specified substring for which *string$find_substring* searches in *source_string*.
string_position	Relative position of the first character of the substring. Zero is the value returned if *string$find_substring* did not find the substring.
start_position	Relative position in the source string at which *string$find_substring* begins the search; the value of one is used by default.

Routine *string$find_substring* returns the unsuccessful status values listed in Table 1–15.

Table 1–15: Status Values Returned from Routine *string$find_substring*

Status Value	Description
string$_invalid_descriptor	Invalid string descriptor.
string$_illegal_start_position	The routine completed successfully, except that *start_position* contained a value less than one; the default value of one was used.

Routine *string$find_substring* raises no conditions.

1.10.1.7 Extract and Replace Routines

Routines *string$extract_element*, *string$extract_substring*, *string$extract_substring_length*, and *string$replace_substring* are routines that extract a substring from a string or replace a substring with another substring.

1.10.1.7.1 The *string$extract_element* Routine

The routine *string$extract_element* extracts an element from a string in which the elements are separated by a specified delimiter.

For example, if *source_string* has the value "ABC | DEF | GHI | JKL", *delimiter_string* is a vertical bar (|), and *element_number* is 2, then *string$extract_element* returns the string "GHI".

Once the specified element is located, all the characters in that delimited element are returned. That is, all characters between the *element_number* and *element_number* plus one delimiters are written to *destination_string*. At least *element_number* delimiters must be found. If exactly *element_number* delimiters are found, then all values from the *element_number* delimiter to the end of the string are returned. If *element_number* equals zero and no delimiters are found, the entire input string is returned. If *element_number* is greater than the number of delimiters found, or if the delimiter string is not exactly one character long, a null string is returned.

The routine *string$extract_element* duplicates the functions of the VAX DCL lexical function F$ELEMENT.

```
PROCEDURE string$extract_element (
              IN  source_string : string (*);
              IN  delimiter_string : string (1);
              IN  element_number : integer;
              OUT destination_string : string (*);
              OUT resultant_string_length : integer OPTIONAL;
              ) RETURNS string$status
              LINKAGE
                  REFERENCE (
                      element_number,
                      resultant_string_length
                      )
                  DESCRIPTOR (
                      source_string,
                      delimiter_string,
                      destination_string
                      );
```

Parameters:

source_string
: Source string from which *string$extract_element* extracts the requested delimited substring.

delimiter_string
: Delimiter string used to separate element substrings; this must be exactly one character long.

element_number
: Element number of the delimited element substring to be returned. Zero is used to represent the first delimited element substring, one is used to represent the second, and so forth.

destination_string
: Destination string into which *string$extract_element* copies the selected substring.

resultant_string_length
: Number of octets written into the output string, not counting padding in the case of a fixed-length string. If the input string is truncated to fit the output string, *resultant_string_length* is set to the size of the output string. Therefore, *resultant_string_length* can always be used by the calling program to access a valid substring of the output string.

Routine *string$extract_element* returns the unsuccessful status values listed in Table 1–16.

Table 1–16: Status Values Returned from Routine *string$extract_element*

Status Value	Description
string$_invalid_descriptor	Invalid string descriptor.
string$_invalid_delimiter	The delimiter string is not exactly one character long; a null string is returned.
string$_element_number_outrange	Not enough delimiter characters found to satisfy requested element number; a null string is returned.
string$_truncation	String truncation warning. The fixed-length destination string could not contain all the characters.
string$insufficient_memory	The routine needed to allocate memory, but was unable to do so.

Routine *string$extract_element* raises no conditions.

1.10.1.7.2 The *string$extract_substring* Routine

The routine *string$extract_substring* extracts a substring from a source string and copies that substring into a destination string. It defines the substring by specifying the relative starting and ending positions of the substring in the source string. The source string is unchanged unless it is also the destination string.

If the starting position is less than one, the relative starting position used is one, the first character in the source string. If the starting position is greater than the length of the source string, the null string is returned. If the ending position is greater than the length of the source string, the length of the source string is used.

```
PROCEDURE string$extract_substring (
              IN  source_string : string (*);
              IN  starting_position : integer;
              IN  ending_position : integer;
              OUT destination_string : string (*);
              OUT resultant_string_length : integer OPTIONAL;
              ) RETURNS string$status
              LINKAGE
                  REFERENCE (
                      starting_position,
                      ending_position,
                      resultant_string_length
                      )
                  DESCRIPTOR (
                      source_string,
                      destination_string
                      );
```

Parameters:

source_string — Source string from which *string$extract_substring* extracts the substring that it copies into the destination string.

starting_position — Relative position in the source string at which *string$extract_substring* begins copying the source string.

ending_position — Relative position in the source string at which *string$extract_substring* stops copying the substring.

destination_string — Destination string into which *string$extract_substring* copies the selected substring.

resultant_string_length — Number of octets written into the output string, not counting padding in the case of a fixed-length string. If the input string is truncated to fit the output string, *resultant_string_length* is set to the size of the output string. Therefore, *resultant_string_length* can always be used by the calling program to access a valid substring of the output string.

Routine *string$extract_substring* returns the unsuccessful status values listed in Table 1–17.

Table 1–17: Status Values Returned from Routine *string$extract_substring*

Status Value	Description
string$_invalid_descriptor	Invalid string descriptor.
string$_truncation	String truncation warning. The fixed-length destination string could not contain all the characters.
string$insufficient_memory	The routine needed to allocate memory, but was unable to do so.

Routine *string$extract_substring* raises no conditions.

1.10.1.7.3 The *string$replace_substring* Routine

The routine *string$replace_substring* copies a source string to a destination string, replacing part of the source string with another string. The substring to be replaced is specified by its starting and ending positions.

If the starting position is less than one, one is used. If the ending position is greater than the length of the source string, the length of the source string is used. If the starting position is greater than the ending position, an error is returned and the replacement operation is not performed.

```
PROCEDURE string$replace_substring (
              IN  source_string : string (*);
              IN  replacement_string : string (*);
              IN  starting_position : integer;
              IN  ending_position : integer;
              OUT destination_string : string (*);
              OUT resultant_string_length : integer OPTIONAL;
              ) RETURNS string$status
              LINKAGE
                  REFERENCE (
                      starting_position,
                      ending_position,
                      resultant_string_length
                      )
                  DESCRIPTOR (
                      source_string,
                      replacement_string,
                      destination_string
                      );
```

Parameters:

source_string — Source string to be copied.

replacement_string — Replacement string with which *string$replace_substring* replaces the substring.

starting_position — Position in the source string at which substring replacement begins. The position is relative to the start of the source string.

ending_position — Position in the source string at which substring replacement ends. The position is relative to the start of the source string.

destination_string — Destination string into which *string$replace_substring* writes the new string created when it replaces the substring.

resultant_string_length — Number of octets written into the output string, not counting padding in the case of a fixed-length string. If the input string is truncated to fit the output string, *resultant_string_length* is set to the size of the output string. Therefore, *resultant_string_length* can always be used by the calling program to access a valid substring of the output string.

Routine *string$replace_substring* returns the unsuccessful status values listed in Table 1–18.

Table 1–18: Status Values Returned from Routine *string$replace_substring*

Status Value	Description
string$_invalid_argument	Invalid argument. The starting position is greater than (comes after) the ending position.
string$_invalid_descriptor	Invalid string descriptor.
string$_truncation	String truncation warning. The fixed-length destination string could not contain all the characters.
string$insufficient_memory	The routine needed to allocate memory, but was unable to do so.

Routine *string$replace_substring* raises no conditions.

1.10.1.8 Miscellaneous Routines

Routines *string$analyze_descriptor*, *string$match_wildcard*, *string$translate*, *string$trim*, and *string$upcase* perform the following operations:

- Analyze string descriptors
- Match wildcard specifications
- Translate matched characters
- Trim trailing spaces and tabs
- Convert strings to uppercase characters

1.10.1.8.1 The *string$analyze_descriptor* Routine

The routine *string$analyze_descriptor* takes as input a string passed by descriptor and extracts the length of the string and the address at which the string is stored for all string categories.

```
PROCEDURE string$analyze_descriptor (
              IN  input_string : string (*);
              OUT string_length : integer;
              OUT string_address : string$address;
              ) RETURNS string$status
              LINKAGE
                  REFERENCE (
                      string_length,
                      string_address
                      )
                  DESCRIPTOR (
                      input_string
                      );
```

Parameters:

input_string	Input string from whose descriptor *string$analyze_descriptor* extracts the length of the string and the address at which the string starts.
string_length	Length of the string.
string_address	Address of the string.

Routine *string$analyze_descriptor* returns the unsuccessful status values listed in Table 1–19.

Table 1–19: Status Values Returned from Routine *string$analyze_descriptor*

Status Value	Description
string$_invalid_descriptor	Invalid string descriptor.

Routine *string$analyze_descriptor* raises no conditions.

1.10.1.8.2 The *string$match_wildcard* Routine

The routine *string$match_wildcard* translates wildcard characters and searches the candidate string to determine if it matches the pattern string. The pattern string may contain either one or both of the two wildcard characters; the default characters are the asterisk (*), and the percent sign (%). The asterisk character maps to zero or more characters. The percent character maps to exactly one character.

The arguments *matches_one* and *matches_any* allow the caller to specify the wildcard characters to be used instead of the default values, the percent sign and the asterisk, respectively. The two wildcard characters may be used only as wildcards. If the candidate string contains an asterisk or percent sign (or a user-specified match character), the candidate string will not match a wildcard pattern, because the wildcard characters are never translated literally.

```
PROCEDURE string$match_wildcard (
              IN  candidate_string : string (*);
              IN  pattern_string : string (*);
              OUT match_result : string$match_result;
              IN  matches_one : string (1) OPTIONAL;
              IN  matches_any : string (1) OPTIONAL;
              ) RETURNS string$status
              LINKAGE
                  REFERENCE (
                      match_result
                      )
                  DESCRIPTOR (
                      candidate_string,
                      pattern_string,
                      matches_one,
                      matches_any
                      );
```

Parameters:

candidate_string String that is compared to the pattern string.

pattern_string String containing wildcard characters. The wildcards in the pattern string are translated when *string$match_wildcard* searches the candidate string to determine if it matches the pattern string.

match_result The result of the match attempt. Possible values are as follows:

Value	Description
string$c_match	Indicates that the candidate string matched the pattern string.
string$c_no_match	Indicates that the candidate string did not match the pattern string.

matches_one The user-specified wildcard character that is mapped to exactly one character; the percent character is the default.

matches_any The user-specified wildcard character that is mapped to zero or more characters; the asterisk character is the default.

Routine *string$match_wildcard* returns the unsuccessful status values listed in Table 1–20.

Table 1–20: Status Values Returned from Routine *string$match_wildcard*

Status Value	Description
string$_invalid_wildcard_char	The wildcard character supplied is invalid. The wildcard character is not exactly one character long or the wildcard character is an illegal character.
string$_invalid_descriptor	Invalid string descriptor.

Routine *string$match_wildcard* raises no conditions.

1.10.1.8.3 The *string$translate* Routine

The routine *string$translate* successively compares each character in a source string to all characters in a match string. If a source character matches any of the characters in the match string, *string$translate* moves a character from the translate string to the destination string. Otherwise, *string$translate* moves the character from the source string to the destination string.

The character taken from the translate string has the same relative position as the matching character had in the match string. When a character appears more than once in the match string, the position of the leftmost occurrence of the multiply defined character is used to select the translate string character. If the translate string is shorter than the match string and the matched character position is greater than the translate string length, the destination character is a space.

```
PROCEDURE string$translate (
              IN  source_string : string (*);
              IN  translation_string : string (*);
              IN  match_string : string (*);
              OUT destination_string : string (*);
              OUT resultant_string_length : integer OPTIONAL;
              ) RETURNS string$status
              LINKAGE
                  REFERENCE (
                      resultant_string_length
                      )
                  DESCRIPTOR (
                      source_string,
                      translation_string,
                      match_string,
                      destination_string
                      );
```

Parameters:

source_string	Source string.
translation_string	Translation string.
match_string	Match string.
destination_string	Destination string.
resultant_string_length	Number of octets written into the output string, not counting padding in the case of a fixed-length string. If the input string is truncated to fit the output string, *resultant_string_length* is set to the size of the output string. Therefore, *resultant_string_length* can always be used by the calling program to access a valid substring of the output string.

Routine *string$translate* returns the unsuccessful status values listed in Table 1–21.

Table 1–21: Status Values Returned from Routine *string$translate*

Status Value	Description
string$_invalid_argument	Invalid argument.
string$_invalid_descriptor	Invalid string descriptor.
string$_truncation	String truncation warning. The fixed-length destination string could not contain all the characters.
string$insufficient_memory	The routine needed to allocate memory, but was unable to do so.

Routine *string$translate* raises no conditions.

1.10.1.8.4 The *string$trim* Routine

The routine *string$trim* copies a source string to a destination string and deletes the trailing space and tab characters.

```
PROCEDURE string$trim (
              IN  source_string : string (*);
              OUT destination_string : string (*);
              OUT resultant_string_length : integer OPTIONAL;
              ) RETURNS string$status
              LINKAGE
                  REFERENCE (
                      resultant_string_length
                      )
                  DESCRIPTOR (
                      source_string,
                      destination_string
                      );
```

Parameters:

source_string	Source string which *string$trim* trims and then copies into the destination string.
destination_string	Destination string into which *string$trim* copies the trimmed string.
resultant_string_length	Number of octets written into the output string, not counting padding in the case of a fixed-length string. If the input string is truncated to fit the output string, *resultant_string_length* is set to the size of the output string. Therefore, *resultant_string_length* can always be used by the calling program to access a valid substring of the output string.

Routine *string$trim* returns the unsuccessful status values listed in Table 1–22.

Table 1–22: Status Values Returned from Routine *string$trim*

Status Value	Description
string$_invalid_argument	Invalid argument.
string$_invalid_descriptor	Invalid string descriptor.
string$_truncation	String truncation warning. The fixed-length destination string could not contain all the characters.
string$insufficient_memory	The routine needed to allocate memory, but was unable to do so.

Routine *string$trim* raises no conditions.

1.10.1.8.5 The *string$upcase* Routine

The routine *string$upcase* converts characters in a source string to uppercase and writes the converted characters into the destination string. The routine converts all characters in the multinational character set.

```
PROCEDURE string$upcase (
              IN  source_string : string (*);
              OUT destination_string : string (*);
              OUT resultant_string_length : integer OPTIONAL;
              ) RETURNS string$status
              LINKAGE
                  REFERENCE (
                      resultant_string_length
                      )
                  DESCRIPTOR (
                      source_string,
                      destination_string
                      );
```

Parameters:

source_string	Source string that *string$upcase* converts to uppercase.
destination_string	Destination string into which *string$upcase* writes the string it has converted to uppercase.
resultant_string_length	Number of octets written into the output string, not counting padding in the case of a fixed-length string. If the input string is truncated to fit the output string, *resultant_string_length* is set to the size of the output string. Therefore, *resultant_string_length* can always be used by the calling program to access a valid substring of the output string.

Routine *string$upcase* returns the unsuccessful status values listed in Table 1–23.

Table 1–23: Status Values Returned from Routine *string$upcase*

Status Value	Description
string$_invalid_descriptor	Invalid string descriptor.
string$_truncation	String truncation warning. The fixed-length destination string could not contain all the characters.
string$insufficient_memory	The routine needed to allocate memory, but was unable to do so.

Routine *string$upcase* raises no conditions.

GLOSSARY

AIA: Application Integration Architecture

ARUS: Application Run-Time Utility Services

CMA: Common Multithread Architecture

PSM: Print System Model

Mica Working Design Document
Application Run-Time Utility Services

Revision 0.7

14–April–1988

Section 4 of 4

Authors:

Al Simons
John Nogrady

Issued by:

Al Simons

digital™

TABLE OF CONTENTS

TABLES

Revision History

Date	Revision Number	Author	Summary of Changes
19-November-1987	0.1	All	First draft
27-December-1987	0.2	Connors	Incorporated review comments. Split original chapter entitled "Applications Run-Time Library" into two chapters: "Application Run-Time Utility Services" and "Miscellaneous Run-Time Library Routines."
8-February-1988	0.3	Simons, Nogrady	Prepare sections on memory management and date/time manipulation for primary review.
4-March-1988	0.4	Simons	Incorporate primary review comments on the above sections.
14-March-1988	0.5	Simons, Nogrady	Prepare sections on condition handling and conversions for primary review.
29-March-1988	0.6	Simons, Nogrady	Prepare sections on environment attributes and strings.
08-April-1988	0.7	Simons, Nogrady, Barker	Prepare sections on process information and text formatting.

CHAPTER 1

APPLICATION RUN-TIME UTILITY SERVICES

1.1 Overview

This section has been intentionally removed from this review copy.

1.2 ARUS Routine Design Philosophy

This section has been intentionally removed from this review copy.

1.3 Error Conditions and Status Returns

This section has been intentionally removed from this review copy.

1.4 Memory Allocation and Deallocation Routines

This section has been intentionally removed from this review copy.

1.5 International Date and Time Routines

This section has been intentionally removed from this review copy.

1.6 General Internationalization Routines

This section has been intentionally removed from this review copy.

1.7 Condition Handling Routines

This section has been intentionally removed from this review copy.

1.8 Data Conversion Routines

This section has been intentionally removed from this review copy.

1.9 Environment Attribute Routines

This section has been intentionally removed from this review copy.

1.10 String Manipulation Routines

This section has been intentionally removed from this review copy.

1.11 Process Information Routines

This section describes the ARUS routines used to obtain process information. The only process information included in this first pass of the ARUS interface definition is CPU time consumed by the process or thread.

1.11.1 Functional Interface and Description

The following sections describe the routines associated with thread and process information.

1.11.1.1 Types Used

The following types are used in the interface to the process information routines.

```
TYPE

    arus$status : status;

    arus$context : POINTER anytype;        /* hidden */

    arus$cpu_mode : (
          arus$c_process_mode,
          arus$c_thread_mode
          );

    !
    ! The following types and structures are defined by the corporate
    ! time representation standard.
    !
    arus$timevalue : large_integer SIZE (QUADWORD);

    arus$inaccuracy : large_integer[0..2**48-1] SIZE (BYTE,6);

    arus$time_diff_factor : integer[-720..780] SIZE (BIT,12);

    arus$version : integer SIZE (BIT,4);

    arus$binary_relative_time :
        RECORD
            time     : arus$timevalue;
            inacc    : arus$inaccuracy;
            reserved : arus$time_diff_factor = 0;
            vers     : arus$version = 1;
            LAYOUT
                time;
                inacc;
                reserved;     ! must be 0
                vers;         ! must be 1
            END LAYOUT
        END RECORD;
    !
    ! End of types and structures defined by the corporate time
    ! representation standard.
    !
```

1.11.1.2 Process Information Initialize and Free Routines

The routines *arus$init_process_information* and *arus$free_process_information* are used to allocate or deallocate a dynamic block of storage which is used to store the CPU time for the current thread or process.

The *flags* argument indicates the mode, either thread or process, for which the CPU time is stored. If the target system does not support threads, the *flags* argument will be ignored and the default, process mode, will be used.

If the stored CPU time is for a thread, the calls to *arus$get_cpu_time* and *arus$get_cpu_time_relative* must be executed from the same thread.

1.11.1.2.1 The *arus$init_process_information* Routine

The routine *arus$init_process_information* creates a dynamic control block for the specified mode in the *flags* argument, and returns the pointer to the block in *context*.

```
PROCEDURE arus$init_process_information (
              IN OUT context : arus$context;
              IN flags : arus$cpu_mode OPTIONAL;
              ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      context
                      flags
                      );
```

Parameters:

context

Context variable required by all process information routines to refer to the control block. The control block is allocated in one of two ways:

- If *context* is zero, a control block is allocated in dynamic heap storage and the pointer to the block is returned in *context*.
- If *context* is nonzero, it is considered to be the pointer to a control block previously allocated by a call to *arus$init_process_information*. If so, the control block is reused.

flags

Indicates the mode of the CPU time which is to be stored; it is either a CPU time for the current thread or CPU time for the current process. The default is CPU time for the current process.

Routine *arus$init_process_information* returns the unsuccessful status values listed in Table 1–1.

Table 1–1: Status Values Returned from Routine *arus$init_process_information*

Status Value	Description
arus$_invalid_context	Invalid argument; *context* is nonzero and the block to which it refers was not initialized on a previous call to *arus$init_process_information*.
arus$insufficient_memory	The routine needed to allocate memory, but was unable to do so.

Routine *arus$init_process_information* raises no conditions.

1.11.1.2.2 The *arus$free_process_information* Routine

The routine *arus$free_process_information* frees a block of storage previously allocated by *arus$init_process_information*. If the block of storage was not allocated by *arus$init_process_information*, *arus$free_process_information* returns with an error. If the routine completes successfully, *arus$init_process_information* sets *context* to zero.

```
PROCEDURE arus$free_process_information (
              IN OUT context : arus$context;
              ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      context
                      );
```

Parameters:

context — Context variable. Pointer to a block of storage containing the value returned by a previous call to *arus$init_process_information*; this is the storage that *arus$free_process_information* deallocates.

Routine *arus$free_process_information* returns the unsuccessful status values listed in Table 1–2.

Table 1–2: Status Values Returned from Routine *arus$free_process_information*

Status Value	Description
arus$_invalid_context	Invalid argument; *context* did not point to a valid timer block.

Routine *arus$free_process_information* raises no conditions.

1.11.1.3 Calculation of Elapsed CPU Time

The routines *arus$get_cpu_time* and *arus$get_cpu_time_relative* calculate and return the elapsed CPU time since the last call to *arus$init_process_information*, for the current thread or process. If *context* refers to a control block for a thread, the call to the routines *arus$get_cpu_time* and *arus$get_cpu_time_relative* must be from the same thread.

The routines *arus$get_cpu_time* and *arus$get_cpu_time_relative* returns the elapsed CPU time as a longword (10-millisecond increments) and a binary relative time format, respectively.

1.11.1.3.1 The *arus$get_cpu_time* Routine

The routine *arus$get_cpu_time* calculates the elapsed CPU time since the last call to *arus$init_process_information*. The calculated value is returned as an integer in 10-millisecond increments.

```
PROCEDURE arus$get_cpu_time (
              IN  context : arus$context;
              OUT cpu_time : integer;
              ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      context,
                      cpu_time
                      );
```

Parameters:

context — Context variable initialized by *arus$init_process_information*.

cpu_time — The elapsed CPU time returned in 10 millisecond increments.

Routine *arus$get_cpu_time* returns the unsuccessful status values listed in Table 1–3.

Table 1–3: Status Values Returned from Routine *arus$get_cpu_time*

Status Value	Description
arus$_invalid_context	Invalid argument; *context* did not point to a valid control block.
arus$_illegal_thread	The *context* argument referred to a control block for a thread other than the current thread of execution.

Routine *arus$get_cpu_time* raises no conditions.

1.11.1.3.2 The *arus$get_cpu_time_relative* Routine

The routine *arus$get_cpu_time_relative* calculates the elapsed CPU time since the last call to *arus$init_process_information*. The calculated value is returned as a binary relative time.

```
PROCEDURE arus$get_cpu_time_relative (
              IN  context : arus$context;
              OUT cpu_time : arus$binary_relative_time
              ) RETURNS arus$status
              LINKAGE
                  REFERENCE (
                      context
                      cpu_time
                      );
```

Parameters:

context	Context variable initialized by *arus$init_process_information*.
cpu_time	The elapsed CPU time returned as a binary relative time.

Routine *arus$get_cpu_time_relative* returns the unsuccessful status values listed in Table 1–4.

Table 1–4: Status Values Returned from Routine *arus$get_cpu_time_relative*

Status Value	Description
arus$_invalid_context	Invalid argument; *context* did not point to a valid control block.
arus$_illegal_thread	The *context* argument referred to a control block for a thread other than the current thread of execution.

Routine *arus$get_cpu_time_relative* raises no conditions.

1.12 Text Formatting

The ARUS library provides routines for formatting text. These routines are similar to the VAX/VMS system service SYS$FAO and the C library routine *fprint*. The ARUS routines embody several improvements over their predecessors, however. Those improvements are as follows:

- The new routines support pluralization much more elegantly, solving the $FAO "!S" problems when converting messages to other languages.
- The new routines support very powerful IF and CASE directives, allowing much more flexible formatting.
- The data type information for the text to be formatted is associated with the data, rather than with the formatting string. This allows modularized routines to capture data for later formatting, because the interim routines have all the necessary information about the data.

1.12.1 Formatting Directives

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

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

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

Table 1–5 describes each formatting directive. In these examples, the following syntax notation is used:

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

  These bit forms are only allowed with parameters of type *arus$c_byte_data*, *arus$c_word_data*, *arus$c_longword_data*, and *arus$c_quadword* data. See the PRISM Calling Standard for a description of parameter data types.

  \This is certainly a poor solution to the problem of extracting bits. Ideally, field names should be used, however, this method provides a usable way to do this, if necessary.\
- "V" is used to represent either a parameter number or a numeric or string constant. Parameter numbers are specified by a number only. Numeric constants are specified by preceeding the value with a "#" sign. String constants are enclosed in double quotation marks ("string"). The *length* directive returns a value that can be used wherever a numeric constant is allowed.

To better facilitate use of repeated directives, the formatting routine maintains a special internal parameter number which may be set, incremented, and decremented. This internal parameter number is accessed as if it were parameter number 0 (zero). Upon entry to either the *arus$format_single_string* or *arus$multiple_strings* routine, its value is set to 1. When the internal parameter number (0) is used in an inserting formatting directive, such as %left% or %binary%, its value is used to refer to a parameter number. For example, if the directive %left(0)% is specified and the value of the internal parameter is 5, the fifth parameter would be formatted as a left-justified string. When the internal parameter number is used in a comparison or controlling formatting directive, such as %if% or %repeat%, its value is used directly.

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

Table 1–5: Formatting Directives

Directive[1]	Description
decimal(N[:W[Z]][,"RT"])	The parameter is formatted in decimal. For floating-point parameters, the width may optionally be specified as "W.P" where "P" specifies the precision. The field is zero filled if "Z" is present. Normally, floating-point parameters are formatted with the user's preferred radix point and thousands separator character. This may be optionally overridden by specifying "RT" as part of the directive, where "R" is the radix point character and "T" is the thousands separator character. Example: `%decimal(5:10.2,",.")%` This specifies that the fifth parameter is to be formatted in decimal in a field of width 10 and a precision of 2. Additionally, the "," character is used to specify the radix point and the "." character is used as the thousands separator. Decimal is the only supported directive for formatting floating-point values. For floating point parameters, the optional brackets used to select certain bits of a parameter are not allowed.
hex(N[:W[Z]])	The parameter is formatted in hexadecimal. The field is zero filled if "Z" is present. Note that no leading characters indicating hexadecimal formatting are inserted. Example: `%hex(2)%` This specifies that the second parameter is to be formatted in hexadecimal in a field just large enough to hold the entire value.
octal(N[:W[Z]])	The parameter is formatted in octal. The field is zero filled if "Z" is present. Note that no leading characters indicating octal formatting are inserted.
binary(N[:W[Z]])	The parameter is formatted in binary. The field is zero filled if "Z" is present. Note that no leading characters indicating binary formatting are inserted.
date([N][:F][,ALIGN])	The specified parameter is formatted in date format. If no parameter is supplied, the current system date is formatted.[2]Normally, the date is formatted using the user's preferred date format. If ":F" is specified, the date is formatted using date format "F". If "ALIGN" is specified, the formatting routine forces the date field width to be the maximum produced by the specified date format. This is useful when text is formatted in columns.
time([N][:F][,ALIGN])	The specified parameter is formatted in time format. If no parameter is supplied, the current system time is formatted.[2]Normally, the time is formatted using the user's preferred time format. If ":F" is specified, the time is formatted using time format "F". If "ALIGN" is specified, the formatting routine forces the time field width to be the maximum produced by the specified time format. This is useful when text is formatted in columns.
date_time([N][:F:G][,ALIGN])	The specified parameter is formatted in date/time format. If no parameter is supplied, the current system date and time are formatted.[2]Normally, the date and time are formatted using the user's preferred date and time formats. If ":F:G" is specified, the date is formatted using date format "F" and the time is formatted using time format "G". If "ALIGN" is specified, the formatting routine forces the date/time field width to be the maximum produced by the specified date/time formats. This is useful when text is formatted in columns.

[1]Table 1–6 presents the rules for which parameters and constants are allowed with which directives.

[2]The *arus$format_single_string* and *arus$format_multiple_strings* routines use the ARUS date/time formatting services to format the date and time. These services provide full internationalization capabilities as well as support for multiple date /time formats.

Table 1–5 (Cont.): Formatting Directives

Directive[1]	Description
right(N[:W])	The string parameter is formatted in a right-justified field "W" characters wide. If the string parameter is longer than "W", it is not truncated.
left(N[:W])	The string parameter is formatted in a left-justified field "W" characters wide. If the string parameter is longer than "W", it is not truncated.
center(N[:W])	The string parameter is centered in a field "W" characters wide. If the string parameter is longer than "W", it is not truncated.
length(directive)	The specified directive is evaluated and the formatted string is returned as an integer constant. Note that this is the only directive that does not insert characters into the resultant string. This directive may be used wherever a numeric constant is allowed.
plural(N[,"zero string"[,"singular string"[,"2 string",...,"n string"]]])	This directive is used to control pluralization. The directive allows specification of different strings to be inserted into the resultant string for different values of the specified parameter. The first string corresponds to a value of zero, the second string to a value of one, the third to a value of two, and so on. The final string is used for values greater than or equal to "n". If only the parameter number is supplied, "" is used when the value of the parameter is one and "s" is used when the value of the parameter is zero or more than one.
system(item[,item...])	Insert the specified system item(s) into the resultant string at this location. System items are typically specific to a particular operating system and should be avoided in cases where format strings are used across multiple systems. Supported system items on Mica are: • *object(N)*—the parameter is the identifier of an object whose name is to be translated and inserted into the resultant string. If no name exists for the object, the identifier is output in hexadecimal.
control(item[,item...])	Insert the specified format control item(s) into the resultant string at this location. Supported format items are: • tab—insert tab character • new_line—new line indicator; for the *arus$format_single_string* routine, this inserts carriage_return and line_feed characters; for the *arus$format_multiple_strings* routine, this advances to the next output string in the resultant string array • form_feed—insert form_feed character
character(V,c)	Insert the character "c" in the resultant string *n* times, where *n* is the value of the specified parameter or the specified constant.
set(V)	Set the internal parameter value to the value of the specified parameter or constant. This is normally used prior to the repeat directive.
increment([V])	Increment the internal parameter value by the value of the specified parameter or the specified constant. If "V" is not specified, the constant value 1 is assumed.
decrement([V])	Decrement the internal parameter value by the value of the specified parameter or the specified constant. If "V" is not specified, the constant value 1 is assumed.

[1]Table 1–6 presents the rules for which parameters and constants are allowed with which directives.

Table 1–5 (Cont.): Formatting Directives

Directive[1]	Description
repeat(V,directive[,directive...])	Repeat the specified list of directives. The number of times to repeat may be specified by the value of a parameter or by constant value. The repeat directive in conjunction with the internal parameter value provides a short way to specify output of a list of parameters.
text(V)	Output the specified text string. This is useful in conjunction with the repeat directive.
if(V{op}V,directive[,directive])	Execute the first directive if the operation specified by {op} is TRUE; otherwise, execute the second directive, if specified. Operations compare the value of the first specified parameter or constant with the second parameter or constant. The following comparison operations are supported:

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

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

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

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

[1]Table 1–6 presents the rules for which parameters and constants are allowed with which directives.

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

1.12.1.1 Formatting Directive and Data Type Compatibility

\The current version of the PRISM Calling Standard does not include a list of supported data types for parameters being formatted by *arus$format_single_string* and *arus$format_multiple_strings*. Following is a list of data types required for these routines:

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

\

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

Table 1–6: Data Type Rules for Formatting Directives

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

1.12.2 Functional Interface and Description

The following sections describe the various routines associated with text formatting.

1.12.2.1 Types Used

```
TYPE
    arus$status : status;

    !
    ! The possible values for field argument_datatype are to be
    ! determined later, in conjunction with the PRISM calling standard.
    !
    arus$condition_argument : RECORD
        argument_datatype : arus$condition_arg_type;
        argument_extent : longword;
        argument : POINTER anytype;
    END RECORD;

    arus$condition_array (n): ARRAY [1..n] OF arus$condition_argument;
```

1.12.2.2 Text Formatting Routines

Two routines are provided to support the text formatting directives presented previously, *arus$format_single_string* for formatting a single string, and *arus$format_multiple_strings* for formatting multiple strings. These two routines are presented in the following subsections.

1.12.2.2.1 The *arus$format_single_string* Routine

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

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

Parameters:

Parameter	Description
source_string	Supplies the source string containing text and formatting directives.
resultant_string	Returns the resultant formatted string.
resultant_length	Returns the number of characters used in the resultant string.
parameters	Optionally supplies an array of parameters to be formatted into the resultant string.
language	An optional string that supplies a language name to override the current default language. Language is used to determine language-dependent formats for parameters formatted into the message string.

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

1.12.2.2.2 The *arus$format_multiple_strings* Routine

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

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

Parameters:

Parameter	Description
source_string	Supplies the source string containing text and formatting directives.
array_size	Supplies the number of strings in the *resultant_string* array.
string_size	Supplies the length of each string in the *resultant_string* array.
resultant_string	Returns the resultant formatted strings.
string_lengths	Returns the length of each of the resultant formatted strings.
strings_used	Returns the number of strings in the array that were actually used in the formatting.
parameters	Optionally supplies an array of parameters to be formatted into the resultant string.
language	An optional string that supplies a language name to override the current default language. Language is used to determine language-dependent formats for parameters formatted into the message string.

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

1.13 Command Language Interpreter Interface Routines

TBS

1.14 Table-Driven Parsing Routines

TBS

1.15 High-Level Math Routines

TBS

GLOSSARY

AIA: Application Integration Architecture

ARUS: Application Run-Time Utility Services

CMA: Common Multithread Architecture

PSM: Print System Model