SPARCompiler Ada
Runtime System Guide

SunSoft
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No.: 802-3642-10
Revision A November, 1995

2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.

Contents

"Now here, you see, it takes all the running you can do, to keep in the same place.
If you want to get somewhere else, you must run at least twice as fast as that."
Lewis Carroll

Overview of the Runtime System 1

There are two implementations of the SPARCompiler Ada (SC Ada) runtime in this product, the VADS threaded runtime and the Solaris multithreaded runtime. These runtime systems deliver significantly different functionality. In fact, they are so different that each runtime system requires its own version of the Ada `standard` library as well as its own version of the SunPOro-supplied libraries and configuration directories.

Before compiling your application you must choose which of the runtime systems you are going to use. If you decide to switch runtime systems after you have built your application, you must recompile your application using the other set of SunPro-supplied libraries.

The two versions of the SunPro-supplied libraries are in the directories `self` and `self_thr` in `SCAda_location` (`SCAda_location` indicates the directory where SC Ada is installed from the distribution media). Your application must be built with SunPro-supplied Ada libraries exclusively from one of these two directories.

The following sections present the structure of the SC Ada runtime system, an overview of the runtime system components, the differences between the two runtime systems delivered with this product, and some miscellaneour information.

References

SPARCompiler Ada Multithreading Ada Application

1

1.1 The SC Ada Runtime Structure

The SC Ada runtime system is fast and portable. Portability has been achieved by using a layered model for constructing the runtime system, with a microkernel as the innermost layer. Figure 1-1 shows the layers of the runtime system.

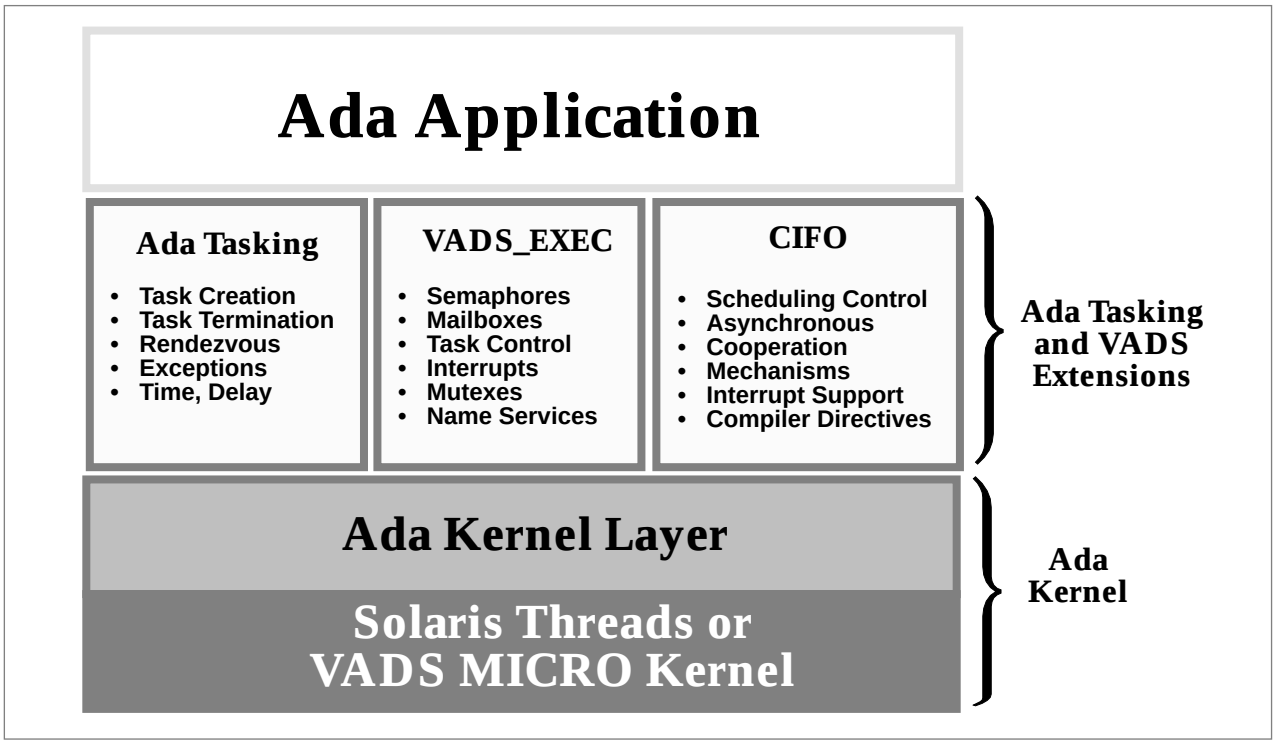

Figure 1-1 Threaded Ada Runtime Layers

1.1.1 The Ada Kernel

The layered model supports the use of different microkernels under the SC Ada runtime, while providing an identical set of services to the Ada application. A microkernel, called VADS MICRO, has been implemented and is supported in embedded systems as well as on workstation hosts. The runtime system has also been ported on top of other kernels, in particular the Solaris Multithread architecture, which we abbreviate to Solaris MT.

The two runtime systems that are part of SC Ada are different because of these two different underlying microkernels, VADS MICRO and Solaris Threads. Later in this chapter there is a section called VADS Threaded vs. Solaris MT Runtime. That section discusses some of the differences between the VADS MICRO microkernel and the Solaris Threads microkernel.

On top of the microkernel is the Ada kernel layer, which maps the microkernel features into the interface required by the Ada tasking and VADS Extensions layer. When the runtime system is ported onto a new microkernel, the main job of porting is writing a new Ada kernel layer for the new microkernel. The microkernel and the Ada kernel layer together are known as the Ada Kernel.

With the layered design of the runtime system, much of the Ada tasking and rendezvous semantics has been moved to a level higher than the kernel, closer to the application. As a result, many of the common operations are handled without going into the Ada kernel at all and the total throughput of the system is enhanced.

The Ada kernel is discussed in the section, Ada Kernel. We recommend that you do not call Ada kernel services directly. These services are described for completeness. There is one exception: there are some object attributes such as tasking attributes that the application can set. The Ada Kernel interface provides routines to initialize these attributes. These routines are supported on all SC Ada implementations, even though the underlying representation of the attributes may vary from microkernel to microkernel.

References

“Ada Kernel Layer” on page 2-1

1.1.2 Ada Tasking and VADS Extensions

This layer of the SC Ada runtime system is common to all implementations of SC Ada although some of the extensions shown are part of products. Each box in the Ada Tasking and VADS Extensions layer represents a set of services that are directly available to the application program. These services are implemented using the interface provided by the Ada Kernel Layer. The boxes are separated to indicate that these services are mostly independent.

The Ada Tasking services differ from the rest of the services in this layer because the Ada Tasking services are called implicitly by the code generated by the SC Ada compiler. In other words, although your Ada tasking application calls the functions in Ada Tasking, the calls are not visible in your source code. You can see the calls if you disassemble certain parts of your program in the debugger. An overview of this interface is given in the Tasking section of the "Runtime Systems Topics" chapter. Do not put calls to any of the routines in Ada Tasking into your source code.

The *VADS EXEC* functions are documented in the "Ada Runtime Services" chapter. The packages in the file `v_vads_exec.a` in the `vads_exec` library provide the user's interface to the following VADS EXEC services:

- Signal Handling
- Mailboxes
- Memory Management
- Semaphores
- Tasking Extensions
- Stack Operations
- Named Objects
- Mutexes / Condition Variables

References

Chapter 4, "Ada Runtime Services

"Tasking" on page 2-63

1.2 How the VADS Threaded Runtime Works

The VADS Threaded runtime system implements Ada tasking inside a single UNIX process. The VADS MICRO kernel supports a very lightweight thread, meaning that the state of a thread is very small and switching between threads is quite fast.

An Ada program using the VADS Threaded runtime system executes as a UNIX process. When the process begins, its first instruction is in `V_START_PROGRAM` which finishes by calling the VADS MICRO Kernel entry point `TS_INITIALIZE`. This kernel entry point (among other functions) creates the task to correspond to the main program and the idle task. After initialization, the user program is resumed in `V_START_PROGRAM_CONTINUE`.

As the application executes, it can create more tasks that can interact. If time slicing is enabled, task execution is preempted by `SIGALRM` signals and VADS MICRO executes tasks of equal priority in a round-robin fashion. All this tasking is taking place inside one UNIX process; the OS is not aware it is going on.

References

`V_START_PROGRAM`, *SPARCompiler Ada Programmer's Guide*

1.3 Debugging and the Runtime System

The SC Ada debugger is aware of the tasking supported in the runtime system, regardless of the microkernel being used. The debugger's `lt` command displays all the tasks in the program with appropriate status. For the Solaris MT runtime, `lt` shows the task-to-thread mapping.

In addition to listing the tasks, the debugger allows you to select each of the tasks and move up and down on the call stack of that task to examine its current execution state, including the values of variables and registers.

References

`lt` command, *SPARCompiler Ada Reference Guide*

`lu` command, *SPARCompiler Ada Reference Guide*

`task` command, *SPARCompiler Ada Reference Guide*

1.4 VADS Threaded vs. Solaris MT Runtime

The two Ada runtime systems supplied with this product are called the VADS Threaded and the Solaris MT runtime systems. The VADS Threaded runtime system is based on the VADS MICRO microkernel. The Solaris MT runtime is based on Solaris Threads as a kernel.

You choose which runtime system you want for your application when you make your application Ada libraries with a.mklib. The Solaris MT runtime is indicated by SCAda_location/self/standard. This runtime has the Solaris Threads as its microkernel. The VADS Threaded runtime is in SCAda_location/self/standard and it is based on the VADS MICRO microkernel.

A complete description of the functions provided by the Solaris MT runtime system is provided in the manual titled Multithreaded Ada.

References

`a.info`, *SPARCompiler Ada Reference Guide*

`a.mklib`, *SPARCompiler Ada Reference Guide*

Getting Started, *SPARCompiler Ada User's Guide*

LINK Directive Names, *SPARCompiler Ada Reference Guide*

SPARCompiler Ada Multithreading Ada Application

1.4.1 Concurrency

One primary difference between the Solaris MT and VADS Threaded runtime systems is *concurrency***.** Concurrency (or concurrency level) measures the number of Ada tasks in your program that the operating system will run at the same time. For the VADS Threaded runtime, there is no operating system concurrency, so the concurrency level is one -- when one task is doing something, all other tasks are stopped. As a consequence, when an Ada task blocks performing an operating system function (e.g., reading a file, opening a socket), the entire Ada program is blocked.

By contrast, the Solaris MT runtime lets you configure the concurrency of your program. For example, if your application has three tasks that do I/O and four tasks that only do computations, you may want a concurrency level of seven.

Or you may want a concurrency of four, three to cover the tasks doing I/O and one more to make sure that some computation is getting done even if all three I/O tasks are blocked.

1.4.2 Multithreaded Ada

If your host computer has symmetric multiprocessors, the operating system concurrency level translates to true concurrency: multiple Ada tasks can be executing at the same time. The Solaris MT Ada runtime has been designed so that your application can exploit symmetric multiprocessors. If your application contains functions that run in parallel, then you should use the Solaris MT runtime system and gain the performance benefits of parallel processing. For a large class of applications containing parallelism, the performance improvements that can be realized by taking advantage of multiple processors can be truly breathtaking!

The VADS Threaded runtime system does not exploit multiple processors.

1.4.3 When to Use the VADS Threaded Runtime

If an Ada application does not contain tasking, it will not benefit from the concurrency of the Solaris MT runtime. Simple programs that read, process, and write information, as well as applications containing no inherent parallelism fall into this category.

It is best to use the VADS Threaded runtime if your program contains no parallelism. The VADS Threaded runtime has been optimized to provide the minimum overhead for an Ada program that doesn't require concurrency. The Solaris MT runtime can also be configured to provide no concurrency. If you are planning to add concurrency to you application at a later time, you may want to start with Solaris Threads, just to maintain a consistent development and debugging environment throughout the application's lifecycle.

1.4.4 Parallel Programming is Tricky

Parallel programming in the context of concurrency is difficult. An Ada program that works with the VADS Threaded runtime may not work in the Solaris MT runtime with a concurrency level greater than one. Here is one pitfall to avoid:

Task priority cannot be used to serialize execution

In the VADS Threaded runtime, raising the priority of a task is a way to guarantee a certain pattern of execution, for example to make sure that a certain operation is not interrupted by lower priority tasks. This does not work with true concurrency, since multiple tasks can execute at the same time. For the same reasons, disabling preemption is another technique that cannot be used to serialize execution. Synchronization methods such as rendezvous, semaphores, mailboxes, or some other method should be used to implement critical sections.

1.5 Runtime System Interfaces

Figure 1-2 is a copy of Figure 1-1 on page 2 with the interface file names shown.

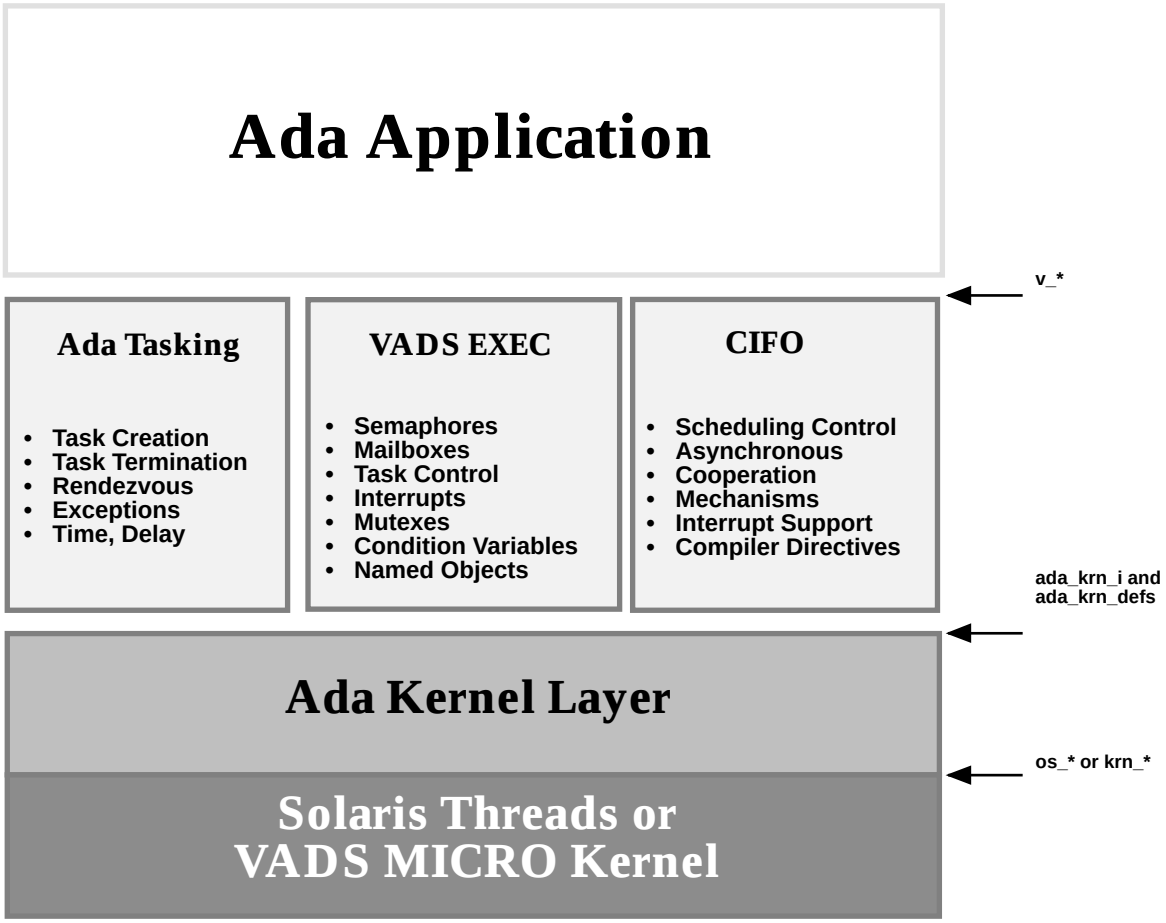

Figure 1-2 Runtime Interfaces

Table 1-1 shows exactly what files in the SC Ada release correspond to the interfaces shown inFigure 1-2 on page 9

Table 1-1 Runtime Interfaces

Common to VADS Threaded and Solaris MT Runtimes				
Ada Tasking /self/standard /self_thr/standard	**VADS EXEC** /self/vads_exec /self_thr/vads_exec	**Ada Kernel** /self/standard /self_thr/standard	**Solaris Threads** /self_thr/standard	**VADS MICRO** /self/standard
v_i_alloc.a	v_intr_b.a	ada_krn_defs.a	os_alloc.a	krn_call_i.a
v_i_bits.a	v_mbox_b.a	ada_krn_i.a	os_signal.a	krn_cpu_defs.a
v_i_callout.a	v_mem_b.a		os_synch.a	krn_defs.a
v_i_cifo.a	v_semaph_b.a		os_thread.a	krn_entries.a
v_i_csema.a	v_stack_b.a		os_time.a	
v_i_except.a	v_vads_exec.a			
v_i_intr.a	v_xtask_b.a			
v_i_libop.a	v_names_b.a			
v_i_mbox.a	v_mutexes_b.a			
v_i_mem.a				
v_i_mutex.a				
v_i_pass.a				
v_i_raise.a				
v_i_sema.a				
v_i_sig.a				
v_i_taskop.a				
v_i_tasks.a				
v_i_time.a				
v_i_timeop.a				
v_i_types.a				

References

"Ada Kernel Layer" on page 2-1

contents of the `standard` library, *SPARCompiler Ada Reference Guide*

SPARCompiler Ada Multithreading Ada Application

VADS EXEC, Chapter 4, "Ada Runtime Services

1.5.1 The Interface Files: ada_krn_defs.a and ada_krn_i.a

We recommend that you do not directly call the interface presented in `ada_krn_i.a` and `ada_krn_defs.a`. This internal interface is documented to provide an understanding of how the Ada Kernel works.

The exception to this is the object attribute initialization routines in `ada_krn_defs.a`. in the section titled Ada Kernel Implementation, where there is a list of mechanisms available to the application programmer to control the attributes of the Ada kernel objects. The attribute initialization routines from `ada_krn_defs.a` are provided for this purpose.

In earlier SC Ada releases, the interface to the low-level kernel services was scattered across multiple `v_i_*` files. In version 3.0, the interface is consolidated in one package, `ADA_KRN_I`. Most of the services are needed to support the Ada tasking semantics. The remaining services are needed to support VADS EXEC. The VADS EXEC services are not supported by all versions of the microkernel.

The services are subdivided into the following groups.

Program
Kernel scheduling
Task management
Task masters synchronization
Task synchronization
Interrupt
Time
Allocation
Mutex
ISR Mutex
Condition variable
Semaphore

Count semaphore
Mailbox
Callout
Task storage
Name

All microkernel-specific type definitions used by the Ada tasking layer are defined in `package ADA_KRN_DEFS`. This package also contains numerous functions for allocating and initializing object attribute records.

File `ada_krn_defs.a` contains type definitions for the following objects:

Mutex
Condition variable
Semaphore
Counting semaphore
Mailbox

We are all framed of flaps and patches and of so shapeless and diverse a texture that every piece and every moment playeth his part”
Montaigne

Runtime System Topics 2

This chapter discusses the following runtime system topics:

- Ada Kernel Layer
- Passive Tasks
- Program Exit or Deadlock
- SC Ada Archive Interface Packages
- Tasking
- Fast Rendezvous Optimization

2.1 Ada Kernel Layer

The Ada Kernel layer is a runtime nucleus with concurrency and synchronization services. It satisfies all the runtime requirements of Ada tasking and the VADS Extensions (VADS EXEC and CIFO).

The Ada Kernel defines and provides operations for the following concurrency and synchronization objects:

- Ada program
- Ada task
- Ada task master
- Ada ”new” allocation
- Kernel scheduling
- Callout
- Task storage
- Interrupts
- Time

- Mutex
- Condition variable
- Binary semaphore
- Counting semaphore
- Mailbox
- Name

For each object there are type definitions and a collection of services. Most objects have a set of user-definable attributes that are used to initialize it. Type and attribute definitions for each object are in package `ADA_KRN_DEFS`. `package ADA_KRN_I` contains the interface to all the object services. Both packages are located in the `standard` directory.

2.1.1 Mutex and Condition Variable Object Types Definition

The mutex and condition variable object types complement semaphores. The definition of these object types is extracted from the POSIX 1003.4a standard, *IEEE Threads Extension for Portable Operating Systems*. See the latest draft of that standard for more details about POSIX threads.

2.1.1.1 Mutex

A *mutex* is a synchronization object used to allow multiple threads to serialize their access to shared data. The name derives from the capability it provides: MUTual EXclusion.

The thread that has locked a mutex becomes its owner and remains the owner until the same thread unlocks the mutex. Other threads that attempt to lock the mutex during this period suspend execution until the original thread unlocks it. The act of suspending the execution of a thread awaiting a mutex does not prevent other threads from making progress in their computations.

In the Ada tasking layer, an `ABORT_SAFE` option has been added to mutexes. A task that has locked an `ABORT_SAFE` mutex is inhibited from being completed by an Ada abort until it unlocks the mutex.

2.1.1.2 Condition Variable

A *condition variable* is a synchronization object that allows a thread to suspend execution until some condition is true. Typically, a thread holding a mutex determines that it cannot proceed by examining the shared data guarded by the mutex. The thread then waits on a condition variable associated with some state of the shared data. Waiting on the condition variable atomically releases the mutex. If another thread modifies the shared data to make the condition true, that thread signals threads waiting on the condition variable. This wakes up the waiting thread which reacquires the mutex, and resumes its execution.

In the Ada tasking layer, an `ABORT_SAFE` option has been added to condition variables. A task locking an `ABORT_SAFE` mutex is inhibited from being completed by an Ada abort until it unlocks the mutex. However, if a task is aborted while waiting at a condition variable (after an implicit mutex unlock), it is allowed to complete.

2.1.2 Ada Kernel Implementation

The Ada Kernel layer can be layered on numerous microkernels. Each Ada Kernel implementation maps the objects, attributes, and services defined in `ADA_KRN_DEFS` and `ADA_KRN_I` onto the underlying microkernel objects and services. By layering Ada tasks on microkernels, Ada tasks can co-exist with and call thread based services written in other languages, such as threaded windows written in C.

In addition, a Sun workstation can have multiple CPUs. Its microkernel services have been designed to allow concurrent thread execution from a single program across multiple CPUs. By layering Ada tasks upon threads, we obtain the multi-CPU capability.

The same `ADA_KRN_DEFS` and `ADA_KRN_I` interface is provided across all the Ada Kernel implementations. However, each implementation has a different representation for the object and attribute type definitions. Microkernel-specific capabilities are made available to the application program through the object attributes. Object attributes can be used in the following places in an application program:

- The address of a mutex attributes record is the second argument of a `PASSIVE` pragma. The passive task's critical region is protected by locking and unlocking the mutex initialized using the mutex attributes. The mutex attributes specify the locking mechanism (test-and-set or disable interrupts)

and the queuing order when the task blocks waiting for the mutex (fifo, priority or priority inheritance). In the CIFO add-on product, VADS MICRO also supports priority ceiling mutexes using the priority ceiling protocol emulation algorithm documented in the POSIX 1003.4a standard.

- A semaphore attributes access value is passed to the VADS EXEC service, `V_SEMAPHORES.CREATE_SEMAPHORE` which returns a binary semaphore `ID`.
- A counting semaphore attributes access value is passed to the VADS EXEC service, `V_SEMAPHORES.CREATE_SEMAPHORE`, which returns a counting semaphore `ID`.
- A mailbox attributes access value is passed to the VADS EXEC service, `V_MAILBOXES.CREATE_MAILBOX`.
- A mutex attributes access value is passed to the VADS EXEC service, `V_MUTEXES.CREATE_MUTEX`, which returns a mutex `ID`.
- The address of a task attributes record is the first or second argument of a `TASK_ATTRIBUTES` pragma.

`ADA_KRN_DEFS` provides a set of subprograms for initializing the object attributes. For example, the following subprograms are provided to initialize mutex attributes:

```
fifo_mutex_attr_init()
prio_mutex_attr_init()
prio_inherit_mutex_attr_init()
prio_ceiling_mutex_attr_init()
intr_attr_init()
```

These subprograms are applicable to all implementations of the Ada Kernel. If an implementation does not support a particular attribute capability, it raises the `PROGRAM_ERROR` exception. Since these subprograms insulate the application software from the underlying attribute type representation, they should be used instead of explicitly initializing each field of an attribute record.

Even though the Ada Kernel exists primarily to satisfy the requirements of Ada tasking and the VADS Extensions, some of its services are of interest to the application programmer. VADS EXEC has services for binary semaphores, counting semaphores, mutexes, condition variables, mailboxes, and names.

These services are layered on the corresponding services in the Ada Kernel. The application may elect to bypass the additional overhead incurred by the VADS EXEC layer and call the Ada Kernel services directly.

Caution – Since the program, master, and task services have complex semantics and implicit dependencies, we advise against calling these services directly.

The following sections discuss each of the objects in the Ada Kernel. For each object, the following topics are addressed:

- Types
- Constants
- Attributes
- Services
- Support Subprograms

The interface to the services is provided in `ada_krn_i.a`. `ada_krn_defs.a`. contains the rest. Both files are in the `standard` directory.

2.1.2.1 *Ada Program*

There is a single Ada program object per executable entity. The Ada program object is automatically created and initialized during program startup.

Types

`PROGRAM_ID`

Type of the Ada program's object. The type definition for `PROGRAM_ID` is in `package SYSTEM`.

`KRN_PROGRAM_ID`

Type of the underlying program / process. There are services for mapping between the Ada `PROGRAM_ID` and `KRN_PROGRAM_ID`.

Attributes

None.

Services

Since the Ada program services should not be called directly, they are only listed here. Consult `ada_krn_i.a` in `standard` for more details.

Program services

```
PROGRAM_INIT
PROGRAM_EXIT
PROGRAM_DIAGNOSTIC
PANIC_EXIT
PROGRAM_IS_ACTIVE
PROGRAM_SELF
```

Program services (VADS EXEC augmentation)

```
PROGRAM_GET
PROGRAM_START
PROGRAM.SET_IS_SERVER
PROGRAM.IS_SERVER
PROGRAM_TERMINATE
PROGRAM_GET_KEY
PROGRAM_GET_ADA_ID
PROGRAM_GET_KRN_ID
PROGRAM.INTER.CALL
```

2.1.2.2 Ada Task

An Ada task object exists for each thread of execution. Most operations on Ada task objects are done implicitly using the tasking semantics of the Ada language.

Types

`TASK_ID`

Type of the Ada task's object. The type definition for `TASK_ID` is in the `package SYSTEM`.

`KRN_TASK_ID`

Type of the underlying thread. There are services for mapping between the Ada `TASK_ID` and `KRN_TASK_ID`.

Attributes

`TASK_ATTR_T`

The address of a `TASK_ATTR_T` record is the first or second argument of the `TASK_ATTRIBUTES` pragma and is passed to the underlying microkernel at task creation. The definition of the `TASK_ATTR_T` record is microkernel-specific. However, all variations of the `TASK_ATTR_T` record contain at least the `prio`, `mutex_attr_address`, and `cond_attr_address` fields. The `prio` field specifies the priority of the task. If the task also had a `pragma PRIORITY(PRIO)`, the `prio` specified in the `TASK_ATTR_T` record takes precedence.

The `mutex_attr_address` field contains the address of the attributes to be used to initialize the mutex object implicitly created for the task. This mutex is used to protect the task's data structure. For example, the task's mutex is locked when another task attempts to rendezvous with it. If `mutex_attr_address` is set to `NO_ADDR`, the `mutex_attr_address` value specified by the `V_USR_CONF.CONFIGURATION_TABLE` parameter, `DEFAULT_TASK_ATTRIBUTES` is used. Otherwise, `mutex_attr_address` must be set to the address of an `ADA_KRN_DEFS.MUTEX_ATTR_T` record. The `MUTEX_ATTR_T` record should be initialized using one of the mutex attribute init subprograms.

References

"Mutex Support Subprograms" on page 2-25

The `cond_attr_address` field contains the address of the attributes to be used to initialize the condition variable object implicitly created for the task. When the task blocks, it waits on this condition variable. If `cond_attr_address` is set to `NO_ADDR`, the `cond_attr_address` value specified by the `V_USR_CONF.CONFIGURATION_TABLE` parameter, `DEFAULT_TASK_ATTRIBUTES` is used. Otherwise, `cond_attr_address` must be set to the address of a `ADA_KRN_DEFS.COND_ATTR_T` record. The `COND_ATTR_T` record should be initialized using one of the condition variable attribute init routines.

References

"Condition Variable Support Subprograms" on page 2-32

The `TASK_ATTR_T` record can be initialized using one of the overloaded `TASK_ATTR_INIT` subprograms.

References

"Ada Task Support Subprograms" on page 2-11

`SPORADIC_ATTR_T`

In the optional CIFO product using VADS MICRO, an Ada task is made sporadic by updating the `sporadic_attr_address` field in the `TASK_ATTR_T` record with the address of a `SPORADIC_ATTR_T` record. The easiest way to initialize both the `TASK_ATTR_T` and `SPORADIC_ATTR_T` records is to use one of the overloaded `SPORADIC_TASK_ATTR_INIT` subprograms. See the CIFO documentation for more details about the attributes and characteristics of a sporadic task.

Services

There are two services for mapping between the Ada TASK_ID and the KRN_TASK_ID:

```
function task_get_Ada_id(krn_tsk: krn_task_id) return
task_id;
 -- Returns NO_TASK_ID if the kernel task is not also an
Ada task
function task_get_krn_id(tsk: task_id) return
krn_task_id;
```

Since the remaining Ada task services should not be called directly, they are only listed here. Consult ada_krn_i.a in standard for more details.

Task management services

```
TASK_SELF
TASK_SET_PRIORITY
TASK_GET_PRIORITY
TASK_CREATE
TASK_GET_SEQUENCE_NUMBER
TASK_ACTIVATE
TASK_STOP
TASK_DESTROY
TASK_STOP_SELF
TASK_DESTROY_SELF
```

Task management services (VADS EXEC augmentation)

```
TASK_DISABLE_PREEMPTION
TASK_ENABLE_PREEMPTION
TASK_SUSPEND
TASK_RESUME
TASK_GET_TIME_SLICE
TASK_SET_TIME_SLICE
TASK_GET_SUPERVISOR_STATE
TASK_ENTER_SUPERVISOR_STATE
TASK_LEAVE_SUPERVISOR_STATE
```

Task synchronization services

```
TASK_LOCK
TASK_UNLOCK
TASK_WAIT
TASK_WAIT_LOCKED_MASTERS
TASK_TIMED_WAIT
TASK_SIGNAL
TASK_WAIT_UNLOCK
TASK_SIGNAL_UNLOCK
TASK_SIGNAL_WAIT_UNLOCK
```

Sporadic task services (CIFO augmentation)

```
TASK_IS_SPORADIC
TASK_SET_FORCE_HIGH_PRIORITY
```

Ada Task Support Subprograms

The TASK_ATTR_T record can be initialized using one of the overloaded TASK_ATTR_INIT subprograms. The `task_attr` parameter identifies the address of the storage allocated for the task attribute record.

```
procedure task_attr_init(
     task_attr   : a_task_attr_t;
     prio        : priority := priority'first;
     -- ...  OS Threads specific fields;
     mutex_attr  : a_mutex_attr_t := null;
     cond_attr   : a_cond_attr_t := null
);

function task_attr_init(
     task_attr   : a_task_attr_t;
     prio        : priority := priority'first;
     -- ...  OS Threads specific fields;
     mutex_attr  : a_mutex_attr_t := null;
     cond_attr   : a_cond_attr_t := null
) return address;

function task_attr_init(
     -- does an implicit "task_attr: a_task_attr_t :=
     --                          new task_attr_t;"
     prio        : priority := priority'first;
     -- ...  OS Threads specific fields;
     mutex_attr  : a_mutex_attr_t := null;
     cond_attr   : a_cond_attr_t := null
 ) return address;
```

Examples

```
with system;
with ada_krn_defs;
package one is
    -- Does an implicit allocation of the task_attr_t record
    task a is
        pragma task_attributes(ada_krn_defs.task_attr_init(
                                prio => 20,
                                ... OS_threads_specific_fields
                               ));
    end;
    task type tt;
    b: tt;
        pragma task_attributes(b, ada_krn_defs.task_attr_init(
                                prio => 30,
                                ... OS_threads_specific_fields
                               ));
end one;


with system;
with ada_krn_defs;
package two is
    -- No implicit allocation is done
    a_attr_rec: ada_krn_defs.task_attr_t;
    a_attr: system.address :=
        ada_krn_defs.task_attr_init(
            task_attr   => ada_krn_defs.to_a_task_attr_t(
                                a_attr_rec'address),
            prio        => 20,
            ... OS_threads_specific_fields
        );
    b_attr_rec: ada_krn_defs.task_attr_t;
    b_attr: system.address :=
        ada_krn_defs.task_attr_init(
            task_attr   => ada_krn_defs.to_a_task_attr_t(
                                b_attr_rec'address),
            prio        => 30,
            ... OS_threads_specific_fields
        );
    task a is
        pragma task_attributes(a_attr);
```

(Continued) (Continued)

```
        -- or
        -- pragma task_attributes(a_attr_rec'address);
    end;
            task type tt;
    b: tt;
        pragma task_attributes(b, b_attr);
        -- or
        -- pragma task_attributes(b, b_attr_rec'address);
end two;
```

Figure 2-1 Initializing the TASK_ATTR_T Record

References

Pragmas, *SPARCompiler Ada Programmer's Guide*

2.1.2.3 Ada Task Master

A single task master object exists to provide mutual exclusion for operations performed across multiple task objects (for example, aborting Ada tasks).

Types

None. There is one master object and it is implied in the services.

Attributes

None.

Services

Since the Ada task master services should not be called directly, they are only listed here. Consult `ada_krn_i.a` in `standard` for more details

Task masters synchronization services

```
MASTERS_LOCK
MASTERS_TRYLOCK
MASTERS_UNLOCK
```

2.1.2.4 Ada "new" Allocation

An object is created for each Ada `new` allocator. This object is freed using Ada's `UNCHECKED_DEALLOCATION`. The Ada Kernel provides services to support Ada's allocation / deallocation needs.

Types

`ADDRESS`

The `ADDRESS` of the object is returned when it has been allocated. This same `ADDRESS` is passed to deallocate the object.

Attributes

None.

Services

The Ada Kernel has the following allocation services:

```
function alloc(size: natural) return address;
 -- Returns NO_ADDR if alloc is unsuccessful.
procedure free(a: address);
```

2.1.2.5 Kernel Scheduling

Services are provided to control the kernel scheduling policies.

Types

None. There is one kernel scheduling object and it is implied in the services.

Attributes

None

Services

The Ada Kernel has the following scheduling services:

```
function kernel_get_time_slicing_enabled return boolean;
procedure kernel_set_time_slicing_enabled(new_value:
boolean);
```

Caution – The above scheduling services are not supported for all implementations of the Ada Kernel.

2.1.2.6 *Callout*

Services are provided that allow a subprogram to be called at a program, task, or idle event.

Types

`CALLOUT_EVENT_T`

All versions of the Ada Kernel are expected to support at least the program events: `EXIT_EVENT` and `UNEXPECTED_EXIT_EVENT`.

Attributes

None.

Services

The following service installs a callout for the specified program, task, or idle event. It returns `FALSE` if the service is not supported or is unable to do the installation.

```
function callout_install(event: callout_event_t; proc:
address)
        return boolean;
```

`V_XTASKING.INSTALL_CALLOUT` in VADS EXEC is layered directly on the `CALLOUT_INSTALL` service. Consult the VADS EXEC documentation for more details.

References

“package V_XTASKING — provide Ada task operations” on page 4-138

2.1.2.7 Task Storage

Some versions of the Ada Kernel support user-defined storage on a per-task basis (currently supported only by the VADS MICRO).

Types

`TASK_STORAGE_ID`

The `ID` or handle of a user-defined object stored in every task.

Attributes

None.

Services

The following task storage allocation services are currently supported only for VADS MICRO:

```
function task_storage_alloc(size: natural) return
task_storage_id;
  -- If service isn't supported or unable to allocate memory, it
  -- returns NO_TASK_STORAGE_ID.
function task_storage_get(tsk: task_id; storage:
task_storage_id)
            return address;
function task_storage_get2(krn_tsk: krn_task_id;
     storage: task_storage_id) return address;
```

The VADS EXEC `V_XTASKING` services (`ALLOCATE_TASK_STORAGE`, `GET_TASK_STORAGE`, and `GET_TASK_STORAGE2`) are layered directly on the above Ada Kernel services. See the `V_XTASKING` documentation for more details.

References

“package V_XTASKING — provide Ada task operations” on page 4-138

2.1.2.8 Interrupts

Services are provided to enable / disable interrupts, get interrupt enabled / disabled status, attach / detach interrupt service routine (ISR), get an attached ISR, and check if it is in an ISR. (On self-hosts, interrupts are UNIX signals.)

Types

`INTR_VECTOR_ID_T`

Signal number range.

`INTR_STATUS_T`

Signal mask.

`INTR_ENTRY_T`

The address of an `INTR_ENTRY_T` object is specified in an interrupt entry address clause. The `INTR_ENTRY_T` record contains two fields: interrupt vector and the task priority for executing the interrupt entry's accept body.

Constants

`DISABLE_INTR_STATUS`

Constant for disabling all asynchronous signals

`ENABLE_INTR_STATUS`

Constant for enabling all asynchronous signals

`BAD_INTR_VECTOR`

Value returned for a bad `INTR_VECTOR` passed to an interrupt service routine.

Attributes

None.

Services

The Ada Kernel has the following interrupt services:

```
  procedure interrupts_get_status(status: out intr_status_t);
  procedure interrupts_set_status(old_status: out intr_status_t;
              new_status: intr_status_t);
  function isr_attach(iv: intr_vector_id_t; isr: address) return
address;
    -- Returns address of previously attached isr.
    -- ADA_KRN_DEFS.BAD_INTR_VECTOR is returned for a bad intr_vector
    -- parameter.
  function isr_detach(iv: intr_vector_id_t) return address;
    -- Returns address of previously attached isr.
    -- ADA_KRN_DEFS.BAD_INTR_VECTOR is returned for a bad intr_vector
    -- parameter.
  function isr_get(iv: intr_vector_id_t) return address;
    -- Returns the address of the currently attached isr.
    -- ada_krn_defs.BAD_INTR_VECTOR is returned for a bad intr_vector
    -- parameter.
  function isr_get_ivt return address;
   -- Returns address of the Interrupt Vector Table (IVT). Normally, the
   -- IVT is an array of ISR addresses. However, the IVT representation
    -- is CPU dependent (for 386 cross, its the IDT).
  function isr_in_check return boolean;
    -- If in an ISR, returns TRUE.
```

The VADS EXEC `V_INTERRUPTS` services (`ATTACH_ISR`, `DETACH_ISR`, `GET.ISR`, `GET.IVT`, `CURRENT_INTERRUPT_STATUS`, and `SET_INTERRUPT_STATUS`) are layered directly on the Ada Kernel services. See the `V_INTERRUPTS` documentation for more details.

Caution – No VADS EXEC service is layered on the `ISR_IN_CHECK` Ada Kernel service.

Interrupt Support Subprograms

The INTR_ENTRY_T record can be initialized using one of the overloaded INTR_ENTRY_INIT subprograms:

```
procedure intr_entry_init(
    intr_entry  : a_intr_entry_t;
    intr_vector : intr_vector_id_t;
    prio        : priority := priority'last);

function intr_entry_init(
    intr_entry  : a_intr_entry_t;
    intr_vector : intr_vector_id_t;
    prio        : priority := priority'last) return address;

function intr_entry_init(
    -- does an implicit "intr_entry: a_intr_entry_t :=
    --                                          new intr_entry_t;"
    intr_vector : intr_vector_id_t;
    prio        : priority := priority'last) return address;
```

Examples

`intr_entry_init()` can be used as follows to define an interrupt entry:

```
with system;
with ada_krn_defs;
package one is
   -- Does an implicit allocation of the intr_entry_t record
task a is
   entry ctrl_c;
   for ctrl_c use at ada_krn_defs.intr_entry_init(
                intr_vector => 2,
                prio => priority'last);
   end;
end one;
with system;
with ada_krn_defs;
package two is
   -- No implicit allocation is done
   ctrl_c_intr_entry_rec: ada_krn_defs.intr_entry_t;
   ctrl_c_intr_entry: system.address :=
       ada_krn_defs.intr_entry_init(
           intr_entry  => ada_krn_defs.to_a_intr_entry_t(
                          ctrl_c_intr_entry_rec'address),
           intr_vector => 2,
           prio        => priority'last);
  task a is
    entry ctrl_c;
    for ctrl_c use at ctrl_c_intr_entry;
    -- OR
    -- for ctrl_c use at ctrl_c_intr_entry_rec'address;
   end;
end two;
```

Figure 2-2 Define an Interrupt Entry

References

"package V_INTERRUPTS — provide interrupt processing" on page 4-9

2.1.2.9 Time

The Ada Kernel provides time services for supporting Ada's delay statement, the predefined CALENDAR package, and the calendar extensions in the XCALENDAR package.

The Ada `delay` statement and the procedure TIME_DELAY produce a delta delay, whereas DELAY_UNTIL produces an absolute delay.

If a task used the Ada `delay` statement or has called TIME_DELAY, calling SET_TIME does not affect how long the task actually delays.

If a task calls TIME_DELAY_UNTIL, calling SET_TIME changes how long the task actually delays by the amount of the adjustment. For example, if a task has called `DELAY_UNTIL(day => 0, sec => 1200.0)`, and the current time is `(day => 0, sec => 1100.0)`, calling SET_TIME with `(day => 0, sec => 1150.0)` shortens the length of time that task delays by 50 seconds. Additionally, if the current time is moved past the DELAY_UNTIL time for a task, that task is placed on the run queue immediately.

Types

DAY_T

Type of the time's day component. The type definition for DAY_T is in `package SYSTEM`.

DURATION

Type of the time's seconds within a day. DURATION is a predefined Ada type.

Attributes

None.

Services

The Ada Kernel has the following time services:

```
procedure time_set(day: day_t;
                   sec: duration;
                 timer_support_arg: address := NO_ADDR);
  -- The input time must be normalized, sec < 86400.0.
  -- The V_I_TIMEOP package in standard has subprograms
  -- for normalizing time.
  -- For cross targets, timer_support_arg is passed
  -- to V_KRN_CONF's V_TIMER_SUPPORT.SET_TIME
  -- procedure.
  -- For self-hosts, if timer_support_arg /= NO_ADDR, then
  -- it's the address of the OS's time record.  This
  -- allows time_set() to be automatically set with the
  -- OS's current time.
procedure time_get(day: out day_t; sec: out duration);
  -- Returned time is normalized, sec < 86400.0
procedure time_delay(sec: duration);
procedure time_delay_until(day: day_t; sec: duration);
  -- The input time must be normalized, sec < 86400.0.
```

2.1.2.10 Mutex

A mutex object is used to protect a passive task's critical region. Mutexes are used in Company implementation of CIFO. Mutexes can also be used explicitly by the user to serialize access to shared data.

The semantics of locking/unlocking a mutex adheres to the POSIX 1003.4a standard, *IEEE Threads Extension for Portable Operating Systems*. See the latest draft of that standard for more details about POSIX mutexes.

An ABORT_SAFE version of the mutex services is provided in the V_I_MUTEX package found in `standard`. A task locking an ABORT_SAFE mutex is inhibited from being completed by an Ada abort until it unlocks the mutex.

Types

MUTEX_T, A_MUTEX_T

MUTEX_T is the type of a mutex object. A_MUTEX_T is the access type of a mutex object. The address of a mutex object can be converted to its access type with the function, ADA_KRN_DEFS.TO_A_MUTEX_T().

Attributes

MUTEX_ATTR_T, A_MUTEX_ATTR_T

MUTEX_ATTR_T is the type definition of the mutex attributes . A_MUTEX_ATTR_T is its access type. The function, ADA_KRN_DEFS.TO_A_MUTEX_ATTR_T( ) can be used to convert the address of the mutex attribute record to its access type.

The mutex attributes are microkernel-dependent. See `ada_krn_defs.a` in `standard` for the options supported. (VADS MICRO locks the mutex by executing a test-and-set instruction or by disabling interrupts. It supports FIFO, priority, or priority inheritance waiting when the mutex is locked by another task. In the optional CIFO product, VADS MICRO also supports priority ceiling mutexes using the priority ceiling protocol emulation algorithm documented in the POSIX 1003.4a standard.)

VADS MICRO has the following variant mutex attribute record type:

INTR_ATTR_T

disables interrupts (signals) to provide mutual exclusion

The function DEFAULT_MUTEX_ATTR is provided to select the default mutex attributes.

To provide mutual exclusion by disabling all interrupts, use DEFAULT_INTR_ATTR. If the underlying microkernel does not support interrupt attributes, the PROGRAM_ERROR exception is raised.

Services

The Ada Kernel has the following mutex services:

```
function mutex_init(mutex: a_mutex_t; attr: a_mutex_attr_t)
            return boolean;
  -- Returns TRUE if mutex was successfully initialized.

procedure mutex_destroy(mutex: a_mutex_t);

procedure mutex_lock(mutex: a_mutex_t);

function mutex_trylock(mutex: a_mutex_t) return boolean;
 -- Returns TRUE if we were able to lock the mutex without
 -- waiting. Otherwise, returns FALSE without locking the mutex.

procedure mutex_unlock(mutex: a_mutex_t);
```

The Ada Kernel has the following services for mutexes that can be locked from an ISR:

```
function isr_mutex_lockable(mutex: a_mutex_t) return boolean;
  -- Returns TRUE if mutex can be locked from an ISR. This
  -- service is called for a passive task with an interrupt
  -- entry. Since the passive task's critical region will be
  -- entered from an ISR, we must be able to lock its mutex
  -- from an ISR.

procedure isr_mutex_lock(mutex: a_mutex_t);
procedure isr_mutex_unlock(mutex: a_mutex_t);
  -- The isr_mutex_lock/isr_mutex_unlock services must only
  -- be called from an ISR using a mutex that is lockable
  -- from an ISR.
```

The Ada Kernel has the following priority ceiling mutex services (supported only in the optional CIFO product):

```
function ceiling_mutex_init(mutex: a_mutex_t; attr:
a_mutex_attr_t;
    ceiling_prio: priority := priority'last) return boolean;
     -- Returns TRUE if underlying threads supports priority
ceiling
     -- protocol and the mutex was successfully initialized.
     --
     -- The attr parameter can be set to DEFAULT_MUTEX_ATTR to use
     -- the default priority ceiling attributes. The VADS MICRO
     -- ignores the attr parameter.
function ceiling_mutex_set_priority(mutex: a_mutex_t;
    ceiling_prio: priority) return boolean;
     -- Returns FALSE if not a priority ceiling mutex
```

Mutex Support Subprograms

The following subprograms are provided to initialize the MUTEX_ATTR_T record:

```
fifo_mutex_attr_init()
prio_mutex_attr_init()
prio_inherit_mutex_attr_init()
prio_ceiling_mutex_attr_init()
intr_attr_init()
```

If the underlying microkernel does not support the type of mutex being initialized, the PROGRAM_ERROR exception is raised.

Here are the overloaded subprograms for initializing the MUTEX_ATTR_T record:

```
procedure fifo_mutex_attr_init(
    attr            : a_mutex_attr_t);

function fifo_mutex_attr_init(
    attr            : a_mutex_attr_t) return a_mutex_attr_t;

function fifo_mutex_attr_init(
    attr            : a_mutex_attr_t) return address;

function fifo_mutex_attr_init return a_mutex_attr_t;
    -- does an implicit
    --      ”attr: a_mutex_attr_t := new mutex_attr_t;”

 function fifo_mutex_attr_init return address;
    -- does an implicit
    --      ”attr: a_mutex_attr_t := new mutex_attr_t;”
 procedure prio_mutex_attr_init(
     attr            : a_mutex_attr_t);

 function prio_mutex_attr_init(
    attr            : a_mutex_attr_t) return a_mutex_attr_t;

function prio_mutex_attr_init(
    attr            : a_mutex_attr_t) return address;

function prio_mutex_attr_init return a_mutex_attr_t;
    -- does an implicit
    --      ”attr: a_mutex_attr_t := new mutex_attr_t;”

function prio_mutex_attr_init return address;
   -- does an implicit
   --      ”attr: a_mutex_attr_t := new mutex_attr_t;”
procedure  prio_inherit_mutex_attr_init(
    attr            : a_mutex_attr_t);

function  prio_inherit_mutex_attr_init(
    attr            : a_mutex_attr_t) return a_mutex_attr_t;

function  prio_inherit_mutex_attr_init(
    attr            : a_mutex_attr_t) return address;
```

(Continued)

```
function prio_inherit_mutex_attr_init return a_mutex_attr_t;
    -- does an implicit
    --      ”attr: a_mutex_attr_t := new mutex_attr_t;”

function prio_inherit_mutex_attr_init return address;

    -- does an implicit
    --      ”attr: a_mutex_attr_t := new mutex_attr_t;”

procedure prio_ceiling_mutex_attr_init(
    attr    : a_mutex_attr_t;
    ceiling_prio: priority := priority’last);

function prio_ceiling_mutex_attr_init(
    attr    : a_mutex_attr_t;
    ceiling_prio: priority := priority’last) return a_mutex_attr_t;
    function prio_ceiling_mutex_attr_init(
    attr    : a_mutex_attr_t;
    ceiling_prio: priority := priority’last) return address;

function prio_ceiling_mutex_attr_init(
    -- does an implicit ”attr: a_mutex_attr_t := new mutex_attr_t;”
    ceiling_prio: priority := priority’last) return a_mutex_attr_t;

function prio_ceiling_mutex_attr_init(
    -- does an implicit ”attr: a_mutex_attr_t := new mutex_attr_t;”
    ceiling_prio: priority := priority’last) return address;
procedure intr_attr_init(
    attr            : a_mutex_attr_t;
    disable_status  : intr_status_t := DISABLE_INTR_STATUS);

function intr_attr_init(
   attr            : a_mutex_attr_t;
   disable_status  : intr_status_t := DISABLE_INTR_STATUS)
                                        return a_mutex_attr_t;

function intr_attr_init(
    attr            : a_mutex_attr_t;
    disable_status  : intr_status_t := DISABLE_INTR_STATUS)
                                        return address;

function intr_attr_init(
```

(Continued)

```
    -- does an implicit "attr: a_mutex_attr_t :=
                                        new mutex_attr_t;"
    disable_status  : intr_status_t := DISABLE_INTR_STATUS)
                                        return a_mutex_attr_t;

function intr_attr_init(
    -- does an implicit "attr: a_mutex_attr_t :=
                                        new mutex_attr_t;"
    disable_status  : intr_status_t := DISABLE_INTR_STATUS)
                                        return address;
```

Examples

The above `init` subprograms can be used as follows in a `PASSIVE` pragma:

```
with system;
with ada_krn_defs;
package one is
   task a is
     pragma passive(ABORT_SAFE,
ada_krn_defs.fifo_mutex_attr_init);
   end;
   prio_mutex_attr_rec: ada_krn_defs.mutex_attr_t;
   prio_mutex_attr: system.address :=
       ada_krn_defs.prio_mutex_attr_init(
           ada_krn_defs.to_a_mutex_attr_t(
              prio_mutex_attr_rec'address));
   task b is
      pragma passive(ABORT_SAFE, prio_mutex_attr);
      -- or
     -- pragma passive(ABORT_SAFE, prio_mutex_attr_rec'address);
   end;
end one;
```

Figure 2-3 Initialize Subprograms in Passive Tasks

References

"Passive Tasks" on page 2-44

2.1.2.11 Condition Variable

Condition variables are used to wait until a particular condition is TRUE. A condition variable must be used in conjunction with a mutex.

If the guard to the called entry in a passive task is closed, the calling task waits on a condition variable. Condition variables are used to supplement mutexes in the implementation of CIFO and the sporadic server. Condition variables can also be used explicitly by the user.

The semantics of waiting on or signaling a condition variable and its interaction with a mutex adheres to the POSIX 1003.4a standard, *IEEE Threads Extension for Portable Operating Systems*. See the latest draft of that standard for more details about POSIX condition variables.

An ABORT_SAFE version of the mutex and condition variable services is provided in package V_I_MUTEX found in standard. A task locking an ABORT_SAFE mutex is inhibited from being completed by an Ada abort until it unlocks the mutex. However, if a task is aborted while waiting at a condition variable (after an implicit mutex unlock), it is allowed to complete. The V_I_MUTEX services also address the case where multiple ABORT_SAFE mutexes are locked. A task is inhibited from being completed until all the mutexes are unlocked or it does a condition variable wait with only one mutex locked.

Types

COND_T, A_COND_T

COND_T is the type of a condition variable object. A_COND_T is the access type of a condition variable object. The address of a condition variable object can be converted to its access type using the function ADA_KRN_DEFS.TO_A_COND_T().

Attributes

COND_ATTR_T, A_COND_ATTR_T

COND_ATTR_T is the type definition of the condition variable attributes. A_COND_ATTR_T is its access type. The function ADA_KRN_DEFS.TO_A_COND_ATTR_T() can be used to convert the address of the condition variable attributes to its access type.

The condition variable attributes are microkernel-dependent. See `ada_krn_defs.a` in `standard` for the options supported. VADS MICRO supports FIFO or priority waiting.

The function `DEFAULT_COND_ATTR` is provided to select the default condition variable attributes.

Services

The Ada Kernel has the following condition variable services:

```
  function cond_init(cond: a_cond_t; attr: a_cond_attr_t) return
boolean;
   -- Returns TRUE if condition variable was successfully initialized.

  procedure cond_destroy(cond: a_cond_t);

  procedure cond_wait(cond: a_cond_t; mutex: a_mutex_t);
    -- COND_WAIT must be called with the mutex already locked by the
   -- calling task. COND_WAIT atomically releases the mutex and causes
    -- the calling task to block on the condition variable. The
    -- blocked task may be awakened by COND_SIGNAL(),
    -- COND_SIGNAL_UNLOCK(), COND_BROADCAST(), or by some OS
    -- Threads specific stimulus (for Sun Threads it may also
   -- be awakened when the task is interrupted by delivery of a signal
    -- or a fork()). Any change in value of a condition associated
    -- with the condition variable cannot be inferred by the return
    -- of COND_WAIT() and any such condition must be reevaluated.
   -- COND_WAIT() always returns with the mutex locked by the calling
    -- task.

  function cond_timed_wait(cond: a_cond_t; mutex: a_mutex_t;
      sec: duration) return boolean;
    -- COND_TIMED_WAIT() is similar to COND_WAIT(), except that
   -- the calling task will only block for the amount of time specified
    -- by the sec parameter. If the condition variable
    -- wasn't signalled, COND_TIMED_WAIT() returns FALSE. For
    -- all cases, COND_TIMED_WAIT() returns with the mutex
    -- locked by the calling task.

  procedure cond_signal(cond: a_cond_t);
  procedure cond_broadcast(cond: a_cond_t);
    -- COND_SIGNAL() unblocks one task that is blocked on
```

(Continued)

```
   -- the condition variable. COND_BROADCAST() unblocks all
   -- tasks that are blocked on the condition variable. If
   -- no tasks are blocked on the condition variable then
   -- COND_SIGNAL() and COND_BROADCAST() have no effect. Both
   -- procedures should be called under the protection of the
   -- same mutex that is used with the condition variable being
   -- signalled. Otherwise the condition variable may be
   -- signalled between the test of the associated condition
   -- and blocking in COND_WAIT(). This can cause an infinite wait.
 procedure cond_signal_unlock(cond: a_cond_t; mutex: a_mutex_t);
   -- COND_SIGNAL_UNLOCK has the same semantics as making the
   -- following two calls:
   --    COND_SIGNAL(cond);
   --    MUTEX_UNLOCK(mutex);
   --
   -- To improve performance, the Ada Kernel implementation may
   -- treat the above condition variable signalling and the mutex
   -- unlocking sequence as an atomic operation.
```

The Ada Kernel has the following service for a condition variable that can be called only from an ISR:

```
  procedure isr_cond_signal(cond: a_cond_t);
    -- ISR_COND_SIGNAL() must only be called from an ISR. It
    -- has the same semantics as COND_SIGNAL(). The condition
    -- variable being signalled must be protected by an ISR
    -- lockable mutex.
    --
    -- At the completion of the accept body in a passive task
    -- called from an ISR, this service is called to signal
    -- an Ada task waiting on an entry whose guard was changed
    -- from closed to open.
```

Condition Variable Support Subprograms

The following subprograms are provided to initialize the COND_ATTR_T record:

```
fifo_cond_attr_init()
prio_cond_attr_init()
```

If the underlying microkernel does not support the type of condition variable being initialized, the PROGRAM_ERROR exception is raised.

Here are the overloaded subprograms for initializing the COND_ATTR_T record:

```
procedure fifo_cond_attr_init(
    attr            : a_cond_attr_t);

function fifo_cond_attr_init(
    attr            : a_cond_attr_t) return a_cond_attr_t;

function fifo_cond_attr_init return a_cond_attr_t;
    -- does an implicit
    --      "attr: a_cond_attr_t := new cond_attr_t;"
procedure prio_cond_attr_init(
    attr            : a_cond_attr_t);

function prio_cond_attr_init(
    attr            : a_cond_attr_t) return a_cond_attr_t;

function prio_cond_attr_init return a_cond_attr_t;
    -- does an implicit
    --      "attr: a_cond_attr_t := new cond_attr_t;"
```

2.1.2.12 *Binary Semaphore*

A binary semaphore is an object that can be in one of two states, full or empty. If a task waits on a full semaphore, then it makes the semaphore empty and continues executing. If a task waits on an empty semaphore, then it blocks until it is signalled. When a semaphore is signalled, the next available task is unblocked. If no task was blocked on the semaphore, the semaphore becomes full.

Binary semaphores are used by the VADS EXEC V_SEMAPHORES services. The overhead added by the VADS EXEC layer can be eliminated by directly using the Ada Kernel binary semaphores.

Types

SEMAPHORE_T, A_SEMAPHORE_T

SEMAPHORE_T is the type of a binary semaphore object. A_SEMAPHORE_T is the access type of a binary semaphore object. The address of a binary semaphore object can be converted to its access type with the function ADA_KRN_DEFS.TO_A_SEMAPHORE_T().

SEMAPHORE_STATE_T

Binary semaphore's state: SEMAPHORE_FULL or SEMAPHORE_EMPTY.

Constants

SEMAPHORE_FULL
SEMAPHORE_EMPTY

Attributes

SEMAPHORE_ATTR_T, A_SEMAPHORE_ATTR_T

SEMAPHORE_ATTR_T is the type definition of the binary semaphore attributes. A_SEMAPHORE_ATTR_T is its access type. The function ADA_KRN_DEFS.TO_A_SEMAPHORE_ATTR_T() can be used to convert the address of the binary semaphore attributes to its access type.

The A_SEMAPHORE_ATTR_T access value is passed to the VADS EXEC service, V_SEMAPHORES.CREATE_SEMAPHORE(), which returns a BINARY_SEMAPHORE_ID.

The semaphore attributes are microkernel-dependent. See ada_krn_defs.a in standard for the different options supported. (VADS MICRO supports FIFO queuing only when the task waits on a semaphore.)

The function DEFAULT_SEMAPHORE_ATTR is provided to select the default semaphore attributes.

Services

The Ada Kernel has the following binary semaphore services:

```
 function semaphore_init(s: a_semaphore_t; init_state:
semaphore_state_t;
   attr: a_semaphore_attr_t) return boolean;
   -- Returns TRUE if semaphore was successfully initialized.

 procedure semaphore_destroy(s: a_semaphore_t);

 procedure semaphore_wait(s: a_semaphore_t);

 function semaphore_trywait(s: a_semaphore_t) return boolean;
  -- Returns TRUE if the semaphore was FULL.

 function semaphore_timed_wait(s: a_semaphore_t;
    sec: duration) return boolean;
  -- Returns TRUE if we didn't timeout waiting for the
  -- semaphore to be FULL or signalled.

 procedure semaphore_signal(s: a_semaphore_t);

 function semaphore_get_in_use(s: a_semaphore_t) return boolean;
  -- Returns TRUE if any task is waiting on the semaphore. If the
  -- Ada Kernel is unable to detect this condition, it returns
  -- TRUE.
  --
  -- SEMAPHORE_GET_IN_USE() is called by the VADS EXEC
  -- V_SEMAPHORES.DELETE_SEMAPHORE() service for a binary
  -- semaphore.
```

References

"package V_SEMAPHORES — provide binary and counting semaphores" on page 4-115

2.1.2.13 Counting Semaphore

A counting semaphore is an object with a non-negative count. If a task waits on a semaphore with a non-zero count, it decrements the count and continues executing. If a task waits on a semaphore with a zero count, then it blocks until

it is signalled. When a semaphore is signalled, the next available task is unblocked. If no task was blocked on the semaphore, the semaphore's count is incremented.

Counting semaphores are used by the VADS EXEC V_SEMAPHORES services. The overhead added by the VADS EXEC layer can be eliminated by directly using the Ada Kernel's counting semaphores.

Types

COUNT_SEMAPHORE_T, A_COUNT_SEMAPHORE_T

COUNT_SEMAPHORE_T is the type of a counting semaphore object. A_COUNT_SEMAPHORE_T is the access type of a counting semaphore object. The address of a counting semaphore object can be converted to its access type with the function ADA_KRN_DEFS.TO_A_COUNT_SEMAPHORE_T().

Attributes

COUNT_SEMAPHORE_ATTR_T, A_COUNT_SEMAPHORE_ATTR_T

COUNT_SEMAPHORE_ATTR_T is the type definition of the counting semaphore attributes. A_COUNT_SEMAPHORE_ATTR_T is its access type. The function ADA_KRN_DEFS.TO_A_COUNT_SEMAPHORE_ATTR_T() can be used to convert the address of the counting semaphore attributes to its access type.

The A_COUNT_SEMAPHORE_ATTR_T access value is passed to the VADS EXEC service, V_SEMAPHORES.CREATE_SEMAPHORE(), which returns a COUNT_SEMAPHORE_ID.

The count_semaphore attributes are microkernel-dependent. See ada_krn_defs.a in standard for the options supported. (VADS MICRO uses a mutex to protect the count. It waits on a condition variable. The COUNT_SEMAPHORE_ATTR_T is a subtype of MUTEX_ATTR_T. The COND_ATTR_T is derived from the MUTEX_ATTR_T. A FIFO condition variable is used for a FIFO mutex. A priority condition variable is used for either a priority, priority inheritance, or priority ceiling mutex.)

VADS MICRO has the following variant counting semaphore attribute record type:

COUNT_INTR_ATTR_T

interrupts are disabled when accessing the semaphore count

The function, DEFAULT_COUNT_SEMAPHORE_ATTR, is provided to select the default counting semaphore attributes.

To protect counting semaphore operations by disabling all interrupts, use DEFAULT_COUNT_INTR_ATTR. If the underlying microkernel does not support interrupt attributes, the PROGRAM_ERROR exception is raised. However, if the DEFAULT_COUNT_SEMAPHORE_ATTR is interrupt safe, then DEFAULT_COUNT_INTR_ATTR returns DEFAULT_COUNT_SEMAPHORE_ATTR and does not raise PROGRAM_ERROR.

Alternatively, the counting semaphore attributes can be initialized to select the disable interrupts options by using one of the overloaded COUNT_INTR_ATTR_INIT subprograms.

References

"Counting Semaphore Support Subprograms" on page 2-37

Services

The Ada Kernel has the following counting semaphore services:

```
function count_semaphore_init(
     s           : a_count_semaphore_t;
     init_count  : integer;
     attr        : a_count_semaphore_attr_t) return boolean;
     -- Returns TRUE if semaphore was successfully initialized.

procedure count_semaphore_destroy(s: a_count_semaphore_t);

function count_semaphore_wait(s: a_count_semaphore_t;
      wait_time: duration) return boolean;
      -- Waits on a counting semaphore.
      --
     -- If semaphore's count > 0, decrements the count and returns TRUE.
      -- Otherwise, returns according to the wait_time parameter:
      --  < 0.0         - the calling task blocks until the semaphore
      --                  is signalled. It always returns TRUE.
      --  = 0.0         - immediately returns FALSE.
      --  > 0.0         - returns TRUE if we didn't timeout waiting for
```

(Continued)

```
      --                  the semaphore to be signalled. For a
      --                  timeout, returns FALSE.

procedure count_semaphore_signal(s: a_count_semaphore_t);

function count_semaphore_get_in_use(s: a_count_semaphore_t)
        return boolean;
      -- Returns TRUE if any task is waiting on the semaphore. If the
      -- Ada Kernel is unable to detect this condition, it returns
      -- TRUE.
      --
      -- SEMAPHORE_GET_IN_USE() is called by the VADS EXEC
      -- V_SEMAPHORES.DELETE_SEMAPHORE() service for a counting
      -- semaphore.
```

Counting Semaphore Support Subprograms

The counting semaphore attributes can be initialized to select the disable interrupts options by using one of the overloaded `COUNT_INTR_ATTR_INIT` subprograms.

```
   procedure count_intr_attr_init(
       attr            : a_count_semaphore_attr_t;
       disable_status  : intr_status_t := DISABLE_INTR_STATUS);

   function count_intr_attr_init(
       attr            : a_count_semaphore_attr_t;
       disable_status  : intr_status_t := DISABLE_INTR_STATUS)
                                 return a_count_semaphore_attr_t;

 function count_intr_attr_init(
       -- does an implicit
       --  "attr: a_count_semaphore_attr_t :=
       --                             new count_semaphore_attr_t;"
       disable_status  : intr_status_t := DISABLE_INTR_STATUS)
                                 return a_count_semaphore_attr_t;
```

References

"package V_SEMAPHORES — provide binary and counting semaphores" on page 4-115

2.1.2.14 Mailbox

A mailbox object is used to queue fixed length messages between tasks or between ISRs and tasks. Any task or ISR can write messages to a mailbox object. Any task can read messages from a mailbox. If no message is in the mailbox, the reader can optionally wait until a message is written, return immediately with no message or wait up to a specified amount of time for a message.

Mailboxes are used by the VADS EXEC `V_MAILBOXES` services. The overhead added by the VADS EXEC layer can be eliminated by directly using the Ada Kernel mailboxes.

Types

`MAILBOX_T`, `A_MAILBOX_T`

`MAILBOX_T` is the type of a mailbox object. `A_MAILBOX_T` is the access type of a mailbox object. The address of a mailbox object can be converted to its access type via the function, `ADA_KRN_DEFS.TO_A_MAILBOX_T()`.

Attributes

`MAILBOX_ATTR_T`, `A_MAILBOX_ATTR_T`

`MAILBOX_ATTR_T` is the type definition of the mailbox attributes. `A_MAILBOX_ATTR_T` is its access type. The function `ADA_KRN_DEFS.TO_A_MAILBOX_ATTR_T()` can be used to convert the address of the mailbox attributes to its access type.

The `A_MAILBOX_ATTR_T` access value is passed to the VADS EXEC service, `V_MAILBOXES.CREATE_SEMAPHORE()`.

The mailbox attributes are microkernel dependent. See `ada_krn_defs.a` in `standard` for the options supported. (VADS MICRO uses a mutex to protect the mailbox. It waits on a condition variable. The `MAILBOX_ATTR_T` is a subtype of `MUTEX_ATTR_T`. The `COND_ATTR_T` is derived from the `MUTEX_ATTR_T`. A FIFO condition variable is used for a FIFO mutex. A priority condition variable is used for either a priority, priority inheritance, or priority ceiling mutex.)

VADS MICRO has the following variant mailbox attribute record type:

`MAILBOX_INTR_ATTR_T`

interrupts are disabled when accessing the mailbox

The function `DEFAULT_MAILBOX_ATTR` is provided to select the default mailbox attributes.

To protect mailbox operations by disabling all interrupts, use `DEFAULT_MAILBOX_INTR_ATTR`. If the underlying microkernel does not support interrupt attributes, the `PROGRAM_ERROR` exception is raised. However, if the `DEFAULT_MAILBOX_ATTR` is interrupt safe, `DEFAULT_MAILBOX_INTR_ATTR` returns `DEFAULT_MAILBOX_ATTR` and does not raise `PROGRAM_ERROR`.

Alternatively, the mailbox attributes can be initialized to select the disable interrupts options by using one of the overloaded `MAILBOX_INTR_ATTR_INIT` subprograms.

References

"Mailbox Support Subprograms" on page 2-40

Services

The Ada Kernel has the following mailbox services:

```
function mailbox_init(
     m           : a_mailbox_t;
     slots_cnt   : positive;
     slot_len    : natural;
     attr        : a_mailbox_attr_t) return boolean;
     -- MAILBOX_INIT() allocates memory for slots_cnt messages
     -- where each message has a fixed length of slot_len bytes.
     --
     -- Returns TRUE if mailbox was successfully initialized.

procedure mailbox_destroy(m: a_mailbox_t);

function mailbox_read(m: a_mailbox_t; msg_addr: address;
     wait_time: duration) return boolean;
     -- Reads a message from a mailbox. Returns TRUE if message was
```

(Continued)

```
     -- successfully read.
     --
    -- If no message is available for reading, then returns according to
     -- the wait_time parameter:
     --  < 0.0          - returns when message was successfully read.
    --                  This may necessitate suspension of current task
     --                   until another task does mailbox write.
     --  = 0.0          - returns FALSE immediately
     --  > 0.0          - if the mailbox read cannot be completed
     --                   within "wait_time" amount of time,
     --                   returns FALSE.

    function mailbox_write(m: a_mailbox_t; msg_addr: address)
     return boolean;
     -- Writes a message to a mailbox. Returns FALSE if no slot is
     -- available for writing.

function mailbox_get_count(m: a_mailbox_t) return natural;
     -- Returns number of unread messages in mailbox.

function mailbox_get_in_use(m: a_mailbox_t) return boolean;
     -- Returns TRUE if any task is waiting to read from the mailbox.
    -- If the Ada Kernel is unable to detect this condition, it returns
     -- TRUE.
     --
     -- MAILBOX_GET_IN_USE() is called by the VADS EXEC
     -- V_MAILBOXES.DELETE_MAILBOX() service.
```

Mailbox Support Subprograms

The mailbox attributes can be initialized to select the disable interrupts options by using one of the overloaded `MAILBOX_INTR_ATTR_INIT` subprograms.

```
  procedure mailbox_intr_attr_init(
       attr            : a_mailbox_attr_t;
       disable_status  : intr_status_t := DISABLE_INTR_STATUS);

 function mailbox_intr_attr_init(
       attr            : a_mailbox_attr_t;
       disable_status  : intr_status_t := DISABLE_INTR_STATUS)
                                    return a_mailbox_attr_t;
```

```
procedure mailbox_intr_attr_init(
function mailbox_intr_attr_init(
     -- does an implicit
     --  "attr: a_mailbox_attr_t :=
     --                            new mailbox_attr_t;"
    disable_status  : intr_status_t := DISABLE_INTR_STATUS)
                                    return a_mailbox_a
                                                       ttr_t;
```

References

"package V_MAILBOXES — provide mailbox operations" on page 4-30

2.1.2.15 *Name*

Objects or procedures can be named to allow them to be shared across multiple programs.

Supports the VADS EXEC V_NAMES services and the BIND/RESOLVE services in the VADS EXEC V_MAILBOXES, V_MUTEXES and V_SEMAPHORES packages.

Types

STRING

A name can be any arbitrary Ada string.

PROGRAM_ID

Program containing object or subprogram.

ADDRESS

Location of the object or subprogram.

NAME_BIND_STATUS_T

Status returned by the name_bind service.

NAME_RESOLVE_STATUS_T

Status returned by the name_resolve service.

Services

The Ada Kernel has the following name services:

```
function name_bind(
    name    : string;
    prg     : program_id;
    addr    : address) return name_bind_status_t;
  -- Bind a name to the program_id and address of a procedure or object.
  --
  -- The name parameter can be any sequence of characters. An exact
  -- match is done for all name searches. (”MY_NAME” diffs from
  -- ”my_name”.)
  --
  -- The prg parameter should be set to NO_PROGRAM_ID if the name
  -- isn’t bound to a particular program or if the current program and
  -- stack limit switch logic are to be eliminated for an
  -- ada_krn_i.program_inter_call(). All procedures and objects in the
  -- kernel program are bound with prg implicitly set to NO_PROGRAM_ID.
  --
  -- If successful, name_bind returns ada_krn_defs.NAME_BIND_OK.
  -- Otherwise, it returns one of the following error codes also found in
ada_krn_defs:
  -- NAME_BIND_NOT_SUPPORTED
  --NAME_BIND_BAD_ARG
  --NAME_BIND_OUT_OF_MEMORY
  --NAME_BIND_ALREADY_BOUND
procedure name_resolve(
    name    : string;
    wait_time: duration;
    prg     : out program_id;
    addr    : out address;
    status  : out name_resolve_status_t);
  -- Resolve the name of a procedure or object into its program_id and
  -- address.
  --
  -- name_resolve first attempts to find an already bound name that
  -- exactly matches the name parameter. For a match, it returns
  -- immediately with the prg and addr out parameters updated and
  -- status set to ada_krn_defs.NAME_RESOLVE_OK. Otherwise, it
  -- returns according to the wait_time parameter:
      --  < 0.0           - waits indefinitely until the name is bound
      --  = 0.0           - returns immediately with status set to
```

(Continued)

```
      --                    NAME_RESOLVE_FAILED
      --  > 0.0            - if the name isn't bound within "wait_time",
      --                  returns with status set to NAME_RESOLVE_TIMED_OUT
  --
  -- If name services aren't supported or name_resolve was called with
  -- a bad argument, then, status is set to NAME_RESOLVE_NOT_SUPPORTED
  -- or NAME_RESOLVE_BAD_ARG.
>lt
Q  TASK             NUM  STATUS
   <interrupt 17>   5    waiting for interrupt
        handler at 0000046e8
   <interrupt 16>   4    waiting for interrupt
        handler at 0000046d0
   <interrupt 2>    3    waiting for interrupt
        handler at 0000046b8
   <debugger task>  2    waiting to exit
*  <main program>   1    executing
```

2.2 Passive Tasks

Passive tasks are a compiler/runtime optimization that reduces the overhead associated with an Ada task. The use of passive tasks usually results in improved performance for the Ada application.

An Ada task under SC Ada is implemented through services provided by the SC Ada RTS. These services create tasks, control rendezvous between tasks, and provide a variety of other capabilities. The Ada RTS provides each Ada task with a thread of control and with its own stack storage space.

These Ada tasks are called *active tasks* in this section, in contrast to passive tasks.

Passive and active tasks are implemented differently. Passive tasks are simply subroutines, and an entry call to a passive task is the same as a simple subprogram call. A passive task is said to exist only while it is in rendezvous with an active task. A passive task does not have a thread of control or its own stack storage space. While an active task is in rendezvous with a passive task, the passive task uses the thread of control and stack storage space from the active task.

Passive tasks are in essence little more than critical regions guarding a sequence of code. The sequence of code is the accept body. Passive tasks use mutexes to guard the critical region.

Passive tasks increase application performance in two ways. Passive task rendezvous is significantly simpler and faster than active task rendezvous. The reduction in the number of active tasks within an application increases kernel performance by reducing the number of tasks on the entry queues, run queues and by reducing the amount of memory consumed by the kernel.

The following topics are covered in this section:

2.2.1 Pragma Passive

A task is marked as a passive task using the implementation-defined pragma PASSIVE. The pragma can have zero, one, or two parameters as follows:

```
pragma PASSIVE;
pragma PASSIVE(ABORT_UNSAFE);
pragma PASSIVE(ABORT_SAFE);
pragma PASSIVE(ABORT_UNSAFE, mutex_attr'address);
pragma PASSIVE(ABORT_SAFE, mutex_attr'address);
```

An active task calling an ABORT_SAFE passive task entry is inhibited from being completed by an Ada abort until it finishes execution of the passive task's accept body. This inhibits the aborted task from holding a lock on a mutex that is never released.

Alternatively, if an active task calling an ABORT_UNSAFE passive task entry is aborted, the lock is never released and other active tasks are indefinitely blocked if they call an entry in the passive task.

Earlier SC Ada releases supported only the ABORT_UNSAFE option. The ABORT_SAFE option is slightly slower. If the first parameter is omitted, the default is ABORT_UNSAFE.

The second parameter is the address of a mutex attributes record. The passive task's critical region is protected by locking a mutex. The mutex attributes record is used to initialize the mutex. Omitting the second argument selects the default mutex attributes. For VADS MICRO, the default is to lock the mutex using a test-and-set instruction and to be FIFO queued when blocked and waiting for the mutex. (In earlier releases, this was specified through pragma PASSIVE(SEMAPHORE).)

The mutex attributes are defined in the Ada Kernel's ADA_KRN_DEFS package. MUTEX_ATTR_T is the type definition of the mutex attributes.

The mutex attributes are microkernel-dependent. See ada_krn_defs.a in standard for the options supported. (VADS MICRO locks the mutex by executing a test-and-set instruction or by disabling interrupts. It supports FIFO, priority, or priority inheritance waiting when the mutex is locked by another task. In the optional CIFO product, VADS MICRO also supports priority ceiling mutexes using the priority ceiling protocol emulation algorithm documented in the POSIX 1003.4a standard.)

The function DEFAULT_MUTEX_ATTR is provided to select the default mutex attributes.

To provide mutual exclusion by disabling all interrupts, use DEFAULT_INTR_ATTR. (In earlier SC Ada releases, this was specified using pragma PASSIVE(INTERRUPT).) If the underlying microkernel does not support interrupt attributes, the PROGRAM_ERROR exception is raised.

ada_krn_i.a has the following functions for initializing the mutex attributes:

```
function intr_attr_init(
    disable_status  : intr_status_t :=
DISABLE_INTR_STATUS)
                    return address;
function fifo_mutex_attr_init return address;
function prio_mutex_attr_init return address;
function prio_inherit_mutex_attr_init return address;
function prio_ceiling_mutex_attr_init return address;
```

Each of the above attribute initialization functions does an implicit allocation of the MUTEX_ATTR_T record and returns its address. ada_krn_i.a has additional overloaded subprograms for each of the initialization functions.

These initialization functions are supported for all versions of the Ada Kernel. If the mutex type is not supported by the underlying microkernel, the PROGRAM_ERROR exception is raised.

Figure 2-4 shows a passive task. It is part of a package providing buffer management services. It's ABORT_UNSAFE and uses the default mutex attributes.

```
package buffer_Pack
    type element_type is ...;
    task type buffer is
        entry Put( element : element_type );
        entry Get( element : out element_type );
        pragma PASSIVE;
    end buffer;
end buffer_pack;
```

Figure 2-4 Passive Task

Figure 2-5 shows a passive task using an attribute initialization function.

```
with ADA_KRN_DEFS;
package prio_pack is
    task prio_task is
        entry e1;
        pragma PASSIVE(ABORT_SAFE,
ADA_KRN_DEFS.PRIO_MUTEX_ATTR_INIT);
    end;
end prio_pack;
```

Figure 2-5 Passive Priority Queuing Task

Figure 2-6 shows two passive tasks whose critical regions are protected by disabling all interrupts.

```
with ADA_KRN_DEFS;
package interrupt_pack is
    task intr_task_1 is
        entry intr;
        pragma PASSIVE(ABORT_UNSAFE,
ADA_KRN_DEFS.DEFAULT_INTR_ATTR);
    end;
    task intr_task_2 is
        entry intr;
       pragma PASSIVE(ABORT_UNSAFE, ADA_KRN_DEFS.INTR_ATTR_INIT(
                DISABLE_STATUS =>
ADA_KRN_DEFS.DISABLE_INTR_STATUS));
    end;
end interrupt_pack;
```

Figure 2-6 Passive Interrupt Tasks

References

“Mutex” on page 2-2

2.2.2 Passive Task Portability

The compiler performs the passive task optimization only for task specifications marked with `pragma PASSIVE`. The compiler enforces a set of restrictions designed to force programs using passive tasks to execute identically whether or not the pragma is supported. Since Ada compilers usually ignore unsupported pragmas, porting code that uses `pragma PASSIVE` to other compilers should be easy.

Earlier versions of SC Ada supported `pragma PASSIVE` in a slightly different form. Some passive task bodies that were accepted by the older versions of VADS are not accepted by this and future versions. Code constructs that are no longer supported include:

- Passive tasks containing multiple `accept` statements.
- Passive tasks with `for` loops as the outermost statement

Tasks containing multiple `accept` statements can be recoded to use a selective wait. Tasks using `for` loops can be recoded to use unbounded loops.

Figure 2-7 demonstrates the transformation needed to turn a passive task containing multiple `accept` statements into a passive task with a guarded `select` statement. The "Old Version" passive task body was supported under earlier versions of SC Ada. The "New Version" passive task body is supported under the current version.

```
task Semaphore is
    entry seize;
    entry release;
    pragma PASSIVE;
end;
-- Old Version -- New Version

task body semaphore istask body semaphore is
begin                  seized: boolean := FALSE;
    loop               begin
    accept seize;loop
    accept release;select
    end loop;          when not seized =>
end;                           accept seize do
                                      seized := TRUE;
                                   end;
                               or when seized =>
                                   accept release do
                                      seized := FALSE;
                               end;
                               end select;
                            end loop;
                         end;
```

Figure 2-7 Passive Task Transformation

2.2.3 Passive Task Restrictions

A number of restrictions exist on the structure of a passive task. Some restrictions exist so that a task behaves in the same manner whether it is passive or active. Other restrictions are required to simplify the implementation or to allow unambiguous semantics.

If any of the restrictions are violated, the compiler emits warning messages and continues normal compilation. The compiler may skip illegal constructs or generate code to raise `TASKING_ERROR`, so it is advisable to heed the warnings and modify the code to eliminate the warnings.

Passive task bodies must have the following structure:

```
task body PT is
  <decls>
begin
  loop
     <accept or select stm>
  end loop;
end PT;
```

See the next section for a detailed discussion of the error handling for passive tasks. The following restrictions are in effect for passive tasks:

- The `loop` statement is required. Note that tasks coded in this fashion never terminate normally.
- The task body itself cannot have an exception handler. Handlers may be provided within `accept` statements within the passive task. Unhandled exceptions within the passive task `accept` body cause the SC Ada runtime system to mark the passive task as uncallable and subsequent entry calls are refused.
- The `<accept or select stm>` can be an `accept` statement or a selective wait. If it is a selective wait, it cannot contain `terminate` or `delay` alternatives and it cannot contain an `else` clause.
- The declarations within the passive task body can contain almost any Ada construct. Certain record type declarations are not supported within passive tasks. No other program units (such as subprograms, packages, or tasks) or other ”later declarative items” can be declared within a passive task.
- Passive tasks can be of named or anonymous task types.

- Passive tasks and passive task types must be declared immediately within library-level packages.
- Objects of named passive task types can appear in any legal context, including as record components or array elements. Passive task objects can be created with allocators.
- Passive tasks can be terminated only through the `abort` statement or by unhandled exceptions.
- When a passive task is terminated, all further attempts to rendezvous with the passive task cause `TASKING_ERROR` to be raised in the calling task. Tasks queued on the passive task's entries also have `TASKING_ERROR` raised within them.
- Storage associated with passive tasks cannot be reclaimed even if the task was created as the result of an allocator. This is a limitation of the current release.
- The entries within passive `select` statements can be guarded. This is true except for `accept` statements within the `select` statements that accept interrupt entries. Passive interrupt entries can never be guarded; they must always be open.
- Passive task entries cannot declare entry families in the current release. Support for entry families may be added in future releases.
- Delay statements can appear within a passive task body. However, when executed, they cause the calling active task to delay during rendezvous. This use of `delay` statements should be avoided.

2.2.4 Compiler Error Messages for Passive Tasks

The compiler emits two warning messages when it detects a violation of the passive task restrictions. The first message specifies the restriction and the second gives the recovery action taken by the compiler.

It is highly recommended that the compilation of units containing passive tasks be performed with compiler warning messages enabled. The compiler's recovery action can include generating code to raise `TASKING_ERROR`, causing the program to fail during execution.

If the compiler detects an illegal `pragma PASSIVE` in a task specification, the pragma is ignored. Code is generated for the task as an active task. The application containing the task runs as designed, but its performance suffers.

If the compiler detects an error in a passive task body before generating any code for the body, it generates code for "dummy" `accept` bodies. These dummy `accept` bodies are always closed, so that normal entry calls made to them block forever, while conditional or timed entry calls always fail. The application containing the task runs, but it may hang, terminate with deadlocked tasks, or experience another serious runtime error.

If the compiler detects an error in a passive task body after code generation for the body has already begun, it ignores the offending construct and instead generates code to raise `TASKING_ERROR`. It continues generating code for the rest of the passive task body. The application containing the passive task finds that the passive task terminates during rendezvous and further attempts only raise `TASKING_ERROR`.

2.2.5 Examples of Passive Task Errors

Figure 2-8 is of the warning messages produced during the compilation of a passive task specification containing an invalid `pragma PASSIVE`. The compiler continues the compilation normally after rejecting the pragma. If the program that contains this package is linked, the task `bv0113a_pack.pt1` is an active task.

```
  1:package bv0113a_pack is
  2:task pt1 is
  3:    entry e1;
  4:    pragma passive( illegal_argument );
A ------------------------------^
A:warning: RM Appendix F: arguments should be ABORT_(UN)SAFE or
task
          attributes
A:warning: RM Appendix F: pragma PASSIVE ignored; task will be an
         active task
  5:end pt1;
  6:end;
```

Figure 2-8 Passive Task Warning Messages

Figure 2-9 is of a passive task body that contains an illegal `select` statement. Delay alternatives are not supported within passive tasks. The compiler detected the error before it began code generation for the passive task body. The compiler issued dummy `accept` bodies instead of generating code for the passive task body as supplied.

```
   1:with report;
   2:package body bv0107a_pack is
   3:task body pt1 is
A ------^
A:warning: RM Appendix F: errors in passive task body; dummy accept
bodies emitted
   4:begin
   5:   loop
   6:      select
   7:          accept e1 do
   8:              report.failed( ”entry e1 accepted” );
   9:          end;
  10:      or
  11:          delay 1.0;
A --------------^
A:warning: RM Appendix F: DELAY is not a legal alternative in passive
task select
  12:          report.failed( ”delay alternative selected” );
  13:      end select;
  14:   end loop;
  15:end;
  16:end;
```

Figure 2-9 Passive Task with Illegal Select Statement

The following example is a case in which the compiler did not detect the invalid passive task body until after it began code generation for the passive task's accept statement. In this case the inner accept of entry `e2` is replaced

with code that causes TASKING_ERROR to be raised. This causes TASKING_ERROR to be raised whenever any other task attempts to rendezvous with entry e1.

```
   1:with report;
   2:package body bv0104a_pack is
   3:task body pt1 is
   4:begin
   5:   loop
   6:       accept e1 do
   7:           accept e2 do
A --------------^
A:warning: RM Appendix F: nested passive accepts not supported
B:warning: RM Appendix F: TASKING_ERROR will be raised
   8:               report.failed( ”nested entry pt1.e1.e2 accepted” );
   9:           end;
  10:       end;
  11:   end loop;
  12:end;
```

Figure 2-10 Invalid Passive Task Body

2.3 Ada Interrupt Entries as Interrupt Handlers

Section 13.5.1 in the Ada LRM defines the syntax and semantics for an interrupt entry. The SC Ada implementation is defined there with the following interpretations and restrictions:

- An interrupt entry cannot have any parameters.
- A passive task that contains one or more interrupt entries must always be trying to accept each interrupt entry, unless it is handling the interrupt. The task must be executing either an `accept` for the entry (if there is only one) or a `select` statement where the interrupt entry accept alternative is open as defined by Ada LRM 9.7.1(4). This is not a restriction on normal tasks (i.e., signal ISRs).
- An interrupt acts as a conditional entry call in that interrupts are not queued.
- No additional requirements are imposed for a `select` statement containing both a `terminate` alternative and an `accept` alternative for an interrupt entry.
- Direct calls to an interrupt entry from another task are allowed and are treated as a normal task rendezvous.
- Interrupts are not queued.

The address clause for an interrupt entry does not specify the priority of the interrupt. It points to an `INTR_ENTRY_T` record defined in `ADA_KRN_DEFS`. The `INTR_ENTRY_T` record contains two fields: the interrupt vector and the task priority for executing the interrupt entry's accept body.

In earlier SC Ada releases the address clause specified the interrupt vector. To preserve backward compatibility, the parameter `OLD_STYLE_MAX_INTR_ENTRY` was added to the configuration table in `v_usr_conf_b.a`. If the value in the address clause is `<= OLD_STYLE_MAX_INTR_ENTRY`, it contains the interrupt vector value and not a pointer to an `ADA_KRN_DEFS.INTR_ENTRY_T` record. Setting `OLD_STYLE_MAX_INTR_ENTRY` to `MEMORY_ADDRESS(0)` disables the old way of interpretation.

The default value for `OLD_STYLE_MAX_INTR_ENTRY` is `MEMORY_ADDRESS(511)`.

ada_krn_defs.a has the following function for initializing the interrupt entry:

Table 2-1

```
function intr_entry_init(
    intr_vector : intr_vector_id_t;
    prio        : priority := priority'last) return address;
```

The above function does an implicit allocation of the INTR_ENTRY_T record and returns its address. ada_krn_defs.a has additional overloaded subprograms for initializing the INTR_ENTRY_T record.

For OLD_STYLE_MAX_INTRY_ENTRY = MEMORY_ADDRESS(511), the following two interrupt entries are identical:

```
task a is
    entry ctrl_c;
    for ctrl_c use at ada_krn_defs.intr_entry_init(
                            intr_vector => 2,
                            prio => priority'last - 1);
end;
or
task a is
    pragma priority(priority'last -1);
    entry ctrl_c;
    for ctrl_c use at system.memory_address(2);
end;
```

Note that for the old style interrupt entry, the task priority for executing the interrupt entry's accept body is always the priority of the attached task containing the interrupt entry (per POSIX 1003.5.) The priority cannot differ as it can for the new style.

Interrupt entries are defined in normal Ada tasks (referred to as *signal ISRs*) or in tasks to which pragma PASSIVE has been applied (referred to as *passive ISRs*).

For an interrupt entry in a passive task, a check is made during elaboration to see if the passive task's mutex can be locked from an ISR. If the mutex is not lockable, the PROGRAM_ERROR exception is raised. Normally, a passive task with an interrupt entry protects its critical region by disabling interrupts. This is achieved by setting the second parameter of pragma PASSIVE to

ADA_KRN_DEFS.DEFAULT_INTR_ATTR to disable all interrupts or by using the mutex attribute initialization function, ADA_KRN_DEFS.INTR_ATTR_INIT to disable some of the interrupts.

Here's an example of a passive task with interrupt entries:

```
with ADA_KRN_DEFS;
with SYSTEM;
package interrupt_entry_pack is
    task intr_task_1 is
        entry ctrl_c;
        -- old style passive tasks
        for ctrl_c use at system.memory_address(2);
        pragma PASSIVE(ABORT_UNSAFE,
ADA_KRN_DEFS.DEFAULT_INTR_ATTR);
    end;
    task intr_task_2 is
        entry usr1;
        for usr1 use at ada_krn_defs.intr_entry_init(
                                intr_vector => 16,
                                prio => priority'last);
       pragma PASSIVE(ABORT_UNSAFE, ADA_KRN_DEFS.INTR_ATTR_INIT(
                DISABLE_STATUS =>
ADA_KRN_DEFS.DISABLE_INTR_STATUS));
    end;
end interrupt_entry_pack;
```

Figure 2-11 Passive Interrupt Entries

The accept body for a passive interrupt entry is executed to actually handle the interrupt. The accept body is executed immediately when the interrupt occurs (without any task rescheduling). Interrupts are still disabled according to the CPU interrupt-processing mechanism. A direct consequence is that if the accept for an interrupt entry is in a select, it must always be an open alternative. The program is abandoned with an unhandleable TASKING_ERROR exception if the passive task is not trying to accept the interrupt entry when the interrupt occurs. Since the accept body of the passive ISR is actually handling an interrupt, it has the same restrictions as a conventional ISR with respect to the kernel services it may call.

References

Queued Interrupts, Ada LRM 13.5.1(2) and 13.5.1(6)

Select Statement, Ada LRM 13.5.1(3)

"Passive Tasks" on page 2-44

2.4 Program Exit or Deadlock

A program normally exits after the main subprogram returns and when the following two conditions are satisfied:

- No task is ready to run.
- No task is suspended at an Ada `delay` statement, at a call to `XCALENDAR.DELAY_UNTIL`, at a timed entry call, or at a `select` statement with an open delay alternative.

However, to accommodate interrupt entries and attached ISRs, either of the following conditions inhibits a program from exiting:

- A task is suspended at an `accept` or `select` with an open interrupt entry. Interrupt entries at a `select` with `terminate` are considered closed. This is not applicable to interrupt entries in a `PASSIVE` task.
- The Ada RTS's `EXIT_DISABLE_FLAG` is `TRUE`. This flag is initialized to `FALSE`. It is normally set to `TRUE` by the application program if it attaches an ISR or has `PASSIVE` tasks with interrupt entries. This flag can be read or set by the VADS EXEC services in V_XTASKING, `CURRENT_EXIT_DISABLED`, and `SET_EXIT_DISABLED`. Alternatively, this flag may be referenced using `V_I_TASKS` in `standard` (`GET_EXIT_DISABLED_FLAG` and `SET_EXIT_DISABLED_FLAG`)

Note – In earlier SC Ada versions, signals mapped to an interrupt entry at a simple accept did not inhibit the program from exiting. If you still want that effect , change the simple `accept` to a "`select` or `terminate`" as illustrated in this example:

```
loop
  select
    accept ctrl_c do
      -- ctrl_c logic, such as following call to terminate
      -- the program
      v_i_tasks.terminate_program(0);
    end;
  or
    terminate;
  end select;
end loop;
```

A program deadlocks if the main subprogram has not returned and the above program exit conditions are satisfied and neither of the interrupt conditions for inhibiting program exit is satisfied.

2.5 SC Ada Archive Interface Packages

SC Ada uses an archive to provide many of the runtime services that are used by the compiler. Interfaces to many of these routines are supplied through the `V_I_*` packages in `standard`.

Note – Most of the services described in the listing of the `standard` library are provided by a higher level interface in the VADS EXEC library. It is possible to eliminate that extra layer of software by calling these services directly. One note of caution however, using the VADS EXEC services better protects you from changes in future versions of SC Ada since Sun Microsystems, Inc. is committed to preserving the VADS EXEC interfaces.

The following packages are described on page

V_I_ALLOC[note 1]	V_I_MBOX[note 1]	V_I_TASKOP
V_I_BITS	V_I_MEM	V_I_TASKS
V_I_CALLOUT[note 1]	V_I_MUTEX[note 3]	V_I_TIME
V_I_CIFO	V_I_PASS	V_I_TIMEOP
V_I_CSEMA[note 1]	V_I_RAISE	V_I_TYPES
V_I_EXCEPT[note 2]	V_I_SEMA[note 1]	V_SEMA[note 1]
V_I_INTR[note 1]	V_I_SIG	V_TAS
V_I_LIBOP		

Note – 1: These packages provide compatibility with earlier SC Ada releases by layering on the Ada Kernel's type definitions and subprograms found in the `ADA_KRN_DEFS` and `ADA_KRN_I` packages.

Note – 2: `package V_I_EXCEPT` contains the interface to the Ada exception services to:
a. get the `ID` and program counter of the current Ada exception
b. return the string name of an exception `ID`

c. install a callout to be called whenever an exception is raised

`package V_I_EXCEPT` also contains the interface to the core dump services to:
a. produce a core file from anywhere in your program
b. enable the generation of a core file for an unhandled Ada exception
c. enable exception traceback registers to be saved for an unhandled core dump

Note – 3: The `V_I_MUTEX` package interfaces to the mutex and condition variable services that are Ada tasking `ABORT_SAFE`. After locking a mutex, the task is inhibited from being completed by an Ada abort until it unlocks the mutex. However, if the task is aborted while waiting at a condition variable (after an implicit mutex unlock), it is allowed to complete. The `V_I_MUTEX` services also address the case where multiple `ABORT_SAFE` mutexes can be locked. A task is inhibited from being completed until all the mutexes are unlocked or it does a condition variable wait with only one mutex locked. There are services to init, destroy, lock, trylock, and unlock an `ABORT_SAFE` mutex. There are services to init, destroy, wait on, timed wait on, signal, and broadcast an `ABORT_SAFE` condition variable. In the optional CIFO product, there are also services to init, set priority of, and get priority of an `ABORT_SAFE` priority ceiling mutex.

2.6 Tasking

The following is a tasking example. The concept is taken from the Ada LRM 9.12. Following it are discussions of its runtime interactions, including creation, activation, startup, `delay` statements, `entry` calls, `accept` and `select` statements, completion, and termination.

```
 1:with text_io;
 2:procedure buffer is
 3:   task buffer is
 4:      entry read(c: out character);
 5:      entry write(c: in character);
 6:   end;
 7:   task producer is
 8:   end;
 9:   task consumer is
10:   end;
11:   task body producer is
12:   begin
13:      ...
14:   buffer.write(c);
15:      ...
16:   end;
17:   task body consumer is
18:      ...
19:   begin
20:      ...
21:         select buffer.read(c);
22:            ...
23:         else delay(0.001);
24:         end  select;
25:      ...
26:   end;
27:   task body buffer is
28:   begin
29:      ...
30:         select when count < pool_size =>
31:            accept write(c: in character) do ...  end;
32:            ...
33:         or  when count > 0 =>
34:            accept read(c: out character) do ...  end;
35:            ...
36:         or terminate;
```

(Continued)

```
37:         end select;
38:      ...
39:   end buffer;
40:begin
41:   ...
42:   while buffer'callable loop
43:      delay(0.001);
44:   end loop;
45:   ...
46:end;
```

Figure 2-12 Tasking

2.6.0.1 *Task Creation*

Tasks come into being in two stages: creation and activation. Tasks are created by the elaboration of declarations of objects either defined as tasks or containing an instance of a task type. Both tasks and task type instances are treated identically by the runtime system.

In the example, the calls emitted by the compiler for lines 3-6, creating the buffer task, are:

```
    function ts_init_activate_list return a_list_t;
    function ts_create_master (
        fp                : address;
        generic_param     : address   )
    return a_master_t;
    function ts_create_task_and_link(
        master            : a_master_t;
        prio              : integer;
        stack_size        : natural;
        start             : address;
        entry_count       : integer;
        activation_list   : a_list_t;
        generic_param     : address;
        task_attr         : ada_krn_defs.a_task_attr_t;
        has_pragma_prio   : integer) return a_task_t;
```

where:

```
a_list_t                    pointer to task activation list head
a_master_t                  pointer to master structure
a_task_t                    pointer to task control data structure
```

The interface to the above and all the Ada tasking routines called by the compiler is provided in the file `v_i_taskop.a` in the `standard` directory. To see the actual code emitted by the compiler in calling the subprograms, enter the debugger and use the **li** (list instruction) command.

The call to `TS_INIT_ACTIVATE_LIST` initializes a task activation list for this scope (for all the tasks created directly within the declarative part of the test procedure). A pointer to the list head is returned.

The call to `TS_CREATE_MASTER` initializes a master structure. This structure will keep track of all the tasks that must terminate before this scope can be completed (in this example, before the program can be completed). A pointer to the master structure is returned.

The call to `TS_CREATE_TASK_AND_LINK` causes the creation of a task control data structure. A pointer to this structure is returned. Parameter values are assigned by the compiler, with `STACK_SIZE`, priority, and `TASK_ATTR` given by default or by Ada pragmas. The task is linked to the activation list but is not activated at this point.

The return value from this task creation function is the task's descriptor. The compiled code stores this descriptor in the storage class implied by the scope of the object declaration. It uses this descriptor as the task's value. Whenever the task must be named by the generated code to the RTS, this descriptor is passed to the RTS. This descriptor is really the pointer to the task's record, an RTS data structure. The generated code does not take advantage of this fact.

The SC Ada debugger now knows of this new task, so the debugger command `lt` elicits the list:

```
Q#  TASK            NUM       STATUS
    buffer'2        2         not yet active
*   <main program>  1         executing
```

The other tasks are created in a similar fashion, for `PRODUCER` by the code at line 7 and `CONSUMER` by the code at line 9. Now `lt` gives:

```
Q#  TASK            NUM     STATUS
    consumer        4       not yet active
    producer        3       not yet active
    buffer'2        2       not yet active
*   <main program>  1       executing
```

2.6.0.2 *Task Activation*

At the `begin` that follows a declarative part that contains tasks, the program calls `TS_ACTIVATE_LIST`. The activation list is the one being used for the declarative part. The RTS activates the tasks on the list and returns. The compiler calls `TS_ACTIVATION_EXCEPTIONS` immediately following the call to `TS_ACTIVATE_LIST`. This makes the activating task wait for the children that were just activated to run and elaborate their declarations. Each child calls `TS_ACTIVATION_COMPLETE`, which increments a count of successfully activated tasks. The last child to do this wakes up the activating task.

At line 40 the following calls are emitted:

```
procedure ts_activate_list(
        activation_list : a_list_id;
        is_allocator    : integer);
function ts_activation_exceptions return act_status_t;
```

where:

```
type act_status_t is (
         act_ok,
         act_elab_err,
         act_except,
         act_elab_err_act_except);
```

If TS_ACTIVATION_EXCEPTIONS returns ACT_ELAB_ERR, then TASKING_ERROR exception is raised. If TS_ACTIVATION_EXCEPTIONS returns ACT_EXCEPT or ACT_ELAB_ERR_ACT_EXCEPT, then PROGRAM_ERROR exception is raised.

The compiler emits the following call for TS_ACTIVATION_COMPLETE:

```
procedure ts_activation_complete(act_status :
act_status_t);
```

When all the tasks from the above example activate and elaborate correctly, the following output is given by an lt command at an instruction breakpoint at the instruction following the call to TS_ACTIVATE_EXCEPTIONS (just after source instruction 40):

```
Q#  TASK            NUM     STATUS
D1  consumer        4       suspended at delay
        on delay queue, until day: 0 sec: 0.399
R1  producer        3       ready
   buffer'2      2      suspended at select (terminate possible)
        open entries:  write  read
*   <main program>  1       executing
```

It is possible for the elaboration of the declarative part of a task body to cause an exception. In this case, TS_ACTIVATION_COMPLETE is called by an anonymous exception handler with a parameter indicating abnormal activation. The following calls are emitted for this implicit anonymous handler:

```
procedure ts_exception_master(master : master_id);
procedure raise_exception(identifier : system.address);
```

where:

The interface to RAISE_EXCEPTION is provided in v_i_raise.a found in standard. The external name assigned to the RAISE_EXCEPTION procedure is RAISE.

The handler saves the exception and calls the TS_EXCEPTION_MASTER service to await termination of any subtasks of the current task (there are none in example PRODUCER). It then raises the exception again.

This wakes up the activating task (which in this case is the main program) even though some of the sibling activated tasks may not yet have called TS_ACTIVATION_COMPLETE. The newly activated tasks are completed and terminated by the TS_ACTIVATION_EXCEPTIONS service executing for that activating main program; it returns to the main program with a non-zero value:

```
0  all is ok during activation
1  elaboration error, raising TASKING_ERROR
2  activation error, raising PROGRAM_ERROR
```

An lt command issued from an instruction breakpoint immediately following TS_EXCEPTION_MASTER in the anonymous handler shows the following after an exception was raised in the declarative part:

```
Q#  TASK       NUM    STATUS
D1  consumer       4    suspended at delay
            on delay queue, until day 0 sec: 0.161
    producer       3   terminated
    buffer'2       2       suspended at select (termination
possible)
            open entries:  write
*   <main program>1    executing
```

2.6.0.3 Task Startup

The code for task startup precedes the text for the task body. The code for `PRODUCER` can be displayed by typing the command `li 11` in the debugger.

The task body code has the same procedure startup as other subprograms, getting stack space, checking for stack limits, and setting up the addressing environment for reaching entities in other parts of the program (these are copied from the caller's stack frame).

The only purely tasking activity before elaboration is the call to `TS_TID` to get and record the `ID` of this task. This value is the task data structure address in the kernel space.

After any elaboration for the task is complete, a call to `TS_ACTIVATION_COMPLETE` informs the parent task of the success of the activation. Were this the last task on the activation list, the activating task (the main program here) would be placed on the run queue for further execution.

2.6.0.4 Delay Statements

Simple `delay` statements (that is, those not in the `else` part of a `select`) are implemented as calls to the `TS_DELAY` service:

```
procedure ts_delay(delay_val : in duration);
```

where:

`DELAY_VAL` is the internal representation for duration in units of 0.1 milliseconds.

The fixed-point number is pushed as the parameter to TS_DELAY. The following lt command output illustrates the main program after it has requested a delay at line 43:

```
Q#  TASK        NUM     STATUS
    consumer    4       in rendezvous
    producer    3       suspended calling buffer'2[04cd14].write
*   buffer'2    2       executing
            in rendezvous with consumer[05b7f8] at entry entry_0
D1 <main program>  1       suspended at delay
            on delay queue, until day: 0 sec: 0.8441
```

Tasks requesting delays are put on a DELAY_QUEUE. Each timer interrupt checks each task on this queue to see if CURRENT_TIME has passed its time to awaken; each such task is then queued on the run queue.

2.6.0.5 *Task Entry Calls*

The compiler translates unconditional and untimed entry calls into calls to TS_CALL. TS_CALL takes three parameters: the task descriptor of the task being called, the entry ID of the entry being called (when the compiler sees a task specification it assigns integers, starting at one, to the declared entries), and the address of a parameter block for the call (parameter blocks are made to look like the stack memory for subprogram parameters).

task PRODUCER contains a simple entry call that provides characters to task BUFFER.

The entry call has the following subprogram interface:

```
procedure ts_call (
    called_task        : in task_id;
    called_entry       : in integer;
    param_block        : in address);
```

The TS_CALL routine tries to do an immediate rendezvous if possible. Rendezvous is possible if the called task, BUFFER, is suspended at an accept or at a select statement and is waiting at the entry being called. For

immediate rendezvous, control is transferred to the called task. This is almost like calling it as a subprogram except that when the rendezvous is completed, both the called task and the calling task are able to execute.

The following is the `lt` output during a rendezvous:

```
Q#  TASK            NUM     STATUS
D2  consumer        4       suspended at delay
        on delay queue, until day: 0 sec: 0.571
    producer        3       in rendezvous buffer'2[2].write
*   buffer'2        2       executing
        in rendezvous with producer[3] at entry write
D1  <main program>  1       ready
```

If the call cannot be done immediately, it is because the called task is not waiting at an `accept` or `select` or because it is not waiting for a call of the entry being called. The called task becomes suspended on the entry queue, waiting for the called task to accept that entry. The following `lt` command illustrates what happens when PRODUCER suspends waiting for BUFFER to accept a WRITE:

```
Q#  TASK       NUM     STATUS
D1  consumer   4       suspended at delay
            on delay queue, until day: 0 sec: 0.121
    producer   3       suspended calling buffer'2[2].write
*   buffer'2   2       executing
   <main program> 1       ready
```

In the example, eventually BUFFER accepts PRODUCER. An `lt` output at that time is identical to an immediate rendezvous except that the status of PRODUCER shows that it is suspended:

```
Q#  TASK            NUM       STATUS
    producer       3        suspended   calling buffer'2[2].write
```

Conditional entry calls are implemented with the TS_CONDITIONAL_CALL service. It is the same as a TS_CALL except that if immediate rendezvous is not possible, it returns FALSE. Otherwise it returns TRUE after the rendezvous.

Timed entry calls are implemented with the TS_TIMED_CALL service. Its additional parameter passes a delay duration. If immediate rendezvous is not possible, it suspends after setting up a delay for the requested duration. Whenever a timer interrupt follows, a task still on the DELAY_QUEUE can be awakened as though from a delay.

The conditional and timed entry calls have the following subprogram interfaces:

```
function ts_conditional_call (
    called_task        : in task_id;
    called_entry       : in integer;
    param_block        : in address) return boolean;
function ts_timed_call (
    timeout            : in duration;
    called_task        : in task_id;
    called_entry       : in integer;
    param_block        : in address) return boolean;
```

2.6.0.6 *Accept and Select Statements*

The example has no simple `accept` statements, but these points are similar to those of the more complex `select` statement.

`select` statement code begins with evaluation of the guards for the `accept` alternatives. Compiled code builds a list of the open alternatives in a data structure called an *entry list*. An entry list is an array of integers. The first integer corresponds to the first alternative in the `select` statement, the second integer corresponds to the second alternative, and so on. If the guard for an entry is closed, a zero is put in the entry list element corresponding to the alternative. If the guard is open, the integer corresponding to the entry being accepted is put in the element corresponding to the alternative.

After the ENTRY_LIST is built, a call is generated to one of the TS_SELECT routines:

```
ts_select             simple selects (no else part)
ts_select_terminate   an else part with a terminate
ts_select_else        an else part (conditional accept)
ts_select_delay       an else part with a delay
```

In this case, TS_SELECT_TERMINATE is called, with the address and length of ENTRY_LIST, a Boolean that is true if the terminate alternative is open and space reserved for two OUT parameters (the result of the select process and the associated parameter block).

The TS_SELECT procedure examines its entry queue to see if any task is waiting for any open entry. If so, immediate rendezvous is possible with the first such task. Otherwise, the task suspends until another task calls an open entry (for all selects) or until its delay expires (for selects with delay alternatives) or until all other dependent tasks are ready to terminate (for selects with open terminate alternatives). selects with else parts fall immediately to their else part unless immediate rendezvous is possible.

```
Q#  TASK             NUM     STATUS
R2  consumer         4       ready
*   producer         3       executing code
    buffer'2       2      suspended at select (terminate possible)
         open entries:  write  read
R1  <main program>  1       ready
```

When the rendezvous takes place, control is returned back to the acceptor task after its call to a TS_SELECT subprogram. At the end of the accept body, the TS_FINISH_ACCEPT procedure is called to allow both the acceptor and caller tasks to execute in parallel. If an unhandled exception is raised in the accept body, it is also propagated back to the caller task.

The accept, select, and finish accept calls have the following subprogram interfaces:

```
function ts_accept(
        accepting_entry    : in    integer) return address;
    procedure ts_select(
        user_entry_list    : in  a_entry_record_t;
        elist_len          : in  integer;
        param_block        : out address;
        result             : out integer);
    procedure ts_select_terminate(
        user_entry_list    : in  a_entry_record_t;
        elist_len          : in  integer;
        termin_open        : in  integer;
        param_block        : out address;
        result             : out integer);
    procedure ts_select_else(
        user_entry_list    : in  a_entry_record_t;
        elist_len          : in  integer;
        param_block        : out address;
        result             : out integer);
    procedure ts_select_delay(
        user_entry_list    : in  a_entry_record_t;
        elist_len          : in  integer;
        dlist_len          : in  integer;
        param_block        : out address;
        result             : out integer);
    procedure ts_finish_accept(
        exception_occurred : in  integer;
        exception_string   : in  address);
    where:
 a_entry_record_t  pointer to entry list
```

2.6.0.7 *Task Completion and Termination*

Tasks come to an end in two stages: first they complete and then they terminate. Once a task is completed, it does not run again. A completed task becomes terminated when all tasks dependent on it either complete or agree to terminate.

The following call is emitted at the end of PRODUCER, a task that has no descendants and simply executes through to completion:

```
procedure ts_complete_task;
```

The TS_COMPLETE_TASK service actually takes tasks all the way through to termination, checking and possibly waiting for any dependent tasks.

In the example, breakpoints can occur just after all the subtasks have terminated or are waiting at a terminate alternative:

```
Q#  TASK            NUM     STATUS
    consumer        4       terminated
    producer        3       terminated
   buffer'2      2      suspended at select (terminate possible)
       open entries:  write
*   <main program>  1       executing
```

BUFFER cannot terminate yet because the main program is still executing; it can still call one of the open entries of BUFFER. The code for the main program tries to shut down all the dependent tasks with a call to the TS_COMPLETE_MASTER service:

```
procedure ts_complete_master(master : master_id);
```

The main program now returns for completion of the whole program, as is described under program exit.

2.7 Fast Rendezvous Optimization

Normally the `accept` body of an Ada rendezvous is executed only in the context of the acceptor task. The fast rendezvous optimization also executes the `accept` body in the context of the caller task. This optimization reduces the number of thread context switches that need to be executed by the underlying microkernel.

Note that if you are using the CIFO archive instead of the default, the fast rendezvous optimization is inhibited.

Here's an overview of the optimization: if the acceptor task gets to the `accept` statement before the caller task makes the call, the acceptor task saves its register and stack context, switches to a wait stack, and does an `ADA_KRN_I.TASK_WAIT`. When the caller task gets around to doing the `accept` call, it saves its register and stack context, restores the acceptor task's register and stack context and returns to execute the `accept` body. When the end of the `accept` body is reached, the caller task overwrites the current register and stack context into the acceptor task's area, does an `ADA_KRN_I.TASK_SIGNAL` of the acceptor task, restores the caller task's register and stack context, and returns to the code in the caller task. Eventually, when the signaled acceptor task is scheduled to run, it restores the acceptor task's register and stack context (this context was updated by the caller task to be at the point where the call was made to finish the `accept` body) and returns to the code in the acceptor task after the call was made to finish the `accept` body.

Two configuration parameters have been added to `v_usr_conf` on behalf of the fast rendezvous optimization:

`FAST_RENDEZVOUS`

setting this parameter to `TRUE` enables the fast rendezvous optimization. This parameter would need to be set to `FALSE` only for multiprocessor Ada, where the accept body must execute in the acceptor task bound to a processor. It defaults to `TRUE`.

`WAIT_STACK_SIZE`

This parameter specifies how much stack is needed when the acceptor task switches from its normal task stack to a special stack it can use to call `ADA_KRN_I.TASK_WAIT`.

When using the debugger, the fast rendezvous optimization (`accept` body is executed by the caller task) has a few subtle differences from the normal rendezvous case (`accept` body is executed by the acceptor task).

Here's an example to illustrate the differences. There are two tasks doing a repetitive rendezvous. The caller task is `RENDEZVOUS_SEND`. The acceptor task is `RENDEZVOUS_RECEIVE`. A breakpoint has been placed in the acceptor body. There are two cases: either the breakpoint is reached when the acceptor task (`RENDEZVOUS_RECEIVE`) is executing the `accept` body or the caller task (`RENDEZVOUS_SEND`) is executing the acceptor body. Here's the debugger output for the two cases.

Case 1: At breakpoint when the acceptor task is executing the accept body (normal rendezvous)

```
[1]  stopped at "/vc/test/task_rend2.a":15 in rendezvous_receive

>lt
Q  TASK                NUM        STATUS
   rendezvous_send     3          in rendezvous
        rendezvous_receive[2].receive_item
*  rendezvous_receive  2          executing
        in rendezvous with rendezvous_send[3] at
           entry receive_item
>lt all
Q  TASK                NUM        STATUS
   rendezvous_send     3          in rendezvous
rendezvous_receive[2].receive_item
        thread id        = 14002bce8
        Ada tcb address  = 140026fd8
        Static priority  = 0
        Current priority = 0
        parent task : <main program>[1]

*  rendezvous_receive  2          executing
   in rendezvous with rendezvous_send[3] at entry receive_item
        ENTRY            STATUS   TASKS WAITING
        receive_item              - no tasks waiting -
        thread id        = 140026de0
        Ada tcb address  = 1400220d0
```

(Continued)

```
        Static priority  = 0
        Current priority = 0
        parent task : <main program>[1]
```

Case 2: At breakpoint when the caller task is executing accept body (fast rendezvous)

```
[1]  stopped at "/vc/test/task_rend2.a":15 in rendezvous_receive

>lt
Q  TASK                NUM        STATUS
   rendezvous_send     3          doing rendezvous
        for rendezvous_receive[2].receive_item
*  rendezvous_receive  2          executing
       in rendezvous via rendezvous_send[3] at entry receive_item
>lt all
Q  TASK               NUM         STATUS
   rendezvous_send    3           doing rendezvous
        for rendezvous_receive[2].receive_item
        thread id        = 14002bce8
        Ada tcb address  = 140026fd8
        Static priority  = 0
        Current priority = 0
        parent task : <main program>[1]

*  rendezvous_receive  2         executing
       in rendezvous via rendezvous_send[3] at entry receive_item
        ENTRY         STATUS   TASKS WAITING
        receive_item           - no tasks waiting -
        thread id        = 140026de0
        Ada tcb address  = 1400220d0
        Static priority  = 0
        Current priority = 0
        parent task : <main program>[1]
```

Here are the subtle differences for the fast rendezvous, case 2:

- Even though the breakpoint occurred in the RENDEZVOUS_SEND task, RENDEZVOUS_RECEIVE is still displayedas the current breakpointed task. You can select the caller task and still get the caller's call stack and see where it was making the rendezvous call from.
- The STATUS for RENDEZVOUS_SEND is "doing rendezvous" instead of "in rendezvous". This indicates that the caller task is executing the accept body.
- The second line for RENDEZVOUS_RECEIVE is "in rendezvous via" instead of "in rendezvous with". This indicates that the caller task is executing the accept body.

The only misleading piece of information is the current underlying thread that is executing. The debugger says that RENDEZVOUS_RECEIVE is the currently executing task. From this you would assume that its thread, 2, is the current one. However, for the fast rendezvous case, it is really RENDEZVOUS_SEND's thread, 3.

"A place for everything
and everything in its place."
Samuel Smiles

Memory Management and Allocation 3

This chapter describes the SC Ada implementation of the Ada memory requirements and the dynamic memory allocation / deallocation support available to the SC Ada user.

3.1 Memory Management/Requirements Implementation

The Ada language has many explicit and implicit memory requirements. Explicit requirements include object declarations and the `new` allocator. Implicit requirements include queues and blocks for tasking control and intermediate storage for initializations. This section describes how SC Ada implements these memory requirements.

Memory requirements are met from one of three areas: static data, the program heap, and the program stack. Use of static data is restricted to what can be allocated at compile time. The program stack is typically restricted to local usage because stack offsets must be known. The program heap is the most flexible. It can be used for any memory requirement, but excessive heap use degrades performance.

3.1.1 Explicit Memory Requirements

Table - 1 lists the explicit memory requirements of Ada and describes how they are implemented by SC Ada.

Table - 1 Explicit Memory Requirements

Explicit Memory Requirement	**Implementation**
Objects declared in package specifications or bodies (not generic).	Placed in static data. Earlier versions of SC Ada placed objects >500 bytes on the heap, but this is no longer done. The only exception to this is unconstrained objects, which are placed on the heap.
Objects declared in local declare blocks and subprograms.	Placed on the stack.
Objects created through the use of the new allocator.	Placed on the heap except for a constant object of an access type declared in a nongeneric package spec or body and initialized with a new allocator. If the initialization is static, the object is placed in static data.
Objects declared in the specification or body of a generic package.	Placement depends on how the generic is instantiated. If instantiated in a nonlocal context (within or as a library-level package), the objects are placed in static data. If instantiated within a local context (subprogram or local declare block), objects are placed on the stack.

3.1.2 Implicit Memory Requirements

Table - 2 lists the implicit memory requirements of Ada and describes how they are implemented by SC Ada.

Table - 2 Implicit Memory Requirements

Implicit Memory Requirement	**Implementation**
Task stacks	Comes off the heap, except for the main task, which uses the program stack. The RTS keeps track of stacks and limits for each task. This can cause serious problems when mixing Ada tasking with other languages. For example, if an Ada task makes a call to a C function that uses more stack than is allocated to the Ada task, the C function corrupts the adjacent heap memory. This corruption is very difficult to track down. This is not a serious problem if tasking is not used, because the program stack is better protected.
Nonstatic aggregate assignments	Assignments to any record or array type is completely checked before any target modification is performed. Each component of the record or array type is constraint-checked. To do this, a temporary image of the destination is built on the program stack. If all components are assigned without an exception, the temporary image is copied to the true destination. This will occur even for the simplest aggregate assignment.
RTS Bookkeeping Cross-Development Kernel	The kernel can have its own stack or it can make use of the current task stack, depending on the target architecture. The kernel has its own allocator for dynamic allocations, such as task control blocks.
RTS Bookkeeping Self-Host Kernel	The kernel uses the current task stack for its local memory requirements and has its own allocators for dynamic allocations such as task control blocks.

3.2 Heap Management

Heap memory is managed through a simple interface using `pragma INTERFACE` and `pragma EXTERNAL_NAME`. When the compiler identifies a need for heap memory, a call is made to a routine identified by the symbol `AA_GLOBAL_NEW`. When `UNCHECKED_DEALLOCATION` is performed on an object, a call is made to a routine identified by the symbol `AA_GLOBAL_FREE`. After this, as part of the elaboration, the compiler generates a call to a routine identified by the symbol `AA_GLOBAL_INIT`. The user is free to supply these routines, thus implementing any memory management algorithm desired.

These heap management routines require the use of another routine to provide them chunks of memory to manage. This routine is identified by the `GET_HEAP_MEMORY_CALLOUT` component of the configuration table within the `v_usr_conf` package. The default is to call the routine `V_GET_HEAP_MEMORY`.

For self-host applications, all programs use a single central-allocation service, `sbrk(2)`. Use of `sbrk(2)` guarantees mutex protection for allocations. In addition, we. supply a package called MALLOC that provides mutex-protected allocation routines.

The current default heap memory implementations described above provide a fast mechanism for getting and reusing memory. They also perform the coalescing of adjacent free blocks. This prevents large amounts of memory fragmentation over time that eventually exhausts memory.

There are several alternatives to the SC Ada default allocation scheme. The first alternative uses pool allocation instead of heap allocation. This alternative is described in the documentation. The second alternative is to make use of `package V_MEMORY` within VADS EXEC. Use of this package does not replace the default scheme entirely so task stacks and applications of the `new` allocator still use `AA_GLOBAL_NEW`. However, this package provides pool-based allocation schemes which, with instantiated generics, create allocators and deallocators. These pools are used to implement a crude form of garbage collection since, when a pool is deallocated, all the objects allocated from that pool are also deallocated. This method is dangerous because there is no check to see if any objects in the pool are still in use.

References

"Memory Management and Underlying Operating Systems" on page 3-44

"Pool-based Allocation: POOL" on page 3-35

3.3 Stack Management

For applications that do not use tasking, stack usage is very straightforward. In self-host applications, the stack is simply the process stack. For cross-development environments, the stack is the top of the heap-stack area specified in the kernel configuration.

For applications that use tasking, stack management is more complicated. The main task still uses the stack as described above. Each additional task, however, gets its stack from the heap. The RTS keeps track of the current stack pointers and stack limits for each task.

It is often important to know how much stack area a task requires and the maximum stack depth a task attains. SC Ada offers a utility to obtain this information. The `use` option to the debugger command `lt` (`lt use`) displays the starting location and the size of each task stack, including the exception stack, and fast rendezvous wait stack. It also shows the maximum stack usage.

References

lt command, *SPARCompiler Ada Reference Guide*

3.4 Memory Allocation Support in the SC Ada Runtime System

This section describes the support for dynamic memory allocation and deallocation in the SC Ada runtime system and describes the ways in which memory allocation support can be configured to suit application-specific requirements.

In Ada, memory is allocated at runtime in one of two basic ways: by using a stack or using a heap. Memory is allocated from a stack when a subprogram is activated. Memory is allocated from a heap when a program employs an allocator or when a program directly or indirectly calls a routine that allocates memory. For example, if a program uses tasks, the compiler generates calls to an RTS task creation routine that allocates task storage from a heap. SC Ada supports user-space allocation (allocators) independently of kernel-space allocation (tasks); this document describes only the user-space *library* support.

This implementation of the RTS goes beyond the strict Ada requirements in that it also supports pool-based allocation and deallocation of memory for user space. The user can also configure in custom allocation support.

Ada allocators (`new`) and deallocators (`UNCHECKED_DEALLOCATION`) normally result in the compiler generating calls to the runtime library routines, `AA_GLOBAL_NEW` and `AA_GLOBAL_FREE`, respectively. (If local heaps are applicable, calls `AA_LOCAL_NEW` and `AA_LOCAL_FREE` are generated instead.)

The user-space allocation routines `AA_GLOBAL_NEW` and `AA_GLOBAL_FREE` are user configurable. Several implementations are supplied, including a default.

`SLIM_MALLOC`

A first-fit heap allocator that provides fast coalescing upon deallocation.

`FAT_MALLOC`

Includes `SLIM_MALLOC`'s features plus small blocks lists for performance improvement, and the capability to allocate and deallocate from an interrupt handler (The default for SPARCompiler Ada).

`DBG_MALLOC`

Includes `FAT_MALLOC`'s features plus facilities for assisting debug of programs using allocation.

In addition, a simple model (SMPL_ALLOC) is supplied as an aid to programmers who want to develop their own allocation / deallocation routines. An application can use any one of these implementations or you can develop your own. With the SC Ada compiler and runtime system, these implementations are mutually exclusive.

For applications that make use of memory pools, package POOL in the usr_conf library enables applications to dynamically create, destroy, and switch pools.

VADS EXEC provides an additional memory allocation mechanism independent of the AA_GLOBAL_NEW/AA_GLOBAL_FREE implementation. Calls to the services in the VADS EXEC package V_MEMORY are not made by the compiler to support allocators / deallocators (that is, new). The application must explicitly call V_MEMORY services, which can be used concurrently with any of the previously described user-space allocation implementations.

References

allocators, Ada LRM 4.8

3.5 Allocators

Consider a user program containing an access type statically defined by the following:

```
type record_type is record
  field_one: integer;
  field_two: integer;
  ...
end record;
type access_to_record is access record_type;
v: access_to_record;
```

A user can create an object of `type RECORD_TYPE` dynamically at execution time and reference it through the variable V. In Ada, this is done by using an allocator construct, as illustrated in this example:

```
v := new record_type;
```

Such allocators reserve a contiguous block of heap memory for each new instance of `RECORD_TYPE`. This memory is not used for anything except that `RECORD_TYPE` instance while the instance is active.

An object becomes active (allocated) by the above mechanism. It becomes inactive (deallocated) by a user action, typically a call to `UNCHECKED_DEALLOCATION` instantiated for the appropriate type. The SC Ada RTS does *not* automatically deallocate objects except in the specific cases of local heaps and task objects.

The block of memory must be large enough to provide storage for every field or element that can legally (after constraint checks) be referenced through variable V. In practice, it is somewhat larger, with extra space for:

- A two-integer-sized header for when the block is allocated.
- Another two-access-type-sized locations for when the block is free, which are used for links.
- Padding, if necessary, to cause the next allocation to be aligned on a double-word boundary.
- Padding, in some cases, to ensure that the block fits in a small blocks list when deallocated.

Be careful about the possible scenario shown by the following example code fragment.

```
Type foo is record
    ...
end record;
type a_foo is access foo;
type foo2 is record
    ...
end record;
type a_foo2 is access foo2;
procedure dealloc_foo is new UNCHECKED_DEALLOCATION(foo, a_foo);
bar1, bar2: a_foo;
bar3: a_foo2;
begin
    bar1 := new foo;
    ...
    bar2 := bar1;
    ...
    dealloc_foo(bar1);
    bar3 := new foo2;
    ...
end;
```

After BAR3 is assigned, there is a high likelihood that, despite being different types, BAR2 and BAR3 point to the same object. The SC Ada RTS does not attempt to detect the dangling reference generated here. Note that the preceding example has this same problem if BAR2 and BAR3 are of the same type. The point here is that the user must be careful with access types and deallocation.

An allocator is implemented as an implicit call to an RTS routine AA_GLOBAL_NEW.

```
new_object_address :=
  AA_GLOBAL_NEW(storage_units_needed_for_record_type);
```

The SC Ada RTS supplies a default implementation for AA_GLOBAL_NEW, as well as for the other memory allocation utilities discussed in this chapter. The SC Ada default is contained in the SC Ada runtime library and is linked with the user program unless the user explicitly supplies an alternative implementation. If the user includes a module that defines the name AA_GLOBAL_NEW, that definition supersedes the default. This process, part of configuration control, is described in more detail under ”Simple Allocation: SMPL_ALLOC”. For the moment, it is important to know that users can redefine the memory allocation implementation and that SC Ada provides the default implementation (in some cases more than one) described here.

References

“Small Block Lists” on page 3-23

“Simple Allocation: SMPL_ALLOC” on page 3-17

3.6 SC Ada Library Memory Management Semantics

The user-space (as opposed to kernel-space) memory management interface defines a set of subprograms called by compiler-generated code and the SC Ada runtime library itself. This section describes when these subprograms are called and the function each performs. SC Ada supplies the following specification in `standard` for its memory management interface:

```
with system;
with v_i_types;
package v_i_alloc is
    function aa_global_new(size : in v_i_types.alloc_t)
        return system.address;
    procedure aa_global_free(a : in system.address);
    function aa_aligned_new(size, dope_size : in v_i_types.alloc_t;
        alignment : in integer) return system.address;
    function aa_local_new(size : in v_i_types.alloc_t;
        lheap: system.address) return system.address;
    procedure aa_local_free(a: in system.address;
        lheap: system.address);
    procedure extend_intr_heap(extension: in integer);
    function get_intr_heap_size return integer;
    function krn_aa_global_new(size: v_i_types.alloc_t)
        return system.address;
    procedure krn_aa_global_free(a: system.address);
    procedure extend_stack;
end v_i_alloc;
```

Figure 3-1 Specification for Memory Management Interface

3.6.0.1 AA_GLOBAL_NEW

A call to `AA_GLOBAL_NEW (`*`size`*`)` returns a pointer (address) to a contiguous block of at least *size* `STORAGE_UNITS`. The space can be located anywhere in memory, but the storage should not be in use for any purpose other than holding this object.

In general, the compiler generates calls to AA_GLOBAL_NEW for each use of an allocator. Currently, there is one exception to this rule: If the corresponding access type is defined directly or indirectly within a subprogram *and* a representation clause has been given for the type:

```
for access_type'storage_size use ...
```

the compiler generates calls to AA_LOCAL_NEW. Future versions of SC Ada will have a second case where AA_GLOBAL_NEW is not used. When an alignment representation clause is given for the type:

```
for record_type use
    record at mod ...
        ...
    end record;
```

the compiler generates calls to AA_ALIGNED_NEW.

3.6.0.2 *AA_ALIGNED_NEW*

AA_ALIGNED_NEW supports allocation of objects on specified storage unit boundaries. For example, an object can be allocated such that it is aligned on a virtual page boundary. A call to AA_ALIGNED_NEW has the following form:

```
object_address := v_i_alloc.AA_ALIGNED_NEW(size,
dope_size, alignment);
```

where size is the size of the object allocated, DOPE_SIZE is the size of the dope vector required (usually zero for anything but unconstrained arrays), and alignment is the number of STORAGE_UNITS to align the object to. DOPE_SIZE, which must be a multiple of four STORAGE_UNITS, is the amount of memory to be allocated and prepended to the object that is not aligned. The amount of the alignment must be a power of two. For example, the call:

```
object_address := v_i_alloc.AA_ALIGNED_NEW(2000, 16,
4096);
```

allocates 2000 STORAGE_UNITS with a 16 STORAGE_UNIT dope area aligned on a 4096 STORAGE_UNIT boundary. Future versions of SC Ada will support the Ada alignment representation clause using this function.

3.6.0.3 AA_GLOBAL_FREE

AA_GLOBAL_FREE supports deallocation of objects allocated by AA_GLOBAL_NEW (or AA_ALIGNED_NEW). Its parameter must be a pointer that was previously returned by AA_GLOBAL_NEW. Passing an arbitrary or previously freed address to AA_GLOBAL_FREE is erroneous and leads to unpredictable results.

The predefined generic subprogram UNCHECKED_DEALLOCATION is implemented with calls to AA_GLOBAL_FREE if allocators for the type call AA_GLOBAL_NEW.

Instances of unconstrained array objects are implemented as an access to the start of the array with a dope vector prepended. If such an object is passed to UNCHECKED_DEALLOCATION, it looks past the dope vector to find the header stored by AA_GLOBAL_NEW. Therefore, *do not use* UNCHECKED_CONVERSION or the address attribute in supplying a value directly to AA_GLOBAL_FREE for these instances.

Access variables must not refer to deallocated objects. The compiler does not verify that access variables refer to allocated objects. The RTS and the compiler expect that any storage freed by AA_GLOBAL_FREE can be reused for other purposes.

References

"AA_LOCAL_NEW" on page 3-13

3.6.0.4 AA_LOCAL_NEW

Access types declared locally in a procedure are given special treatment if they have a STORAGE_SIZE representation specification. In these cases, the elaboration of the access type declaration cause a request for a contiguous block of the required size to be pushed onto the procedure stack along with a descriptor for the local heap. The descriptor is defined by the following record.

```
type local_heap_t is record
  size: integer;           -- size of the local heap in storage_units
  head: address;           -- first storage_unit of local heap
  next: address;           -- next available storage_unit of local heap
  free_list: address;      -- free list populated by AA_LOCAL_FREE
end record;
type a_local_heap is access local_heap_t;
```

Allocators for local heap access types are implemented as calls to AA_LOCAL_NEW. The size parameter of AA_LOCAL_NEW specifies the size in storage units of the object to be allocated. The LHEAP parameter is a pointer of type A_LOCAL_HEAP to the appropriate local heap descriptor for that access type.

The RTS does not require that a local heap be used, but it allocates the indicated amount of storage in any case.

In its search to find a suitably sized block to satisfy the allocation, the RTS first tries to allocate from the end of the local heap by examining the next field. If there is not enough room, it searches the free list (created as blocks are deallocated into this heap). If still no block is found, a STORAGE_ERROR exception is raised.

The STORAGE_SIZE required follows the simple formula:

```
RF(SO + 2 * SI) * number of objects
Where:
    RF is a function that rounds its argument up to a
    multiple of the size of a float type.
      SO is the size of the object
      SI is the size of an integer
```

All sizes are in SYSTEM.STORAGE_UNITs.

Here is a simple example where the compiler creates and uses a local heap:

```
procedure foo is
    type node;
    type a_node is access node;
    type node is record
        value: integer;
        link: a_node;
    end record;
    for a_node'STORAGE_SIZE use 400;
    head, cur_node: a_node;
begin
    -- head is allocated from the local heap
    head := new node;
    cur_node := head;
    for i in 1 .. 20 loop
        cur_node.value := i ** 2;
        -- each node is allocated from the local heap
        cur_node.link := new node;
        cur_node := cur_node.link;
    end loop;
    cur_node.link := null;
    ...
end foo; -- whole linked list deallocated
```

3.6.0.5 AA_LOCAL_FREE

The SC Ada compiler generates calls to AA_LOCAL_FREE for uses of UNCHECKED_DEALLOCATION applied to an access type whose objects are allocated by means of AA_LOCAL_NEW. The object is linked into the local heap's free list. Coalescing of adjacent free objects in a local pool is not supported. It is assumed that the object pointer passed to AA_LOCAL_FREE is a pointer supplied by AA_LOCAL_NEW; this is not checked and programs that disobey this rule are erroneous.

3.6.0.6 EXTEND_INTR_HEAP

This is a user-callable routine that allows the number of blocks available for allocation from an interrupt handler to be increased.

References

"Allocation from Interrupt Handlers" on page 3-25

3.6.0.7 GET_INTR_HEAP_SIZE

This is a user-callable function that allows the number of blocks available for allocation from an interrupt handler to be accessed. This routine is used to detect when the number of blocks available is dangerously low and must be increased (using `EXTEND_INTR_HEAP`).

3.6.0.8 KRN_AA_GLOBAL_NEW and KRN_AA_GLOBAL_FREE

These routines are called by a user program to allocate and deallocate memory directly from the kernel.

These routines are provided to provide compatibility with earlier SC Ada releases. They are layered upon the Ada Kernel's `ALLOC` and `FREE` services.

3.6.0.9 EXTEND_STACK

In earlier SC Ada releases this routine allowed the caller to extend the main program's stack space. It is no longer supported and always raises a `STORAGE_ERROR` exception.

3.6.0.10 AA_INIT

In the SC Ada memory allocation default implementations, a call to `AA_INIT` must precede *any* call to `AA_GLOBAL_NEW`. Users can supply their own implementations that relax this rule. The default RTS does call `AA_INIT` as essentially its first operation. Note that no interface is provided for this routine because you should not call it from a user-level program. Note that if you write your own `AA_INIT` routine, it must not involve calls to any other RTS service, because these are not yet initialized.

3.7 Overview of SC Ada-Supplied Memory Management

SC Ada supplies four implementations of memory management. In order of size they are called SMPL_MALLOC, SLIM_MALLOC, FAT_MALLOC, and DBG_MALLOC. SMPL_MALLOC is a very simple example implementation whose source is provided as a basis for those who want to write their own allocation routines. FAT_MALLOC adds features to SLIM_MALLOC and likewise DBG_MALLOC adds features to FAT_MALLOC. Additionally, SC Ada supplies a package that layers upon SLIM_MALLOC, FAT_MALLOC, or DBG_MALLOC that allows you to do "pool-based" allocation. Note the term "heap allocation" refers to all three SLIM_MALLOC, FAT_MALLOC, and DBG_MALLOC implementations.

SC Ada has several configuration parameters in the v_usr_conf_b.a file that allow tuning the performance and behavior of the memory allocation routines.

3.8 Simple Allocation: SMPL_ALLOC

Source is provided for a very simple memory allocation implementation to illustrate how you can write your own version of the memory allocation routines. Refer to the files smpl_alloc.a, smpl_alloc_b.a, and smpl_alloc_s.a in the usr_conf directory. Note that because of the different block header structures and because SMPL_ALLOC doesn't support deallocation, the POOL package cannot be layered on top of SMPL_ALLOC.

3.9 Slim Heap Allocation: SLIM_MALLOC

In heap allocation, all calls to AA_GLOBAL_NEW not originating from interrupt handlers allocate space from one global heap. For a request of *N* storage units, AA_GLOBAL_NEW allocates at least *N* storage units from the global heap. For most systems, the number of storage units is rounded to an even multiple of words or longwords. On many systems, memory references to address at even multiples of storage units are faster than references to odd addresses. Other systems require addresses to be aligned.

A two-word header is also allocated to hold the size of the object being allocated, its status (free or allocated), and the size of the previous adjacent block for use when coalescing. This header is located adjacent to the allocated object, immediately preceding it. AA_GLOBAL_FREE uses this header. The header is described by the following data structures:

```
type block_header is record     -- Header used by all blocks
    size: integer;              -- +/- total block size incl header
                                -- If negative, then block is used.
    prev_blk_size: integer;     -- +/- previous block size incl
                                -- header,
                                -- If negative then block was
                                -- allocated from an interrupt
                                -- handler.
end record;
type free_block;                -- Free memory block
type a_free_block is access free_block;
type free_block is record
    header: block_header;       -- Size = +block size ( >0 )
    next_free: a_free_block;    -- Forward link to next free memory
    prev_free: a_free_block;    -- Backward link to previous free
                                -- memory
end record;
```

A method similar to the boundary-tag system is used to obtain constant-time coalescing with adjacent free blocks, if any, into one block. AA_GLOBAL_FREE links the deallocated storage onto a doubly linked free list. The links are placed in the deallocated object so the minimum size of an object is always rounded up to the size of two access variables.

The minimum size of an allocation is therefore the number of storage units required to hold the two-word header and the two linked list pointers.

AA_GLOBAL_NEW searches the free list for an area large enough to satisfy each storage request as it occurs. A “first fit” algorithm is used by default.

The following series of illustrations in Figure 3-1 on page 11 and Figure 3-4 on page 27 is an example of how memory looks while doing memory allocations and deallocations. To make it easier to follow, we assume that there are 1000

bytes of memory to begin with and that we ignore the `start` and `end` blocks used for checking the boundary conditions. The first and second fields of every block are the size of the block and the size of the previous block respectively.

Note – This example does not show the case where there are more than two free blocks in memory. In these cases, the `PREV_FREE` and `NEXT_FREE` fields are no longer identical (they form a doubly-linked circular list).

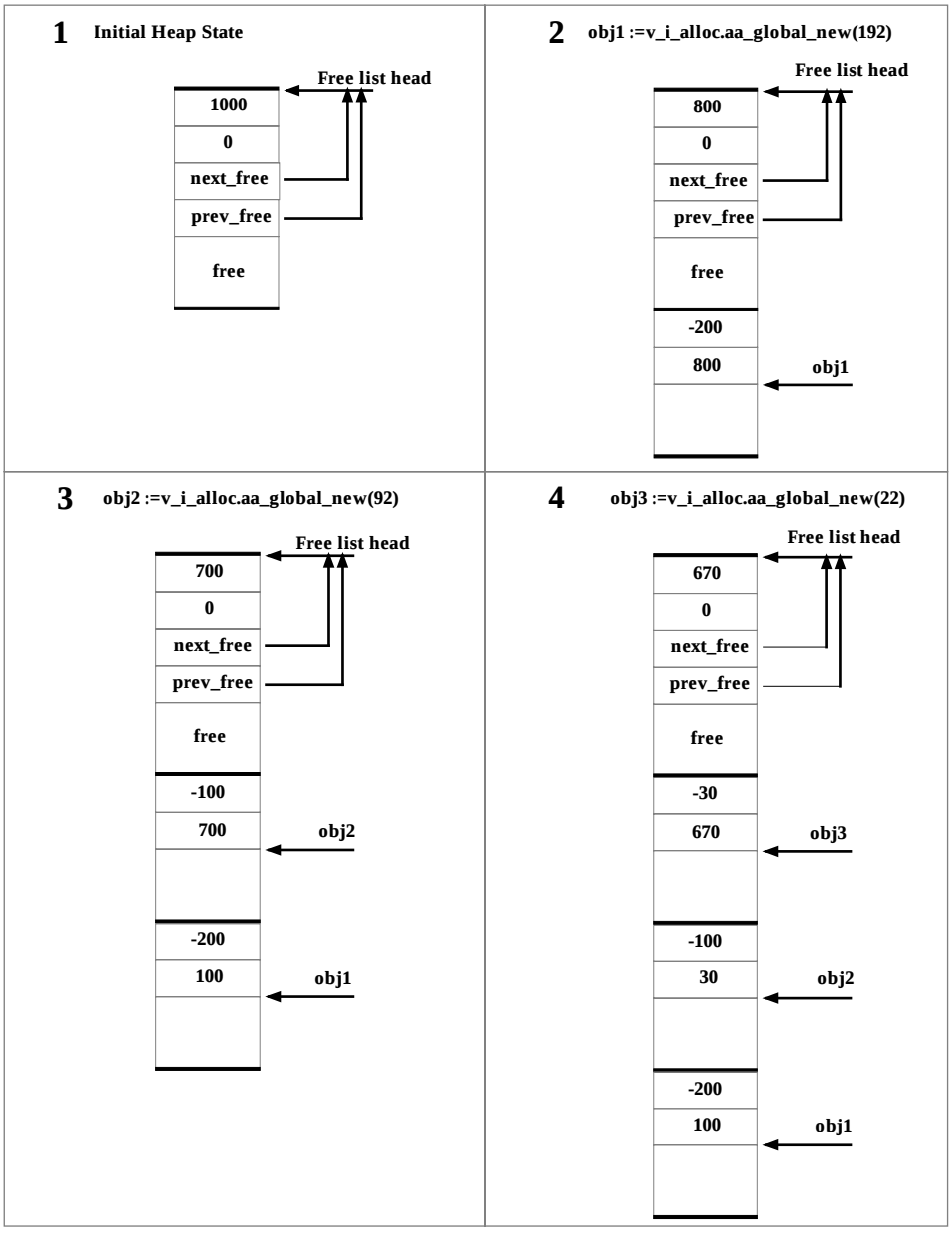

Figure 3-2 Memory Allocation and Deallocation Process-1

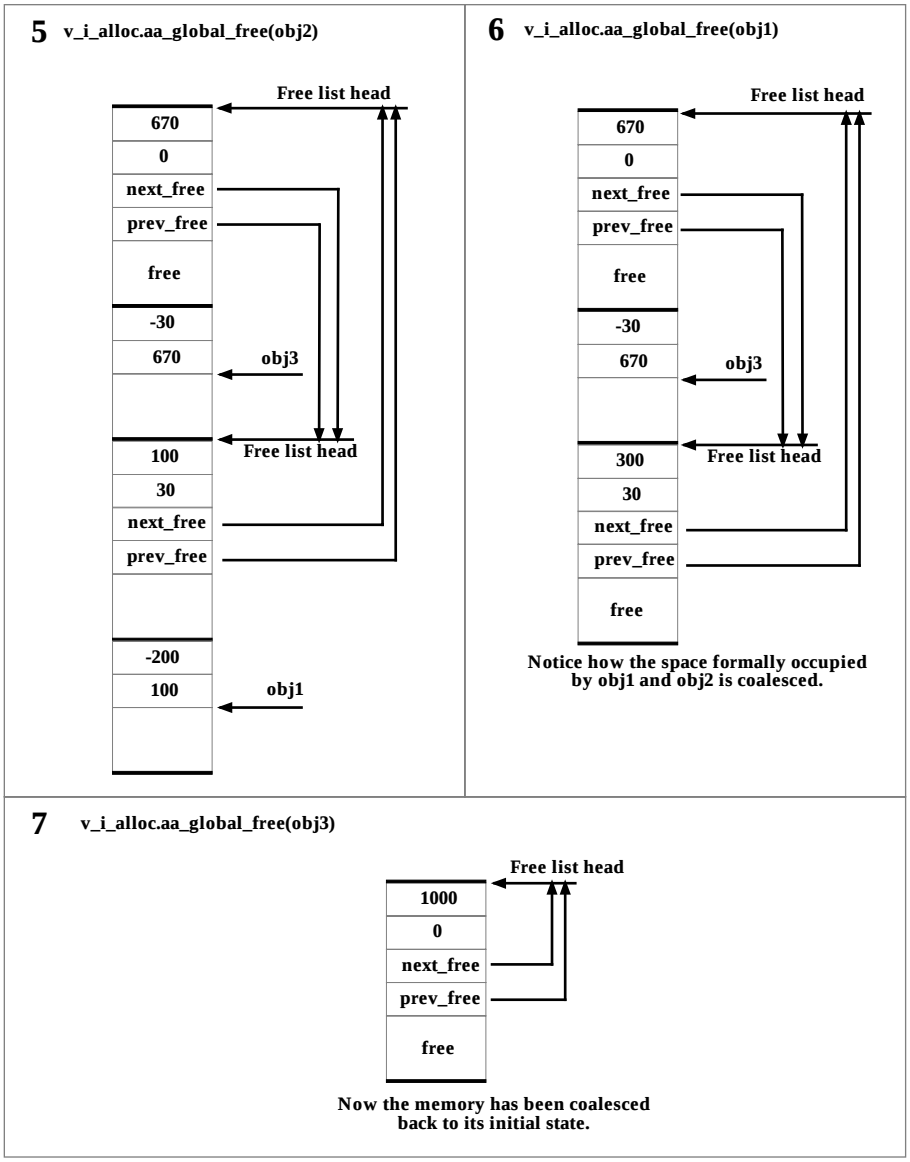

Figure 3-3 Memory Allocation and Deallocation Process-2

Heap allocation uses the user-configurable variable MIN_SIZE. If an area in the free list is larger than the user request by less than MIN_SIZE storage units, the entire object is given to the user even though the extra storage is wasted. However, if the waste is MIN_SIZE or larger, remaining storage is split into a new free area.

The intent of MIN_SIZE is to reduce memory fragmentation, whereby the free list becomes clogged with numerous free areas too small to satisfy memory requests. It is recommended that MIN_SIZE be configured to be in the range of 8 to 128 storage units. However, in no case can it be smaller than the size of two linked access variables (8 storage units) or larger than the smallest small block list.

The parameter ALLOCATION_STRATEGY allows you to instruct AA_GLOBAL_NEW to search the free list using either of two strategies: "first-fit" or "best-fit". Generally, setting it to FIRST_FIT (the default) optimizes for high speed versus low fragmentation, whereas setting it to BEST_FIT optimizes for low fragmentation versus high speed.

The default value for MIN_SIZE is 8 storage units. To reconfigure this value, package V_USR_CONF in the file v_usr_conf_b.a (located in the usr_conf directory) must be changed and then linked with the program.

V_USR_CONF has two additional parameters controlling heap allocation: GET_HEAP_MEMORY_CALLOUT and HEAP_EXTEND. The routine indicated by GET_HEAP_MEMORY_CALLOUT is called when the heap memory is exhausted. HEAP_EXTEND is the minimum amount of memory that should be requested from GET_HEAP_MEMORY_CALLOUT. Typically, GET_HEAP_MEMORY_CALLOUT is called when AA_GLOBAL_NEW fails to find sufficient heap space to satisfy a memory request. In this case, the value of the size parameter for GET_HEAP_MEMORY_CALLOUT is set to AA_GLOBAL_NEW's size parameter or HEAP_EXTEND, whichever is larger. Also, the heap area is initialized at program startup by calling GET_HEAP_MEMORY_CALLOUT using HEAP_EXTEND as the size parameter.

After the package is modified and recompiled, it is included in a program either by a with clause in the Ada source or a WITH*n* directive in the ada.lib file.

References

"Small Block Lists" on page 3-23

3.10 Fat Heap Allocation: FAT_MALLOC

FAT_MALLOC is the default memory management archive for SC Ada. In addition to the services of SLIM_MALLOC, FAT_MALLOC provides small block lists and allocation from an interrupt handler. FAT_MALLOC is considerably larger and more complex than SLIM_MALLOC.

3.10.1 Small Block Lists

Space for small block lists is allocated from the heap. Users can configure any number of lists, each with a different element size. Each list contains elements of the same size. A small block is allocated from the list whose elements are just large enough to store the object. For example, if lists are established for blocks of size 16, 32, and 128 STORAGE_UNITs, requests of up to 16 storage units come from the first list, requests for17 through 32 storage units from the second list, and requests for 33 through 128 storage units from the third list. There are two cases where objects are allocated from the heap using a “first fit” strategy as in SLIM_MALLOC. The first case is if the object is larger than 128 STORAGE_UNITs. The second case is if the properly sized small blocks list is empty when examined; in this case when the object is deallocated, it is returned to that small block list.

UNCHECKED_DEALLOCATION is used in the normal way to free space occupied by small objects. The compiler generates a call to AA_GLOBAL_FREE to implement UNCHECKED_DEALLOCATION. The user-configurable parameter MIN_LIST_LENGTH defines the minimum number of objects to be kept on each small block list. If the current number of blocks on the proper small block list is less than MIN_LIST_LENGTH, the small block returns to that list with no attempt to coalesce it with its neighbors. Otherwise, coalescing is attempted. If the block can’t be coalesced with either of its neighbors, it returns to the small block list. Coalescing is considered impossible if the neighbor to be coalesced is currently on a small block list and the number of blocks on that list is less than MIN_LIST_LENGTH. The rationale for providing a minimum list length is that the small blocks lists is of little use if small blocks are always coalesced when they are deallocated, that is, the small block list would be empty too often.

The number of small object lists can be configured to zero, in which case they are not used.

The rationale for using small blocks is that, if the user knows in advance that most allocations are of a specific size, allocation can be facilitated for that size. Overall performance therefore improves both in memory usage and speed.

List-based allocation can increase internal fragmentation if the sizes of objects are truly random. Thus, it should be used only if the user knows ahead of time that many allocation requests are of a specific size or group of sizes or memory waste can be tolerated.

To reconfigure individual list element sizes, the `SMALL_BLOCK_SIZES_TABLE` structure in the package body `V_USR_CONF` in the file `v_usr_conf_b.a` (located in the `usr_conf` directory) must be changed.

`V_USR_CONF` has the following default sizes:

```
:= (
      8, 16, 24, 32
   );
```

`V_USR_CONF` has three other parameters for controlling allocation: `HEAP_EXTEND`, `GET_HEAP_MEMORY_CALLOUT`, and `MIN_SIZE`. Their usage here is the same as for `SLIM_MALLOC`.

References

“Slim Heap Allocation: SLIM_MALLOC” on page 3-17

3.10.2 Allocation from Interrupt Handlers

FAT_MALLOC (and DBG_MALLOC) contains code for handling memory allocation from an interrupt handler. Using SLIM_MALLOC, a deadlock arises if a task was doing a new operation (or calling V_I_ALLOC.AA_GLOBAL_NEW) when an interrupt occurs and the interrupt handler does a new. This is because a mutex guards the allocator and an interrupt handler cannot be suspended (except by another interrupt). To solve this problem, FAT_MALLOC provides a very simple memory allocation scheme when memory is allocated from an interrupt handler. This condition is detected by using the Ada Kernel service ADA_KRN_I.ISR_IN_CHECK( ). This scheme uses a separate heap of preallocated, fixed-size blocks (user configurable). Instead of using mutexes to protect the data structures, it disables interrupts.

This scheme allocates memory very quickly since all blocks are the same size. The blocks are tagged as having been allocated from an interrupt handler and can thus either be deallocated during task time or interrupt time. The current number of blocks available for allocation is accessible by using the function V_I_ALLOC.GET_INTR_HEAP_SIZE. Additionally, if the interrupt handler runs low or out of blocks (encounters a STORAGE_ERROR exception), a task can call V_I_ALLOC.EXTEND_INTR_HEAP to get more blocks. Note that V_I_ALLOC.EXTEND_INTR_HEAP should not be called at interrupt time or a deadlock could result. If an interrupt handler attempts to allocate an object that is larger than the fixed size blocks in the interrupt heap, a STORAGE_ERROR exception will be raised.

The default initial size of the interrupt heap size is determined by the INITIAL_INTR_HEAP field in the MEM_ALLOC_CONF record in v_usr_conf_b.a file. Its default setting is 100 objects. The size of each object in the heap is determined by the INTR_OBJ_SIZE field. The default size is 128 storage units. These can be set to any reasonable value, but keep in mind that this storage is allocated before the program actually starts running and affects the amount of runtime memory used.

Figure 3-4 uses V_I_ALLOC.EXTEND_INTR_HEAP:

```
package RECEIVE is
end RECEIVE;
with V_INTERRUPTS;
with SYSTEM;
with V_SEMAPHORES;
with V_I_ALLOC;
package body RECEIVE is
    low_water_mark: constant integer := 10;
    please_extend_intr_heap: V_SEMAPHORES.binary_semaphore_id :=
        V_SEMAPHORES.CREATE_SEMAPHORE;
    task INTR_HEAP_EXTEND_SLAVE is
    end INTR_HEAP_EXTEND_SLAVE;
    task body INTR_HEAP_EXTEND_SLAVE is
    begin
        loop
            V_SEMAPHORES.WAIT_SEMAPHORE(
                semaphore => please_extend_intr_heap,
                wait_time => V_SEMAPHORES.WAIT_FOREVER);
            V_I_ALLOC.EXTEND_INTR_HEAP(100);
        end loop;
    end INTR_HEAP_EXTEND_SLAVE;
    procedure RECEIVE_HANDLER is
    begin
        -- interrupt handler code
        -- ...
        -- buffer.next := new buffer;
        -- ...
        -- check to see if we are running low on allocation blocks:
        --
        if V_I_ALLOC.GET_INTR_HEAP_SIZE <= low_water_mark then
            -- we are running low, tell the slave we want more
            V_SEMAPHORES.SIGNAL_SEMAPHORE(please_extend_intr_heap);
        end if;
    end RECEIVE_HANDLER;
    procedure RECEIVE_ISR is new V_INTERRUPTS.ISR(RECEIVE_HANDLER);
```

(Continued)

```
begin
    -- Attach the instantiated RECEIVE_ISR
    V_INTERRUPTS.ATTACH_ISR(16#80#, RECEIVE_ISR'address);
end RECEIVE;
```

Figure 3-4 V_I_ALLOC_EXTEND_INTR_HEAP

If you want to write your own allocator to allocate memory for interrupt handlers, you should call the Ada Kernel servic, `ADA_KRN_I.ISR_IN_CHECK()` to see if the allocator was called from an interrupt handler or a task.

3.11 Debug Heap Allocation: DBG_MALLOC

This archive includes all of the features of the FAT_MALLOC plus some features that help to debug memory allocation or display usage problems. Calls to AA_GLOBAL_NEW return a piece of memory that has some extra information in the header. When AA_GLOBAL_FREE is called, this extra information is checked along with the other fields for consistency. If an inconsistency is detected, a STORAGE_ERROR exception is raised. These checks can increase the probability of detecting an attempt to deallocate a dangling access value. Because of this extra checking, performance suffers.

In addition, through package ALLOC_DEBUG in the usr_conf directory, you can call routines that allow the entire heap to be combed for inconsistencies. The procedure is called VERIFY_HEAP. This routine raises a STORAGE_ERROR if an inconsistency is detected. The ALLOC_DEBUG package has two other features of interest. First it contains a routine, PRINT_HEAP_MAP, for printing the HEAP_MAP. The address of each block in the heap is printed as well as its status (free or allocated) and size. It contains another routine , PRINT_HEAP_STATS, that prints a histogram of block sizes and some information about memory utilization. Using the Ada command with to

include this package in your program causes the default allocator to be automatically replaced by the DBG_MALLOC at link time. The package specification and some examples are shown below:

```
package alloc_debug is
    procedure verify_heap;
    procedure print_heap_map;
    procedure print_heap_stats(
            lower_size_cutoff: integer := 1;
            upper_size_cutoff: integer := 512);
end;
```

Figure 3-5 Package Specification for ALLOC_DEBUG

```
with alloc_debug;
procedure verify_and_print_heap is
begin
    alloc_debug.verify_heap;
    alloc_debug.print_heap_map;
end verify_and_print_heap;
```

Figure 3-6 Example program using VERIFY_HEAP and PRINT_HEAP_MAP:

The output of the preceding example is:

```
Heap Map:
Address: 16#45120#
    Status: allocated
    Size: 24
Address: 16#45138#
    Status: free
    Size: 127232
Address: 16#64238#
```

(Continued)

```
    Status: allocated
    Size: 32
Address: 16#64258#
    Status: allocated
    Size: 1072
Address: 16#64688#
    Status: allocated
    Size: 24
.
.
.
```

Here's an example of using PRINT_HEAP_STATS called from within a large program. The routine is called with the following line in the source code:

```
alloc_debug.print_heap_stats(lower_size_cutoff => 1,
upper_size_cutoff => 192);
```

The output is:

```
Heap Stats:
Histogram for blocks in size range 1 .. 192
          Size Range     Count    Freq
-------------------------------------
       1 ..       24:         5 : 0.010 -
      25 ..       48:       184 : 0.362 - ************
      49 ..       72:       137 : 0.270 - **********
      73 ..       96:        12 : 0.024 - *
      97 ..      120:         0 : 0.000 -
     121 ..      144:        47 : 0.093 - ***
     145 ..      168:       120 : 0.236 - *********
     169 ..      192:         3 : 0.006 -
Memory allocated by RTS from kernel: 131072 bytes.
Memory allocated by program from RTS: 104848 bytes.
Memory utilization efficiency: 80.0%
```

Some notes about use:

- The upper_size_cutoff is rounded up so that the eight size intervals are of equal and integral sizes.
- The block size includes the header size; the actual object size is at least two words smaller (depending on the `SMALL_BLOCK_SIZES` and `MIN_SIZE` configuration parameters. To eliminate the effects of small block sizes on the statistics, set the list to empty and `MIN_SIZE` to 8 in package `V_USR_CONF`).
- Other tasks are blocked from doing allocations while the heap map is being printed or the heap stats are being calculated to avoid running into data structure inconsistencies.
- The use of this package itself can change the actual statistics somewhat. Although this package does no allocations of its own, it uses `TEXT_IO`, which does. In practice, programs that do a lot of allocation and deallocation notice only a slight variation in the statistics. To get exact results, the following program can be compiled and executed:

```
with alloc_debug;
procedure print_base_heap_stats is
begin
  alloc_debug.print_heap_stats(...);
end print_base_heap_stats;
```

Note – with **...** being filled in with the same upper and lower size cutoffs used in the real program. The results from running this program can be deducted from the results of the real program to derive the real program allocation behavior.

SunPro supplies the source code for this package in the `usr_conf` library so that it can be customized. If you modify and/or recompile this package, you must alter the following line in `alloc_dbg_b.a`:

```
pragma link_with("$wrong_SCAda_path/standard/.objects/dbg_malloc.a");
      -- where $wrong_SCAda_path is the path shipped in the source
```

to

```
pragma link_with("SCAda_path/standard/.objects/dbg_malloc.a");
```

We advise that you copy the files `dbg_heap.a`, `alloc_dbg.a`, and `alloc_dbg_b.a`.

3.12 Configuring in a Non-default Allocation Archive

To use an archive other than the default, you must include a WITH*n* directive in the `ada.lib`. The most convenient way to do this is to use the `a.info` command. For example:

```
a.info -a WITH /user2/SCada/self/standard/.objects/dbg_malloc.a
```

Note, as mentioned before, that if you use the Ada command `with` in the `ALLOC_DEBUG` package, the default allocator automatically gets replaced with `DBG_MALLOC`. Therefore it is not necessary to add the WITH*n* INFO directive to the `ada.lib`.

Note – It is not necessary to perform the above step if you `with` the `ALLOC_DEBUG` package, because this automatically links in the `dbg_malloc.a` archive.

3.13 pragma RTS_INTERFACE

Before describing package POOL package, it is a useful digression to describe pragma RTS_INTERFACE. This pragma allows the replacement of the default calls made implicitly at runtime to the underlying RTS routines. For example, as described before, the compiler normally generates calls to AA_GLOBAL_NEW to implement the Ada new operator. With this pragma, you cause the compiler to generate calls to any routine as long as its parameters and return value match the original. As for the allocation routines, they must match the specification in the V_I_ALLOC package in standard. Here is a code fragment demonstrating this:

```
package body foo is
    ...
    function my_global_new(size: v_i_types.alloc_t)
            return system.address is
        obj_addr: system.address;
    begin
        obj_addr := v_i_alloc.aa_global_new(size);
        text_io.put("Allocating ")
        my_num_io.put(size, base => 16, width => 1);
        text_io.put(" storage_units from address ")
        my_num_io.put(to_integer(obj_addr), base => 16, width => 1);
        text_io.newline;
        return obj_addr;
    end my_global_new;
    procedure bar is
        pragma RTS_INTERFACE(AA_GLOBAL_NEW, my_global_new);
        type a_integer is access integer;
        count: a_integer;
    begin
        ...
        count := new integer;  -- my_global_new gets called here
        ...
    end bar;
    ...
end foo;
```

Note – The subprogram name used in the pragma must be a simple name (no package name qualification). If the subprogram is in another package, this is easily achieved by an Ada `use` clause.

`pragma RTS_INTERFACE` supports replacing the RTS routines whose interfaces are defined in the following packages:

```
v_i_alloc.aV_I_ALLOC.A          Memory allocation routines
v_i_libop.aV_I_LIBOP.A          Library routines (such as block compares)
v_i_pass.aV_I_PASS.A            Passive tasking routines
v_i_raise.aV_I_RAISE.A          Exception routines
v_i_taskop.aV_I_TASKOP.A        Tasking routines
```

Because of the powerful nature of this pragma, use it with caution.

3.14 Pool-based Allocation: POOL

SunPro also supplies POOL, an optional pool-based memory allocation package (pools are sometimes referred to in the literature as *arenas*). POOL layers on top of any of SLIM_MALLOC, FAT_MALLOC, or DBG_MALLOC. Using this package along with pragma RTS_INTERFACE, memory is separated into multiple storage pools. All objects allocated from a particular pool are deallocated simultaneously with a single call. The source to this package is provided as well to show one way to layer on the underlying RTS memory allocation routines. Shown below is the package specification:

```
with system;use system;
with v_i_types;
package pool is
    pragma suppress(ALL_CHECKS);
    subtype alloc_t is v_i_types.alloc_t;
    deallocated_pool                    : exception;
    noextend_pool                       : exception;
    attempt_to_deallocate_object_in_pool : exception;
    function aa_global_pool_new
            (size : in     alloc_t)
        return address;
    procedure aa_global_pool_free
            (a : in     address);
    type a_pool_descr is private;
    function create_pool
            (initial_size   : alloc_t;
             extension_size : alloc_t;
             base           : address := NO_ADDR)
        return a_pool_descr;
    procedure reset_pool
            (p : in     a_pool_descr);
    procedure deallocate_pool
            (p : in out a_pool_descr);
    procedure switch_pool
            (old_pool:    out a_pool_descr;
             new_pool: in     a_pool_descr);
```

```
    procedure restore_pool
           (old_pool: in     a_pool_descr);
    function aa_pool_new
           (pool : a_pool_descr;
            size : alloc_t)
        return address;
    function is_same_pool
           (pool_1 : a_pool_descr;
            pool_2 : a_pool_descr)
        return boolean;
    function amount_allocated
           (p : a_pool_descr)
        return alloc_t;
    function current_pool
        return a_pool_descr;
    function heap_pool
        return a_pool_descr;
    procedure enable_unsafe_allocation;
    procedure disable_unsafe_allocation;
    procedure enable_deallocation_exception;
    procedure disable_deallocation_exception;
private
    type pool_descr;
    type a_pool_descr is access pool_descr;
end pool;
```

Figure 3-7 package POOL

These pool control facilities enable a program to create and deallocate pools of memory and to direct its memory requests to specific pools. The interface to these facilities is contained in the `usr_conf` library `package POOL`.

The default pool, called the *heap pool*, is just the underlying allocator (`SLIM_MALLOC`, `FAT_MALLOC` or `DBG_MALLOC`). Objects allocated from the heap pool may be deallocated with either `AA_GLOBAL_FREE`, or `AA_GLOBAL_POOL_FREE`. In contrast to the pools created with `CREATE_POOL`, the heap pool cannot be deallocated.

The only appropriate way to deallocate objects allocated from pools is to deallocate the entire pool. Any use of AA_GLOBAL_POOL_FREE to free an individual object allocated from any pool besides the heap pool is ignored. Any use of AA_GLOBAL_FREE on an object allocated from any pool besides the heap pool is erroneous and results in unpredictable behavior.

Pools other than the heap pool do not support memory coalescing or small block lists, since individual objects are never deallocated.

3.14.1 Pool Control

3.14.1.1 AA_POOL_NEW

```
function aa_pool_new(pool: in a_pool_descr;
                     size: in alloc_t)
   return address;
```

For applications that need to switch from one pool to another quickly without incurring the overhead of the SWITCH_POOL and RESTORE_POOL routines, AA_POOL_NEW provides a better alternative. Although it cannot be used with the pragma RTS_INTERFACE, it can be called explicitly to allocate memory from a particular pool. The caller supplies the pool descriptor and the amount of memory to be allocated and is returned an address that points to the newly allocated memory.

3.14.1.2 AMOUNT_ALLOCATED

```
function amount_allocated
(p : a_pool_descr)
return alloc_t;
```

AMOUNT_ALLOCATED returns the number of bytes currently allocated from the specified pool.

3.14.1.3 CREATE_POOL

```
function create_pool(initial_size      alloc_t;
                     extension_size :  alloc_t := 0;
                   base_address   :  address := no_addr)
                return a_pool_descr;
```

CREATE_POOL creates an internal data structure for a pool and returns a descriptor or identifier for the pool. Pools generally use contiguous memory where possible to prevent fragmentation. INITIAL_SIZE gives the number of storage units to allocate to the pool initially. Pools grow as necessary, allocating from the heap in lots of EXTENSION_SIZE STORAGE_UNITS. If EXTENSION_SIZE is zero, the pool is not extended and exhaustion raises STORAGE_ERROR. By specifying the base address of a pool, a user can assign a specific region of memory to the pool, perhaps to an area containing faster memory. SC Ada does not check to see if pools created in this manner collide with other objects (e.g., other pools): this must be done by the user. The default NO_ADDR causes the base to be arbitrary, because the pool storage block is obtained from an arbitrary storage area in the heap. Pools obtained by using NO_ADDR are guaranteed not to collide with other objects. The POOL package creates the descriptor for the heap pool at elaboration time.

3.14.1.4 CURRENT_POOL

```
function current_pool return a_pool_descr;
```

CURRENT_POOL returns the pool descriptor for the current pool.

3.14.1.5 DEALLOCATE_POOL

```
procedure deallocate_pool(p: a_pool_descr);
```

DEALLOCATE_POOL causes all blocks of storage, including any extensions, obtained for the named pool to be returned to the heap's free list. Such memory is no longer reserved for that pool and can be used for any heap activity. The heap pool can never be deallocated.

3.14.1.6 DISABLE_DEALLOCATION_EXCEPTION

```
procedure disable_deallocation_exception;
```

DISABLE_DEALLOCATION_EXCEPTION causes an exception not to be raised if the caller attempts to deallocate an object in any pool other than the heap pool. This is the default condition.

3.14.1.7 DISABLE_UNSAFE_ALLOCATION

```
procedure disable_unsafe_allocation;
```

DISABLE_UNSAFE_ALLOCATION causes the headers not to be omitted from pool-allocated objects. This is the default condition.

3.14.1.8 ENABLE_DEALLOCATION_EXCEPTION

```
procedure enable_deallocation_exception;
```

ENABLE_DEALLOCATION_EXCEPTION raises an exception if the caller attempts to deallocate an object in any pool other than the heap pool. This works only when unsafe deallocation is disabled. This can be useful for identifying deallocation code to be commented out before switching to Unsafe_Allocation mode.

3.14.1.9 ENABLE_UNSAFE_ALLOCATION

```
procedure enable_unsafe_allocation;
```

ENABLE_UNSAFE_ALLOCATION causes the header to be omitted from objects allocated from a pool to reduce overhead. The header is used to tell the difference between objects allocated from a pool and objects allocated from the heap. It is called unsafe because if the caller then attempts to free an object allocated with unsafe allocation enabled, memory will become corrupted or a memory fault will occur.

In addition to omitting the header, this option relaxes the alignment restriction. Objects 8 bytes or larger in size are aligned on 8-byte boundaries as always; objects 3 to 7 bytes are aligned on 4-byte boundaries; 2- byte objects are aligned on 2-byte boundaries; 1-byte objects are byte aligned.

3.14.1.10 HEAP_POOL

```
function heap_pool return a_pool_descr;
```

HEAP_POOL returns the pool descriptor for the heap pool. You might want to switch to the HEAP_POOL to allocate objects that must persist.

3.14.1.11 IS_SAME_POOL

```
function is_same_pool
(pool_1 : a_pool_descr;
 pool_2 : a_pool_descr)
return boolean;
```

IS_SAME_POOL compares the two pool descriptions: pool_1 and pool_2. If they reference the same pool, the function returns TRUE; otherwise, it returns FALSE.

3.14.1.12 RESET_POOL

```
procedure reset_pool(p:  a_pool_descr);
```

RESET_POOL causes the objects in the pool to become deallocated without actually freeing the pool memory back to the underlying allocator. This causes a performance increase if you intend to reuse the pool, rather than calling DEALLOCATE_POOL and then CREATE_POOL again. The heap pool cannot be reset.

3.14.1.13 RESTORE_POOL

```
procedure restore_pool(old_pool: in a_pool_descr);
```

RESTORE_POOL switches the current pool back to OLD_POOL.

3.14.1.14 SWITCH_POOL

```
procedure switch_pool(old_pool: out  a_pool_descr;
                      new_pool: in   a_pool_descr);
```

For performance reasons, AA_GLOBAL_POOL_NEW does not use an explicit parameter to specify the pool used for allocation requests. Instead, it uses an implicit parameter defined by the CURRENT_POOL. At system startup, the heap is the current pool. However, by calling SWITCH_POOL, a program selects an arbitrary pool as the current pool. OLD_POOL is loaded with the previous current pool descriptor. In a multitask program, each task has its own current pool. All **new**s performed within the scope of pragma RTS_INTERFACE using AA_GLOBAL_POOL_NEW are obtained from the current pool. This means that if a procedure using pools calls another procedure that also uses pools, the called procedure must be sure to bracket its allocation from its pool with calls to SWITCH_POOL and RESTORE_POOL. SWITCH_POOL and RESTORE_POOL are designed to clearly bracket the use of a pool, increasing readability and easing maintenance.

```
with POOL;
with V_I_ALLOC;
with V_I_TYPES;
with UNCHECKED_DEALLOCATION;
package body window is
 procedure construct_window_descr(window: out a_window_descr) is
    use POOL;
    pragma RTS_INTERFACE(AA_GLOBAL_NEW, aa_global_pool_new);
    save_pool: a_pool_descr;
  begin
    -- allocate the window descriptor itself from the heap pool
    switch_pool(old_pool => save_pool, new_pool => POOL.heap_pool);
         window := new window_descr;
    restore_pool(save_pool);
    window.pool_descr :=
       create_pool(initial_size => 10_000, extension_size => 5000);
    -- allocate the window's data structures from the pool we just
    -- created
  switch_pool(old_pool => save_pool, new_pool => window.pool_descr);
    window.font_descr := new font_descriptor'(default_font);
```

```
    window.color_descr := new color_descriptor'(default_colors);
    window.feature_descr := new feature_descriptor'(default_features);
            ...
      restore_pool(save_pool);
  end construct_window_descr;
  procedure destruct_window_descr(window: in a_window_descr) is
     use POOL;
     -- the following pragma is unnecessary, since the window
     -- descriptor was allocated from the heap_pool; it is shown as
     -- an example only.
     pragma RTS_INTERFACE(AA_GLOBAL_FREE, aa_global_pool_free);
     procedure free_window_descr is new
         UNCHECKED_DEALLOCATION(a_window_descr);
 begin
     -- deallocate the window's pool first
     deallocate_pool(window.pool_descr);
     -- deallocate the descriptor itself;
     free_window_descr(window);
  end destruct_window_descr;
  ...
end window;
```

Figure 3-8 Using Pools

3.15 Suggestions for Use of SC Ada Memory Allocation

Heap allocation is generally suited to most applications for which deallocation is straightforward or unimportant. Deallocation requires the user be able to find every object that requires deallocation and call an UNCHECKED_DEALLOCATION instantiation to free it.

Worst case fragmentation occurs if something like the following occurs:

```
i := 1;
loop until storage error
    small(i) := new small_object;
    large(i) := new large_object;
    i := i + 1;
end loop;
i := 1;
loop until all large_object's are deallocated
    deallocate(large(i));
    i := i + 1;
end loop;
larger := new larger_object;
```

The last allocation fails even if the LARGER_OBJECT is only a little larger than LARGE_OBJECT. In this case, you can have megabytes of free memory but no hole large enough for one LARGER_OBJECT.

The more complicated FAT_MALLOC implementation is somewhat slower than SLIM_MALLOC for programs that do little or no deallocation. It is faster if the process lasts long enough to collect and reuse a lot of deallocated storage. The small object lists prevent fragmentation but cost time and space if the user does not know in advance which object sizes to use.

Pool-based memory deallocation has been demonstrated to be one of the easiest memory deallocation mechanisms to use. Programmers tend to have a general feel for how long their objects are needed and so can define a set of pools for each lifetime. They do not need to be able to walk through their data structures to locate garbage and dispose of it explicitly, object by object. Pool-based allocation should not be used where object lifetimes are random or unpredictable.

3.16 Mutual Exclusion During Allocation

User-space memory allocation, as implemented in the supplied SMPL_MALLOC, SLIM_MALLOC, FAT_MALLOC, DBG_MALLOC archives and POOL package, is a critical region protected via ABORT_SAFE mutexes. The implementations define an ABORT_SAFE mutex that must be locked before a task can enter the allocation / deallocation code. If the mutex is locked, the task must queue itself on that mutex's wait list.

Before return is made from allocation utilities such as AA_GLOBAL_NEW, the allocation mutex must be released, indicating that other tasks can now queue up to get their allocation. The mutex must also be released if a STORAGE_ERROR occurs during a call to AA_GLOBAL_NEW.

Since an ABORT_SAFE mutex is used, the task doing a memory allocation operation is inhibited from being completed by an Ada abort until it finishes the allocation operation and releases the mutex. In earlier releases of SC Ada, the memory allocations were not ABORT_SAFE.

package V_I_MUTEX in standard provides the interface to the ABORT_SAFE mutex services used by the allocation utilities.

References

"Allocation from Interrupt Handlers" on page 3-25

3.17 Memory Management and Underlying Operating Systems

When a program runs on top of an underlying operating system, there can be requests for heap memory from other libraries (for example, requests for buffers for disk transfers). It is imperative that memory allocations be internally consistent or these unknown buffers could be allocated in active program memory.

On UNIX-based systems, a single central allocation service, sbrk(2), is used by all programs. The default implementation of V_GET_HEAP_MEMORY in V_USR_CONF preserves this behavior by making a kernel call that in turn calls sbrk(2). Ada programs are thus consistent: user allocators call AA_GLOBAL_NEW, which probably satisfies the request directly and is protected by a mutex. Otherwise, V_GET_HEAP_MEMORY calls the Ada Kernel's alloc service, which is protected by the kernel's prohibition against executing multiple single threads and the request is satisfied by a call to sbrk(2).

If that package does not suit your needs, use the mutex mechanism or coding practice of serializing accesses to protect tasks that call programs written in other languages. For example, you should perform all I/O from a single task. Of particular concern are I/O operations, because they allocate large blocks of memory, either during file opens or during the first reads and writes to the files. To be absolutely safe, you must use the allocation mutex to make critical regions around all calls to foreign language utilities.

SunPro supplies an `ABORT_SAFE` mutex-protected `malloc` package discussed in the next section, *Protected Malloc.*

Without that `malloc` package, SC Ada does not protect its underlying calls to host OS utilities. It is therefore possible that multitasking, time-sliced programs will run into conflicts with unpredictable results if tasks are allowed to suspend while in the host OS. It is recommended that time-slicing (or other interrupts) be turned off for these tasks, that mutexes be used to protect them, or that all such activities be confined to a single task at any time.

3.18 Protected Malloc

In the `usr_conf` library in the file `malloc.a` there is a package called `MULTITASK_SAFE_MALLOC`. This package provides `ABORT_SAFE` mutex-protected versions of the C memory allocation routines (`malloc`, `calloc`, `valloc`, `memalign`, `realloc`, and `free`). The underlying allocation scheme is from the chosen allocation archive (`FAT_`, `SLIM_`, or `DBG_MALLOC`).

To use this package add the `usr_conf` library to your `ADAPATH` and `with MULTITASK_SAFE_MALLOC` somewhere in your program. This will cause the C `malloc` routines normally linked from the `libc.a` library to be superseded by the routines defined in this package when your program is linked. (You will also be using the user configuration from `v_usr_conf_b.a`.)

Alternatively, you can add a WITHn directive to one of your Ada libraries so that all programs linked in libraries where this directive is visible will use the routines from `MULTITASK_SAFE_MALLOC`. To add the directive use:

```
a.info -a WITH /user2/SCada/usr_conf/.objects/malloc02
```

The source to `MULTITASK_SAFE_MALLOC` is supplied so that, if necessary, users may modify it to suit the needs of their particular application.

Note – This package does not support the older semantics of REALLOC, which stated that any block of memory freed since the last call to MALLOC is valid as a parameter to REALLOC. The semantics of this in a multitasking system are undefined (e.g., which thread or threads of execution define “the last call to MALLOC?”) and are therefore not supported. SC Ada considers any block already freed to be untouchable and any program that violates this is erroneous. Programs using the REALLOC function should be checked for these semantics and fixed as necessary.

3.19 Replacing User-space Memory Allocation

Some users may have specialized performance needs that are not met by any of the supplied memory allocation packages. These users may want to implement their own algorithms. They should compile a new body for the V_I_ALLOC package specifications given earlier using SMPL_ALLOC as a starting point.

Some considerations:

- If multiple tasks can do memory allocation simultaneously, the critical region of the user algorithms should be protected via ABORT_SAFE mutexes. `package V_I_MUTEX` in `standard` contains the interface to the ABORT_SAFE mutex services.
- On a host operating system, be aware that standard memory allocators are not protected as critical regions and can be interrupted. Trouble arises if another task enters the allocator before the interrupted task completes its allocation. Also, calls to other languages can lead to unprotected calls for memory. For example, if TASK1 opens file1 and TASK2 opens file2, both tasks will probably call the OS to get memory for buffers and so forth. If TASK1 is suspended during the allocation, unpredictable results can occur. SC Ada does not protect users from this effect.
- Packages in the files `v_i_*.a`, found in `standard` are supplied to provide access to library services.

3.20 Memory Allocation Exerciser

Sun provides a package called `ALLOC_EXERCISER` that allows you to tune the small block sizes and other memory configuration parameters, compare the performance of one memory allocation implementation against another, or test the robustness of a user-written memory allocator.

One possible use is in tuning the memory allocation parameters for a specific application, if the memory allocation behavior has been previously characterized. In this way, without actually having to run the application, you can tune the memory allocation. It may not be able to reproduce the behavior of a specific application exactly, the memory allocation exerciser may come close enough to get some useful information.

The memory allocation exerciser provides the ability to specify:

- Object size
- Object lifetime
- Object alignment and the percentage of objects that need to be aligned
- Number of objects allocated / deallocated
- Whether blocks will be filled with a key value and checked at the time of deallocation for a proper key

Object size, alignment, and lifetime are specifiable either as a randomly chosen value in a specified range or as a set of specific values randomly chosen with weighting specified. For example, size could be specified as in the range of 20 to 1000 bytes or as a set of weighted specific sizes as in:

```
size (in bytes)   weight (in percentage)
20                  10
40                   7
128                 50
512                 30
4096                 3
```

The weights are not necessarily percentages, but it is convenient to think of them that way. The probability of a specific value coming up is `weight` / (sum of `weights`).

Object lifetime is measured as the number of allocations before deallocating a particular object. For example, a lifetime of one means that the object is deallocated immediately after it was allocated. A special value —eternity— is used as a specific lifetime when the object should never be deallocated.

Note – This program is not designed to exercise allocation from pools using `package POOL`.

Figure 3-9 is a program that uses the allocation exerciser.

```
with alloc_exerciser; use alloc_exerciser;
procedure example_exercise is
    s: object_size_descriptor_t(specific_sizes, 5);
    l: object_lifetime_descriptor_t(specific_lifetimes, 4);
    a: object_alignment_descriptor_t(random_alignments, 0);
begin
    s.sizes_and_weights := (
            (value => 16, weight => 40),
            (value => 50, weight => 25),
            (value => 1200, weight => 10),
            (value => 50800, weight => 17),
            (value => 4000, weight => 8));
    l.lifetimes_and_weights := (
            (value => 1, weight => 60),
            (value => 10, weight => 5),
            (value => 100, weight => 25),
            (value => eternity, weight => 10));
    -- Alignment is specified as the power of two to align
    -- to (non power of two alignments are not valid)
    -- e.g. 4 -> 2 ** 4 = 16; 14 -> 2 ** 14 = 16384
    a.low := 4; a.high := 14;
    a.percent_aligned := 1;
    exercise(
        size_desc => s,
        lifetime_desc => l,
        alignment_desc => a,
        use_keys => false,
        num_transactions => 3000);
end example_exercise;
```

Figure 3-9 Allocation Exerciser

Notice how each one of the descriptors is a record with two discriminants. The first discriminant says what sort of distribution to use and the second, if applicable, determines how many specific values are going to be used. The USE_KEYS flag is useful for testing an allocator to see if it is corrupting other blocks as it allocates and deallocates. If it is set to TRUE, the program fills the block with a random key value after the block has been allocated, and when the block is deallocated, it checks the block to make sure the key is in all locations of the block. In this mode it takes considerably more time to run.

The source is in the `examples` library as the files named `alloc_exer.a` and `alloc_exer_b.a`. The above example is in file `example_exer.a`. With some minor modifications, `example_exer.a` could be made into a benchmark by using calls to `package CALENDAR`.

References

`package ALLOC_DEBUG`, “Debug Heap Allocation: DBG_MALLOC” on page 3-28

"... this too has to be extended in order to display its patterns..."
Themistocles"

Ada Runtime Services 4

The packages in the file `v_vads_exec.a` in the `vads_exec` library provide the user interface to the following VADS EXEC services:

- Signal Handling (`package V_INTERRUPTS`)
- Mailboxes (`package V_MAILBOXES`)
- Memory Management (`package V_MEMORY`)
- Mutexes and Condition Variables (`package V_MUTEXES`)
- Name Services (`package V_NAMES`)
- Semaphores (`package V_SEMAPHORES`)
- Stack Operations (`package V_STACK`)
- Tasking Extensions (`package V_XTASKING`)

The packages contain subprograms, generics, data types, and exceptions that enable the user to directly invoke these services. These packages are layered on the Ada Kernel. The Ada Kernel provides all the threads and synchronization services needed to implement the VADS EXEC services.

The interface to the Ada Kernel services is defined in `package ADA_KRN_I`. Its type definitions are provided in `package ADA_KRN_DEFS`. Both packages are found in `standard`.

The remainder of this chapter provides an overview of the interface, followed by reference pages for each of the packages and their services.

References

"Ada Kernel Layer" on page 2-1

4

Overview of Interface

The VADS EXEC user interface provides these packages, functions, and procedures:

`package V_INTERRUPTS` — Provide interrupt processing for VADS EXEC with user-defined interrupt service routines (ISRs)

- `function ATTACH_ISR` — Attach an interrupt service routine to a given interrupt vector and return the previously attached ISR
- `procedure ATTACH_ISR` — Attach an interrupt service routine to a given interrupt vector
- `function CURRENT_INTERRUPT_STATUS` — Retrieve the current CPU interrupt status mask or priority level
- `function CURRENT_SUPERVISOR_STATE` — Return the supervisor / user state
- `function DETACH_ISR` — Detach an interrupt service routine from a given interrupt vector and return the previously attached ISR
- `procedure DETACH_ISR` — Detach an interrupt routine from a given interrupt vector
- `procedure ENTER_SUPERVISOR_STATE` — Enter the supervisor state for the current task, enabling execution of privileged instructions (This procedure is not supported for self hosts.)
- `generic procedure FAST_ISR` — Provide a faster version of the ISR generic. However, restrictions are imposed on the interrupt handler code
- `generic procedure FLOAT_WRAPPER` — Save and restore the state of the floating-point coprocessor (This generic procedure is not supported; it raises a `TASKING_ERROR` exception.)
- `function GET_ISR` — Return address of currently attached ISR
- `function GET_IVT` — Return address of current Interrupt Vector Table (IVT)
- `generic procedure ISR` — Provide the entry and exit code required of all interrupt service routines and perform the processing required upon entry and exit from an ISR

`procedure LEAVE_SUPERVISOR_STATE` — Exit the supervisor state for the current task, disabling execution of privileged instructions (This procedure is not supported.)

`function SET_INTERRUPT_STATUS` — Change the current interrupt status and return the previous interrupt status

`function SET_SUPERVISOR_STATE` — Set the supervisor/user state of the current task

`generic package V_MAILBOXES` — Provide mailbox operations for asynchronous passing of data between tasks or between an interrupt handler and a task

`procedure/function BIND_MAILBOX` — Bind a name to a mailbox

`procedure/function CREATE_MAILBOX` — Create and initialize a mailbox

`procedure DELETE_MAILBOX` — Delete a mailbox

`procedure READ_MAILBOX` — Retrieve a message from a mailbox

`procedure RESOLVE_MAILBOX` — Resolve a name into a mailbox

`procedure WRITE_MAILBOX` — Write a message into a mailbox

`function CURRENT_MESSAGE_COUNT` — Return the number of unread messages

`package V_MEMORY` — Provide memory management operations for FixedPools, FlexPools and HeapPools

`procedure CREATE_FIXED_POOL` — Create a FixedPool

`procedure CREATE_FLEX_POOL` — Create a FlexPool

`procedure CREATE_HEAP_POOL` — Create a HeapPool

`procedure DESTROY_FIXED_POOL` — Delete a FixedPool

`procedure DESTROY_FLEX_POOL` — Delete a FlexPool

`procedure DESTROY_HEAP_POOL` — Delete a Heap Pool

`generic function FIXED_OBJECT_ALLOCATION` — Allocate an object from the given FixedPool, initializing it with a specified value

`generic procedure FIXED_OBJECT_DEALLOCATION` — Deallocate the memory at the given location

`generic function FLEX_OBJECT_ALLOCATION` — Allocate an object from the given FlexPool, initializing it with a specified value

`generic procedure FLEX_OBJECT_DEALLOCATION` — Deallocate the memory at the given location

`generic function HEAP_OBJECT_ALLOCATION` — Allocate an object from the given HeapPool, initializing it with a specified value

`procedure INITIALIZE_SERVICES` — Initialize the memory management services

`package V_MUTEXES` — Provide mutexes and condition variables

`procedure BIND_COND` — Bind a name to a condition variable

`procedure BIND_MUTEX` — Bind a name to a mutex

`procedure BROADCAST_COND` — Broadcast a condition variable

`procedure/function CREATE_COND` — Create and initialize a condition variable

`procedure/function CREATE_MUTEX` — Create and initialize a mutex

`procedure DELETE_COND` — Delete a condition variable

`procedure DELETE_MUTEX` — Delete a mutex

`function GET_PRIORITY_CEILING_MUTEX` — Return a mutex priority ceiling

`procedure LOCK_MUTEX` — Attempt to lock a mutex

`procedure/function RESOLVE_COND` — Resolve a name into a condition variable

`procedure/function RESOLVE_MUTEX` — Resolve a name into a mutex

`procedure SET_PRIORITY_CEILING_MUTEX` — Set a mutex priority ceiling

`procedure SIGNAL_COND` — Signal a condition variable

procedure SIGNAL_UNLOCK_COND — Signal a condition variable / unlock mutex

procedure/function TIMED_WAIT_COND — Perform a timed block of a calling task

function TRYLOCK_MUTEX — Attempt to lock a mutex

procedure UNLOCK_MUTEX — Unlock a mutex

procedure WAIT_COND — Block a task on a condition variable

package V_NAMES — Provide name services

procedure BIND_OBJECT — Bind a name to the address of an object

procedure BIND_PROCEDURE — Bind a name to the program_id and address of a procedure

procedure/function RESOLVE_OBJECT — Resolve the name of an object into its address

procedure RESOLVE_PROCEDURE — Resolve the name of a procedure into its program_id and address

package V_SEMAPHORES — Provide binary and counting semaphores

procedure/function BIND_SEMAPHORE — Bind a name to a semaphore

procedure/function CREATE_SEMAPHORE — Create and initialize a semaphore

procedure DELETE_SEMAPHORE — Delete a semaphore

procedure RESOLVE_SEMAPHORE — Resolve a name into a semaphore

procedure SIGNAL_SEMAPHORE — Perform a signal operation on a semaphore

procedure WAIT_SEMAPHORE — Perform a wait operation on a semaphore

package V_STACK — Provide operations to control the stack

procedure CHECK_STACK — Determine the current value of the stack pointer and the stack lower bound

`procedure EXTEND_STACK` — Extend the current stack

`package V_XTASKING` — Provide operations to perform on Ada tasks and programs (extended tasking)

`function ALLOCATE_TASK_STORAGE` - Allocate storage in the task control block. The function returns the storage `ID` to be used in subsequent `GET_TASK_STORAGE` or GET_TASK_STORAGE2 service calls

`function CALLABLE` — Return the `P'CALLABLE` attribute for the specified task

`function CURRENT_EXIT_DISABLED` — Return the current value

`function CURRENT_FAST_RENDEZVOUS_ENABLED` — Return the value of the `FAST_RENDEZVOUS_ENABLED` flag of the current task. For details about the fast rendezvous optimization see page .

`function CURRENT_PRIORITY` — Return the priority of the specified task

`function CURRENT_PROGRAM` — Return the program ID of the current task

`function CURRENT_TASK` — Return the task ID of the current task

`function CURRENT_TIME_SLICE` — Return the current time slice interval of the specified task

`function CURRENT_TIME_SLICING_ENABLED` — Return the current value of the kernel `TIME_SLICING_ENABLED` configuration parameter

`function CURRENT_USER_FIELD` — Return the current value for the user-modifiable field of the specified task

`procedure DISABLE_PREEMPTION` — Inhibit the current task from being preempted. This procedure does not disable interrupts

`procedure DISABLE_TASK_COMPLETE` — Disable the current task from being completed and terminated when aborted. (Must be paired with `ENABLE_TASK_COMPLETE`.)

`procedure ENABLE_PREEMPTION` — Allow the current task to be preempted

`procedure ENABLE_TASK_COMPLETE` — Enable the current task to be completed and terminated when aborted. (Must be paired with `DISABLE_TASK_COMPLETE`.)

`function GET_PROGRAM` — Return the program ID of the specified task

`function GET_PROGRAM_KEY` - Return the user-defined key for the specified program

`function GET_TASK_STORAGE` — Return the starting address of the task storage area for the specified task and storage ID

`function GET_TASK_STORAGE2` — Return the starting address of the task storage area using the OS ID of the task

`function ID` — Return an identifier for an Ada program or task given the underlying OS program or task ID

`procedure INSTALL_CALLOUT` — Install a procedure to be called at a program exit, program switch, task create, task switch, task complete or idle event

`procedure INTER_PROGRAM_CALL` —Call a procedure in another program

`function OS_ID` — Return the underlying OS program or task ID given the Ada program or task ID

`procedure RESUME_TASK` — Ready the named task for execution

`procedure SET_EXIT_DISABLED` — Change the kernel `EXIT_DISABLE_FLAG` to inhibit or allow the program to exit

`procedure SET_FAST_RENDEZVOUS_ENABLED` — Change the `FAST_RENDEZVOUS_ENABLED` flag for the current task to a new value

`procedure SET_IS_SERVER_PROGRAM` — Mark the current program as a server

`procedure SET_PRIORITY` — Change the priority of the specified task to a new priority

`procedure SET_TIME_SLICE` — Change the time slice interval of the specified task

`procedure SET_TIME_SLICING_ENABLED` — Set the kernel `TIME_SLICING_ENABLED` configuration parameter

procedure SET_USER_FIELD — Set the user-modifiable field of the specified task to a new value

`function START_PROGRAM` — Start another, separately linked program identified by its link block and return the program `ID` of the just started program

`procedure SUSPEND_TASK` — Cause a running task to become suspended

`function TERMINATED` — Return the `P'TERMINATED` attribute for the specified task

`procedure TERMINATE_PROGRAM` — Terminate the specified program

`generic function V_ID` — Return an identifier for a task given a task object of the task type used to instantiate the generic

package V_INTERRUPTS — provide interrupt processing

Description

The `package V_INTERRUPTS` provides interrupt processing support for the user interfac, through user-defined interrupt service routines (ISRs).

Note – Interrupts are Solaris signals and the interrupt status mask is the signal mask. Throughout, an interrupt or interrupt vector is a Solaris signal and the interrupt status mask is the Solaris signal mask.

The VADS EXEC interface supports only signal handlers installed as an ISR with the `ATTACH_ISR` service. Alternatively, install signal handlers as a task interrupt entry. Avoid direct use of the Solaris `sigvec` or signal services. Upon entry to an ISR, the program performs the following actions:

- All asynchronous signals are blocked (they must remain blocked; VADS EXEC does not support nested asynchronous signals).
- The scratch registers are saved.
- If supported by the OS signal handler logic, the floating-point registers are saved.
- The frame pointer is updated so that the debugger and exception unwinding can deduce that this is the top of the stack for a signal handler.

The ISR performs user-defined signal handling actions. (Floating-point operations cannot be executed within an ISR unless your Solaris implementation saves and restores floating-point context for signals.) The ISR returns to enable VADS EXEC to complete signal handling and restore the saved context.sufficient

A generic procedure ISR provides a wrapper for the signal handler. The procedure should be instantiated with a parameterless procedure that performs the user-defined signal handling actions.

Pass the address of the procedure created by the ISR instantiation to `ATTACH_ISR` to attach the interrupt service routine to the appropriate signal. The ISR is subsequently detached by a call to `DETACH_ISR`.

References

"Ada Interrupt Entries as Interrupt Handlers" on page 2-56

Data References in ISRs

The user-defined interrupt handler must not reference non-local stack-relative data (including tasks declared within a subprogram). Instead, all references must be to objects declared in library-level package specifications or bodies, in the interrupt handler, or in subprograms called by the handler.

Exception Propagation in ISRs

Exceptions raised in an interrupt handler must be handled locally. An `others` exception handler should be provided to prevent attempting to propagate an exception from an interrupt handler. If you do not provide a handler for a raised exception (for example, `NUMERIC_ERROR`), the entire program is abandoned.

ISR/Ada Task Interaction

An ISR executes in the context of the task it interrupted. Since the ISR does not have its own task state, it must not perform any Ada tasking operations that affect the state of the interrupted task. Consequently, these Ada operations cannot be performed from inside an ISR:

- task creations
- accept statements
- entry calls
- delay statements
- abort statements
- evaluation of an allocator or deallocator (i.e., use of `new` or an instantiation of `UNCHECKED_DEALLOCATION`)

In addition, the ISR cannot invoke any VADS EXEC service that might block the interrupted task:

- `WAIT_SEMAPHORE`
- `READ_MAILBOX`
- any `V_MEMORY` allocator or deallocator
- `LOCK_MUTEX/UNLOCK_MUTEX` (depends on the attribute associated with the mutex)
- `WAIT_COND`

If the ISR attempts to call a service that would block, the TASKING_ERROR exception is raised.

The ISR may, however, invoke any nonblocking VADS EXEC service, including these:

- RESUME_TASK or SUSPEND_TASK
- SIGNAL_SEMAPHORE
- WRITE_MAILBOX
- SIGNAL_COND

Floating-point registers do not need to be saved before invoking VADS EXEC services from an ISR.

VADS EXEC tasking and interrupt services support preemptive task scheduling. If a call to a VADS EXEC service from an interrupt handler causes a task with a priority higher than that of the interrupted task to become ready to run, the higher priority task runs immediately upon return from the outermost interrupt service routine.

If the ISR calls only the kernel service IST_ENTER and ISR_COMPLETE, control is returned directly to the interrupted task.

Package Procedures and Functions

`function ATTACH_ISR`

Attach an interrupt service routine to a given interrupt vector and return the previously attached ISR

`procedure ATTACH_ISR`

Attach an interrupt service routine to a given interrupt vector

`function CURRENT_INTERRUPT_STATUS`

Retrieve the current CPU interrupt status mask or priority level

`function CURRENT_SUPERVISOR_STATE`

Return the supervisor / user state

`function DETACH_ISR`

Detach an interrupt service routine from a given interrupt vector and return the previously attached ISR

`procedure DETACH_ISR`

Detach an interrupt routine from a given interrupt vector

`procedure ENTER_SUPERVISOR_STATE`

Enter the supervisor state for the current task, enabling execution of privileged instructions. (This procedure is not supported for self hosts)

`generic procedure FAST_ISR`

Provide a faster version of the ISR. However, restrictions are imposed on the interrupt handler code

generic `procedure FLOAT_WRAPPER`

Save and restore the floating-point coprocessor. (This generic procedure is not supported for self hosts; it raises a TASKING_ERROR exception.)

`function GET_ISR`

Return address of currently attached ISR

`function GET_IVT`

Return address of current Interrupt Vector Table (IVT)

`generic procedure ISR`

Provide the entry and exit code required of all interrupt service routines and perform all processing required upon entering and exiting an ISR

`procedure LEAVE_SUPERVISOR_STATE`

Exit the supervisor state for the current task, disabling execution of privileged instructions (This procedure is not supported for self-hosts.)

`function SET_INTERRUPT_STATUS`

Change the current interrupt status and return the previous interrupt status

`function SET_SUPERVISOR_STATE`

Set the supervisor / user state of the current task

Types

`type VECTOR_ID`

Specifies the valid range for signal numbers

`type INTERRUPT_STATUS_T`

Specifies the interrupt status

Constants

`DISABLE_INTERRUPT`

Passed to SET_INTERRUPT_STATUS to disable all interrupts

`ENABLE_INTERRUPT`

Passed to SET_INTERRUPT_STATUS to enable all interrupts

Exceptions

`INVALID_INTERRUPT_VECTOR`

The vector number passed to an operation is an invalid vector number

`UNEXPECTED_V_INTERRUPTS_ERROR`

An unexpected error occurs in a `V_INTERRUPTS` routine

`VECTOR_IN_USE`

You’re attempting to attach an ISR to a vector already attached to an ISR

Example

```
  package ISR_EXAMPLE is
  end;
  with V_INTERRUPTS;
  with SYSTEM;
  package body ISR_EXAMPLE is
  INTEGER_CNT: integer := 0;
  FLOAT_CNT: float := 0.0;
  procedure INTEGER_HANDLER is
  begin
      -- No floating point operations
    INTEGER_CNT := INTEGER_CNT + 1;
  end;
  procedure FLOAT_HANDLER is
  begin
      -- Does floating point operations
    FLOAT_CNT := FLOAT_CNT + 1.0;
  end
 procedure INTEGER_ISR is new V_INTERRUPTS.ISR(INTEGER_HANDLER);
      -- An instantiated isr that doesn't perform any floating
      -- point operations.
  procedure WRAPPED_FLOAT_HANDLER is new
            V_INTERRUPTS.FLOAT_WRAPPER(FLOAT_HANDLER);
      -- An instantiated subprogram with floating point state
      -- saved and initialized upon entry and then restored
      -- upon return. This wrapped procedure is passed when
      -- the isr is instantiated.
  procedure FLOAT_ISR is new
V_INTERRUPTS.ISR(WRAPPED_FLOAT_HANDLER);
      -- An instantiated isr that performs floating point
operations.
  begin
    declare
         prev_isr: SYSTEM.address;
    begin
      -- Attach above instantiated isr's
      prev_isr := V_INTERRUPTS.ATTACH_ISR(16#80#,
INTEGER_ISR'address);
      prev_isr := V_INTERRUPTS.ATTACH_ISR(16#81#,
FLOAT_ISR'address);
    end;
 end ISR_EXAMPLE;
```

Figure 4-1 Utilizing Services within a Handling Routine

Package Specification

```
with ADA_KRN_DEFS;
with SYSTEM;
package V_INTERRUPTS is
pragma SUPPRESS(ALL_CHECKS);
type VECTOR_ID is new ADA_KRN_DEFS.intr_vector_id_t;
type INTERRUPT_STATUS_T is new ADA_KRN_DEFS.intr_status_t;
ENABLE_INTERRUPT  : constant INTERRUPT_STATUS_T :=
    INTERRUPT_STATUS_T(ADA_KRN_DEFS.ENABLE_INTR_STATUS);
DISABLE_INTERRUPT : constant INTERRUPT_STATUS_T :=
    INTERRUPT_STATUS_T(ADA_KRN_DEFS.DISABLE_INTR_STATUS);
INVALID_INTERRUPT_VECTOR         : exception;
VECTOR_IN_USE                    : exception;
UNEXPECTED_V_INTERRUPTS_ERROR : exception;
generic
    with procedure INTERRUPT_HANDLER;
procedure ISR;
pragma SHARE_CODE(ISR, FALSE);
generic
    with procedure INTERRUPT_HANDLER;
procedure FAST_ISR;
pragma SHARE_CODE(FAST_ISR, FALSE);
generic
    with procedure FLOAT_HANDLER;
procedure FLOAT_WRAPPER;
pragma SHARE_CODE(FLOAT_WRAPPER, FALSE);
function ATTACH_ISR
    (vector    : in     vector_id;
    isr        : in     system.address) return system.address;

procedure ATTACH_ISR
    (vector    : in     vector_id;
    isr        : in     system.address);

function DETACH_ISR(vector : in vector_id) return system.address;

procedure DETACH_ISR(vector : in vector_id);
function GET_ISR
          (vector     : in     vector_id) return system.address;
function GET_IVT return system.address;
function CURRENT_INTERRUPT_STATUS return interrupt_status_t;
```

(Continued)

```
function SET_INTERRUPT_STATUS(new_status : in
interrupt_status_t)
    return interrupt_status_t;

function CURRENT_SUPERVISOR_STATE return boolean;

procedure ENTER_SUPERVISOR_STATE;

procedure LEAVE_SUPERVISOR_STATE;

function SET_SUPERVISOR_STATE(new_state : in boolean) return
boolean;

end V_INTERRUPTS;
```

procedure/function ATTACH_ISR — attach ISR to interrupt vector

Syntax

```
procedure ATTACH_ISR
        (vector     : in VECTOR_ID;
        isr         : in SYSTEM.ADDRESS);
function ATTACH_ISR
        (vector     : in VECTOR_ID;
        isr         : in SYSTEM.ADDRESS)
        return SYSTEM.ADDRESS;
```

Arguments

isr

The address of the interrupt service routine.

vector

The interrupt vector number to which the interrupt service routine is to be attached. The vector number is the number of the interrupt vector, not the vector offset. For your operating system, the signal number is 1-32.

Description

`procedure ATTACH_ISR` is used to attach the ISR at *isr* to the interrupt vector indicated by *vector*.

`function ATTACH_ISR` attaches the ISR and returns the address of the previously attached ISR.

Exceptions

`INVALID_INTERRUPT_VECTOR`

The vector number is out-of-range.

`VECTOR_IN_USE`

Indicates that an interrupt handler has already been assigned. (`ATTACH_ISR` no longer raises this exception because `ATTACH_ISR` overrides any previously attached ISRs. For compatibility purposes, this exception is still described.)

Threaded Runtime

`ATTACH_ISR` is layered upon `ADA_KRN_I.ISR_ATTACH`.

function CURRENT_INTERRUPT_STATUS — return mask/priority setting

Syntax

```
function CURRENT_INTERRUPT_STATUS
        return interrupt_status_t;
```

Description

`CURRENT_INTERRUPT_STATUS` returns the current CPU interrupt status mask or priority level.

Threaded Runtime

`CURRENT_INTERRUPT_STATUS` is layered upon `ADA_KRN_I.INTERRUPTS_GET_STATUS`.

function CURRENT_SUPERVISOR_STATE — return supervisor/user state

Syntax

```
function CURRENT_SUPERVISOR_STATE
        return boolean;
```

Description

CURRENT_SUPERVISOR_STATE returns the supervisor/user state of the current task. If the task is in supervisor state, TRUE is returned. If the task is in user state, FALSE is returned.

Threaded Runtime

CURRENT_SUPERVISOR_STATE is layered upon ADA_KRN_I.TASK_GET_SUPERVISOR_STATE.

4

function/procedure DETACH_ISR — detach ISR from interrupt vector

Syntax

```
procedure DETACH_ISR
        (vector : in  VECTOR_ID);
function DETACH_ISR
        (vector : in  VECTOR_ID)
        return SYSTEM.ADDRESS;
```

Arguments

vector

The number of the interrupt vector to be detached. The vector number is the number of the interrupt vector, not the offset of the vector. For your operating system, the signal number is 1-32.

Description

`procedure DETACH_ISR` detaches an interrupt routine from a specified interrupt vector. The attachment is performed during kernel initialization if the vector is given an initial value in the interrupt vector table.

The default handler is reinstated.

`function DETACH_ISR` detaches the ISR and returns the address of the previously attached ISR.

Exceptions

`INVALID_INTERRUPT_VECTOR`

The vector number is out-of-range.

Threaded Runtime

`DETACH_ISR` is layered upon `ADA_KRN_I.ISR_DETACH`.

procedure ENTER_SUPERVISOR_STATE — enter task supervisor state

Syntax

```
procedure ENTER_SUPERVISOR_STATE
```

Description

`ENTER_SUPERVISOR_STATE` enables you to enter the supervisor state for the current task. While in supervisor state, you can execute privileged instructions.

Threaded Runtime

`ENTER_SUPERVISOR_STATE` is layered upon `ADA_KRN_I.TASK_ENTER_SUPERVISOR_STATE`.

Caution – `ENTER_SUPERVISOR_STATE` is not supported for self-hosts.

generic procedure FAST_ISR — faster version of the ISR generic

Syntax

```
generic
        with procedure INTERRUPT_HANDLER;
procedure FAST_ISR;
pragma SHARE_CODE (FAST_ISR, FALSE);
```

Description

The generic procedure `FAST_ISR` is instantiated with a parameterless procedure that provides the handler for a particular interrupt. The address of the resulting instantiation is passed to `ATTACH_ISR` to attach the interrupt service routine to a particular vector.

For SPARC cross targets, `FAST_ISR` is identical to the ISR generic.

`FAST_ISR` is identical to the ISR generic.

generic procedure FLOAT_WRAPPER — save/restore floating-point state

Syntax

```
generic
    with procedure FLOAT_HANDLER;
procedure FLOAT_WRAPPER;
pragma SHARE_CODE (FLOAT_WRAPPER, FALSE);
```

Description

`FLOAT_WRAPPER` saves and restores the floating-point state. Before procedure `FLOAT_HANDLER` is called, the floating-point state is reset and float exceptions enabled according to the `FLOATING_POINT_CONTROL` parameter in the kernel program configuration package `V_KRN_CONF`. (This generic procedure is not supported; it raises a `TASKING_ERROR` exception.)

Caution – `FLOAT_WRAPPER` is not supported for self-hosts.

function GET_ISR — return address of currently attached ISR

Syntax

```
function GET_ISR
          (vector : in vector_id) return system.address;
```

Arguments

vector

The interrupt vector number of the ISR.

Description

`function GET_ISR` returns the address of the currently attached ISR for the given interrupt vector.

Exceptions

`INVALID_INTERRUPT_VECTOR`

Raised if the vector number is out-of-range.

function GET_IVT — return address of current Interrupt Vector Table (IVT)

Syntax

```
function GET_IVT return system.address;
```

Description

`GET_IVT` returns the address of the current Interrupt Vector Table (IVT). Normally, the IVT is an array of ISR addresses. However, the IVT representation is CPU dependent.

generic procedure ISR — provide entry and exit codes for all ISRs

Syntax

```
generic
    with procedure INTERRUPT_HANDLER;
procedure ISR;
pragma SHARE_CODE(ISR, FALSE);
```

Description

ISR provides the wrapper required for all interrupt service routines. It can be instantiated with a parameterless procedure that provides the handler for a particular interrupt. The address of the resulting instantiation is passed to ATTACH_ISR to attach the interrupt service routine to a particular vector.

Caution – This wrapper does not save/restore the floating-point context.

Caution – This generic must be instantiated at the library package level.

procedure LEAVE_SUPERVISOR_STATE — exit task supervisor state

Syntax

```
procedure LEAVE_SUPERVISOR_STATE
```

Description

`LEAVE_SUPERVISOR_STATE` exits you from the supervisor state for the current task. After exiting the supervisor state, you can no longer execute privileged instructions. Any attempt to execute a privileged instruction when you are not in supervisor state causes a CPU exception.

If the configuration variable `V_KRN_CONF.SUPERVISOR_TASKS_ENABLED` is `TRUE`, all tasks are in supervisor state all the time. In this situation, the procedure `LEAVE_SUPERVISOR_STATE` has no effect.

Threaded Runtime

`LEAVE_SUPERVISOR_STATE` is layered upon `ADA_KRN_I.TASK_LEAVE_SUPERVISOR_STATE`.

Caution – `LEAVE_SUPERVISOR_STATE` is not supported for self hosts.

function SET_INTERRUPT_STATUS — change mask or priority setting

Syntax

```
function SET_INTERRUPT_STATUS
        (new_status :in INTERRUPT_STATUS_T)
        return INTERRUPT_STATUS_T;
```

Arguments

new_status

The new interrupt status setting

Description

The function SET_INTERRUPT_STATUS changes the setting of the interrupt status mask or priority and returns the previous interrupt status.

Threaded Runtime

SET_INTERRUPT_STATUS is layered upon ADA_KRN_I.INTERRUPTS_SET_STATUS.

function SET_SUPERVISOR_STATE — change supervisor/user state of task

Syntax

```
function SET_SUPERVISOR_STATE
        (new_state :  in BOOLEAN)
        return boolean;
```

Arguments

new_state

The new supervisor/user state. If TRUE, the task is placed in supervisor state. If FALSE, the task is placed in user state.

Description

The function SET_SUPERVISOR_STATE changes the supervisor/user state for the current task. The previous state is returned.

Threaded Runtime

SET_SUPERVISOR_STATE is layered upon the following services in package ADA_KRN_I:

TASK_GET_SUPERVISOR_STATE

TASK_ENTER_SUPERVISOR_STATE

TASK_LEAVE_SUPERVISOR_STATE

Caution – SET_SUPERVISOR_STATE is not supported forself-hosts.

4

package V_MAILBOXES — provide mailbox operations

Syntax

```
generic
  type MESSAGE_TYPE is private;
package V_MAILBOXES
```

Description

`package V_MAILBOXES` is a generic package that provides mailbox operations. Use mailboxes for the unsynchronized passing of data between tasks or between an interrupt handler and a task. `V_MAILBOXES` has been layered upon the Ada Kernel mailbox objects.

The procedure `CREATE_MAILBOX` creates a mailbox for passing objects of the type used to instantiate the generic package.

`READ_MAILBOX` reads messages from a mailbox on a first-in/first-out (FIFO) basis. A parameter can be used to specify how long the task is willing to wait for a message if one is not immediately available. The queuing order for tasks waiting to read a message depends on the attributes passed to `CREATE_MAILBOX`. The default is FIFO.

`WRITE_MAILBOX` writes a message to the mailbox, awakening a task waiting on the mailbox if any are waiting. If the awoken task is of sufficient priority, it preempts the current running task.

`DELETE_MAILBOX` deletes a mailbox. The user must specify the action to be taken if there are currently tasks waiting on the mailbox.

`READ_MAILBOX` and `WRITE_MAILBOX` are declared as overloaded procedures to allow the user to select the method of error notification. If a result parameter is provided in the call, a result status is returned in the parameter; otherwise an exception is raised if an error occurs.

`BIND_MAILBOX` and `RESOLVE_MAILBOX` allow a mailbox to be shared between programs. `BIND_MAILBOX` binds a name to a mailbox previously created in the current program. `RESOLVE_MAILBOX` resolves a name into a mailbox that was created and bound in another program.

`MESSAGE_TYPE` is the type of object that can be passed using the operations provided by an instantiation.

If a task is aborted while accessing a mailbox, the resource is permanently locked.

Package Procedures And Functions

`procedure BIND_MAILBOX`

Bind a name to a mailbox.

`procedure CREATE_MAILBOX`

Create and initialize a mailbox.

`procedure DELETE_MAILBOX`

Delete a mailbox.

`procedure READ_MAILBOX`

Retrieve a message from a mailbox.

`procedure/function RESOLVE_MAILBOX`

Resolve a name into a mailbox.

`procedure WRITE_MAILBOX`

Write a message into a mailbox.

`function CURRENT_MESSAGE_COUNT`

Return the number of unread messages.

Types

`MAILBOX_DELETE_OPTION`

Specifies the action to take during a call to `DELETE_MAILBOX` if tasks are waiting on the mailbox that is to be deleted or if the mailbox contains messages. Possible values are:

```
DELETE_OPTION_FORCE
DELETE_OPTION_WARNING
```

MAILBOX_ID

A private type used to identify mailboxes. The compiler type checks mailbox parameters to ensure that the buffer type is consistent with the mailbox ID.

MESSAGE_TYPE

Can be passed using the operations provided by an instantiation

MAILBOX_RESULT

Returns the completion status of the versions of READ_MAILBOX and WRITE_MAILBOX that return a status. Possible values are:

```
DELETED
EMPTY
FULL
RECEIVED
SENT
TIMED_OUT
```

References

"procedure DELETE_MAILBOX — remove mailbox" on page 4-43

"procedure READ_MAILBOX — retrieve message" on page 4-45

"procedure WRITE_MAILBOX — deposit message" on page 4-49

Constants

WAIT_FOREVER

Passed to the WAITTIME parameter of READ_MAILBOX to specify that the mailbox should be waited on until a message is received . Additionally, this constant may be used with RESOLVE_MAILBOX to specify waiting indefinitely or not at all for the name to be bound.

DO_NOT_WAIT

Passed to the WAITTIME parameter of READ_MAILBOX to specify that a non-waited read of the mailbox should be performed . Additionally, this constant may be used with RESOLVE_MAILBOX to specify waiting indefinitely or not at all for the name to be bound.

Exceptions

`BIND_MAILBOX_BAD_ARGUMENT`

Raised if `BIND_MAILBOX` is called with a null name.

`BIND_MAILBOX_NOT_SUPPORTED`

Raised if name services are not supported by the underlying RTS.

`INVALID_MAILBOX`

Raised if the mailbox ID passed to a mailbox operation does not identify an existing mailbox.

`MAILBOX_DELETED`

Raised by `READ_MAILBOX` if the mailbox is deleted during the read operation.

`MAILBOX_EMPTY`

Raised by `READ_MAILBOX` if an unwaited read is performed and there is no message in the mailbox.

`MAILBOX_FULL`

Raised by `WRITE_MAILBOX` if an attempt is made to write to a full mailbox.

`MAILBOX_IN_USE`

Raised by `DELETE_MAILBOX` if you attempt to delete a mailbox that a task is waiting for and the delete option is `DELETE_OPTION_WARNING`.

`MAILBOX_NAME_ALREADY_BOUND`

Raised if an attempt is made to bind a name to a mailbox and the name has already been bound to another object or procedure.

`MAILBOX_NOT_EMPTY`

Raised by `DELETE_MAILBOX` when you attempt to delete a mailbox that is not empty and the delete option is `DELETE_OPTION_WARNING`.

MAILBOX_TIMED_OUT

Raised by READ_MAILBOX if a timed wait is performed and a message does not arrive in the specified time interval.

NO_MEMORY_FOR_MAILBOX

Raised by CREATE_MAILBOX if insufficient memory is available to create the mailbox.

NO_MEMORY_FOR_MAILBOX_NAME

Raised if an attempt is made to bind a name to a mailbox and there is insufficient memory.

NOT_A_MAILBOX_NAME

Raised by RESOLVE_MAILBOX if the name is bound, but not to a mailbox object.

RESOLVE_MAILBOX_BAD_ARGUMENT

Raised if RESOLVE_MAILBOX is called with a null name.

RESOLVE_MAILBOX_FAILED

Raised by RESOLVE_MAILBOX if a non-waited attempt was made to resolve a name to a mailbox, and the name wasn't already bound.

RESOLVE_MAILBOX_NOT_SUPPORTED

Raised if name services are not supported by the underlying RTS.

RESOLVE_MAILBOX_TIMED_OUT

Raised by RESOLVE_MAILBOX if a timed wait was attempted and the name wasn't bound to a mailbox in the given time interval.

UNEXPECTED_V_MAILBOX_ERROR

Raised if an unexpected error occurs during a V_MAILBOX operation.

Example

This declaration creates a package that provides operations to pass characters via mailboxes.

```
with V_MAILBOXES;
package CHAR_BOX is new V_MAILBOXES
  (MESSAGE_TYPE => CHARACTER);
```

The following declaration creates a package providing signal operations via mailboxes (i.e., messages of null length);

```
type signal_t is array(integer 1..0) of integer;
package signal is new V_MAILBOXES(MESSAGE_TYPE =>
signal_t);
```

Package Specification

```
with ADA_KRN_DEFS;
  generic
type MESSAGE_TYPE is private;
package V_MAILBOXES is

    pragma suppress(ALL_CHECKS);
    type mailbox_id is private;

    type mailbox_delete_option is
(DELETE_OPTION_FORCE,DELETE_OPTION_WARNING);

    WAIT_FOREVER : constant duration := -1.0;
    DO_NOT_WAIT  : constant duration := 0.0;
    NO_MEMORY_FOR_MAILBOX:exception;
    INVALID_MAILBOX:exception;
    MAILBOX_TIMED_OUT:exception;
    MAILBOX_EMPTY:exception;
    MAILBOX_DELETED:exception;
    MAILBOX_FULL:exception;
    MAILBOX_NOT_EMPTY:exception;
    MAILBOX_IN_USE:exception;
    UNEXPECTED_V_MAILBOX_ERROR:exception;
    BIND_MAILBOX_NOT_SUPPORTED:exception;
    BIND_MAILBOX_BAD_ARGUMENT:exception;
    NO_MEMORY_FOR_MAILBOX_NAME:exception;
    MAILBOX_NAME_ALREADY_BOUND:exception;
    RESOLVE_MAILBOX_NOT_SUPPORTED:exception;
    RESOLVE_NAILBOX_BAD_ARGUMENT:exception;
    RESOLVE_MAILBOX_TIMED_OUT:exception;
    RESOLVE_MAILBOX_FAILED:exception;
```

(Continued)

```
    NOT_A_MAILBOX_NAME:exception;
    type mailbox_result is (SENT,RECEIVED,TIMED_OUT,FULL,EMPTY,DELETED);
    procedure BIND_MAILBOX
    (name    : in  string;
    mailbox   :in  mailbox_id);
    procedure CREATE_MAILBOX
    (numberslots:inpositive := 1;
    mailbox    :out   mailbox_id;
    attr       :in    ADA_KRN_DEFS.a_mailbox_attr_t :=
                          ADA_KRN_DEFS.DEFAULT_MAILBOX_ATTR);
    function CREATE_MAILBOX
    (numberslots:inpositive := 1;
    attr       :in    ADA_KRN_DEFS.a_mailbox_attr_t :=
                          ADA_KRN_DEFS.DEFAULT_MAILBOX_ATTR);return
mailbox_id;
    function CURRENT_MESSAGE_COUNT
    (mailbox:  in  mailbox_id) return natural;

    procedure DELETE_MAILBOX
    (mailbox:  in  mailbox_id;
    delete_option:inmailbox_delete_option);
    procedure READ_MAILBOX
    (mailbox:  in  mailbox_id;
    waittime:  in  duration;
    message    :out message_type);

    procedure READ_MAILBOX
    (mailbox:  in  mailbox_id;
    waittime:  in  duration;
    message    :out   message_type;
    result     :out   mailbox_result);
    procedure RESOLVE_MAILBOX
    (name          : in     string;
    mailbox       : outmailbox_id;
    wait_time     : in     duration := WAIT_FOREVER);
    function RESOLVE_MAILBOX
    (name          : in     string;
    wait_time     :in     duration := WAIT_FOREVER)
    return mailbox_id;
    procedure WRITE_MAILBOX
    (mailbox:  in  mailbox_id;
    message    :in    message_type);
    procedure WRITE_MAILBOX
```

(Continued)

```
    (mailbox:  in  mailbox_id;
    message    :in    message_type;
    result     :out   mailbox_result);
private
    type mailbox_rec;
    type mailbox_id is access mailbox_rec;
end V_MAILBOXES;
pragma SHARE_BODY(V_MAILBOXES, FALSE);
```

procedure BIND_MAILBOX — bind name to a mailbox

Syntax

```
procedure BIND_MAILBOX
            (name : in string;
             mailbox : in mailbox_id);
```

Arguments

name

Mailbox's name. The name can be any sequence of characters. An exact match is done for all name searches. (MY_MAILBOX differs from my_mailbox.)

mailbox

The mailbox to be bound.

Description

BIND_MAILBOX binds a name to a previously created mailbox.

Exceptions

NO_MEMORY_FOR_MAILBOX_NAME

name could not be bound because of insufficient memory.

BIND_MAILBOX_NOT_SUPPORTED

name services are not supported by the underlying RTS.

BIND_MAILBOX_BAD_ARGUMENT

A null name was passed.

MAILBOX_NAME_ALREADY_BOUND

name was already bound to another object or procedure.

Note – For this product, names are only known within a single program. Name binding and resolution services are more applicable to the cross-target environment.

function/procedure CREATE_MAILBOX — create mailbox

Syntax

```
function CREATE_MAILBOX
        (numberslots: in positive := 1;
        attr : in ADA_KRN_DEFS.a_mailbox_attr_t :=
                                    ADA_KRN_DEFS.DEFAULT_MAILBO
X_ATTR)
        return mailbox_id;
procedure CREATE_MAILBOX
        (numberslots: in positive := 1;
        mailbox     : out mailbox_id;
        attr        : in ADA_KRN_DEFS.a_mailbox_attr_t :=
                              ADA_KRN_DEFS.DEFAULT_MAILBOX_A
TTR);
```

Arguments

attr

Points to an ADA_KRN_DEFS.MAILBOX_ATTR_T record. This record contains the mailbox attributes, which depend on the underlying threaded runtime. It is defined in ada_krn_defs.a found in standard.

The attr parameter has been defaulted to DEFAULT_MAILBOX_ATTR. Unless you want to do something special, the default will suffice. VADS MICRO defaults to FIFO queuing when a task blocks waiting to read a message.

For VADS MICRO, use ADA_KRN_DEFS.DEFAULT_MAILBOX_INTR_ATTR to protect the mailbox critical region by disabling all interrupts. Use the subprogram ADA_KRN_DEFS.MAILBOX_INTR_ATTR_INIT to initialize the attributes so that a subset of the interrupts are disabled.

If the mailbox is accessed from an ISR, it must be protected by disabling interrupts.

mailbox

The created mailbox.

numberslots

The number of message slots to have in the mailbox (maximum number of messages the mailbox can hold).

Description

CREATE_MAILBOX creates and initializes a mailbox. Two versions of the call are supplied: a procedure that returns the mailbox ID as an out parameter and a function returning the mailbox ID. To notify other tasks of the mailbox ID, place the MAILBOX_ID variable at a package level.

The created mailbox can be shared with other programs by using the BIND_MAILBOX/RESOLVE_MAILBOX services.

Exceptions

NO_MEMORY_FOR_MAILBOX

Insufficient memory is available to create the mailbox.

Threaded Runtime

CREATE_MAILBOX is layered upon ADA_KRN_I.MAILBOX_INIT. See the source code for ADA_KRN_I.MAILBOX_INIT for more details about mailbox attributes.

function CURRENT_MESSAGE_COUNT — return number of unread messages

Syntax

```
function CURRENT_MESSAGE_COUNT
        (mailbox  : in  mailbox_id)
 return natural;
```

Arguments

mailbox

The mailbox for which to get the message count.

Description

The function CURRENT_MESSAGE_COUNT takes one parameter, the mailbox ID, and returns the number of unread messages currently in the specified mailbox.

Threaded Runtime

CURRENT_MESSAGE_COUNT is layered upon ADA_KRN_I.MAILBOX_GET_COUNT.

procedure DELETE_MAILBOX — remove mailbox

Syntax

```
procedure DELETE_MAILBOX
        (mailbox            : in mailbox_id;
        delete_option       : in mailbox_delete_option);
```

Arguments

delete_option

The action to be taken if the mailbox is in use or not empty. Possible values are

DELETE_OPTION_FORCE

Ready all waiting tasks. These task calls to READ_MAILBOX raise the exception MAILBOX_DELETED or return the value DELETED.

The mailbox is deleted.

DELETE_OPTION_WARNING

If there are messages in the mailbox, raise the exception MAILBOX_NOT_EMPTY in the calling task.

If tasks are waiting at the mailbox, raise the exception MAILBOX_IN_USE in the calling task.

The mailbox is not deleted.

mailbox

Mailbox to be deleted

Description

The procedure DELETE_MAILBOX removes a mailbox from the system. Deleting a mailbox releases all of the buffer space used for the mailbox.

If a mailbox is shared between programs by using the BIND_MAILBOX/RESOLVE_MAILBOX services, it must be deleted only in the program where it was created.

Exceptions

`INVALID_MAILBOX`

`mailbox_id` does not identify an existing mailbox.

`MAILBOX_IN_USE`

`mailbox` is not empty and *`delete_option`* is `DELETE_OPTION_WARNING`.

`MAILBOX_NOT_EMPTY`

Tasks waiting at mailbox and *`delete_option`* is `DELETE_OPTION_WARNING`.

Threaded Runtime

`DELETE_MAILBOX` is layered upon `ADA_KRN_I.MAILBOX_DESTROY`.

procedure READ_MAILBOX — retrieve message

Syntax

```
procedure READ_MAILBOX
 (mailbox : in mailbox_id;
 waittime : in duration;
 message  :out message_type);

procedure READ_MAILBOX
 (mailbox : in mailbox_id;
 waittime : in duration;
 message: out message_type;
 result  :out mailbox_result);
```

Arguments

mailbox

Mailbox to be read.

message

Message read from the mailbox.

waittime

Time (in duration seconds) to wait for a message.

result

Result of the read from the mailbox.

Description

procedure READ_MAILBOX retrieves messages from a specified mailbox. To indicate whether the task waits forever or not at all for a message, specify the constants WAIT_FOREVER and DO_NOT_WAIT, respectively.

Do not call READ_MAILBOX from an interrupt service routine unless *waittime* is specified as DO_NOT_WAIT.

Exceptions/Results

INVALID_MAILBOX

mailbox_id specifies a non-existent mailbox.

MAILBOX_DELETED/DELETED

mailbox is deleted during the read operation.

MAILBOX_EMPTY/EMPTY

An unwaited read is performed and there is no message in the mailbox.

MAILBOX_TIMED_OUT/TIMED_OUT

Timed read is performed and no message is received in the specified time interval.

/RECEIVED

Result returned if a message is read.

Threaded Runtime

READ_MAILBOX is layered upon ADA_KRN_I.MAILBOX_READ.

procedure/function RESOLVE_MAILBOX — resolve name into a mailbox

Syntax

```
procedure RESOLVE_MAILBOX
            (name          : in string;
             mailbox       : out mailbox_id;
             wait_time     : in duration := WAIT_FOREVER);
function RESOLVE_MAILBOX
            (name          : in     string;
            wait_time     : in     duration := WAIT_FOREVER)
      return mailbox_id;
```

Arguments

name

Mailbox's name.

mailbox

The mailbox resolved.

wait_time

Amount of time the user is willing to wait for the name to be bound. Two predefined values that are allowable are:

WAIT_FOREVER

Wait forever for the name to be bound.

DO_NOT_WAIT

Do not wait if the name is not already bound.

Description

RESOLVE_MAILBOX resolves a name into a mailbox that was created and bound in another program. Two versions are supplied, a procedure that returns mailbox id as an out parameter or a function returning the mailbox id. RESOLVE_MAILBOX first attempts to find an already bound name that exactly matches the name parameter. For a match, it returns immediately. Otherwise, it returns according to the wait_time parameter.

Exceptions

`NOT_A_MAILBOX_NAME`

Raised by `RESOLVE_MAILBOX` if the name is bound, but not to a mailbox object.

`RESOLVE_MAILBOX_BAD_ARGUMENT`

Raised if `RESOLVE_MAILBOX` is called with a null name.

`RESOLVE_MAILBOX_FAILED`

Raised by `RESOLVE_MAILBOX` if a non-waited attempt was made to resolve a name to a mailbox, and the name wasn't already bound.

`RESOLVE_MAILBOX_NOT_SUPPORTED`

Raised if name services are not supported by the underlying RTS.

`RESOLVE_MAILBOX_TIMED_OUT`

Raised by `RESOLVE_MAILBOX` if a timed wait was attempted and the name wasn't bound to a mailbox in the given time interval.

Note – For this product, names are only known within a single program. Name binding and resolution services are more applicable to the cross-target environment.

---

procedure WRITE_MAILBOX — deposit message

Syntax

```
procedure WRITE_MAILBOX
        (mailbox    : in mailbox_id;
        message     : in message_type);

procedure WRITE_MAILBOX
        (mailbox    : in mailbox_id;
        message     : in message_type;
        result      : out mailbox_result);
```

Arguments

mailbox

Mailbox to be written to.

message

Message to be written to the mailbox.

result

Result of the write to the mailbox.

Description

If any tasks are waiting at the mailbox, the first-queued waiting task receives the message and becomes ready-to-run. If that task is of higher priority than the current task, it preempts the current task.

WRITE_MAILBOX can be called from an interrupt service routine. If the write operation awakens a task with a higher priority than the interrupted task, the interrupted task is preempted upon completion of interrupt processing.

The interrupt protected form of mailbox must be used when the mailbox is written from an ISR.

Exceptions/Results

`INVALID_MAILBOX`

`mailbox_id` specifies a non-existent mailbox.

`MAILBOX_FULL/FULL`

The mailbox is full.

`/SENT`

Result returned if the message is sent.

Threaded Runtime

`WRITE_MAILBOX` is layered upon `ADA_KRN_I.MAILBOX_WRITE`.

References

“package V_INTERRUPTS — provide interrupt processing” on page 4-9

package V_MEMORY — provide memory management operations

Description

`package V_MEMORY` provides memory management operations through three distinct methods: *FixedPools*, *FlexPools* and *HeapPools*.

For each type of pool, operations are provided to enable creation, deletion, allocation and deallocation (with the exception that deallocation is not provided for *HeapPools*). The allocation and deallocation operations are implemented as generics that can be instantiated to create allocators and deallocators for a given type of object for a given type of pool.

*FixedPool*s are made up of constant-sized memory blocks, providing fast allocation and deallocation with constant time overheads. Fixed memory pools are best suited for storage of objects of relatively homogeneous size.

FlexPools contain variably sized memory blocks, providing efficient use of memory. The problem of fragmentation, inherent in variably sized allocation schemes, is diminished by the VADS EXEC concept of granularity, whereby blocks are allocated on specified boundaries. *FlexPools* are organized around a standard boundary tag scheme, providing for compaction of adjacent blocks.

HeapPools provide the fastest memory allocation in VADS EXEC. There is no deallocation of heap memory and therefore no space overhead in managing the pool and minimal processing overhead in allocating from a heap. *HeapPools* are intended for circumstances where a pool of memory is required for a short period of time. After use, the entire pool can be deleted.

Although memory for pool usage is allocated from a fixed address by default, you can also allocate memory from a dynamic address. Simply perform an Ada `new` allocation and use this allocation as the starting address for the pool.

The maximum number of pools that can exist at any given time is limited to 20. This limit applies only to `package V_MEMORY`.

References

input parameter, “procedure INITIALIZE_SERVICES — initialize memory services” on page 4-69

granularity, “procedure INITIALIZE_SERVICES — initialize memory services” on page 4-69

Package Procedures and Functions

`procedure CREATE_FIXED_POOL`

Create a FixedPool.

`procedure CREATE_FLEX_POOL`

Create a FlexPool.

`procedure CREATE_HEAP_POOL`

Create a HeapPool.

`procedure DESTROY_FIXED_POOL`

Delete a FixedPool.

`procedure DESTROY_FLEX_POOL`

Delete a FlexPool.

`procedure DESTROY_HEAP_POOL`

Delete a HeapPool.

`generic function FIXED_OBJECT_ALLOCATION`

Allocate an object from the given FixedPool, initializing it with a specified value.

`generic procedure FIXED_OBJECT_DEALLOCATION`

Deallocate the memory at the given location.

`generic function FLEX_OBJECT_ALLOCATION`

Allocate an object from the given FlexPool, initializing it with a specified value.

`generic procedure FLEX_OBJECT_DEALLOCATION`

Deallocate the memory at the given location.

`generic function HEAP_OBJECT_ALLOCATION`

Allocate an object from the given HeapPool, initializing it with a specified value.

`procedure INITIALIZE_SERVICES`

Initialize memory services.

Types

`FIXED_POOL_ID`

A private type used to identify *FixedPools*.

`FLEX_POOL_ID`

A private types used to identify *FlexPools*.

`HEAP_POOL_ID`

A private types used to identify *HeapPools*.

Exceptions

`BAD_BLOCK`

A deallocation request is made of memory not identifiable as an allocated block from a pool.

`BAD_POOL_CREATION_PARAMETER`

A pool cannot be created because a parameter passed to the `create` operation is unacceptable.

`INVALID_POOL_ID`

The pool ID passed to a pool operation does not correspond to an existing pool.

`NO_AVAILABLE_POOL`

A pool cannot be created because the maximum number of pools had already been allocated.

`NO_MEMORY`

A memory request cannot be honored because of insufficient memory in the pool.

OBJECT_LARGER_THAN_FIXED_BLOCK_SIZE

An attempt is made to allocate from a *FixedPool* an object whose size is greater than the size of the blocks in the pool.

UNCONSTRAINED_OBJECT

Raised by the allocation / deallocation subprograms if instantiated with an unconstrained OBJECT type.

UNEXPECTED_V_MEMORY_ERROR

an unexpected error (an internal V_MEMORY bug) occurs during a V_MEMORY operation.

Package Specification

```
with V_I_MEM;
with SYSTEM;
package V_MEMORY is

  pragma SUPPRESS(ALL_CHECKS);

  type fixed_pool_id is private;
  type flex_pool_id is private;
  type heap_pool_id is private;

  BAD_POOL_CREATION_PARAMETER         : exception;
  NO_AVAILABLE_POOL                   : exception;
  INVALID_POOL_ID                     : exception;
  NO_MEMORY                           : exception;
  BAD_BLOCK                           : exception;
  OBJECT_LARGER_THAN_FIXED_BLOCK_SIZE : exception;
  UNCONSTRAINED_OBJECT                : exception;
  UNEXPECTED_V_MEMORY_ERROR           : exception;

  procedure INITIALIZE_SERVICES
    (top_of_memory: in  system.address :=
system.”+”(16#FFFF_FFFF#);
    machine_boundary: in  integer := integer’size;
    granularity: in  integer := 16 );
  pragma INLINE_ONLY(INITIALIZE_SERVICES);
  procedure CREATE_FIXED_POOL
```

(Continued)

```
    (base_address: in    system.address;
    pool_size: in    natural;
    block_size: in    natural;
    pool    : out   fixed_pool_id);
  pragma INLINE_ONLY(CREATE_FIXED_POOL);
 generic
    type OBJECT is private;
    type POINTER is access OBJECT;
  function FIXED_OBJECT_ALLOCATION
    (pool   : in    fixed_pool_id;
    value   : in    object)
    return pointer;
  -- pragma INLINE(FIXED_OBJECT_ALLOCATION);
generic
    type OBJECT is private;
    type POINTER is access OBJECT;
  procedure FIXED_OBJECT_DEALLOCATION
    (location: in    pointer);
  -- pragma INLINE(FIXED_OBJECT_DEALLOCATION);
 procedure DESTROY_FIXED_POOL
              (pool              : in     fixed_pool_id);
  pragma INLINE_ONLY(DESTROY_FIXED_POOL);
  procedure CREATE_FLEX_POOL
              (base_address      : in     system.address;
               pool_size         : in     natural;
               pool              : out    flex_pool_id);
  pragma INLINE_ONLY(CREATE_FLEX_POOL);

  generic
    type OBJECT is private;
    type POINTER is access OBJECT;
  function FLEX_OBJECT_ALLOCATION
              (pool              : in     flex_pool_id;
               value             : in     object)
               return pointer;
  -- pragma INLINE(FLEX_OBJECT_ALLOCATION);
  generic
    type OBJECT is private;
    type POINTER is access OBJECT;

  procedure FLEX_OBJECT_DEALLOCATION
              (location          : in     pointer);
  -- pragma INLINE(FLEX_OBJECT_DEALLOCATION);
```

(Continued)

```
  procedure DESTROY_FLEX_POOL
              (pool              : in     flex_pool_id);
  pragma INLINE_ONLY(DESTROY_FLEX_POOL);
  procedure CREATE_HEAP_POOL
              (base_address      : in     system.address;
               pool_size         : in     natural;
               pool              : out    heap_pool_id);
  pragma INLINE_ONLY(CREATE_HEAP_POOL);
generic
    type OBJECT is private;
    type POINTER is access OBJECT;
  function HEAP_OBJECT_ALLOCATION
              (pool              : in     heap_pool_id;
               value             : in     object)
               return pointer;
  -- pragma INLINE(HEAP_OBJECT_ALLOCATION);
 procedure DESTROY_HEAP_POOL
              (pool              : in     heap_pool_id);
  pragma INLINE_ONLY(DESTROY_HEAP_POOL);

private
  type fixed_pool_id is new V_I_MEM.pool_id;
  type flex_pool_id is new V_I_MEM.pool_id;
  type heap_pool_id is new V_I_MEM.pool_id;
end V_MEMORY;
```

procedure CREATE_FIXED_POOL — create FixedPool

Syntax

```
procedure CREATE_FIXED_POOL
              (base_address   : in system.address;
               pool_size      : in natural;
               block_size     : in natural;
               pool           : out fixed_pool_id);
pragma INLINE_ONLY(CREATE_FIXED_POOL);
```

Arguments

base_address

Address at which the pool starts.

block_size

Size of each block, in bytes, in the pool.

pool

Identifier associated with the created pool.

pool_size

Amount of memory, in bytes, to allocate to the pool.

Description

Upon creation, the pool is subdivided into blocks. Each block is of the size specified by the *block_size* parameter. Each block is preceded by a 20-byte header that contains information used to manage the pool.

Thus, a 16#10_000# byte pool of 16#400# byte blocks provides 62 blocks with 1240 bytes used for headers.

Exceptions

BAD_POOL_CREATION_PARAMETER

A pool creation parameter is invalid.

NO_AVAILABLE_POOL

The maximum number of pools that can exist simultaneously in the system has already been allocated.

procedure CREATE_FLEX_POOL — create FlexPool

Syntax

```
procedure CREATE_FLEX_POOL
            (base_address      : in system.address;
             pool_size         : in natural;
             pool              : out flex_pool_id);
pragma INLINE_ONLY(CREATE_FLEX_POOL);
```

Arguments

base_address

Address at which the pool starts.

pool

Identifier associated with the created pool.

pool_size

Amount of memory, in bytes, to allocate to the pool.

Description

Each block in a *FlexPool* requires 28 bytes of overhead for pool management. Upon creation, blocks are created at the front and back of the pool to simplify the pool management operations. Because these are null blocks, they consume a total of 56 bytes.

Thus, after creation, a 1000-byte pool contains a 916- byte block. Subsequent allocations split this block into smaller blocks.

Exceptions

BAD_POOL_CREATION_PARAMETER

A pool creation parameter is invalid.

NO_AVAILABLE_POOL

The maximum number of pools that can exist simultaneously in the system has already been allocated.

procedure CREATE_HEAP_POOL — create HeapPool

Syntax

```
procedure CREATE_HEAP_POOL
            (base_address      : in     system.address;
             pool_size         : in     natural;
             pool              : out    heap_pool_id);
pragma INLINE_ONLY(CREATE_HEAP_POOL);
```

Arguments

base_address

Address at which the pool starts.

pool

Identifier associated with the created pool.

pool_size

Amount of memory to allocate to the pool.

Description

No internal management is performed on *HeapPools* and thus no memory within the pool is reserved for pool management. A 1000-byte pool provides 1000 bytes of available memory.

Exceptions

BAD_POOL_CREATION_PARAMETER

A pool creation parameter is invalid.

NO_AVAILABLE_POOL

The maximum number of pools that can exist simultaneously in the system has already been allocated.

procedure DESTROY_FIXED_POOL — delete FixedPool

Syntax

```
procedure DESTROY_FIXED_POOL (pool :in fixed_pool_id);
pragma INLINE_ONLY(DESTROY_FIXED_POOL);
```

Arguments

pool

Identity of the pool to be deleted.

Exceptions

`INVALID_POOL_ID`

`pool` does not identify a *FixedPool*.

procedure DESTROY_FLEX_POOL — delete FlexPool

Syntax

```
procedure DESTROY_FLEX_POOL(pool : in  flex_pool_id);
pragma INLINE_ONLY(DESTROY_FLEX_POOL);
```

Arguments

pool

Identity of the pool to be deleted.

Exceptions

INVALID_POOL_ID

pool does not identify a *FlexPool.*

procedure DESTROY_HEAP_POOL — delete HeapPool

Syntax

```
procedure DESTROY_HEAP_POOL(pool : in  heap_pool_id);
pragma INLINE_ONLY(DESTROY_HEAP_POOL);
```

Arguments

pool

Identity of the pool to be deleted.

Exceptions

INVALID_POOL_ID

pool does not identify a *HeapPool.*

generic function FIXED_OBJECT_ALLOCATION — allocate object

Syntax

```
generic
  type OBJECT is private;
  type POINTER is access OBJECT;

function FIXED_OBJECT_ALLOCATION
       (pool       : in fixed_pool_id;
       value       : in object)
       return pointer;
```

Arguments

pool

Pool from which to allocate the object.

value

Initial value for the object.

Description

FIXED_OBJECT_ALLOCATION allocates an object of type OBJECT from a specified *FixedPool*, initializing it with the specified *value*. Multiple instantiations of FIXED_OBJECT_ALLOCATION can use the same fixed pool.

Exceptions

INVALID_POOL_ID
pool does not identify a *FixedPool*.

NO_MEMORY
The object cannot be allocated from the pool because of insufficient memory.

OBJECT_LARGER_THAN_FIXED_BLOCK_SIZE
The size of the object exceeds the size of the pool blocks.

UNCONSTRAINED_OBJECT
The generic is instantiated with an unconstrained OBJECT type.

generic procedure FIXED_OBJECT_DEALLOCATION — deallocate memory

Syntax

```
generic
        type OBJECT is private;
        type POINTER is access OBJECT;

procedure FLEX_OBJECT_DEALLOCATION
        (location   : in pointer);
```

Arguments

location

Pointer to the object whose space is to be deallocated.

Description

`FIXED_OBJECT_DEALLOCATION` deallocates memory at the given location.

Exceptions

`BAD_BLOCK`

The pointer does not point to a block in a *FixedPool.*

generic function FLEX_OBJECT_ALLOCATION — allocate object

Syntax

```
generic
        type OBJECT is private;
        type POINTER is access OBJECT;

function FLEX_OBJECT_ALLOCATION
        (pool       : in flex_pool_id;
        value       : in object)
        return pointer;
```

Arguments

pool

Pool from which to allocate the object.

value

Initial value for the object.

Description

FLEX_OBJECT_ALLOCATION allocates an object of type OBJECT from a specified *FlexPool*, initializing it with the specified *value*.

The object is allocated from the pool using a first-fit algorithm. If, after taking into account the pool granularity, enough unused memory in the selected block exists to create a new block, the block is split into two blocks.

Exceptions

INVALID_POOL_ID

pool does not identify a *FlexPool*.

NO_MEMORY

The object cannot be allocated from the pool because of insufficient memory.

UNCONSTRAINED_OBJECT

The generic is instantiated with an unconstrained OBJECT type.

References

input parameter, “procedure INITIALIZE_SERVICES — initialize memory services” on page 4-69

granularity, “procedure INITIALIZE_SERVICES — initialize memory services” on page 4-69

generic procedure FLEX_OBJECT_DEALLOCATION — deallocate memory

Syntax

```
generic
        type OBJECT is private;
        type POINTER is access OBJECT;

procedure FLEX_OBJECT_DEALLOCATION
        (location   : in pointer);
```

Arguments

location

Pointer to the object whose space is to be deallocated.

Description

This procedure combines the deallocated block with any adjoining free blocks to form a single, larger free block.

Exceptions

BAD_BLOCK

The pointer does not point to a block in a *FlexPool.*

generic function HEAP_OBJECT_ALLOCATION — allocate object

Syntax

```
generic
        type OBJECT is private;
        type POINTER is access OBJECT;

function HEAP_OBJECT_ALLOCATION
        (pool       : in heap_pool_id;
        value       : in object)
        return pointer;
```

Arguments

pool

Pool from which to allocate the object.

value

Initial value for the object.

Description

HEAP_OBJECT_ALLOCATION allocates an object of type OBJECT from a specified *HeapPool*, initializing it with the specified *value*. Heap objects are allocated through the use of a pointer, which indicates the next available block and the number of bytes available.

Exceptions

INVALID_POOL_ID

pool does not identify a *HeapPool*.

NO_MEMORY

The object cannot be allocated from the pool because of insufficient memory.

UNCONSTRAINED_OBJECT

The generic is instantiated with an unconstrained OBJECT type.

procedure INITIALIZE_SERVICES — initialize memory services

Syntax

```
procedure INITIALIZE_SERVICES
        (top_of_memory :in system.address :=
                                system.”+”(16#FFFF_FFFF#);
        machine_boundary: in integer := integer’size;
        granularity : in integer := 16);
pragma INLINE_ONLY(INITIALIZE_SERVICES);
```

Arguments

granularity

Controls memory fragmentation within the flex memory pools.

machine_boundary

CPU natural boundary in BITS.

top_of_memory

Highest memory location addressable by the target/host (default value = 16#FFFF_FFFF#).

Description

The procedure INITIALIZE_SERVICES initializes memory services by supplying them with top-of-memory, machine boundary and granularity information. The *machine_boundary* parameter is particularly important on machines that impose specific bit alignment for certain data types. For INITIALIZE_SERVICES, the *machine_boundary* parameter defaults to INTEGER’SIZE.

Within the INITIALIZE_SERVICES procedure, the granularity parameter controls flex pool memory fragmentation. Flex pools provide storage in blocks of various sizes, differing in BYTES, causing all allocation requests to be rounded to the nearest multiple of 16. Unless specified otherwise, INITIALIZE_SERVICES defaults the granularity to 16.

INITIALIZE_SERVICES must be called before any of the memory services. The V_MEMORY package body calls INITIALIZE_SERVICES, using the default parameter values. Consequently, you must call INITIALIZE_SERVICES only when you want to override the default parameters.

The *top_of_memory* parameter is used to validate the BASE_ADDRESS parameter for the CREATE_FIXED_POOL, CREATE_FLEX_POOL, and CREATE_HEAP_POOL services. The default value of 16#FFFF_FFFF# ensures that any BASE_ADDRESS parameter is valid.

package V_MUTEXES — provide mutexes and condition variables

Description

`package V_MUTEXES` provides mutexes and condition variables. It is layered directly on top of the `V_I_MUTEX` package, which provides abort-safe mutexes and condition variables.

There are two varieties of mutexes: normal and recursive.

Normal mutexes can be used in conjunction with condition variables and provide the functionality described in Posix 1003.4a.

Recursive mutexes cannot be used in conjunction with condition variables; this is enforced by the typing system. Recursive mutexes are layered on top of normal mutexes and support recursive locking by a given task. That is, when a task succeeds in locking a given mutex, it can recursively lock the mutex without blocking. Subsequent unlocks do not release the lock until the recursion depth has reached 0. Recursive mutexes detect and raise exceptions for invalid unlock and delete operations.

Mutexes (both normal and recursive) can be created with specific attributes. If the underlying kernel does not support the attribute, an exception is raised during creation.

Condition variables can also be created with specific queuing behavior. If the underlying kernel does not support the specified behavior, an exception is raised during creation.

Mutexes and condition variables can be shared between programs by using the bind and resolve name services.

Mutex operations are provided to create, delete, bind, resolve, lock, and unlock mutexes.

Priority ceiling specific mutex operations are provided to set and return the priority ceiling of a mutex.

Condition variable operations are provided to create, delete, bind, resolve, signal, broadcast, and wait.

Package Procedures And Functions

`procedure BIND_COND`

Bind name to condition variable.

`procedure BIND_MUTEX`

Bind name to mutex.

`procedure BROADCAST_COND`

Broadcast condition variable.

`procedure/function CREATE_COND`

Create and initialize condition variable.

`procedure/function CREATE_MUTEX`

Create and initialize mutex.

`procedure DELETE_COND`

Delete condition variable.

`procedure DELETE_MUTEX`

Delete mutex.

`function GET_PRIORITY_CEILING_MUTEX`

Return mutex priority ceiling.

`procedure LOCK_MUTEX`

Attempt to lock mutex.

`procedure/function RESOLVE_COND`

Resolve name into condition variable.

`procedure/function RESOLVE_MUTEX`

Resolve name into mutex.

`procedure SET_PRIORITY_CEILING_MUTEX`

Set mutex priority ceiling.

`procedure SIGNAL_COND`

Signal condition variable.

`procedure SIGNAL_UNLOCK_COND`

Signal condition variable / unlock mutex.

`procedure/function TIMED_WAIT_COND`

Provide timed block of calling task.

`function TRYLOCK_MUTEX`

Attempt to lock mutex.

`procedure UNLOCK_MUTEX`

Unlock mutex.

`procedure WAIT_COND`

Block task on condition variable.

Types

`MUTEX_ID`

A private type used to identify a mutex.

`R_MUTEX_ID`

A private type used to identify a recursive mutex.

`COND_ID`

A private type used to identify a condition variable.

`COND_RESULT`

Returned by the functional version of `TIMED_WAIT_COND` to indicate the result of the operation.

Constants

`WAIT_FOREVER`

Used with the `RESOLVE_MUTEX` or `RESOLVE_COND` services to specify an indefinite wait for the name to be bound.

`DO_NOT_WAIT`

Used with the `RESOLVE_MUTEX` or `RESOLVE_COND` services to specify no wait at all for the name to be bound.

Mutex Exceptions

`NO_MEMORY_FOR_MUTEX`

Raised by `CREATE_MUTEX` if an attempt is made to create a mutex and there is insufficient memory for the mutex object in the system pool.

`MUTEX_ATTR_NOT_SUPPORTED`

Raised by `CREATE_MUTEX` if the underlying kernel does not support the specified attribute.

`MUTEX_LOCKED`

Raised by `DELETE_MUTEX` if an attempt is made to delete a mutex and the mutex is locked. This is supported only for recursive mutexes.

`BIND_MUTEX_NOT_SUPPORTED`

Raised if name services are not supported by the underlying RTS.

`BIND_MUTEX_BAD_ARGUMENT`

Raised if `BIND_MUTEX` is called with a null name.

`NO_MEMORY_FOR_MUTEX_NAME`

Raised if an attempt is made to bind a name to a mutex and there is insufficient memory.

`MUTEX_NAME_ALREADY_BOUND`

Raised if an attempt is made to bind a name to a mutex and the name is already bound to another object or procedure.

RESOLVE_MUTEX_NOT_SUPPORTED

Raised if name services are not supported by the underlying RTS.

RESOLVE_MUTEX_BAD_ARGUMENT

Raised if RESOLVE_MUTEX is called with a null name.

RESOLVE_MUTEX_TIMED_OUT

Raised by RESOLVE_MUTEX if a timed wait is attempted and the name is not bound to a mutex in the given time interval.

RESOLVE_MUTEX_FAILED

Raised by RESOLVE_MUTEX if a non-waited attempt is made to resolve a name to a mutex, and the name is not already bound.

NOT_A_MUTEX_NAME

Raised by RESOLVE_MUTEX if the name is bound, but not to a mutex object.

UNEXPECTED_V_MUTEX_ERROR

Raised by a mutex operation if an unexpected error occurs during a V_MUTEXES operation.

Condition Variable Exceptions

NO_MEMORY_FOR_COND

Raised by CREATE_COND if an attempt is made to create a condition variable and there is insufficient memory for the condition variable object in the system pool.

COND_QUEUING_NOT_SUPPORTED

Raised by CREATE_COND if the underlying kernel does not support the specified queuing behavior.

COND_TIMED_OUT

Raised by the procedural form of TIMED_WAIT_COND if the specified duration is exceeded.

BIND_COND_NOT_SUPPORTED
Raised if name services are not supported by the underlying RTS.

`BIND_COND_BAD_ARGUMENT`

Raised if `BIND_COND` is called with a null name.

`NO_MEMORY_FOR_COND_NAME`

Raised if an attempt is made to bind a name to a condition variable and there is insufficient memory.

`COND_NAME_ALREADY_BOUND`

Raised if an attempt is made to bind a name to a condition variable and the name has already been bound to another object or procedure.

`RESOLVE_COND_NOT_SUPPORTED`

Raised if name services are not supported by the underlying RTS.

`RESOLVE_COND_BAD_ARGUMENT`

Raised if `RESOLVE_COND` is called with a null name

`RESOLVE_COND_TIMED_OUT`

Raised by `RESOLVE_COND` if a timed wait is attempted and the name is not bound to a condition variable in the given time interval.

`RESOLVE_COND_FAILED`

Raised by `RESOLVE_COND` if a non-waited attempt is made to resolve a name to a condition variable, and the name is not already bound.

`NOT_A_COND_NAME`

Raised by `RESOLVE_COND` if the name is bound, but not to a condition variable object.

`UNEXPECTED_V_COND_ERROR`

Raised by a condition variable operation if the condition variable parameter is not a valid condition variable.

Package Specification

```
with SYSTEM;            use SYSTEM;
with ADA_KRN_DEFS;
with V_I_CIFO;
with V_INTERRUPTS;
package V_MUTEXES is
  pragma SUPPRESS( ALL_CHECKS );
  type mutex_id is private;
  type r_mutex_id is private;
  type cond_id is private;
  type cond_result is ( OBTAINED, TIMED_OUT );
  WAIT_FOREVER : constant duration := -1.0;
  DO_NOT_WAIT  : constant duration := 0.0;
  NO_MEMORY_FOR_MUTEX         : exception;
  MUTEX_ATTR_NOT_SUPPORTED    : exception;
  MUTEX_LOCKED                : exception;
  BIND_MUTEX_NOT_SUPPORTED    : exception;
  BIND_MUTEX_BAD_ARGUMENT     : exception;
  NO_MEMORY_FOR_MUTEX_NAME    : exception;
  MUTEX_NAME_ALREADY_BOUND    : exception;
  RESOLVE_MUTEX_NOT_SUPPORTED : exception;
  RESOLVE_MUTEX_BAD_ARGUMENT  : exception;
  RESOLVE_MUTEX_TIMED_OUT     : exception;
  RESOLVE_MUTEX_FAILED        : exception;
  NOT_A_MUTEX_NAME            : exception;
  UNEXPECTED_V_MUTEX_ERROR    : exception;
  NO_MEMORY_FOR_COND          : exception;
  COND_QUEUING_NOT_SUPPORTED  : exception;
  COND_TIMED_OUT              : exception;
  BIND_COND_NOT_SUPPORTED     : exception;
  BIND_COND_BAD_ARGUMENT      : exception;
  NO_MEMORY_FOR_COND_NAME     : exception;
  COND_NAME_ALREADY_BOUND     : exception;
  RESOLVE_COND_NOT_SUPPORTED  : exception;
  RESOLVE_COND_BAD_ARGUMENT   : exception;
  RESOLVE_COND_TIMED_OUT      : exception;
  RESOLVE_COND_FAILED         : exception;
  NOT_A_COND_NAME             : exception;
  UNEXPECTED_V_COND_ERROR     : exception;
  procedure CREATE_MUTEX
              (mutex:    out mutex_id;
               attr: in     ADA_KRN_DEFS.a_mutex_attr_t :=
```

(Continued)

```
                              ADA_KRN_DEFS.DEFAULT_MUTEX_ATTR);
procedure CREATE_MUTEX
            (mutex:    out r_mutex_id;
             attr: in     ADA_KRN_DEFS.a_mutex_attr_t :=
                              ADA_KRN_DEFS.DEFAULT_MUTEX_ATTR);
function  CREATE_MUTEX
            (attr: in    ADA_KRN_DEFS.a_mutex_attr_t :=
                              ADA_KRN_DEFS.DEFAULT_MUTEX_ATTR)
  return mutex_id;
function  CREATE_MUTEX
            (attr: in    ADA_KRN_DEFS.a_mutex_attr_t :=
                              ADA_KRN_DEFS.DEFAULT_MUTEX_ATTR)
  return r_mutex_id;
procedure DELETE_MUTEX
            (mutex : in out mutex_id);
procedure DELETE_MUTEX
            (mutex : in out r_mutex_id);
procedure BIND_MUTEX
            (name     : in     string;
             mutex  : in     mutex_id);
procedure BIND_MUTEX
            (name     : in     string;
             mutex    : in     r_mutex_id);
procedure RESOLVE_MUTEX
            (name   : in     string;
             mutex  :    out mutex_id;
             wait_time : in     duration := WAIT_FOREVER);
procedure RESOLVE_MUTEX
            (name        : in     string;
             mutex       :    out r_mutex_id;
             wait_time   : in     duration := WAIT_FOREVER);
function RESOLVE_MUTEX
            (name        : in     string;
             wait_time   : in     duration := WAIT_FOREVER)
      return mutex_id;
function RESOLVE_MUTEX
            (name        : in     string;
             wait_time   : in     duration := WAIT_FOREVER)
      return r_mutex_id;
procedure LOCK_MUTEX
            (mutex : in    mutex_id);
procedure LOCK_MUTEX
            (mutex : in    r_mutex_id);
```

(Continued)

```
pragma INLINE_ONLY(LOCK_MUTEX);
function  TRYLOCK_MUTEX
            (mutex : in     mutex_id)
  return boolean;
function  TRYLOCK_MUTEX
            (mutex : in     r_mutex_id)
  return boolean;
pragma INLINE_ONLY(TRYLOCK_MUTEX);
procedure UNLOCK_MUTEX
            (mutex : in     mutex_id);
procedure UNLOCK_MUTEX
            (mutex : in     r_mutex_id);
pragma INLINE_ONLY(UNLOCK_MUTEX);
procedure SET_PRIORITY_CEILING
            (mutex : in     mutex_id;
             ceil_pri : in     priority);
procedure SET_PRIORITY_CEILING
            (mutex : in     r_mutex_id;
             ceil_pri : in     priority);
function  GET_PRIORITY_CEILING
            (mutex : in     mutex_id)
  return integer;
function  GET_PRIORITY_CEILING
            (mutex : in     r_mutex_id)
  return integer;
procedure CREATE_COND
            (cond :    out cond_id;
             queuing  : in     V_I_CIFO.queuing_t :=
                                    V_I_CIFO.ARBITRARY_QUEUING;
             abort_callout_proc : in     address := NO_ADDR;
             abort_arg   : in     address := NO_ADDR);

function  CREATE_COND
            (queuing : in     V_I_CIFO.queuing_t :=
                        V_I_CIFO.ARBITRARY_QUEUING;
             abort_callout_proc : in     address := NO_ADDR;
             abort_arg : in     address := NO_ADDR)
  return cond_id;
procedure DELETE_COND
            (cond : in out cond_id);
procedure BIND_COND
            (name     : in     string;
             cond  : in     cond_id);
```

(Continued)

```
  procedure RESOLVE_COND
              (name  : in     string;
               cond   :    out cond_id;
               wait_time  : in     duration := WAIT_FOREVER);
  function RESOLVE_COND
              (name  : in     string;
               wait_time : in     duration := WAIT_FOREVER)
        return cond_id;
  procedure SIGNAL_COND
              (cond : in     cond_id);
  pragma INLINE_ONLY(SIGNAL_COND);
  procedure SIGNAL_UNLOCK_COND
              (cond  :  in     cond_id;
               mutex : in     mutex_id);
  pragma INLINE_ONLY(SIGNAL_UNLOCK_COND);
  procedure BROADCAST_COND
              (cond : in     cond_id);
  pragma INLINE_ONLY(BROADCAST_COND);
  procedure WAIT_COND
              (cond : in     cond_id;
               mutex : in     mutex_id);
  pragma INLINE_ONLY(WAIT_COND);
  procedure TIMED_WAIT_COND
              (cond : in     cond_id;
               mutex : in     mutex_id;
               sec : in     duration);
  function  TIMED_WAIT_COND
              (cond : in     cond_id;
               mutex : in     mutex_id;
               sec : in     duration)
    return cond_result;
private
    type mutex_rec;
    type mutex_id is access mutex_rec;
    type r_mutex_rec;
    type r_mutex_id is access r_mutex_rec;
    type cond_rec;
    type cond_id is access cond_rec;
end V_MUTEXES;
```

procedure BIND_COND — bind name to condition variable

Syntax

```
procedure BIND_COND
      (name       : in string;
      cond        : in cond_id);
```

Arguments

name

Condition variable name. The name can be any sequence of characters. An exact match is done for all name searches. (MY_COND differs from my_cond.)

cond

Condition variable to be bound.

Description

BIND_COND binds a name to a previously created condition variable.

Exceptions

NO_MEMORY_FOR_COND_NAME

Raised if an attempt is made to bind a name to a condition variable and there is insufficient memory.

BIND_COND_NOT_SUPPORTED

Raised if name services are not supported by the underlying RTS.

BIND_COND_BAD_ARGUMENT

Raised if BIND_COND is called with a null name.

COND_NAME_ALREADY_BOUND

Raised if an attempt is made to bind a name to a condition variable and the name has already been bound to another object or procedure.

Note – For this product, names are only known within a single program. Name binding and resolution services are more applicable to the cross-target environment.

procedure BIND_MUTEX — bind name to mutex

Syntax

```
procedure BIND_MUTEX
        (name       : in string;
        mutex       : in mutex_id);
procedure BIND_MUTEX
        (name       : in string;
        mutex       : in r_mutex_id);
```

Arguments

name

Mutex name. The name can be any sequence of characters. An exact match is done for all name searches. (MY_MUTEX differs from my_mutex.)

mutex

Mutex to be bound.

Description

BIND_MUTEX binds a name to a previously created mutex.

Exceptions

NO_MEMORY_FOR_MUTEX_NAME
Raised if an attempt is made to bind a name to a mutex and there is insufficient memory.

BIND_MUTEX_NOT_SUPPORTED
Raised if name services are not supported by the underlying RTS.

BIND_MUTEX_BAD_ARGUMENT
Raised if BIND_MUTEX is called with a null name.

MUTEX_NAME_ALREADY_BOUND
Raised if an attempt is made to bind a name to a mutex and the name has already been bound to another object or procedure.

Note – For this product, names are only known within a single program. Name binding and resolution services are more applicable to the cross-target environment.

procedure BROADCAST_COND — broadcast condition variable

Syntax

```
procedure BROADCAST_COND
        (cond       : in cond_id);
pragma INLINE_ONLY(BROADCAST_COND);
```

Arguments

cond

Condition variable to be broadcast.

Description

`procedure BROADCAST_COND` broadcasts a condition variable, ”waking up” all tasks that are waiting on the condition variable. However, no awakened task resumes execution until the associated mutex is unlocked.

procedure/function CREATE_COND — create and initialize condition variable

Syntax

```
procedure CREATE_COND
        (cond       out cond_id;
        queuing     : in V_I_CIFO.queuing_t :=

V_I_CIFO.ARBITRARY_QUEUING;
        abort_callout_proc: in address := NO_ADDR;
        abort_arg   : in address := NO_ADDR);
function  CREATE_COND
              (queuing: in     V_I_CIFO.queuing_t :=
                               V_I_CIFO.ARBITRARY_QUEUING;
        abort_callout_proc: in address := NO_ADDR;
        abort_arg   : in address := NO_ADDR)
return cond_id;
```

Arguments

abort_callout_proc

Procedure to call before completing the task abort sequence. Default is NO_ADDR, which implies that no call is done. The procedure has the following specification:

```
procedure abort_callout_proc( abort_arg : in address )
```

If not NO_ADDR, the procedure is called with the *abort_arg* argument.

abort_arg

Argument (of arbitrary type) to the ABORT_CALLOUT_PROC. [Default: NO_ADDR]

cond

Condition variable created.

queuing

Queuing associated with the condition variable.

Description

subprogram CREATE_COND creates and initializes a condition variable. Two versions are supplied: a procedure that returns COND_ID as an out parameter, and a function that returns COND_ID as the result.

The created condition variable can be shared with other programs by using the BIND_COND/RESOLVE_COND services.

For a description of condition variable queuing behaviors, see v_i_cifo.a in the standard directory.

Here are some examples of usage:

```
cond : cond_id;
queueing: V_I_CIFO.queuing_t := ...
procedure abort_proc( arg : in address );
i: integer;
```

- Creates a condition variable with default queuing behavior:

```
cond := create_cond;
```

- Creates a condition variable with priority queuing behavior. Note that the exception COND_QUEUING_NOT_SUPPORTED is raised if the underlying kernel does not support priority queuing on condition variables:

```
create_cond( cond, V_I_CIFO.PRIORITY_QUEUING );
```

- Creates a condition variable (with default queuing behavior) for which the specified abort procedure is called with the specified argument, if the task is aborted:

```
      create_cond( abort_callout_proc :=
abort_proc'address,
      abort_arg   := i'address );
```

Exceptions

NO_MEMORY_FOR_COND
Raised by CREATE_COND if an attempt is made to create a condition variable and there is insufficient memory for the condition variable object in the system pool.

COND_QUEUING_NOT_SUPPORTED
Raised by CREATE_COND if the underlying kernel does not support the specified queuing behavior.

procedure/function CREATE_MUTEX — create and initialize mutex

Syntax

```
procedure CREATE_MUTEX
        (mutex      : out mutex_id;
        attr        : in ADA_KRN_DEFS.a_mutex_attr_t :=

ADA_KRN_DEFS.DEFAULT_MUTEX_ATTR);

procedure CREATE_MUTEX
 (mutex : out r_mutex_id;
 attr   : in ADA_KRN_DEFS.a_mutex_attr_t :=
                  ADA_KRN_DEFS.DEFAULT_MUTEX_ATTR);
function  CREATE_MUTEX
        (attr       : in ADA_KRN_DEFS.a_mutex_attr_t :=
                        ADA_KRN_DEFS.DEFAULT_MUTEX_ATTR)
        return mutex_id;
function  CREATE_MUTEX
        (attr       : in ADA_KRN_DEFS.a_mutex_attr_t :=

ADA_KRN_DEFS.DEFAULT_MUTEX_ATTR)
        return r_mutex_id;
```

Arguments

attr

Attribute to be associated with the mutex.

mutex

Mutex created.

Description

CREATE_MUTEX creates and initializes a mutex. Two versions are supplied: a procedure that returns MUTEX_ID as an out parameter, and a function returning MUTEX_ID as the result.

The created mutex can be shared with other programs by using the BIND_MUTEX/RESOLVE_MUTEX services.

For a description of mutex attributes, see `ada_krn_defs.a` in the `standard` directory.

Here are some examples of usage:

```
mutex : mutex_id;
r_mutex: r_mutex_id;
intr_mask: ADA_KRN_DEFS.intr_status_t := ...
```

- Creates a (recursive) mutex with default attributes:

  ```
  r_mutex := create_mutex;
  ```

- Creates a mutex with priority queue waiting:

  ```
  create_mutex( mutex, ADA_KRN_DEFS.prio_mutex_attr_init
  );
  ```

- Creates a mutex suitable for using in an interrupt service routine, which uses the mask specified by the caller:

  ```
  mutex :=
  create_mutex(ADA_KRN_DEFS.intr_attr_init(intr_mask));
  ```

- Creates a (recursive) mutex that uses the priority ceiling emulation algorithm described in Posix 1003.4a. Note that this results in the exception `MUTEX_ATTR_NOT_SUPPORTED` being raised if the underlying kernel does not support the priority ceiling algorithm:

  ```
  create_mutex( r_mutex, prio_inherit_mutex_attr_init );
  ```

Exceptions

`NO_MEMORY_FOR_MUTEX`

Raised by `CREATE_MUTEX` if an attempt is made to create a mutex and there is insufficient memory for the mutex object in the system pool.

`MUTEX_ATTR_NOT_SUPPORTED`

Raised by `CREATE_MUTEX` if the underlying kernel does not support the specified attribute.

procedure DELETE_COND — delete condition variable

Syntax

```
procedure DELETE_COND
            (cond : in out cond_id);
```

Arguments

cond

Condition variable to delete.

Description

`procedure DELETE_COND` deletes a previously created condition variable.

If a condition variable is shared between programs by using the `BIND_COND/RESOLVE_COND` services, it must be deleted only in the program where it was created.

procedure DELETE_MUTEX — delete mutex

Syntax

```
procedure DELETE_MUTEX
            (mutex: in out mutex_id);
procedure DELETE_MUTEX
            (mutex: in out r_mutex_id);
```

Arguments

mutex

Mutex to delete.

Description

procedure DELETE_MUTEX deletes a previously created mutex.

If a mutex is shared between programs by using the BIND_MUTEX/RESOLVE_MUTEX services, it must be deleted only in the program where it was created.

Exceptions

MUTEX_LOCKED

Raised if the mutex is a recursive mutex, and the mutex is locked by the calling task.

function GET_PRIORITY_CEILING_MUTEX — return mutex priority ceiling

Syntax

```
function  GET_PRIORITY_CEILING_MUTEX
       (mutex      : in mutex_id)
return integer;

function  GET_PRIORITY_CEILING_MUTEX
       (mutex      : in r_mutex_id)
return integer;
```

Arguments

mutex

Mutex from which to get the ceiling priority.

Description

function `GET_PRIORITY_CEILING_MUTEX` returns the ceiling priority of the given mutex.

Exceptions

`MUTEX_ATTR_NOT_SUPPORTED`

Raised if either the priority ceiling protocol is not supported, or the mutex is not a priority ceiling mutex.

procedure LOCK_MUTEX — attempt to lock mutex

Syntax

```
procedure LOCK_MUTEX
            (mutex: in    mutex_id);

procedure LOCK_MUTEX
            (mutex: in    r_mutex_id);
pragma INLINE_ONLY(LOCK_MUTEX);
```

Arguments

mutex

Mutex to lock.

Description

`procedure LOCK_MUTEX` attempts to lock the mutex.

For non-recursive mutexes, if the mutex is currently locked, the task is blocked until the mutex is no longer locked and there are no other tasks waiting "ahead" associated with the mutex. Attempting to lock a mutex that is already locked by the calling task results in an infinite wait for the calling task.

For recursive mutexes, if the mutex is currently locked by the calling task, the depth is incremented and no blocking occurs. If the mutex is locked by another task, the task is blocked until the mutex is no longer locked and there are no other tasks waiting ahead of it.

procedure/function RESOLVE_COND — resolve name into condition variable

Syntax

```
procedure RESOLVE_COND
            (name          : in     string;
             cond          :    out cond_id;
            wait_time    : in    duration := WAIT_FOREVER);

function RESOLVE_COND
            (name          : in     string;
            wait_time     : in    duration := WAIT_FOREVER)
      return cond_id;
```

Arguments

name

Condition variable name.

cond

Resolved condition variable.

wait_time

Amount of time the user is willing to wait for the name to be bound. Two allowable predefined values are:

WAIT_FOREVER

Wait forever for the name to be bound.

DO_NOT_WAIT

Do not wait if the name is not already bound.

Description

subprogram RESOLVE_COND resolves a name into a condition variable that is created and bound in another program. Two versions are supplied: a procedure that returns COND_ID as an out parameter and a function that returns COND_ID as the result.

subprogram RESOLVE_COND first attempts to find an already bound name that exactly matches the name parameter. For a match, it returns immediately. Otherwise, it returns according to the *wait_time* parameter.

Exceptions

RESOLVE_COND_NOT_SUPPORTED

Raised if name services are not supported by the underlying RTS.

RESOLVE_COND_BAD_ARGUMENT

Raised if RESOLVE_COND is called with a null name.

RESOLVE_COND_TIMED_OUT

Raised by RESOLVE_COND if a timed wait is attempted and the name is not bound to a condition variable in the given time interval.

RESOLVE_COND_FAILED

Raised by RESOLVE_COND if a non-waited attempt is made to resolve a name to a condition variable, and the name is not already bound.

NOT_A_COND_NAME

Raised by RESOLVE_COND if the name is bound, but not to a condition variable object.

Note – For this product, names are only known within a single program. Name binding and resolution services are more applicable to the cross-target environment.

---

procedure/function RESOLVE_MUTEX — resolve name into mutex

Syntax

```
procedure RESOLVE_MUTEX
            (name          : in string;
             mutex         : out mutex_id;
             wait_time     : in duration := WAIT_FOREVER);

procedure RESOLVE_MUTEX
            (name          : in string;
             mutex         : out r_mutex_id;
             wait_time     : in duration := WAIT_FOREVER);

function RESOLVE_MUTEX
            (name          : in string;
             wait_time     : in duration := WAIT_FOREVER)
      return mutex_id;

function RESOLVE_MUTEX
            (name          : in     string;
            wait_time     : in     duration := WAIT_FOREVER)
      return r_mutex_id;
```

Arguments

mutex

Resolved mutex.

name

Mutex name.

wait_time

Amount of time the user is willing to wait for the name to be bound. Two allowable predefined values are:

WAIT_FOREVER
Wait forever for the name to be bound.

DO_NOT_WAIT
Do not wait if the name is not already bound.

Description

subprogram RESOLVE_MUTEX resolves a name into a mutex that is created and bound in another program. Two versions are supplied: a procedure that returns MUTEX_ID as an out parameter and a function that returns MUTEX_ID as the result.

subprogram RESOLVE_MUTEX first attempts to find an already bound name that exactly matches the name parameter. For a match, it returns immediately. Otherwise, it returns according to the wait_time parameter.

Exceptions

RESOLVE_MUTEX_NOT_SUPPORTED

Raised if name services are not supported by the underlying RTS.

RESOLVE_MUTEX_BAD_ARGUMENT

Raised if RESOLVE_MUTEX is called with a null name.

RESOLVE_MUTEX_TIMED_OUT

Raised by RESOLVE_MUTEX if a timed wait is attempted and the name is not bound to a mutex in the given time interval.

RESOLVE_MUTEX_FAILED

Raised by RESOLVE_MUTEX if a non-waited attempt is made to resolve a name to a mutex, and the name is not already bound.

NOT_A_MUTEX_NAME

Raised by RESOLVE_MUTEX if the name is bound, but not to a mutex object.

Note – For this product, names are only known within a single program. Name binding and resolution services are more applicable to the cross-target environment.

---

procedure SET_PRIORITY_CEILING_MUTEX — set mutex priority ceiling

Syntax

```
procedure SET_PRIORITY_CEILING_MUTEX
            (mutex        : in     mutex_id;
             ceil_pri: in     priority);
procedure SET_PRIORITY_CEILING_MUTEX
            (mutex        : in     r_mutex_id;
             ceil_pri: in     priority);
```

Arguments

ceil_pri

New ceiling priority.

mutex

Mutex with which to set priority.

Description

`procedure SET_PRIORITY_CEILING` sets the given mutex priority ceiling.

Exceptions

`MUTEX_ATTR_NOT_SUPPORTED`

Raised if the priority ceiling protocol is not supported, or the given mutex is not a priority ceiling mutex.

procedure SIGNAL_COND — signal condition variable

Syntax

```
procedure SIGNAL_COND
            (cond : in     cond_id);
pragma INLINE_ONLY(SIGNAL_COND);
```

Arguments

cond

Signalled condition variable.

Description

`procedure SIGNAL_COND` signals a condition variable, ”waking up” the next (as defined by the queuing behavior) task that is waiting on the condition variable. However, the awakened task does not resume execution until the associated mutex is unlocked.

procedure SIGNAL_UNLOCK_COND — signal condition variable/unlock mutex

Syntax

```
procedure SIGNAL_UNLOCK_COND
            (cond : in     cond_id;
             mutex: in     mutex_id);
pragma INLINE_ONLY(SIGNAL_UNLOCK_COND);
```

Arguments

cond

Signalled condition variable.

mutex

Mutex associated with the condition variable.

Description

`procedure SIGNAL_UNLOCK_COND` signals a condition variable and unlocks the specified mutex. This is more efficient than separate calls to `SIGNAL_COND` and `UNLOCK_MUTEX`.

procedure/function TIMED_WAIT_COND — provide timed block of calling task

Syntax

```
procedure TIMED_WAIT_COND
            (cond : in     cond_id;
             mutex: in     mutex_id;
             sec           : in     duration);
function  TIMED_WAIT_COND
            (cond : in     cond_id;
             mutex: in     mutex_id;
             sec           : in     duration)
  return cond_result;
```

Arguments

cond

Condition variable on which to wait.

mutex

Mutex associated with the condition variable.

sec

Time duration (in seconds).

Description

`subprogram TIMED_WAIT_COND` blocks the calling task on the condition variable until either a corresponding signal/broadcast is performed on the condition variable or the time duration has expired (assuming the duration is positive). If the duration is zero, it returns immediately, without blocking. If the duration is negative, it does not time out; that is, it waits indefinitely. There are two versions: the procedure raises an exception on timeout (otherwise, it returns normally), and the function returns the result of the wait. If successful, the calling task holds the lock on the mutex upon return.

Exceptions

`COND_TIMED_OUT`

Raised by the procedural version if the wait times out.

function TRYLOCK_MUTEX — attempt to lock mutex

Syntax

```
function  TRYLOCK_MUTEX
            (mutex: in     mutex_id)
  return boolean;

function  TRYLOCK_MUTEX
            (mutex: in     r_mutex_id)
  return boolean;
pragma INLINE_ONLY(TRYLOCK_MUTEX);
```

Arguments

mutex

Mutex to lock.

Description

`function TRYLOCK_MUTEX` attempts to lock the mutex. If the mutex is already locked, it returns `FALSE`. Otherwise, it locks the mutex (as specified by the `LOCK_MUTEX` routine) and returns `TRUE`. No blocking ever occurs as a result of this call.

procedure UNLOCK_MUTEX — unlock mutex

Syntax

```
procedure UNLOCK_MUTEX
        (mutex      : in mutex_id);

procedure UNLOCK_MUTEX
        (mutex      : in r_mutex_id);
pragma INLINE_ONLY(UNLOCK_MUTEX);
```

Arguments

mutex

Mutex to unlock.

Description

`procedure UNLOCK_MUTEX` unlocks the mutex, enabling the next task waiting to lock the mutex to proceed. If the mutex is a recursive mutex, however, the depth is first decremented. If the subsequent depth is greater than zero, the calling task retains the lock. If the depth is zero, then the lock is given up.

Exceptions

`UNEXPECTED_V_MUTEX_ERROR`

Raised if a recursive mutex is unlocked by some task other than the owner or when the depth is 0.

procedure WAIT_COND — block task on condition variable

Syntax

```
procedure WAIT_COND
            (cond : in     cond_id;
             mutex: in     mutex_id);
pragma INLINE_ONLY(WAIT_COND);
```

Arguments

cond

Condition variable on which to wait.

mutex

Mutex associated with the condition variable.

Description

`procedure WAIT_COND` blocks the calling task until a corresponding signal/broadcast is performed on the condition variable, and the mutex is unlocked. It is possible, however, that a task may return from `WAIT_COND` "early", because of an abort and/or other operating-system events (see `V_I_MUTEX`). It is therefore essential that calls to `WAIT_COND` be wrapped inside a loop that tests for the associated predicate before proceeding. Upon return from `WAIT_COND`, the mutex is locked by the calling task.

package V_NAMES — provide name services

Description

V_NAMES provides name services. These services allow objects to be shared between programs or, in conjunction with V_XTASKING.INTER_PROGRAM_CALL, allow a procedure to be called from another program. V_NAMES has been layered on the Ada Kernel name services.

BIND_OBJECT binds a name to the address of an object.

BIND_PROCEDURE binds a name to the program ID and address of a procedure.

RESOLVE_OBJECT resolves the name of an object into its address.

RESOLVE_PROCEDURE resolves the name of a procedure into its program ID and address allowing the procedure to be called using the V_XTASKING.INTER_PROGRAM_CALL service.

RESOLVE_OBJECT and RESOLVE_PROCEDURE first attempt to find an already bound name that exactly matches the name parameter. For a match, they return immediately. Otherwise, according to a wait_time parameter, they wait indefinitely until the name is bound, return immediately and raise the NAME_RESOLVE_FAILED exception, or raise the NAME_RESOLVE_TIMED_OUT exception if the name is not bound within WAIT_TIME.

Note – For this product, names are only known within a single program. Name binding and resolution services are more applicable to the cross-target environment.

Package Procedures And Functions

`procedure BIND_OBJECT`
Bind a name to the address of an object.

`procedure BIND_PROCEDURE`
Bind a name to the program ID and address of a procedure.

`procedure/function RESOLVE_OBJECT`
Resolve the name of an object into its address.

`procedure RESOLVE_PROCEDURE`
Resolve the name of a procedure into its program ID and address.

Constants

`WAIT_FOREVER`
Used with the `RESOLVE_OBJECT` or `RESOLVE_PROCEDURE` services to specify waiting indefinitely for the name to be bound.

`DO_NOT_WAIT`
Used with the `RESOLVE_OBJECT` or `RESOLVE_PROCEDURE` services to specify not waiting at all for the name to be bound.

Exceptions

`NO_MEMORY_FOR_NAME`
Raised if an attempt is made to bind a name and there is insufficient memory.

`NAME_SERVICE_NOT_SUPPORTED`
Raised if name services are not supported by the underlying RTS.

`BAD_ARGUMENT_FOR_NAME_SERVICE`
Raised if name service is called with a null name.

`NAME_ALREADY_BOUND`
Raised if an attempt is made to bind a name and the name has already been bound to another object or procedure.

`NAME_RESOLVE_TIMED_OUT`
Raised by `RESOLVE_OBJECT` or `RESOLVE_PROCEDURE` if a timed wait is attempted and the name is not bound in the given time interval.

NAME_RESOLVE_FAILED
Raised by RESOLVE_OBJECT or RESOLVE_PROCEDURE if a non-waited attempt is made to resolve a name, and the name is not already bound.

Package Specification

```
with SYSTEM;
package V_NAMES is
  pragma SUPPRESS(ALL_CHECKS);
  WAIT_FOREVER : constant duration := -1.0;
  DO_NOT_WAIT  : constant duration := 0.0;
  NO_MEMORY_FOR_NAME              : exception;
  NAME_SERVICE_NOT_SUPPORTED      : exception;
  BAD_ARGUMENT_FOR_NAME_SERVICE   : exception;
  NAME_ALREADY_BOUND              : exception;
  NAME_RESOLVE_TIMED_OUT          : exception;
  NAME_RESOLVE_FAILED             : exception;
  procedure BIND_OBJECT
             (name         : in     string;
              object_address   : in     system.address);
  procedure BIND_PROCEDURE
             (name  : in     string;
              procedure_address : in     system.address);
  procedure BIND_PROCEDURE
             (name      : in     string;
              procedure_program: in     system.program_id;
              procedure_address: in     system.address);
  procedure RESOLVE_OBJECT
             (name: in     string;
              object_address : out    system.address;
              wait_time: in     duration := WAIT_FOREVER);
  function RESOLVE_OBJECT
             (name: in     string;
              wait_time: in     duration := WAIT_FOREVER)
       return system.address;
  procedure RESOLVE_PROCEDURE
             (name: in     string;
              procedure_program: out    system.program_id;
              procedure_address: out    system.address;
              wait_time : in     duration := WAIT_FOREVER);
end V_NAMES;
```

procedure BIND_OBJECT — bind name to object address

Syntax

```
procedure BIND_OBJECT
          (name                    : in     string;
           object_address: in     system.address);
```

Arguments

name

Object name. The name can be any sequence of characters. An exact match is done for all name searches. (`MY_OBJECT` is different from `my_object`.)

object_address

Object address.

Description

`BIND_OBJECT` binds a name to the address of an object.

Exceptions

`NO_MEMORY_FOR_NAME`

Raised if the name could not be bound because of insufficient memory.

`NAME_SERVICE_NOT_SUPPORTED`

Raised if name services are not supported by the underlying RTS.

`BAD_ARGUMENT_FOR_NAME_SERVICE`

Raised if a null name is passed.

`NAME_ALREADY_BOUND`

Raised if the name is already bound to another object or procedure.

Note – For this product, names are only known within a single program. Name binding and resolution services are more applicable to the cross-target environment.

---

procedure BIND_PROCEDURE — bind name to program ID and address

Syntax

```
procedure BIND_PROCEDURE
          (name                    : in     string;
           procedure_address: in     system.address);
procedure BIND_PROCEDURE
          (name                    : in     string;
           procedure_program : in     system.program_id;
           procedure_address: in     system.address);
```

Arguments

name

Procedure name. The name can be any sequence of characters. An exact match is done for all name searches. (MY_PROCEDURE is different from my_procedure.)

procedure_program

Procedure's program ID. Set *procedure_program* to NO_PROGRAM_ID if the current program and stack limit switch logic are to be omitted for a V_XTASKING.INTER_PROGRAM_CALL().

procedure_address

Procedure's address.

Description

BIND_PROCEDURE binds a name to the PROGRAM_ID and address of a procedure. When the *procedure_program* parameter is omitted, it is set to the current program.

Exceptions

NO_MEMORY_FOR_NAME

Raised if the name could not be bound because of insufficient memory.

NAME_SERVICE_NOT_SUPPORTED

Raised if name services are not supported by the underlying RTS.

BAD_ARGUMENT_FOR_NAME_SERVICE

Raised if a null name is passed.

NAME_ALREADY_BOUND

Raised if the name is already bound to another object or procedure.

Note – For this product, names are only known within a single program. Name binding and resolution services are more applicable to the cross-target environment.

procedure/function RESOLVE_OBJECT — resolve name to address

Syntax

```
  procedure RESOLVE_OBJECT
             (name                    : in     string;
              object_address: out    system.address;
              wait_time      : in     duration :=
WAIT_FOREVER);
  function RESOLVE_OBJECT
             (name                    : in     string;
              wait_time      : in     duration :=
WAIT_FOREVER)
        return system.address;
```

Arguments

name

Object name

object_address

Object address

wait_time

Amount of time the user is willing to wait for the name to be bound. Two allowable predefined values are

WAIT_FOREVER

Wait forever for the name to be bound.

DO_NOT_WAIT

Do not wait if the name is not already bound.

Description

RESOLVE_OBJECT resolves the name of an object into its address. Two versions are supplied, a procedure that returns the address as an out parameter or a function returning the object address.

RESOLVE_OBJECT first attempts to find an already bound name that exactly matches the name parameter. For a match, it returns immediately. Otherwise, it returns according to the *wait_time* parameter.

Exceptions

NAME_SERVICE_NOT_SUPPORTED

Raised if name services are not supported by the underlying RTS.

BAD_ARGUMENT_FOR_NAME_SERVICE

Raised if a null name is passed.

NAME_RESOLVE_TIMED_OUT

Raised if a timed wait is attempted and the name does not become bound in the given time interval.

NAME_RESOLVE_FAILED

Raised if a non-waited attempt is made and the name is not already bound.

Note – For this product, names are only known within a single program. Name binding and resolution services are more applicable to the cross-target environment.

---

procedure RESOLVE_PROCEDURE — resolve name

Syntax

```
  procedure RESOLVE_PROCEDURE
             (name                    : in     string;
              procedure_program: out    system.program_id;
              procedure_address: out    system.address;
              wait_time                : in     duration :=
WAIT_FOREVER);
```

Arguments

name

Procedure name.

procedure_address

Procedure's address.

procedure_program

Procedure's program ID.

wait_time

Amount of time the user is willing to wait for the name to be bound. Two allowable predefined values are

WAIT_FOREVER

wait forever for the name to be bound.

DO_NOT_WAIT

do not wait if the name is not already bound.

Description

RESOLVE_PROCEDURE resolves the name of a procedure into its program ID and address. The program_id and address can be used to call the procedure with the V_XTASKING.INTER_PROGRAM_CALL service.

RESOLVE_PROCEDURE first attempts to find an already bound name that exactly matches the name parameter. For a match, it returns immediately. Otherwise, it returns according to the wait_time parameter.

Exceptions

NAME_SERVICE_NOT_SUPPORTED

Raised if name services are not supported by the underlying RTS.

BAD_ARGUMENT_FOR_NAME_SERVICE

Raised if a null name is passed.

NAME_RESOLVE_TIMED_OUT

Raised if a timed wait is attempted and the name does not become bound in the given time interval.

NAME_RESOLVE_FAILED

Raised if a non-waited attempt is made and the name is not already bound.

Note – For this product, names are only known within a single program. Name binding and resolution services are more applicable to the cross-target environment.

---

package V_SEMAPHORES — provide binary and counting semaphores

Description

The `package V_SEMAPHORES` provides operations for binary and counting semaphores. The semaphore data structure is allocated by using the Ada `new` allocator.

`V_SEMAPHORES` has been layered on the Ada Kernel binary and counting semaphore objects.

The following operations are overloaded and apply to both binary and counting semaphores.

The operation `CREATE_SEMAPHORE` creates a semaphore with an initial semaphore count. The semaphore count indicates the number of available resources; for example, an initial value of 1 allows only one task at a time to access the semaphore.

`WAIT_SEMAPHORE` decrements the semaphore count, causing the task to block on the semaphore if the semaphore is not available (the count becomes negative). The queuing order for blocked tasks depends on the attributes passed to `CREATE_SEMAPHORE`. VADS MICRO only supports first-in / first-out (FIFO) queuing for binary semaphores. For counting semaphores, FIFO or priority ordered queues are supported with FIFO as the default. The task must specify how long it is willing to wait for the semaphore.

`WAIT_SEMAPHORE` enables you to select the error notification method. If a result parameter is provided in the call, a result status is returned in the parameter; otherwise, an exception is raised if an error occurs.

`SIGNAL_SEMAPHORE` increments the semaphore count, awakening nya task that is waiting on the semaphore. If the awakened task is of sufficient priority, it preempts the current task.

`DELETE_SEMAPHORE` removes a semaphore. The user must specify the action to be taken if tasks are currently waiting on the semaphore.

`BIND_SEMAPHORE` and `RESOLVE_SEMAPHORE` allow a semaphore to be shared between programs. `BIND_SEMAPHORE` binds a name to a semaphore previously created in the current program. `RESOLVE_SEMAPHORE` resolves a name into a semaphore that was created and bound in another program.

If a task is aborted while accessing a semaphore, the resource is permanently locked.

Package Procedures And Functions

`procedure` `BIND_SEMAPHORE`

Bind a name to a semaphore.

`procedure` `CREATE_SEMAPHORE`

Create and initialize a semaphore.

`procedure` `DELETE_SEMAPHORE`

Delete a semaphore.

`procedure/function` `RESOLVE_SEMAPHORE`

Resolve a name into a semaphore.

`procedure` `SIGNAL_SEMAPHORE`

Perform a signal operation on a semaphore.

`procedure` `WAIT_SEMAPHORE`

Perform a wait operation on a semaphore.

Types

`BINARY_COUNT_T`

Restricts the range for the initial count of a binary semaphore.

`COUNT_SEMAPHORE_ID`

A private type used to identify a counting semaphore.

`BINARY_SEMAPHORE_ID`

A private type used to identify a binary semaphore.

SEMAPHORE_DELETE_OPTION

Specifies the action to take during a call to DELETE_SEMAPHORE if tasks are waiting on the semaphore that is to be deleted. Possible values are:

```
DELETE_OPTION_FORCE
DELETE_OPTION_WARNING
```

SEMAPHORE_RESULT

Return the completion status of the version of WAIT_SEMAPHORE that returns a status. Possible values are:

```
OBTAINED
TIMED_OUT
NOT_AVAILABLE
DELETED
```

Use pragma NO_IMAGE to eliminate the excess space overhead for enumeration types. The image array associated with an enumeration type is allocated in its own CONST subsection and is deleted by selective linking, if there are no uses of ENUM_TYPE'image.

References

"procedure DELETE_SEMAPHORE — delete semaphore" on page 4-128

"procedure WAIT_SEMAPHORE — perform wait operation" on page 4-133

Constants

WAIT_FOREVER

Passed to the WAIT_TIME parameter of WAIT_SEMAPHORE to specify the semaphore should be waited on until it is signaled . Additionally, this constant may be used with RESOLVE_SEMAPHORE to specify waiting indefinitely or not at all for the name to be bound.

DO_NOT_WAIT

Passed to the WAIT_TIME parameter of WAIT_SEMAPHORE to specify that the call should not block on the semaphore. If the semaphore is not available, an appropriate status is returned or an exception is raised. Additionally, this constant may be used with RESOLVE_SEMAPHORE to specify waiting indefinitely or not at all for the name to be bound.

Exceptions

`BIND_SEMAPHORE_BAD_ARGUMENT`

Raised if `BIND_SEMAPHORE` is called with a null name.

`BIND_SEMAPHORE_NOT_SUPPORTED`

Raised if name services are not supported by the underlying RTS.

`NO_MEMORY_FOR_SEMAPHORE`

Raised by `CREATE_SEMAPHORE` if an attempt is made to create a semaphore and there is insufficient memory for the semaphore object.

`NO_MEMORY_FOR_SEMAPHORE_NAME`

Raised if an attempt is made to bind a name to a mailbox and there is insufficient memory.

`NOT_A_SEMAPHORE_NAME`

Raised by `RESOLVE_SEMAPHORE` if the name is bound, but not to a mailbox object.

`RESOLVE_SEMAPHORE_BAD_ARGUMENT`

Raised if `RESOLVE_SEMAPHORE` is called with a null name.

`RESOLVE_SEMAPHORE_FAILED`

Raised by `RESOLVE_SEMAPHORE` if a non-waited attempt was made to resolve a name to a mailbox, and the name wasn't already bound.

`RESOLVE_SEMAPHORE_NOT_SUPPORTED`

Raised if name services are not supported by the underlying RTS.

`RESOLVE_SEMAPHORE_TIMED_OUT`

Raised by `RESOLVE_SEMAPHORE` if a timed wait was attempted and the name wasn't bound to a mailbox in the given time interval.

`SEMAPHORE_DELETED`

Raised by `WAIT_SEMAPHORE` if the semaphore is deleted while the task is waiting.

SEMAPHORE_IN_USE

Raised by DELETE_SEMAPHORE if an attempt is made to delete a semaphore when tasks are waiting on the semaphore and DELETE_OPTION_WARNING has been specified

SEMAPHORE_NAME_ALREADY_BOUND

Raised if an attempt is made to bind a name to a mailbox and the name has already been bound to another object or procedure.

SEMAPHORE_NOT_AVAILABLE

Raised by WAIT_SEMAPHORE if a non-waited attempt is made to obtain the semaphore and the semaphore is not available.

SEMAPHORE_TIMED_OUT

Raised by WAIT_SEMAPHORE if a timed wait is attempted and the semaphore did not become available in the given time interval.

UNEXPECTED_V_SEMAPHORE_ERROR

Raised if a semaphore routine is called with an invalid pointer to a semaphore or called using a deleted semaphore

Package Specification

```
with ADA_KRN_DEFS;
package V_SEMAPHORES is

pragma SUPPRESS(ALL_CHECKS);

type binary_semaphore_id is private;

type count_semaphore_id is private;

subtype binary_count_t is integer range 0 .. 1;

type semaphore_delete_option is (DELETE_OPTION_FORCE,DELETE_OPTION_WARNING);

WAIT_FOREVER : constant duration := -1.0;
DO_NOT_WAIT  : constant duration := 0.0;
```

(Continued)

```
NO_MEMORY_FOR_SEMAPHORE         : exception;
SEMAPHORE_IN_USE                : exception;
SEMAPHORE_DELETED               : exception;
SEMAPHORE_NOT_AVAILABLE         : exception;
SEMAPHORE_TIMED_OUT             : exception;
UNEXPECTED_V_SEMAPHORE_ERROR    : exception;
BIND_SEMAPHORE_NOT_SUPPORTED    : exception;
BIND_SEMAPHORE_BAD_ARGUMENT     : exception;
NO_MEMORY_FOR_SEMAPHORE_NAME    : exception;
SEMAPHORE_NAME_ALREADY_BOUND    : exception;
RESOLVE_SEMAPHORE_NOT_SUPPORTED : exception;
RESOLVE_SEMAPHORE_BAD_ARGUMENT    : exception;
RESOLVE_SEMAPHORE_TIMED_OUT     : exception;
RESOLVE_SEMAPHORE_FAILED        : exception;
NOT_A_SEMAPHORE_NAME            : exception;

type semaphore_result is (OBTAINED, TIMED_OUT, NOT_AVAILABLE, DELETED);
procedure CREATE_SEMAPHORE
    (initial_count: in  binary_count_t := 1;
    semaphore: out binary_semaphore_id;
    attr       : in  ADA_KRN_DEFS.a_semaphore_attr_t :=
                       ADA_KRN_DEFS.DEFAULT_SEMAPHORE_ATTR);
function CREATE_SEMAPHORE
    (initial_count: in  binary_count_t := 1;
    attr       : in  ADA_KRN_DEFS.a_semaphore_attr_t :=
                      ADA_KRN_DEFS.DEFAULT_SEMAPHORE_ATTR)
    return binary_semaphore_id;
procedure CREATE_SEMAPHORE
    (initial_count: in  integer := 1;
    semaphore: out count_semaphore_id;
    attr       : in  ADA_KRN_DEFS.a_count_semaphore_attr_t :=
                      ADA_KRN_DEFS.DEFAULT_COUNT_SEMAPHORE_ATTR);
function CREATE_SEMAPHORE
    (initial_count: in integer := 1;
    attr       : in ADA_KRN_DEFS.a_count_semaphore_attr_t :=
                      ADA_KRN_DEFS.DEFAULT_COUNT_SEMAPHORE_ATTR)
    return count_semaphore_id;


procedure DELETE_SEMAPHORE
    (semaphore: in   binary_semaphore_id;
    delete_option: in   semaphore_delete_option);
```

(Continued)

```
procedure DELETE_SEMAPHORE
    (semaphore: in   count_semaphore_id;
    delete_option: in   semaphore_delete_option);

procedure SIGNAL_SEMAPHORE
    (semaphore: in   binary_semaphore_id);

procedure SIGNAL_SEMAPHORE
    (semaphore in   count_semaphore_id);


procedure WAIT_SEMAPHORE
    (semaphore       : in   binary_semaphore_id;
    wait_time        : in   duration);

procedure WAIT_SEMAPHORE
    (semaphore       : in   binary_semaphore_id;
    wait_time        : in   duration;
    result           : out  semaphore_result);

procedure WAIT_SEMAPHORE
    (semaphore       : in   count_semaphore_id;
    wait_time        : in   duration);

procedure WAIT_SEMAPHORE
    (semaphore       : in   count_semaphore_id;
    wait_time        : in   duration;
    result           : out  semaphore_result);
procedure BIND_SEMAPHORE
    (name      : in   string;
    semaphore        : in   binary_semaphore_id);
procedure BIND_SEMAPHORE
    (name      : in   string;
    semaphore        : in   count_semaphore_id);
procedure RESOLVE_SEMAPHORE
    (name            : in   string;
    semaphore        : out  binary_semaphore_id;
    wait_time        : in   duration := WAIT_FOREVER);
function RESOLVE_SEMAPHORE
    (name            : in   string;
    wait_time        : in   duration := WAIT_FOREVER)
    return binary_semaphore_id;
```

(Continued)

```
procedure RESOLVE_SEMAPHORE
    (name            : in   string;
    semaphore        : out  count_semaphore_id;
    wait_time        : in   duration := WAIT_FOREVER);
function RESOLVE_SEMAPHORE
    (name            : in   string;
    wait_time        : in   duration := WAIT_FOREVER)
    return count_semaphore_id;
private
    type binary_semaphore_rec;
    type binary_semaphore_id is access binary_semaphore_rec;

    type count_semaphore_rec;
    type count_semaphore_id is access count_semaphore_rec;
end V_SEMAPHORES;
```

procedure BIND_SEMAPHORE — bind name to a semaphore

Syntax

```
procedure BIND_SEMAPHORE
        (name       : in string;
        semaphore   : in binary_semaphore_id);
procedure BIND_SEMAPHORE
        (name       : in string;
        semaphore   : in count_semaphore_id);
```

Arguments

name

Semaphore's name. The name can be any sequence of characters. An exact match is done for all name searches. (MY_SEMA differs from my_sema.)

semaphore

Identifies the semaphore to be bound.

Description

BIND_SEMAPHORE binds a name to a previously created semaphore.

Exceptions

NO_MEMORY_FOR_SEMAPHORE_NAME

Name could not be bound because of insufficient memory.

BIND_SEMAPHORE_NOT_SUPPORTED

Name services are not supported by the underlying RTS.

BIND_SEMAPHORE_BAD_ARGUMENT

A null name was passed.

SEMAPHORE_NAME_ALREADY_BOUND

Name was already bound to another object or procedure.

Note – For this product, names are only known within a single program. Name binding and resolution services are more applicable to the cross-target environment.

procedure/function CREATE_SEMAPHORE — create semaphore

Syntax

```
procedure CREATE_SEMAPHORE
        (initial_count: in  binary_count_t := 1;
        semaphore    : out binary_semaphore_id;
        attr         : in  ADA_KRN_DEFS.a_semaphore_attr_t :=
                             ADA_KRN_DEFS.DEFAULT_SEMAPHORE_ATTR);
function CREATE_SEMAPHORE
        (initial_count: in  binary_count_t := 1;
        attr         : in  ADA_KRN_DEFS.a_semaphore_attr_t :=

ADA_KRN_DEFS.DEFAULT_SEMAPHORE_ATTR)
        return binary_semaphore_id;
procedure CREATE_SEMAPHORE
        (initial_count: in  integer := 1;
        semaphore    : out count_semaphore_id;
        attr         : in
ADA_KRN_DEFS.a_count_semaphore_attr_t :=

ADA_KRN_DEFS.DEFAULT_COUNT_SEMAPHORE_ATTR);
function CREATE_SEMAPHORE
        (initial_count: in integer := 1;
        attr         : in
ADA_KRN_DEFS.a_count_semaphore_attr_t :=

ADA_KRN_DEFS.DEFAULT_COUNT_SEMAPHORE_ATTR)
        return count_semaphore_id;
```

Arguments

attr

For the binary CREATE_SEMAPHORE, *attr* points to an ADA_KRN_DEFS.SEMAPHORE_ATTR_T record. For the counting CREATE_SEMAPHORE, *attr* points to an ADA_KRN_DEFS.COUNT_SEMAPHORE_ATTR_T record. These records contain the binary/counting attributes. These are dependent on the

underlying threaded runtime. The type definitions of SEMAPHORE_ATTR_T and COUNT_SEMAPHORE_ATTR_T and the different options supported are found in ada_krn_defs.a in standard.

The *attr* parameter has been defaulted to DEFAULT_SEMAPHORE_ATTR or DEFAULT_COUNT_SEMAPHORE_ATTR. Unless you want to do something special, the default will suffice. VADS MICRO defaults to FIFO queuing when a task blocks waiting for a semaphore.

For the VADS MICRO counting CREATE_SEMAPHORE, use ADA_KRN_DEFS.DEFAULT_COUNT_INTR_ATTR to protect the critical region for updating the semaphore count by disabling all interrupts.

Use the subprogram ADA_KRN_DEFS.COUNT_INTR_ATTR_INIT to initialize the attributes so that a subset of the interrupts are disabled.

If the counting semaphore is to be signaled from an ISR, it must be protected by disabling interrupts.

initial_count

Initial number of resources allocated to the semaphore .(Binary semaphores are restricted to an initial count of 0 or 1. Binary semaphores with an initial value of 0 can be used for event posting.)

semaphore

Identifier for the created semaphore.

Description

CREATE_SEMAPHORE creates and initializes a binary or counting semaphore. Two versions are supplied for each semaphore type: a procedure that returns the semaphore ID as an out parameter or a function that returns the semaphore ID.

The created semaphore can be shared with other programs by using the BIND_SEMAPHORE/RESOLVE_SEMAPHORE services.

Exceptions

NO_MEMORY_FOR_SEMAPHORE

Semaphore cannot be created because of insufficient memory.

Threaded Runtime

CREATE_SEMAPHORE is layered upon ADA_KRN_I.SEMAPHORE_INIT or ADA_KRN_I.COUNT_SEMAPHORE_INIT. See them for more details about semaphore and counting semaphore attributes.

procedure DELETE_SEMAPHORE — delete semaphore

Syntax

```
procedure DELETE_SEMAPHORE
          (semaphore     : in     binary_semaphore_id;
         delete_option : in    semaphore_delete_option);

procedure DELETE_SEMAPHORE
          (semaphore     : in     count_semaphore_id;
         delete_option : in    semaphore_delete_option);
```

Arguments

delete_option

Action to be taken *if* the semaphore is in use. Possible values are:

DELETE_OPTION_FORCE

Ready all waiting tasks. These tasks' calls to WAIT_SEMAPHORE raise the exception SEMAPHORE_DELETED or return the result value DELETED. The semaphore is deleted.

DELETE_OPTION_WARNING

Raise the exception SEMAPHORE_IN_USE in the calling task. The semaphore is not deleted.

semaphore

Identifier of the semaphore to be deleted.

Description

DELETE_SEMAPHORE is used to delete either a binary or a counting semaphore. Note that the memory allocated for the semaphore is always freed after the semaphore is deleted.

If a semaphore is shared between programs by using the BIND_SEMAPHORE/RESOLVE_SEMAPHORE services, it must be deleted only in the program where it was created.

Exceptions

SEMAPHORE_IN_USE

An attempt is made to delete a semaphore when tasks are waiting on the semaphore and DELETE_OPTION_WARNING has been specified.

Threaded Runtime

DELETE_SEMAPHORE is layered upon ADA_KRN_I.SEMAPHORE_DESTROY or ADA_KRN_I.COUNT_SEMAPHORE_DESTROY.

procedure/function RESOLVE_SEMAPHORE — resolve name into a semaphore

Syntax

```
procedure RESOLVE_SEMAPHORE
            (name              : in     string;
             semaphore         :    out binary_semaphore_id;
           wait_time       : in    duration := WAIT_FOREVER);
function RESOLVE_SEMAPHORE
            (name              : in     string;
           wait_time       : in    duration := WAIT_FOREVER)
      return binary_semaphore_id;
procedure RESOLVE_SEMAPHORE
            (name              : in     string;
             semaphore         :    out count_semaphore_id;
           wait_time       : in    duration := WAIT_FOREVER);
function RESOLVE_SEMAPHORE
            (name              : in     string;
           wait_time       : in    duration := WAIT_FOREVER)
      return count_semaphore_id;
```

Arguments

name

Semaphore's name.

semaphore

The semaphore resolved.

wait_time

Amount of time the user is willing to wait for the name to be bound. Two allowable predefined values are:

WAIT_FOREVER

Wait forever for the name to be bound.

DO_NOT_WAIT

Do not wait if the name is not already bound.

Description

RESOLVE_SEMAPHORE resolves a name into a semaphore that was created and bound in another program. Two versions are supplied, a procedure that returns semaphore id as an out parameter or a function returning the semaphore id. RESOLVE_SEMAPHORE first attempts to find an already bound name that exactly matches the name parameter. For a match, it returns immediately. Otherwise, it returns according to the wait_time parameter.

Exceptions

NOT_A_SEMAPHORE_NAME

Raised by RESOLVE_SEMAPHORE if the name is bound, but not to a semaphore object.

RESOLVE_SEMAPHORE_BAD_ARGUMENT

Raised if RESOLVE_SEMAPHORE is called with a null name.

RESOLVE_SEMAPHORE_FAILED

Raised by RESOLVE_SEMAPHORE if a non-waited attempt was made to resolve a name to a semaphore, and the name wasn't already bound.

RESOLVE_SEMAPHORE_NOT_SUPPORTED

Raised if name services are not supported by the underlying RTS.

RESOLVE_SEMAPHORE_TIMED_OUT

Raised by RESOLVE_SEMAPHORE if a timed wait was attempted and the name wasn't bound to a semaphore in the given time interval.

Note – For this product, names are only known within a single program. Name binding and resolution services are more applicable to the cross-target environment.

---

procedure SIGNAL_SEMAPHORE — perform signal operation

Syntax

```
procedure SIGNAL_SEMAPHORE
          (semaphore    : in     binary_semaphore_id);

procedure SIGNAL_SEMAPHORE
          (semaphore    : in     count_semaphore_id);
```

Arguments

semaphore

The semaphore to be signaled.

Description

SIGNAL_SEMAPHORE performs a signal operation on a semaphore. If tasks are waiting on the semaphore, the signal operation causes a waiting task to be readied. If the readied task has a higher priority than the current task, the waiting task preempts the current task.

This service can be called from an interrupt service routine. However, for a counting semaphore, the interrupt protected form must be used.

Threaded Runtime

SIGNAL_SEMAPHORE is layered upon ADA_KRN_I.SEMAPHORE_SIGNAL or ADA_KRN_I.COUNT_SEMAPHORE_SIGNAL.

procedure WAIT_SEMAPHORE — perform wait operation

Syntax

```
procedure WAIT_SEMAPHORE
            (semaphore    : in     binary_semaphore_id;
             wait_time    : in     duration);

procedure WAIT_SEMAPHORE
            (semaphore    : in     binary_semaphore_id;
             wait_time    : in     duration;
             result       : out    semaphore_result);

procedure WAIT_SEMAPHORE
            (semaphore    : in     count_semaphore_id;
             wait_time    : in     duration);

procedure WAIT_SEMAPHORE
            (semaphore    : in     count_semaphore_id;
             wait_time    : in     duration;
             result       : out    semaphore_result);
```

Arguments

result

Result of the wait operation

semaphore

The semaphore to wait on

wait_time

Amount of time the user is willing to wait for the resource governed by the semaphore. Two allowable predefined values are:

WAIT_FOREVER

Wait forever for the semaphore

DO_NOT_WAIT

Do not wait if semaphore is not available

Other values are also allowable.

Description

Do not call WAIT_SEMAPHORE from an interrupt service routine, unless the *wait_time* is specified as DO_NOT_WAIT.

Exceptions/Results

SEMAPHORE_DELETED/DELETED

Semaphore deleted while the task is waiting.

SEMAPHORE_NOT_AVAILABLE/NOT_AVAILABLE

A non-waited attempt is made to obtain the semaphore and the semaphore is not available.

SEMAPHORE_TIMED_OUT/TIMED_OUT

A timed wait is attempted and the semaphore did not become available in the given time interval.

/OBTAINED

Result returned if the semaphore is obtained.

Threaded Runtime

WAIT_SEMAPHORE is layered upon ADA_KRN_I.SEMAPHORE_WAIT or ADA_KRN_I.COUNT_SEMAPHORE_WAIT.

package V_STACK — provide stack operations

Description

The `package V_STACK` provides stack operations. The procedure `CHECK_STACK` returns the current value of the stack pointer and the lower limit of the stack. Another procedure is provided to extend the current stack.

Package Procedures And Functions

`procedure CHECK_STACK`

Returns the current value of the stack pointer and the lower limit of the stack.

`procedure EXTEND_STACK`

Extends the current stack.

Package Specification

```
with SYSTEM;
package V_STACK is

  pragma SUPPRESS(ALL_CHECKS);

  procedure CHECK_STACK
            (current:out system.address;
             limit:out system.address);
  pragma INLINE_ONLY(CHECK_STACK);

  procedure EXTEND_STACK;
  pragma INLINE_ONLY(EXTEND_STACK);

end V_STACK;
```

procedure CHECK_STACK — return stack pointer value and upper limit

Syntax

```
procedure CHECK_STACK
        (current    :        out system.address;
        limit       :        out system.address);
pragma INLINE_ONLY(CHECK_STACK);
```

Arguments

current

Current value of the stack pointer.

limit

Value of the stack upper boundary.

Description

CHECK_STACK returns the current value of the stack pointer and the stack upper bound. This information is used to determine the amount of space remaining on the stack. The current out parameter returned by CHECK_STACK is a close approximation of the current SP. Because of compiler code, CHECK_STACK may return a value that is a few bytes lower than the actual SP.

Under certain conditions, code generated by this procedure cannot use the stack. However, the compiler can implicitly generate code that uses the stack to generate a reference to the parameters. Disassembly of the inline expansion can be used to determine if a given call performs stack allocation.

procedure EXTEND_STACK — extend current stack

Syntax

```
procedure EXTEND_STACk;
pragma INLINE_ONLY(EXTEND_STACK);
```

Description

This service is not supported. It always raises `STORAGE_ERROR` exception.

4

package V_XTASKING — provide Ada task operations

Description

XTASKING (extended tasking) provides operations that are performed on Ada tasks. XTASKING services augment the services defined by the language.

SUSPEND_TASK immediately inhibits the task from running regardless of its current state. RESUME_TASK allows a suspended task to run when it is in a ready-to-run state. Neither SUSPEND_TASK or RESUME_TASK affects the task state.

In addition to SUSPEND_TASK and RESUME_TASK, V_XTASKING provides operations to determine the current task and get and set these task parameters: priority, time slice interval, user field, fast rendezvous enabled, callable attribute, and terminated attribute. (The attribute parameters cannot be set.) Additionally, V_XTASKING provides operations to get and set the global time slicing enabled configuration parameter.

Services to disable and enable task preemption are provided. In addition, services to disable and enable the current task from being completed/terminated by an Ada ABORT are available.

Ada tasks and programs are layered on the underlying OS tasks and programs. Services are provided for mapping an Ada task ID to an OS task ID and vice-versa. Services are also provided for mapping program IDs.

For cross versions, V_XTASKING provides a service to start and terminate another separately linked program. It also has services to get the ID of the current program or any task.

A service is provided to call a procedure in another program. This service is normally used in conjunction with the V_NAMES bind/resolve procedure services.

Services are also provided to install subprogram callouts resident in the user program for program, task, and idle events. A callout can be installed for the following program events: exit, unexpected exit (main program abandoned because of an unhandled exception), or switch. A callout can be installed for the following task events: creation, switch and completion. A callout can be called whenever the kernel is in its idle state. Additionally, user-defined storage can be allocated in the control block for a task.

Ada tasking has the global flag EXIT_DISABLED_FLAG, which can be set to TRUE to inhibit the program from exiting. Services are provided to get and set this flag.

Package Procedures And Functions

`function ALLOCATE_TASK_STORAGE`

Allocate storage in the task control block. The function returns the storage ID to be used in subsequent GET_TASK_STORAGE or GET_TASK_STORAGE2 service calls.

`function CALLABLE`

Return the P'CALLABLE attribute for the specified task.

`function CURRENT_EXIT_DISABLED`

Return current value.

`function CURRENT_FAST_RENDEZVOUS_ENABLED`

Return the value of the FAST_RENDEZVOUS_ENABLED flag of the current task. For details about the fast rendezvous optimization see page .

`function CURRENT_PRIORITY`

Return the priority of the specified task .

`function CURRENT_PROGRAM`

Return the program ID of the current task.

`function CURRENT_TASK`

Return the task ID of the current task.

`function CURRENT_TIME_SLICE`

Return the current time slice interval of the specified task.

`function CURRENT_TIME_SLICING_ENABLED`

Return the current value of the kernel TIME_SLICING_ENABLED configuration parameter.

`function CURRENT_USER_FIELD`

Return the current value of the specified task user-modifiable field.

`procedure DISABLE_PREEMPTION`

Inhibit the current task from being preempted. The procedure does not disable interrupts.

`procedure DISABLE_TASK_COMPLETE`

Disable the current task from being completed and terminated when aborted. Must be paired with `ENABLE_TASK_COMPLETE`.

`procedure ENABLE_PREEMPTION`

Allow the current task to be preempted.

`procedure ENABLE_TASK_COMPLETE`

Enable the current task to be completed and terminated when aborted. Must be paired with `DISABLE_TASK_COMPLETE`.

`function GET_PROGRAM`

Return the program `ID` of the specified task.

`function GET_PROGRAM_KEY`

Return the user-defined key for the specified program.

`function GET_TASK_STORAGE`

Return the starting address of the task storage area of the specified task and storage ID.

`function GET_TASK_STORAGE2`

Return the starting address of the task storage area using the OS `ID` of the task.

`function ID`

Return an identifier for an Ada program or task given the underlying OS program or task ID.

`procedure INSTALL_CALLOUT`

Install a procedure to be called at a program exit, program switch, task create, task switch, task complete ,or idle event.

`procedure INTER_PROGRAM_CALL`

Call a procedure in another program.

`function OS_ID`

Return the underlying OS program or task ID given the Ada program or task ID.

`procedure RESUME_TASK`

Ready the named task for execution.

`procedure SET_EXIT_DISABLED`

Change the kernel `EXIT_DISABLE_FLAG` to inhibit or allow the program to exit.

`procedure SET_FAST_RENDEZVOUS_ENABLED`

Change the `FAST_RENDEZVOUS_ENABLED` flag for the current task to a new value.

`procedure SET_IS_SERVER_PROGRAM`

Mark the current program as a server program.

`procedure SET_PRIORITY`

Change the priority of the specified task.

`procedure SET_TIME_SLICE`

Change time slice interval of specified task.

`procedure SET_TIME_SLICING_ENABLED`

Set kernel configuration parameter `TIME_SLICING_ENABLED`.

`procedure SET_USER_FIELD`

Change the value of the specified task user-modifiable field.

`function START_PROGRAM`

Start another, separately linked program identified by its link block, and return the PROGRAM_ID of the just started program.

`procedure SUSPEND_TASK`

Cause a running task to become suspended.

`function TERMINATED`

Return the `P'TERMINATED` attribute for the specified task.

procedure `TERMINATE_PROGRAM`

Terminate the specified program.

`generic function V_ID`

Return an identifier for a task, given a task object of the task type used to instantiate the generic.

Types

`CALLOUT_EVENT_T`

Type of program, tasking, or idle event that can have a callout procedure installed with the `INSTALL_CALLOUT` service. Possible values are:

```
EXIT_EVENT
IDLE_EVENT
PROGRAM_SWITCH_EVENT
TASK_COMPLETE_EVENT
TASK_CREATE_EVENT
TASK_SWITCH_EVENT
UNEXPECTED_EXIT_EVENT
```

`OS_PROGRAM_ID`

Type of the underlying OS program control block.

`OS_TASK_ID`

Type of the underlying OS task control block.

`PROGRAM_ID`

Type of the Ada program control block. This type is defined in package SYSTEM.

TASK_ID

Type of the Ada task control block. This type is defined in package SYSTEM.

TASK_STORAGE_ID

Provides an identifier for user-defined storage in a task control block created and returned by the ALLOCATE_TASK_STORAGE service and passed to the GET_TASK_STORAGE or GET_TASK_STORAGE2 extended tasking services.

XTASKING_RESULT

Type of the result codes returned by the non-exception-raising versions of the SUSPEND_TASK and RESUME_TASK routines. Possible values are:

```
NOT_RESUMED
NOT_SUSPENDED
RESUMED
SUSPENDED
```

Subtypes

USER_FIELD_T

Type of user-modifiable field stored in task control block where USER_FIELD_T'SIZE = INTEGER'SIZE.

USER_FIELD2_T

Type of user-modifiable field stored in the task control block where USER_FIELD2_T'SIZE = ADDRESS'SIZE.

Constants

NULL_TASK_NAME

null task constant.

NO_TASK_STORAGE_ID

Returned by ALLOCATE_TASK_STORAGE when no more space is available in the task control block.

NULL_OS_TASK_NAME

null OS task constant.

NULL_OS_PROGRAM_NAME

null OS program constant.

Exceptions

INVALID_RESUME

Raised by RESUME_TASK if the task cannot be resumed. This occurs if the task is not currently suspended. Not used with the VADS MICRO kernel.

INVALID_SUSPEND

Raised by SUSPEND_TASK if the task cannot be suspended. This occurs if the task is not currently runnable. Not used with the VADS MICRO kernel.

UNEXPECTED_V_XTASKING_ERROR

Raised if an unexpected error occurs in an V_XTASKING routine.

Package Specification

```
with ADA_KRN_DEFS;
with SYSTEM;
package V_XTASKING is
pragma suppress(ALL_CHECKS);
NULL_TASK_NAME: constant system.task_id := system.NO_TASK_ID;
type os_task_id is new ADA_KRN_DEFS.krn_task_id;
NULL_OS_TASK_NAME: constant os_task_id :=
    os_task_id(ADA_KRN_DEFS.NO_KRN_TASK_ID);
type os_program_id is new ADA_KRN_DEFS.krn_program_id;
NULL_OS_PROGRAM_NAME: constant os_program_id :=
    os_program_id(ADA_KRN_DEFS.NO_KRN_PROGRAM_ID);
type task_storage_id is private;
NO_TASK_STORAGE_ID: constant task_storage_id;
```

(Continued)

```
type callout_event_t is new ADA_KRN_DEFS.callout_event_t;
subtype user_field_t is integer;
subtype user_field2_t is SYSTEM.address;
INVALID_SUSPEND             : exception;
INVALID_RESUME              : exception;
UNEXPECTED_V_XTASKING_ERROR : exception;
type xtasking_result is (SUSPENDED, RESUMED, NOT_SUSPENDED, NOT_RESUMED);


function ID(os_task_name    : in os_task_id) return system.task_id;
function ID(os_program_name : in os_program_id) return system.program_id;
function ID(task_address    : in system.task_id) return system.task_id;
pragma INLINE_ONLY(ID);
function OS_ID(task_name    : in system.task_id) return os_task_id;
generic
  type task_type is limited private;
function V_ID(task_object   : in task_type) return system.task_id;
pragma INLINE_ONLY(V_ID);
function OS_ID(program_name : in system.program_id) return os_program_id;
pragma INLINE_ONLY(OS_ID);
function CURRENT_TASK return system.task_id;
pragma INLINE_ONLY(CURRENT_TASK);
procedure RESUME_TASK (task_name     : in     system.task_id);
procedure RESUME_TASK (task_name     : in     system.task_id;
                       result        : out    xtasking_result);
pragma INLINE_ONLY(RESUME_TASK);

procedure SUSPEND_TASK (task_name    : in     system.task_id);
procedure SUSPEND_TASK (task_name    : in     system.task_id;
                        result       : out    xtasking_result);
pragma INLINE_ONLY(SUSPEND_TASK);
function CURRENT_PRIORITY (task_name : in system.task_id := CURRENT_TASK)
     return system.priority;
pragma INLINE_ONLY(CURRENT_PRIORITY);
procedure SET_PRIORITY
           (new_priority : in system.priority;
            task_name    : in system.task_id := CURRENT_TASK);
pragma INLINE_ONLY(SET_PRIORITY);
function CURRENT_TIME_SLICE
           (task_name  : in system.task_id := CURRENT_TASK) return duration;
pragma INLINE_ONLY(CURRENT_TIME_SLICE);
procedure SET_TIME_SLICE
           (new_interval : in duration;
            task_name    : in system.task_id := CURRENT_TASK);
```

(Continued)

```
pragma INLINE_ONLY(SET_TIME_SLICE);
function CURRENT_USER_FIELD
           (task_name    : in system.task_id := CURRENT_TASK)
            return user_field_t;
function CURRENT_USER_FIELD
            (task_name   : in system.task_id := CURRENT_TASK)
             return user_field2_t;
pragma INLINE_ONLY(CURRENT_USER_FIELD);
procedure SET_USER_FIELD
           (new_value    : in user_field_t;
            task_name    : in system.task_id := CURRENT_TASK);
procedure SET_USER_FIELD
           (new_value    : in user_field2_t;
            task_name    : in system.task_id := CURRENT_TASK);
pragma INLINE_ONLY(SET_USER_FIELD);


function CALLABLE
           (task_name    : in system.task_id) return boolean;
pragma INLINE_ONLY(CALLABLE);
function TERMINATED
           (task_name    : in system.task_id) return boolean;
pragma INLINE_ONLY(TERMINATED);
function CURRENT_TIME_SLICING_ENABLED return boolean;
pragma INLINE_ONLY(CURRENT_TIME_SLICING_ENABLED);
procedure SET_TIME_SLICING_ENABLED
           (new_value    : in  boolean := TRUE);
pragma INLINE_ONLY(SET_TIME_SLICING_ENABLED);
function CURRENT_EXIT_DISABLED return boolean;
pragma INLINE_ONLY(CURRENT_EXIT_DISABLED);
procedure SET_EXIT_DISABLED
           (new_value    : in  boolean := TRUE);
pragma INLINE_ONLY(SET_EXIT_DISABLED);
function START_PROGRAM
    (link_block_address: in system.address;
    key               : in system.address := system.memory_address(2);
    terminate_callout: in system.address := system.NO_ADDR)
               return system.program_id;
pragma INLINE_ONLY(START_PROGRAM);
function CURRENT_PROGRAM return system.program_id;
pragma INLINE_ONLY(CURRENT_PROGRAM);
function GET_PROGRAM(task_name: in system.task_id)
               return system.program_id;
pragma INLINE_ONLY(GET_PROGRAM);
```

(Continued)

```
procedure TERMINATE_PROGRAM
        (status    : in integer;
        program_name: in system.program_id := current_program);
pragma INLINE_ONLY(TERMINATE_PROGRAM);
function GET_PROGRAM_KEY
        (program_name: in system.program_id := CURRENT_PROGRAM)
                return system.address;
pragma INLINE_ONLY(GET_PROGRAM_KEY);
procedure SET_IS_SERVER_PROGRAM;
pragma INLINE_ONLY(SET_IS_SERVER_PROGRAM);
procedure INTER_PROGRAM_CALL
            (procedure_program: in     system.program_id;
             procedure_address: in     system.address;
             argument: in     system.address);
pragma INLINE_ONLY(INTER_PROGRAM_CALL);
procedure INSTALL_CALLOUT
            (event: in callout_event_t;
             proc: in system.address);
pragma INLINE_ONLY(INSTALL_CALLOUT);
function ALLOCATE_TASK_STORAGE
             (size: in natural) return task_storage_id;
pragma INLINE_ONLY(ALLOCATE_TASK_STORAGE);
function GET_TASK_STORAGE
             (task_name: in system.task_id;
              storage_id: in task_storage_id) return system.address;
pragma INLINE_ONLY(GET_TASK_STORAGE);
function GET_TASK_STORAGE2
              (os_task_name: in os_task_id;
               storage_id: in task_storage_id) return system.address;
pragma INLINE_ONLY(GET_TASK_STORAGE2);
procedure DISABLE_PREEMPTION;
pragma INLINE_ONLY(DISABLE_PREEMPTION);

procedure ENABLE_PREEMPTION;
pragma INLINE_ONLY(ENABLE_PREEMPTION);


  procedure DISABLE_TASK_COMPLETE;
  procedure ENABLE_TASK_COMPLETE;
  pragma INLINE_ONLY(DISABLE_TASK_COMPLETE);
  pragma INLINE_ONLY(ENABLE_TASK_COMPLETE);

  function CURRENT_FAST_RENDEZVOUS_ENABLED return boolean;
  pragma INLINE_ONLY(CURRENT_FAST_RENDEZVOUS_ENABLED);
```

(Continued)

```
  procedure SET_FAST_RENDEZVOUS_ENABLED
              (new_value      : in boolean);
  pragma INLINE_ONLY(SET_FAST_RENDEZVOUS_ENABLED);
private
  type task_storage_id is new ADA_KRN_DEFS.task_storage_id;
  NO_TASK_STORAGE_ID: constant task_storage_id :=
    task_storage_id(ADA_KRN_DEFS.NO_TASK_STORAGE_ID);
end V_XTASKING;
```

function ALLOCATE_TASK_STORAGE — allocate storage in task control block

Syntax

```
function ALLOCATE_TASK_STORAGE
        (size       : in natural) return task_storage_id;
pragma INLINE_ONLY(ALLOCATE_TASK_STORAGE);
```

Arguments

size

Number of bytes to be allocated in the task control block

Description

This service allocates storage in the task control block. It returns the ID to be used in subsequent GET_TASK_STORAGE or GET_TASK_STORAGE2 service calls.

The task storage allocation is applicable only to tasks in the current program.

Exceptions

STORAGE_ERROR

Raised if there is not enough memory in the task control block for the task storage. The configuration parameter TASK_STORAGE_SIZE defines the size of the storage area set aside in the control block for each task. Each allocation from this area is aligned on a 4- or 8- byte boundary (the alignment is CPU dependent).

Threaded Runtime

ALLOCATE_TASK_STORAGE is provided for VADS MICRO. However, this service may not be supported by other underlying threaded runtimes.

function CALLABLE — return value of task P'CALLABLE attribute

Syntax

```
function CALLABLE
        (task_name  : in system.task_id)
        return boolean;
pragma INLINE_ONLY(CALLABLE);
```

Arguments

task_name

ID of the task for which the value of the P'CALLABLE attribute is returned

Description

The function CALLABLE returns the value of the specified task's P'CALLABLE attribute. When the specified task is completed, terminated ,or executes abnormally, CALLABLE returns the boolean value FALSE. Under all other conditions, CALLABLE returns the boolean value TRUE. Referencing a nonexistent task or a task that has not yet been created yields indeterminate values.

function CURRENT_EXIT_DISABLED — return value

Syntax

```
function CURRENT_EXIT_DISABLED return boolean;
pragma INLINE_ONLY(CURRENT_EXIT_DISABLED);
```

Description

This function returns the current value for the Ada tasking global variable, `EXIT_DISABLED_FLAG`. When `TRUE`, the Ada application program is inhibited from exiting.

function CURRENT_FAST_RENDEZVOUS_ENABLED — return value of flag

Syntax

```
function CURRENT_FAST_RENDEZVOUS_ENABLED return boolean;
pragma  INLINE_ONLY(CURRENT_FAST_RENDEZVOUS_ENABLED);
```

Description

This function returns the value for the current Ada task FAST_RENDEZVOUS_ENABLED flag. See FAST_RENDEZVOUS_ENABLED in the v_usr_conf.a for details about the fast rendezvous optimization.

References

“Fast Rendezvous Optimization” on page 2-76

function CURRENT_PRIORITY — return priority of task

Syntax

```
function CURRENT_PRIORITY
        (task_name  : in system.task_id := CURRENT_TASK)
        return system.priority;
pragma INLINE_ONLY(CURRENT_PRIORITY);
```

Arguments

task_name

ID of the task whose priority is to be returned to the caller. If no task name is specified, *task_name* defaults to the current task

Description

CURRENT_PRIORITY returns the current priority of the specified task. This may be the task priority as specified by a pragma PRIORITY, the default task priority, the priority of a task it is in rendezvous with (if the task is in rendezvous with a higher priority task), or the task priority changed via SET_PRIORITY.

function CURRENT_PROGRAM — return current program identifier

Syntax

```
function CURRENT_PROGRAM
        return system.program_id;
pragma INLINE_ONLY(CURRENT_PROGRAM);
```

Description

This function returns the program `ID` of the current program.

Threaded Runtime

`CURRENT_PROGRAM` is provided for VADS MICRO. However, this service may not be supported by other underlying threaded runtimes.

function CURRENT_TASK — return current task identifier

Syntax

```
function CURRENT_TASK
        return system.task_id;
pragma INLINE_ONLY(CURRENT_TASK);
```

Description

This function returns the task `ID` of the current task.

Example

The following call suspends the currently running task:

```
V_XTASKING.SUSPEND_TASK(V_XTASKING.CURRENT_TASK);
```

function CURRENT_TIME_SLICE — return current time slice interval

Syntax

```
function CURRENT_TIME_SLICE
        (task_name  : in system.task_id := CURRENT_TASK)
        return duration;
pragma INLINE_ONLY(CURRENT_TIME_SLICE);
```

Arguments

task_name

`ID` of the task whose time slice interval is returned. If no *`task_name`* is specified, the routine defaults to the current task

Description

The `function CURRENT_TIME_SLICE` returns the value of the specified task's current time slice interval. `CURRENT_TIME_SLICE` requires a task `ID` for a parameter. If you do not supply a task ID, the routine defaults to the current task.

Threaded Runtime

This service is provided for VADS MICRO. However, this service may not be supported by other underlying threaded runtimes.

function CURRENT_TIME_SLICING_ENABLED — check time slicing enabled

Syntax

```
function CURRENT_TIME_SLICING_ENABLED
        return boolean;
pragma INLINE_ONLY(CURRENT_TIME_SLICING_ENABLED)
```

Description

The `function CURRENT_TIME_SLICING_ENABLED` returns a boolean value indicating the current value of the kernel *`time_slicing_enabled`* configuration parameter.

Threaded Runtime

This service is provided for VADS MICRO. However, this service may not be supported by other underlying threaded runtimes.

function CURRENT_USER_FIELD — return value of user-modifiable field

Syntax

```
function CURRENT_USER_FIELD
        (task_name  : in system.task_id := CURRENT_TASK)
        return user_field_t;
function CURRENT_USER_FIELD
        (task_name  : in system.task_id := CURRENT_TASK)
        return user_field2_t;
pragma INLINE_ONLY(CURRENT_USER_FILED);
```

Arguments

task_name

ID of the task for which the value of the user-modifiable field is returned. The default is the current task.

Description

The function CURRENT_USER_FIELD returns the value of the user-modifiable field of a specified task. If no task ID is specified for the parameter ***task_name***, the routine defaults to the current task. Either an INTEGER'SIZE value (via user_field_t) or an ADDRESS'SIZE value (via user_field2_t) can be retrieved from the task's USER_FIELD. The SC Ada runtime system does not use the USER_FIELD.

procedure DISABLE_PREEMPTION — inhibit current task preemption

Syntax

```
procedure DISABLE_PREEMPTION;
pragma INLINE_ONLY(DISABLE_PREEMPTION);
```

Description

The `procedure DISABLE_PREEMPTION` inhibits the current task from being preempted. This service does not disable signals. Task switching may still occur through the direct action of the currently running task making another higher priority task available to run. Signals still occur and ISR code is executed, but no other tasks run as a result of a signal.

Threaded Runtime

VADS MICRO maintains a preemption depth count for each task. This preemption depth is saved/restored at a task switch. Each call to `DISABLE_PREEMPTION` increments the depth. A nonzero depth count inhibits preemption. However, another task may run if the task calls a kernel service that causes it to block. Each call to `ENABLE_PREEMPTION` decrements the depth. When the depth is zero, the task can be preempted.

Other threaded runtimes may not support the disabling of preemption.

procedure DISABLE_TASK_COMPLETE — disable task completion/termination

Syntax

```
procedure DISABLE_TASK_COMPLETE;
pragma INLINE_ONLY(DISABLE_TASK_COMPLETE);
```

Description

`DISABLE_TASK_COMPLETE` disables the current task from being completed and terminated when aborted. If the task is aborted after `DISABLE_TASK_COMPLETE` is called, its completion is deferred until its mate, `ENABLE_TASK_COMPLETE`, is called. Calls to `DISABLE_TASK_COMPLETE` and `ENABLE_TASK_COMPLETE` can be nested.

These services can be used as follows to inhibit a task from being completed after it has acquired a sharable resource such as a semaphore:

```
DISABLE_TASK_COMPLETE;
        acquire_resource;
        use_resource;
        release_resource;
ENABLE_TASK_COMPLETE;
```

Caution – This procedure must always be paired with procedure `ENABLE_TASK_COMPLETE`.

procedure ENABLE_PREEMPTION — allow current task to be preempted

Syntax

```
procedure ENABLE_PREEMPTION;
pragma INLINE_ONLY(ENABLE_PREEMPTION);
```

Description

The `procedure ENABLE_PREEMPTION` allows the current task to be preempted.

Threaded Runtime

VADS MICRO maintains a preemption depth count for each task. This preemption depth is saved/restored at a task switch. Each call to `DISABLE_PREEMPTION` increments the depth. A nonzero depth count inhibits preemption. However, another task may run if the task calls a kernel service that causes it to block. Each call to `ENABLE_PREEMPTION` decrements the depth. When the depth is zero, the task can be preempted.

Other threaded runtimes may not support the disabling of preemption.

procedure ENABLE_TASK_COMPLETE — enable task completion/termination

Syntax

```
procedure ENABLE_TASK_COMPLETE;
pragma INLINE_ONLY(ENABLE_TASK_COMPLETE);
```

Description

`ENABLE_TASK_COMPLETE` enables the current task to be completed and terminated when aborted. This procedure must be paired with `DISABLE_TASK_COMPLETE`. They can be nested. There is no return if they are not nested and the current task has been marked abnormal by a previous abort.

These services can be used as follows to inhibit a task from being completed after it has acquired a sharable resource such as a semaphore:

```
DISABLE_TASK_COMPLETE;
        acquire_resource;
        use_resource;
        release_resource;
ENABLE_TASK_COMPLETE;
```

Caution – This procedure must always be paired with procedure `DISABLE_TASK_COMPLETE`.

function GET_PROGRAM — return task program identifier

Syntax

```
function GET_PROGRAM
        (task_name  : in system.task_id)
        return system.program_id;
pragma INLINE_ONLY(GET_PROGRAM);
```

Arguments

task_name

`ID` of the task for which the value of the program `ID` is returned.

Description

This function returns the program `ID` for the specified task.

Threaded Runtime

This service is provided for VADS MICRO. However, this service may not be supported by other underlying threaded runtimes.

function GET_PROGRAM_KEY — return user defined key

Syntax

```
function GET_PROGRAM_KEY
        (program_name:in system.program_id :=
CURRENT_PROGRAM)
        return system.address;
pragma INLINE_ONLY(GET_PROGRAM_KEY);
```

Arguments

program_name

ID of the program whose key is returned.

Description

This function returns the user-defined key for the specified program. The key is stored when the program is started via the START_PROGRAM service. The main program has a predefined key of SYSTEM.MEMORY_ADDRESS(0).

Threaded Runtime

This service is provided for VADS MICRO. However, this service may not be supported by other underlying threaded runtimes.

function GET_TASK_STORAGE — return starting address of task storage area

Syntax

```
function GET_TASK_STORAGE
        (task_name  : in system.task_id)
        storage_id  : in task_storage_id)
        return system.address;
pragma INLINE_ONLY(GET_TASK_STORAGE);
```

Arguments

storage_id

Value returned by a previous call to ALLOCATE_TASK_STORAGE. It is applicable only to tasks in the program where the ALLOCATE_TASK_STORAGE service is called from

task_name

ID of the task for which the address of the task storage area is returned.

Description

The service returns the starting address of the task storage area associated with the *storage_id* using the Ada task identifier (instead of the OS task identifier).

Threaded Runtime

This service is provided for VADS MICRO. However, this service may have different semantics or may not be supported by other underlying threaded runtimes.

function GET_TASK_STORAGE2 — get task storage using OS ID of task

Syntax

```
function GET_TASK_STORAGE2
        (os_task_name: in os_task_id)
        storage_id  : in task_storage_id)
        return system.address;
pragma INLINE_ONLY(GET_TASK_STORAGE2);
```

Arguments

os_task_name

OS identifier of the task for which the address of the task storage area is returned

storage_id

Value returned by a previous call to ALLOCATE_TASK_STORAGE. It is applicable only to tasks in the program where the ALLOCATE_TASK_STORAGE service is called from.

Description

The service returns the starting address of the task storage area associated with the *storage_id* using the OS ID of the task (instead of the Ada task identifier).

Threaded Runtime

This service is provided for VADS MICRO. However, this service may not be supported by other underlying threaded runtimes.

function ID — return task or program identifier

Syntax

```
function ID
      (task_address: in system.task_id)
      return task_id;
pragma INLINE_ONLY(ID);
function ID
      (os_task_name: in os_task_id)
      return system.task_id;
pragma INLINE_ONLY(ID);
function ID
      (os_program_name: in os_program_id)
      return system.program_id;
pragma INLINE_ONLY(ID);
```

Arguments

os_program_name

OS identifier of the program.

os_task_name

OS identifier of the task.

task_address

Address of a task obtained by applying the 'TASK_ID attribute to a task name.

The T'ADDRESS attribute has been changed in Sun Ada 1.1 and later releases to be the starting address of the task body machine code. Therefore, the type of the TASK_ADDRESS parameter has changed from SYSTEM.ADDRESS to SYSTEM.TASK_ID.

Description

If a task address is given, `ID` returns a task identifier for any task. The `ID` of a task object of a task type may be determined using either this function or a function created by an instantiation of the `generic function V_ID`. However, the `ID` of a task that is not a task object of a task type may only be determined by the ID function. This function can be used for all tasks.

`ID` also converts from an underlying thread or operating system identifier to an Ada task `ID` or Ada program `ID`.

Note – The `ID` function for converting a `TASK_ADDRESS` no longer needs to be called. The `T'TASK_ID` attribute returns the `ID` for a task, which may be used by the `V_XTASKING` services.

Example

The following code fragment uses the ID function with the `RESUME_TASK` operation to resume the task `OTHER_TASK`:

```
V_XTASKING.RESUME_TASK(V_XTASKING.ID(OTHER_TASK'task_id));
```

Error Results

`NULL_TASK_NAME`

Returned if the OS task is not also an Ada task.

`NO_PROGRAM_ID`

Returned if the OS program is not also an Ada program.

procedure INSTALL_CALLOUT — install procedure

Syntax

```
procedure INSTALL_CALLOUT
       (event      : in  callout_event_t;
       proc        : in  system.address);
pragma INLINE_ONLY(INSTALL_CALLOUT)
```

Arguments

event

Program, task, or idle event at which time the installed procedure is called.

proc

Address of the procedure to be called. The EXIT_EVENT or UNEXPECTED_EXIT_EVENT callout procedures are called as follows:

```
procedure exit_callout_proc
(status : in integer);  -- main subprogram
                        -- return status
```

The PROGRAM_SWITCH_EVENT callout procedure is called as follows:

```
procedure program_switch_callout_proc
    (new_os_program_name  : in os_program_id;
            key  : in address);
```

Note that this procedure is called with the OS PROGRAM_NAME, and not the Ada PROGRAM_NAME. Use V_XTASKING.ID(os_program_name) to get the Ada PROGRAM_ID.

The TASK_CREATE_EVENT, TASK_SWITCH_EVENT and TASK_COMPLETE_EVENT callout procedures are called as follows:

```
procedure task_callout_proc
          (os_task_name     : in os_task_id);
```

Note that this procedure is called with the OS TASK_NAME and not the Ada TASK_NAME. You can use V_XTASKING.ID(os_task_name) to get the Ada TASK_ID.

The IDLE_EVENT callout procedure is called as follows:

```
procedure idle_callout_proc;
```

Description

This service installs a procedure to be called at a program, task, or idle event. Callouts can be installed for EXIT_EVENT, UNEXPECTED_EXIT_EVENT, PROGRAM_SWITCH_EVENT, TASK_CREATE_EVENT, TASK_SWITCH_EVENT, TASK_COMPLETE_EVENT, or IDLE_EVENT.

The EXIT_EVENT, UNEXPECTED_EXIT_EVENT and IDLE_EVENT callout procedures are called in LIFO (last-in/first-out) order. The remaining callout procedures are called in the order in which they were installed.

The callouts reside in the user program space. The EXIT_EVENT and UNEXPECTED_EXIT_EVENT callouts are called in the context of the program's main task. The remaining callouts are called directly from kernel logic and can only call kernel services that are reentrant, the same services callable from ISRs. The service of most interest is CALENDAR.CLOCK, which is called for time stamping.

Before any non-PROGRAM_SWITCH_EVENT callout procedure is invoked, the STACK_LIMIT in the user program is set to 0 to negate any stack limit checking. Therefore, the callout procedures do not need to be compiled with stack limit checking suppressed. However, the STACK_LIMIT is not zeroed before calling the PROGRAM_SWITCH_EVENT callout. It needs to be compiled with stack limit checking suppressed.

Except for PROGRAM_SWITCH_EVENT or IDLE_EVENT, the callouts are installed and called only for the program in which they reside.

An overview of the different callout events is as follows:

EXIT_EVENT

Called when the program exits or terminates itself. Not called when the program is terminated from another program. Still called when the UNEXPECTED_EXIT_EVENT callout is called.

IDLE_EVENT

Called whenever there are not any ready-to-run tasks.

PROGRAM_SWITCH_EVENT

Called before switching to a task that resides in a program different from the current program. Called for all program switches, not just switches to and from the program containing the callout Also called before switching to the kernel program's IDLE task. However, not called when a procedure in another program is called via V_XTASKING.INTER_PROGRAM_CALL.

TASK_COMPLETE_EVENT

Called whenever any task in the callout program completes or is aborted.

TASK_CREATE_EVENT

Called whenever a task is created in the program containing the callout. Since the TASK_CREATE_EVENT callout can be installed after numerous tasks have already been created, the INSTALL_CALLOUT service loops through all existing tasks invoking the just installed TASK_CREATE_EVENT callout.

TASK_SWITCH_EVENT

Called before switching to a different task in the same program. For a program switch, the TASK_SWITCH_EVENT callout is called with the *os_task_name* parameter set to NULL_OS_TASK_NAME.

UNEXPECTED_EXIT_EVENT

Called when the program is abandoned because of an unhandled Ada exception.

Exceptions

STORAGE_ERROR

Raised if not enough memory is available for the callout data structures.

Threaded Runtime

This service is provided as described for VADS MICRO. Other underlying threaded runtimes may support different callout events. However, all implementations are expected to support EXIT_EVENT and UNEXPECTED_EXIT_EVENT.

procedure INTER_PROGRAM_CALL — call procedure in another program

Syntax

```
procedure INTER_PROGRAM_CALL
          (procedure_program: in     system.program_id;
           procedure_address: in     system.address;
           argument: in system.address);
pragma INLINE_ONLY(INTER_PROGRAM_CALL);
```

Arguments

argument

The only parameter passed to the called procedure.

procedure_address

Address of the called procedure.

procedure_program

Program ID of the called procedure.

Description

procedure INTER_PROGRAM_CALL does an indirect call to a procedure in another program.

procedure_program can be set to three different values: PROGRAM_SELF(), NO_PROGRAM_ID, or the program ID of the program containing the called procedure.

If procedure_program is set to PROGRAM_SELF() or NO_PROGRAM_ID, the procedure is called directly without switching the current program or stack limit.

If procedure_program is set to NO_PROGRAM_ID and the called procedure is in another program, the called procedure must use pragma SUPPRESS(ALL_CHECKS) and it cannot raise any Ada exceptions. In addition, if it calls any kernel services, the calling program is still the parent and owner of any created tasks or memory allocated from the kernel. Any Ada "new" memory allocations will cause heap corruption if more memory is

needed from the kernel and the calling program terminates before the called program does. Therefore, Ada new memory allocation is strongly discouraged for this case.

If procedure_program is set to the program ID of the program containing the called procedure, and if the called procedure does any task creation or kernel memory allocations, the program containing the called procedure is the parent and owner.

Normally, `INTER_PROGRAM_CALL` is used in conjunction with the `V_NAMES.BIND_PROCEDURE` and `V_NAMES.RESOLVE_PROCEDURE` services where a name has been bound to the procedure to be called.

If the program containing the called procedure doesn't have any active tasks, the `V_XTASKING.SET_IS_SERVER_PROGRAM` service should be called there during elaboration to mark it as a server.

The argument parameter is passed as the only argument to the called procedure. The called procedure has the following profile:

```
procedure called_procedure(argument: system.address);
```

During the call, the current task is disabled from being completed and terminated by an Ada abort.

Before doing the call, the current program is switched. Also, the *stack_limit* in the program containing the called procedure is switched. Before returning, everything is restored.

Note that the `PROGRAM_SWITCH_EVENT` callouts are not called. The task's parent program is not switched. The `PROGRAM_SWITCH_EVENT` callouts are only called when the parent program switches (i.e., when you switch to another task that is in another parent program).

Caution – `INTER_PROGRAM_CALL` should not be used within the context of an ISR, passive task interrupt entry, or kernel event callout. Doing so will cause unpredictable results.

Ada exceptions can be raised and handled in the called procedure. However, `INTER_PROGRAM_CALL` does not handle the propagation of Ada exceptions across inter_program calls. Therefore, the called procedure must have a handler for all possible Ada exceptions. An Ada exception raised in the called

procedure can have an outer handler that maps the exception to error status returned to the calling program. The calling program can then decode the error status and reraise the Ada exception.

Threaded Runtime

This service is provided as described for the VADS MICRO kernel. However, this service may not be supported by other underlying OS threads.

References

"procedure BIND_PROCEDURE — bind name to program ID and address" on page 4-109

"procedure RESOLVE_PROCEDURE — resolve name" on page 4-113

"procedure SET_IS_SERVER_PROGRAM — mark current program" on page 4-180

Caution – For this product, names are only known within a single program. Name binding and resolution services are more applicable to the cross-target environment.

function OS_ID — return underlying OS task or program identifier

Syntax

```
function OS_ID
        (task_name  : in system.task_id)
        return os_task_id;
pragma INLINE_ONLY(OS_ID);
function OS_ID
        (program_name: in system.program_id)
        return os_program_id;
pragma INLINE_ONLY(OS_ID);
```

Arguments

program_name

Name of Ada program to be converted.

task_name

Name of Ada task to be converted.

Description

`OS_ID` returns the underlying OS task or program identifier given either the Ada task `ID` or the Ada program `ID`.

procedure RESUME_TASK — resume task execution

Syntax

```
procedure RESUME_TASK
      (task_name: in     system.task_id);
pragma INLINE_ONLY(RESUME_TASK);

procedure RESUME_TASK
      (task_name: in     system.task_id;
      result: out    xtasking_result)
pragma INLINE_ONLY(RESUME_TASK);
```

Arguments

result

Result of the task resumption.

task_name

ID of the task to be readied for execution.

Description

RESUME_TASK only readies a task suspended by SUSPEND_TASK. The task does not execute unless it is the highest priority ready task.

This service can only be used to activate a task suspended by a call to SUSPEND_TASK.

Exceptions/Results

INVALID_RESUME/NOT_RESUMED

Task is not resumed (Not used with VADS MICRO kernel).

/RESUMED

The result value RESUMED is returned if the task is resumed.

Threaded Runtime

Note that exceptions and results are OS dependent. For the VADS MICRO kernel, RESUME_TASK always returns RESUMED (or no exception is raised). Each task has a suspend flag. RESUME_TASK sets the task suspend flag to FALSE. If the task is ready to run, it is put on the run queue.

The semantics of the RESUME_TASK service are dependent on the underlying operating system. RESUME_TASK is layered on ADA_KRN_I.TASK_RESUME.

procedure SET_EXIT_DISABLED — change kernel EXIT_DISABLED_FLAG

Syntax

```
procedure SET_EXIT_DISABLED
        (new_value    : in boolean := TRUE);
pragma INLINE_ONLY(SET_EXIT_DISABLED);
```

Arguments

new_value

Boolean value to which the EXIT_DISABLED_FLAG flag is to be set.

Description

SET_EXIT_DISABLED sets the Ada tasking global variable, EXIT_DISABLED_FLAG. This flag is initialized to FALSE, allowing the application program to exit when no tasks are on either the run or delay queue. This service is called, with new_value := TRUE, to inhibit the program from exiting. It is normally called after the application program attaches an ISR. The program is allowed to exit with a subsequent call, where new_value := FALSE.

procedure SET_FAST_RENDEZVOUS_ENABLED — set flag

Syntax

```
procedure SET_FAST_RENDEZVOUS_ENABLED
        (new_value : boolean);
pragma INLINE_ONLY(SET_FAST_RENDEZVOUS_ENABLED);
```

Arguments

new_value

New flag setting.

Description

SET_FAST_RENDEZVOUS_ENABLED sets the FAST_RENDEZVOUS_ENABLED flag for the current Ada task. See FAST_RENDEZVOUS_ENABLED in v_usr_conf.a for details about the fast rendezvous optimization.

If FAST_RENDEZVOUS_ENABLED is disabled in the configuration table, it can never be enabled.

Normally, fast rendezvous would need to be disabled only for multiprocessor Ada where the accept body must execute in the acceptor task bound to a processor.

References

“Fast Rendezvous Optimization” on page 2-76

procedure SET_IS_SERVER_PROGRAM — mark current program

Syntax

```
procedure SET_IS_SERVER_PROGRAM;
pragma INLINE_ONLY(SET_IS_SERVER_PROGRAM);
```

Description

`procedure SET_IS_SERVER_PROGRAM` marks the current program as a server program. If the current program won't have any active tasks and it contains procedures called via `V_XTASKING.INTER_PROGRAM_CALL`, this procedure should be called.

A server program has the following attributes. It is automatically terminated when no nonserver program is active. It is inhibited from exiting prematurely or being terminated. Also, when its main procedure returns (at end of server's elaboration), the main task's stack is freed and its microkernel thread is stopped/freed.

Threaded Runtime

`SET_IS_SERVER_PROGRAM` is provided as described for the VADS MICRO kernel. However, this service may not be supported by other underlying OS threads.

References

"procedure INTER_PROGRAM_CALL — call procedure in another program" on page 4-172

procedure SET_PRIORITY — change task priority

Syntax

```
procedure SET_PRIORITY
        (new_priority: in  system.priority;
        task_name   : in  system.task_id := CURRENT_TASK);
pragma INLINE_ONLY(SET_PRIORITY);
```

Arguments

new_priority

New priority setting.

task_name

`ID` of the task that has its priority changed.

Description

`SET_PRIORITY` enables you to change the priority of a specified task. Task scheduling is then reevaluated.

Warning – Use this service with extreme caution because it may interfere with the kernel scheduling.

procedure SET_TIME_SLICE — change task time slice interval

Syntax

```
procedure SET_TIME_SLICE
        (new_interval: in    duration;
        task_name    : in    system.task_id := CURRENT_TASK);
pragma INLINE_ONLY(SET_TIME_SLICE);
```

Arguments

new_interval

New time slice duration. An interval of 0.0 seconds disables time slicing for the task.

task_name

`ID` of the task that is to have the time slice interval changed. The default is the current task.

Description

`SET_TIME_SLICE` enables you to change the time slice interval of a specified task. `SET_TIME_SLICE` requires two parameters: the duration of the new time slice interval (*new_interval*) and the ID of the task that is to have the time slice interval changed (*task_name*).

Threaded Runtime

This service is provided for VADS MICRO. However, this service may have different semantics or may not be supported by other underlying threaded runtimes.

procedure SET_TIME_SLICING_ENABLED — enable/disable time slicing

Syntax

```
procedure SET_TIME_SLICING_ENABLED
    (new_value: in     boolean := TRUE);
pragma INLINE_ONLY(SET_TIME_SLICING_ENABLED);
```

Arguments

new_value

New value of the TIME_SLICING_ENABLED parameter.

Description

The procedure SET_TIME_SLICING_ENABLED enables you to change the value of the kernel TIME_SLICING_ENABLED configuration parameter. If no new value is specified for this function, the function defaults to a boolean value of TRUE.

Threaded Runtime

This service may not be supported by other underlying threaded runtimes. for more information.

procedure SET_USER_FIELD — change task user-modifiable field

Syntax

```
procedure SET_USER_FIELD
        (new_value  : in    user_field_t;
        task_name   : in    system.task_id := CURRENT_TASK);
procedure SET_USER_FIELD
        (new_value  : in    user_field2_t;
        task_name   : in    system.task_id := CURRENT_TASK);
pragma INLINE_ONLY(SET_USER_FIELD);
```

Arguments

new_value

New value to be written to the user-modifiable field.

task_name

ID of the task to have the user-modifiable field changed. The default is the current task.

Description

The procedure `SET_USER_FIELD` changes the value of the user-modifiable field for a specified task. If the task `ID` is not specified, the routine defaults to the current task. Either an `INTEGER'SIZE` value (via `user_field_t`) or an `ADDRESS'SIZE` value (via `user_field2_t`) can be stored in the task's `USER_FIELD`. The SC Ada runtime system does not use the `USER_FIELD`.

function START_PROGRAM — start separately linked program

Syntax

```
function START_PROGRAM
        (link_block_address: in system.address;
        key          : in system.address :=
system.memory_address(2);
        terminate_callout: in system.address :=
system.NO_ADDR);
        return system.program_id;
pragma INLINE_ONLY(START_PROGRAM);
```

Arguments

key

User-defined values are stored in the new program control block. This key is passed to the PROGRAM_SWITCH_EVENT callouts. The key may be obtained by routines in the new program via the GET_PROGRAM_KEY service. One use for the key is to have it point to a list of program arguments. The value for the main program key is SYSTEM.MEMORY_ADDRESS(0). For cross targets, the kernel program has a predefined key of SYSTEM.MEMORY_ADDRESS(1).

link_block_address

LINK_BLOCK address of the program to be started.

terminate_callout

Address of the procedure to be called when the program exits or is terminated. A value of NO_ADDR indicates no callout. The callout procedure is called as follows:

```
procedure terminate_callout_proc
     (os_program_name  : in os_program_id;
      key              : in address
```

Note that this procedure is called with the OS PROGRAM_NAME, not the Ada PROGRAM_NAME. Use V_XTASKING.ID(os_program_name) to get the Ada PROGRAM_ID.

TERMINATE_CALLOUT_PROC must be compiled with stack limit checking suppressed. The PROGRAM_SWITCH_EVENT callout is not called before the TERMINATE_CALLOUT_PROC is called.

Description

The function START_PROGRAM starts separately linked programs identified by their link block and enables concurrent execution of those programs. This function takes the parameter LINK_BLOCK_ADDRESS, which specifies the address of the LINK_BLOCK for the program to be started. It returns the ID of the just started program. Note that the Ada PROGRAM_ID and not the OS program or process ID is returned. This function assumes that the program to be started has previously been loaded.

Each program has a main task and a set of tasks it creates. Tasks from all programs execute from the same run queue.

The kernel has a linked list of programs with each program pointing to a linked list of the program tasks. Tasks from all programs compete for CPU time, according to their priority.

Threaded Runtime

This service is provided for VADS MICRO. However, this service may have different semantics or may not be supported by other underlying threaded runtimes.

procedure SUSPEND_TASK — suspend execution of task

Syntax

```
procedure SUSPEND_TASK
       (task_name  : in     system.task_id);
pragma INLINE_ONLY(SUSPEND_TASK);

procedure SUSPEND_TASK
       (task_name  : in     system.task_id;
       result      : out    xtasking_result);
pragma INLINE_ONLY(SUSPEND_TASK);
```

Arguments

result

Result of the task suspension.

task_name

ID of the task to be suspended.

Description

SUSPEND_TASK is used to suspend a task that is either currently running or ready-to-run.

Exceptions/Results

INVALID_SUSPEND/NOT_SUSPENDED

Task is not suspended (Not used with VADS MICRO kernel).

SUSPENDED

The result value SUSPENDED is returned if the task is suspended.

Threaded Runtime

For the VADS MICRO kernel, SUSPEND_TASK always returns SUSPENDED (or no exception is raised). Each task has a suspend flag. SUSPEND_TASK always sets the task suspend flag to TRUE. If the task is on the run queue, it is removed. The task is inhibited from being placed on the run queue until it is resumed with the RESUME_TASK service.

The semantics of the SUSPEND_TASK service are dependent on the underlying operating system. SUSPEND_TASK is layered on ADA_KRN_I.TASK_SUSPEND.

function TERMINATED — return value of P'TERMINATED attribute

Syntax

```
function TERMINATED
      (task_name: in system.task_id)
      return boolean;
pragma INLINE_ONLY(TERMINATED);
```

Arguments

task_name

ID of the task for which the value of the P'TERMINATED attribute is returned.

Description

The function TERMINATED returns the boolean value of the P'TERMINATED attribute for the specified task. If the task is terminated, TERMINATED returns the boolean value TRUE. Under all other conditions, TERMINATED returns the boolean value FALSE. Referencing a nonexistent task or a task that has not yet been created yields indeterminate values.

procedure TERMINATE_PROGRAM — terminate specified program

Syntax

```
procedure TERMINATE_PROGRAM
        (status      : in integer;
        program_name: in system.program_id :=
current_program);
pragma INLINE_ONLY(TERMINATE_PROGRAM);
```

Arguments

program_name

ID of the program to terminate.

status

Program exit status.

Description

This procedure terminates the specified program. If and only if the program to be terminated is the current program, the EXIT_EVENT callouts installed for the program are called before the program is terminated. After the program is terminated, the TERMINATE_CALLOUT passed to START_PROGRAM is called.

When a program is terminated, all its tasks are terminated and all memory allocated by the program is freed.

Threaded Runtime

This service is provided for VADS MICRO. However, this service may not be supported by other underlying threaded runtimes.

generic function V_ID — return identifier for task of specified type

Syntax

```
generic
        type task_type is limited private;
function V_ID
        (task_object: in task_type)
        return task_id;
pragma INLINE_ONLY(V_ID);
```

Arguments

task_object

Object of the task type used to instantiate this generic.

Description

This generic can be instantiated with a task type to create a function that returns the task identifier for any object of that task type. The resulting task identifier can be used as a parameter to the SUSPEND_TASK and RESUME_TASK operations. V_ID is applicable only to tasks declared as types.

Example

The following code fragment declares a task type DATA_RECEIVER and creates a function RECEIVER_ID, which returns a task identifier for a DATA_RECEIVER:

```
task type DATA_RECEIVER;
function RECEIVER_ID is new
V_XTASKING.V_ID(DATA_RECEIVER);
```

The following code fragment uses the function RECEIVER_ID to obtain a task identifier for a task object, DATA1, that is type DATA_RECEIVER. The resulting task ID is used to resume the task.

```
V_XTASKING.RESUME_TASK(RECEIVER_ID(DATA1));
```

"Let the great world spin forever down the ringing grooves of change."
Tennyson

Summary of RTS Changes

The Ada RTS is completely redesigned in this release so that Ada tasking can be layered upon any POSIX threads-like kernel. In prior releases, the Ada tasking subprograms resided in the kernel nucleus. Now, Ada tasking has been moved into user program space.

The RTS is partitioned into the following three layers:

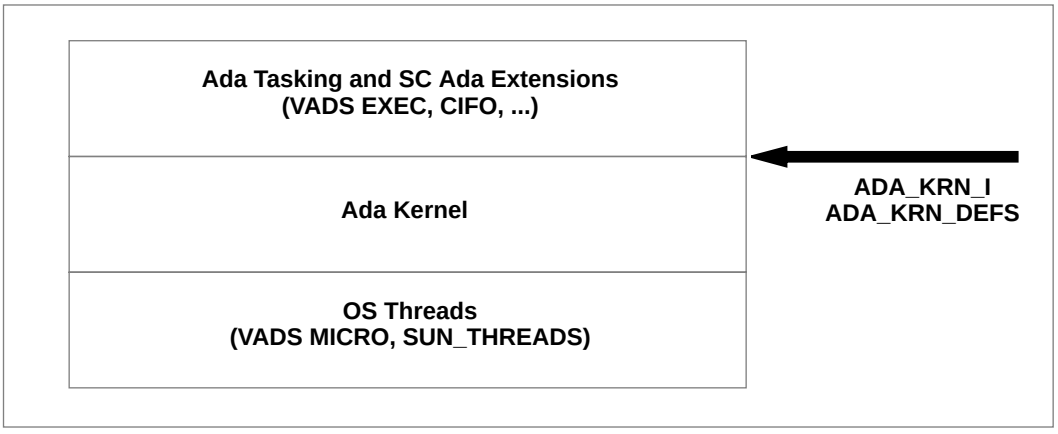

Figure 4-2 Ada RTS Layers

A

The Ada Kernel provides all the threads and synchronization services needed by the Ada Tasking layer. The interface to the Ada Kernel services is defined in the ADA_KRN_I package. Its type definitions are provided in the ADA_KRN_DEFS package.

The OS threads layer contains either the OS provided threads services (SUN_THREADS) or the Sun Microsystems, Inc. micro kernel (VADS MICRO). VADS MICRO does its own threads management independent of any threads services provided by the underlying OS. VADS MICRO combined with the other two layers replaces the monolithic Ada RTS provided in releases prior to SC Ada 2.1.

The Ada Tasking layer is only dependent upon the services defined in ADA_KRN_I and the types in ADA_KRN_DEFS. It makes no direct calls to services in the OS Threads layer. No changes are made in the Ada Tasking layer when the Ada RTS is ported to a new OS Threads. All porting changes are made in the Ada Kernel layer. The Ada Kernel layer does what it takes to map the ADA_KRN_I services (PROGRAM_INIT, TASK_CREATE, TASK_LOCK, TASK_WAIT, ...) onto the services provided by the underlying OS.

A.1 Why Redesign the Ada RTS?

By layering Ada tasks upon threads, Ada tasks can coexist with and call thread based services written in other languages, such as threaded windows written in C.

In addition, a SUN workstation can have multiple CPUs. Their threads services have been designed to allow concurrent thread execution from a single program across multiple CPUs. By layering Ada tasks upon threads, we also obtain the multi-CPU capability.

A.2 Mutex and Condition Variable

The new Ada RTS has added two object types to complement semaphores: mutexes and condition variables. The definition of these object types is extracted from the POSIX 1003.4a standard, “IEEE Threads Extension for Portable Operating Systems.” See the latest draft of that standard for more details about POSIX threads.

A.2.1 mutex

`mutex` is a synchronization object used to allow multiple threads to serialize their access to shared data. The name derives from the capability it provides, MUTual EXclusion.

The thread that has locked a mutex becomes its owner and remains the owner until the same thread unlocks the mutex. Other threads that attempt to lock the mutex during this period suspend execution until the original thread unlocks it. The act of suspending the execution of a thread awaiting a mutex does not prevent other threads from making progress in their computations.

Sun Microsystems, Inc. has also added an ABORT_SAFE option to mutexes. A task that has locked an ABORT_SAFE mutex is inhibited from being completed by an Ada abort until it unlocks the mutex.

A.2.2 condition variable

A *condition variable* is a synchronization object that allows a thread to suspend execution until some condition is true. Typically, a thread holding a mutex determines that it cannot proceed by examining the shared data guarded by the mutex. The thread then waits on a condition variable associated with some state of the shared data. Waiting on the condition variable atomically releases the mutex. If another thread modifies the shared data to make the condition true, that thread signals threads waiting on the condition variable. This wakes up the waiting thread which reacquires the mutex and resumes its execution.

An ABORT_SAFE option has been added to condition variables. A task locking an ABORT_SAFE mutex is inhibited from being completed by an Ada abort until it unlocks the mutex. However, if a task is aborted while waiting at a condition variable (after an implicit mutex unlock), then it is allowed to complete.

A.3 Impact upon Existing Tasking Applications

Re-doing the Ada RTS to be layered upon threads has necessitated numerous changes in the user interface to Ada tasking and VADS EXEC. Furthermore, we have continued to improve and enhance the Ada RTS.

Changes and/or extensions have been made in the following areas:

- Link directives in `ada.lib`
- `pragma PASSIVE`
- `pragma TASK_ATTRIBUTES`
- Interrupt entry
- Passive interrupt entry
- Ada I/O
- Memory allocation
- Fast rendezvous optimization
- `VADS EXEC: V_INTERRUPTS` services
- `VADS EXEC: V_XTASKING` services
- `VADS EXEC: V_SEMAPHORES` services
- `VADS EXEC: V_MAILBOXES` services
- `VADS EXEC: V_STACK` services
- `VADS_EXEC: V_NAMES` services
- `VADS_EXEC: V_MUTEXES` services
- `ADA_KRN_I`: interface to the Ada kernel services
- `ADA_KRN_DEFS`: Ada kernel type definitions
- Files added to standard
- `v_i_*` low level interfaces
- User program configuration (`v_usr_conf`)
- Kernel program configuration (`v_krn_conf`)
- TDM program configuration (`v_tdm_conf`)

Changes and improvements in each area are summarized below.

A.3.1 Link Directives in ada.lib

We have added the link directive, MIN_TASKING to the ada.lib in standard. MIN_TASKING points to the archive selected when the program does tasking but doesn't have any "aborts" or "select with terminates". Since the program doesn't need this capability, its been a.app ifdefed away in the MIN_TASKING archive to reduce the size of the program and to eliminate unnecessary checks.

A.3.2 pragma PASSIVE

Previously, the PASSIVE pragma took the following arguments:

```
pragma passive; -- defaults to SEMAPHORE
pragma passive(SEMAPHORE);
pragma passive(INTERRUPT,
v_i_types.disable_intr_status);
```

It has been changed to:

```
pragma passive; -- defaults to ABORT_UNSAFE,
                -- ada_krn_defs.DEFAULT_MUTEX_ATTR
pragma passive(ABORT_SAFE);
         -- defaults to ada_krn_defs.DEFAULT_MUTEX_ATTR
pragma passive(ABORT_SAFE, mutex_attr'address);
pragma passive(ABORT_UNSAFE);
         -- defaults to ada_krn_defs.DEFAULT_MUTEX_ATTR
pragma passive(ABORT_UNSAFE, mutex_attr'address);
```

An active task calling an ABORT_SAFE passive task entry is inhibited from being completed by an Ada abort until it finishes execution of the passive task's accept body. This inhibits the aborted task from holding a lock that is never released.

Alternatively, if an active task calling an ABORT_UNSAFE passive task entry is aborted, the lock isn't released and other active tasks are indefinitely blocked if they call an entry in the passive task. Previous releases supported only the ABORT_UNSAFE option. The ABORT_SAFE option is slightly slower.

The second argument is the address of an ADA_KRN_DEFS.MUTEX_ATTR_T record. The passive task's critical region is protected by locking and unlocking a mutex. The MUTEX_ATTR_T record is used to initialize the mutex. Omitting the second argument or setting it to ADA_KRN_DEFS.DEFAULT_MUTEX_ATTR selects the default mutex attributes.

The mutex attributes are OS dependent. See ada_krn_defs.a in *SCAda_location*/self/standard (single processor) or *SCAda_location*/self_thr/standard (multiprocessor or multithreaded Ada) for the type definition of MUTEX_ATTR_T and the different options supported.

ada_krn_defs.a has the following overloaded functions for initializing the mutex attributes. These functions will be supported for all versions of OS Threads. If the mutex type is not supported by the underlying OS, the PROGRAM_ERROR exception is raised.

```
function DEFAULT_MUTEX_ATTR return address;
function intr_attr_init(
  disable_status : intr_status_t := DISABLE_INTR_STATUS)
                                    return address;
function DEFAULT_INTR_ATTR return address;
function fifo_mutex_attr_init return address;
function prio_mutex_attr_init return address;
function prio_inherit_mutex_attr_init return address;
function prio_ceiling_mutex_attr_init return address;
```

The following new passive arguments:

```
pragma passive(ABORT_UNSAFE,
   ada_krn_defs.intr_attr_init(disable_intr_status));
```

or

```
pragma passive(ABORT_UNSAFE,
   ada_krn_defs.DEFAULT_INTR_ATTR);
         -- all interrupts disabled
```

are equivalent to the previous passive arguments:

```
pragma passive(INTERRUPT, disable_intr_status);
```

A.3.3 pragma TASK_ATTRIBUTES

The TASK_ATTRIBUTES pragma has been added. It has the following arguments:

```
pragma task_attributes(task_attr'address);
pragma task_attributes(task_object_name,
task_attr'address);
```

The first or second argument is the address of an ADA_KRN_DEFS.TASK_ATTR_T record. The TASK_ATTR_T record is passed to the underlying OS at task create.

The task attributes are OS dependent. See ada_krn_defs.a in standard for the type definition of TASK_ATTR_T and the different options supported. When there isn't a TASK_ATTRIBUTES pragma for a task, the DEFAULT_TASK_ATTRIBUTES found in the v_usr_conf_b.a configuration table are used.

All variations of the TASK_ATTR_T record contain at least the PRIO field. PRIO specifies the priority of the task. If the task also had a pragma PRIORITY(PRIO), the PRIO specified in the TASK_ATTR_T record takes precedence.

The first argument in the second form of the pragma is the name of a task object. This allows task objects of the same task type to have different task attributes (including different task priorities).

ada_krn_defs.a has the following overloaded functions for initializing the task attributes:

```
function task_attr_init(
   prio            : priority;
    .
    .  OS dependent fields
    .
) return address;
```

A.3.4 Interrupt Entry

In the 2.1 Ada RTS, the address specified in the interrupt entry `for use at` clause points to an `INTR_ENTRY_T` record defined in `ADA_KRN_DEFS`. The `INTR_ENTRY_T` record contains two fields: interrupt vector and the task priority for executing the interrupt entry's accept body. In previous release the address in the `for use at` clause specified the interrupt vector.

To preserve backwards compatibility, the parameter, `OLD_STYLE_MAX_INTR_ENTRY` was added to the configuration table in `v_usr_conf_b.a`. If the address in the `for use at` clause is <= `OLD_STYLE_MAX_INTR_ENTRY`, then, it contains the interrupt vector value and not a pointer to an `ADA_KRN_DEFS.INTR_ENTRY_T` record. Setting `OLD_STYLE_MAX_INTR_ENTRY` to 0 (NO_ADDR), disables the old way of interpretation.

The default value for `OLD_STYLE_MAX_INTR_ENTRY` is 511.

`ada_krn_defs.a` has the following overloaded function for initializing the interrupt entry:

```
function intr_entry_init(
   intr_vector : intr_vector_id_t;
   prio        : priority := priority'last) return address;
```

For `OLD_STYLE_MAX_INTR_ENTRY` = 511, the following two interrupt entries are identical:

```
task a is
   entry ctrl_c;
   for ctrl_c use at ada_krn_defs.intr_entry_init(
                              intr_vector => 2,
                              prio => priority'last - 1);
end;
```

or

```
task a is
   pragma priority(priority'last -1);
   entry ctrl_c;
   for ctrl_c use at system.memory_address(2);
end;
```

Note – for the old style interrupt entry, the task priority for executing the interrupt entry's accept body is always the priority of the attached task containing the interrupt entry (per POSIX 1003.5). The priority cannot differ as it can for the new style.

A.3.5 Passive Interrupt Entry

Previously, you specified a passive interrupt entry as follows:

```
task a is
  entry ctrl_c;
  for ctrl_c use at system.memory_address(2);
  pragma passive(INTERRUPT, disable_intr_status);
end;
```

In this version of SC Ada, you indicate a passive interrupt entry by:

```
task a is
  entry ctrl_c;
  for ctrl_c use at ada_krn_defs.intr_entry_init(
                           intr_vector => 2,
                           prio => priority'last);
  pragma passive(ABORT_UNSAFE,
      ada_krn_defs.intr_attr_init(disable_intr_status));
```

OR, to disable all interrupts

```
  pragma passive(ABORT_UNSAFE,
  ada_krn_defs.DEFAULT_INTR_ATTR);
                                -- all interrupts disabled
end;
```

If the underlying OS doesn't support passive interrupt entries, then, the PROGRAM_ERROR exception is raised.

A.3.6 Ada I/O

All `SEQUENTIAL_IO`, `DIRECT_IO`, `TEXT_IO`, `INTEGER_IO`, `FLOAT_IO`, `FIXED_IO` and `ENUMERATION_IO` file operations were changed to be Ada tasking safe and abort safe. All I/O operations are locked on a per file basis.

You now are guaranteed atomic file operations on a per task basis. Furthermore, the task initiating the I/O request is inhibited from being completed by an Ada Abort until it finishes.

Since I/O operations were not protected in the past, you could call the `TEXT_IO put` subprograms from a passive ISR accept body or an ISR (signal) handler. Now, if you call `text_io.put()` from an ISR, a task may already be doing an I/O operation to the same file and holding its lock. Since an ISR cannot block waiting for the lock, a `TASKING_ERROR` exception must be raised. Note that this restriction does not apply to non-passive interrupt entries.

To allow diagnostic output from an ISR, we have added `package SIMPLE_IO`, to `standard` for self hosts. (It already exists in `CROSS_IO` for cross targets.) The `SIMPLE_IO` package contains a subset of the `TEXT_IO put` subprograms. The `SIMPLE_IO` subprograms are unprotected and make direct calls to the OS I/O services.

A.3.7 Memory Allocation

Before this version of SC Ada, all memory allocations were task safe. Now they are also abort safe. The task doing a memory allocation operation is inhibited from being completed by an Ada abort until it finishes the allocation operation and releases the lock.

A.3.8 Fast Rendezvous Optimization

Normally the accept body of an Ada rendezvous is only executed in the context of the acceptor task. The fast rendezvous optimization also executes the accept body in the context of the caller task. This optimization reduces the number of thread context switches that need to be executed by the underlying OS Threads.

Here's an overview of the optimization: if the acceptor task gets to the accept statement before the caller task makes the call, the acceptor task saves its register and stack context, switches to a wait stack and does an `ADA_KRN_I.TASK_WAIT`. When the caller task gets around to doing the accept call, it saves its register and stack context, restores the acceptor task's register and stack context and returns to execute the accept body. When the end of the accept body is reached, the caller task overwrites the current register and stack context into the acceptor task's area, does an `ADA_KRN_I.TASK_SIGNAL` of the acceptor task, restores the caller task's

register and stack context and returns to the code in the caller task. Eventually, when the signaled acceptor task is scheduled to run, it restores the acceptor task's register and stack context (this context was updated by the caller task to be at the point where the call was made to finish the accept body) and returns to the code in the acceptor task after the call was made to finish the accept body.

Two configuration parameters have been added to `v_usr_conf` on behalf of the fast rendezvous optimization:

`FAST_RENDEZVOUS`

setting this parameter to `TRUE` enables the fast rendezvous optimization. This parameter would only need to be set to `FALSE`, for multiprocessor Ada, where the accept body must execute in the acceptor task bound to a processor. It defaults to `TRUE`.

`WAIT_STACK_SIZE`

This parameter specifies how much stack is needed for when the acceptor task switches from its normal task stack to a special stack it can use to call `ADA_KRN_I.TASK_WAIT`.

When using the debugger, the fast rendezvous optimization (accept body is executed by the caller task) has a few subtle differences from the normal rendezvous case (accept body is executed by the acceptor task).

Here's an example to illustrate the differences. I have two tasks doing a repetitive rendezvous. The caller task is `RENDEZVOUS_SEND`. The acceptor task is `RENDEZVOUS_RECEIVE`. I have placed a breakpoint in the acceptor body. I have two cases, either the breakpoint is reached when the acceptor task (`RENDEZVOUS_RECEIVE`) is executing the accept body or the caller task (`RENDEZVOUS_SEND`) is executing the acceptor body. Here's the debugger output for the two cases.

Case 1: at breakpoint when the acceptor task is executing the accept body (normal rendezvous)

```
[1]  stopped at "/vc/test/task_rend2.a":15 in rendezvous_receive

>lt
Q  TASK                ADDR       STATUS
   rendezvous_send     01006e63c  in rendezvous
        rendezvous_receive[010068fbc].receive_item
*  rendezvous_receive  010068fbc  executing
        in rendezvous with rendezvous_send[01006e63c] at
           entry receive_item
>lt all
Q  TASK                ADDR       STATUS
   rendezvous_send     01006e63c  in rendezvous
        rendezvous_receive[010068fbc].receive_item
        thread id = 01006e7c0
*  rendezvous_receive  010068fbc  executing
        in rendezvous with rendezvous_send[01006e63c] at
           entry receive_item
        ENTRY         STATUS   TASKS WAITING
        receive_item           - no tasks waiting -
        thread id = 010069140
```

Case 2: at breakpoint when the caller task is executing accept body (fast rendezvous)

```
[1]  stopped at "/vc/test/task_rend2.a":15 in rendezvous_receive

>lt
Q  TASK                ADDR       STATUS
   rendezvous_send     01006e63c  doing rendezvous
        for rendezvous_receive[010068fbc].receive_item
*  rendezvous_receive  010068fbc  executing
        in rendezvous via rendezvous_send[01006e63c] at
           entry receive_item
>lt all
Q  TASK                ADDR       STATUS
   rendezvous_send     01006e63c  doing rendezvous
        for rendezvous_receive[010068fbc].receive_item
        thread id = 01006e7c0
*  rendezvous_receive  010068fbc  executing
        in rendezvous via rendezvous_send[01006e63c] at
           entry receive_item
        ENTRY         STATUS   TASKS WAITING
        receive_item           - no tasks waiting -
        thread id = 010069140
```

Here are the subtle differences for the fast rendezvous, case 2:

- Even though the breakpoint occurred in the RENDEZVOUS_SEND task, we still display RENDEZVOUS_RECEIVE as the current breakpointed task. You can select the caller task and still get the caller's callstack and see where it was making the rendezvous call from.
- The STATUS for RENDEZVOUS_SEND is "doing rendezvous" instead of "in rendezvous". "doing" instead of "in" indicates that the caller task is executing the accept body.
- The second line for RENDEZVOUS_RECEIVE is "in rendezvous via" instead of "in rendezvous with". "via" instead of "with" indicates that the caller task is executing the accept body.
- The only misleading piece of information is the current underlying thread that is executing. The debugger says that RENDEZVOUS_RECEIVE is the currently executing task. From this you would assume that its thread, 010069140, is the current one. However, for the fast rendezvous case, it is really RENDEZVOUS_SEND's thread, 01006e7c0.

A.3.9 VADS EXEC: V_INTERRUPTS Services

The following services were added to V_INTERRUPTS:

```
function ATTACH_ISR
            (vector     : in     vector_id;
             isr        : in     system.address)
                    return system.address;
-- Overloads the existing ATTACH_ISR procedure.
-- The new function version returns the previously
-- attached vector.
function DETACH_ISR(vector : in vector_id)
                    return system.address;
-- Overloads the existing DETACH_ISR procedure.
-- The new function version returns the previously
-- attached vector.
function CURRENT_SUPERVISOR_STATE return boolean;
-- Returns the supervisor/user state of the current task.
-- If the task is in supervisor state, returns TRUE.
function SET_SUPERVISOR_STATE(new_state : in boolean)
                    return boolean;
-- Sets the supervisor/user state for the current task
-- to a different setting.  The previous state is returned.
```

A.3.10 VADS EXEC: V_XTASKING Services

package SYSTEM contains the type definitions for the Ada task control block (TCB), TASK_ID and the Ada program control block (PCB), PROGRAM_ID. Since Ada tasking and VADS EXEC are layered upon the OS's tasks and programs, we also need to define types for the OS's TCB and PCB. The following types have been added:

```
type os_task_id is new ADA_KRN_DEFS.krn_task_id;
NULL_OS_TASK_NAME: constant os_task_id :=
 os_task_id(ADA_KRN_DEFS.NO_KRN_TASK_ID);
  -- Type of the underlying OS's task control block.
  -- The os_task_name parameter passed to the task callouts
  -- is of this type. The services ID() and OS_ID() are
  -- provided to map OS task id's to/from Ada task id's.
```

```
 (Continued)
  --
  -- Note, except for the task callout procedures and the
  -- GET_TASK_STORAGE2 service, the task_name parameters
  -- identify the Ada task and not the underlying OS task
  -- or thread.
type os_program_id is new ADA_KRN_DEFS.krn_program_id;
NULL_OS_PROGRAM_NAME: constant os_program_id :=
 os_program_id(ADA_KRN_DEFS.NO_KRN_PROGRAM_ID);
  -- Type of the underlying OS's program control block. The
  -- os_program_name parameter passed to the program
  -- callouts is of this type. The services ID() and
  -- OS_ID() are provided to map OS program id's
  -- to and from Ada program id's.
  --
  -- Note, except for the program callout procedures,
  -- the program_name parameters identify the Ada program
  -- and not the underlying OS program or process.
```

The following services have been added for converting from an Ada TASK_ID to an underlying thread/OS_TASK_ID and vice versa. Services have also been added for converting program IDs.

```
function ID(os_task_name : in os_task_id)
             return system.task_id;
 -- Returns an id for an Ada task given the underlying OS's
 -- task id. The os_task_name is passed as a parameter
 -- to the task callouts. Returns NULL_TASK_NAME if the
 -- OS task isn't also an Ada task.
function OS_ID(task_name : in system.task_id)
             return os_task_id;
  -- Returns the underlying OS's task id given the
  -- Ada task id.
function ID(os_program_name : in os_program_id)
             return system.program_id;
  -- Returns an id for an Ada program given the underlying
  -- OS's program id. The os_program_name is passed as a
  -- parameter to the program callouts. Returns
  -- NO_PROGRAM_ID if the OS program isn't also an
  -- Ada program.
function OS_ID(program_name : in system.program_id)
```

(Continued)

```
 (Continued)
            return os_program_id;
  -- Returns the underlying OS's program id given
  -- the Ada program id.
```

The following ABORT_SAFE services were added:

```
procedure DISABLE_TASK_COMPLETE;
procedure ENABLE_TASK_COMPLETE;
  -- Disables/enables the current task from being
  -- completed and terminated when aborted. These services
  -- must be paired.  They can be nested. No return, if
  -- not nested and the current task has been marked
  -- abnormal by a previous abort.
  --
  -- These services would be used as follows to inhibit a
  -- task from being completed after it has acquired a
  -- sharable resource such as a semaphore.
  --
  --      DISABLE_TASK_COMPLETE;
  --          acquire_resoure;
  --          use_resource;
  --          release_resource;
  --      ENABLE_TASK_COMPLETE;
```

The following FAST_RENDEZVOUS services have been added:

```
function CURRENT_FAST_RENDEZVOUS_ENABLED return boolean;
procedure SET_FAST_RENDEZVOUS_ENABLED (new_value : boolean);
  -- Gets/sets the fast rendezvous enabled
  -- flag for the current task.  See the
  -- "Fast Rendezvous Optimization" section
  -- in the overview for more details
  -- about the optimization.
  --
  -- Normally, fast rendezvous would only need
  -- to be disabled for multiprocessor Ada where
  -- the accept body must execute in the
  -- acceptor task that is bound to a
  -- processor.
```

The following service has been added to support calls to procedures in other programs:

```
procedure INTER_PROGRAM_CALL
          (procedure_program: in     system.program_id;
           procedure_address: in     system.address;
           argument: in     system.address);
pragma INLINE_ONLY(INTER_PROGRAM_CALL);
```

The following service has been added to mark the current program as a server program:

```
procedure SET_IS_SERVER_PROGRAM;
pragma INLINE_ONLY(SET_IS_SERVER_PROGRAM);
```

The semantics of the RESUME_TASK and SUSPEND_TASK services depend on the underlying OS. These VADS EXEC services are layered upon ADA_KRN_I.TASK_RESUME and ADA_KRN_I.TASK_SUSPEND.

For the VADS MICRO kernel, RESUME_TASK always returns RESUMED (or no exception is raised). Each task has a suspend flag. RESUME_TASK sets the task's suspend flag to FALSE. If the task is READY to run, it is put on the run queue.

For the VADS MICRO kernel, SUSPEND_TASK always returns SUSPENDED (or no exception is raised). SUSPEND_TASK sets the task's suspend flag to TRUE. If the task is on the run queue, it is removed. The task is inhibited from being placed on the run queue until it is resumed via the RESUME_TASK service.

For SUN_THREADS, RESUME_TASK/SUSPEND_TASK map directly upon the Solaris thread service THR_CONTINUE/THR_SUSPEND.

The time slicing services are supported only by the VADS MICRO kernel. These are the services: CURRENT_TIME_SLICE, SET_TIME_SLICE, CURRENT_TIME_SLICING_ENABLED and SET_TIME_SLICING_ENABLED.

The program services, START_PROGRAM and GET_PROGRAM_KEY are not supported.

The TERMINATE_CALLOUT passed to START_PROGRAM is called with the OS's PROGRAM_NAME and not the Ada PROGRAM_NAME.

For the INSTALL_CALLOUT service, only the events, EXIT_EVENT and UNEXPECTED_EXIT_EVENT are supported by all the underlying OSs. The IDLE_EVENT was added for the VADS MICRO. Also, only the VADS MICRO supports the task storage services.

For VADS MICRO, the PROGRAM_SWITCH_CALLOUT is called with the OS's PROGRAM_NAME and not the Ada PROGRAM_NAME.

For VADS MICRO, the TASK_CALLOUTs are called with the OS's TASK_NAME and not the Ada PROGRAM_NAME. The newly added service, GET_TASK_STORAGE2, must be called to get task storage using the OS_TASK_NAME.

The following task storage service was added:

```
function GET_TASK_STORAGE2
           (os_task_name : in     os_task_id;
            storage_id   : in     task_storage_id)
                          return system.address;
-- Returns the starting address of the task storage area
-- associated with the storage_id using OS's id of the task
-- (instead of the Ada task_id)
```

The implementation of the DISABLE_PREEMPTION/ENABLE_PREEMPTION services was changed for the VADS MICRO. The VADS MICRO kernel maintains a preemption depth count for each task. This preemption depth is saved/restored at a task switch. Each call to DISABLE_PREEMPTION increments the depth. A nonzero depth count inhibits preemption. However, another task may run if the task calls a kernel service that causes it to block. Each call to ENABLE_PREEMPTION decrements the depth. When the depth is zero, the task may be preempted.

A.3.11 VADS EXEC: V_SEMAPHORES Services

The ATTR parameter has been added to the binary CREATE_SEMAPHORE services. The INTERRUPT_FLAG and INTERRUPT_STATUS parameters for the counting CREATE_SEMAPHORE services have been replaced with the ATTR parameter. For the binary CREATE_SEMAPHORE, ATTR points to an ADA_KRN_DEFS.SEMAPHORE_ATTR_T record. For the counting CREATE_SEMAPHORE, ATTR points to an ADA_KRN_DEFS.COUNT_SEMAPHORE_ATTR_T record.

The binary/counting semaphore attributes are OS dependent. See ada_krn_defs.a in standard for the type definition of SEMAPHORE_ATTR_T/COUNT_SEMAPHORE_ATTR_T and the different options supported.

The ATTR parameter has been defaulted to DEFAULT_SEMAPHORE_ATTR or DEFAULT_COUNT_SEMAPHORE_ATTR. Unless you want to do something special, the default should suffice.

For the VADS MICRO counting CREATE_SEMAPHORE: use the ADA_KRN_DEFS.COUNT_INTR_ATTR_T to protect the critical region for updating the semaphore's count by disabling interrupts. Setting ATTR to ADA_KRN_DEFS.DEFAULT_COUNT_INTR_ATTR disables all interrupts.

Fixed the DELETE_SEMAPHORE logic to always free memory allocated for the semaphore. Previously, memory wasn't freed if the semaphore was used for signaling.

A.3.12 VADS EXEC: V_MAILBOXES Services

The INTERRUPT_FLAG and INTERRUPT_STATUS parameters for the CREATE_MAILBOX services have been replaced with the ATTR parameter. ATTR points to an ADA_KRN_DEFS.MAILBOX_ATTR_T record.

The mailbox attributes are OS dependent. See ada_krn_defs.a in standard for the type definition of MAILBOX_ATTR_T and the different options supported.

The ATTR parameter has been defaulted to DEFAULT_MAILBOX_ATTR. Unless you want to do something special, the default should suffice.

For VADS MICRO, use the ADA_KRN_DEFS.MAILBOX_INTR_ATTR_T to protect the mailbox's critical region by disabling interrupts. Setting ATTR to ADA_KRN_DEFS.DEFAULT_MAILBOX_INTR_ATTR disables all interrupts.

A.3.13 VADS EXEC: V_STACK Services

The EXTEND_STACK service isn't supported. When called, it always raises the STORAGE_ERROR exception.

A.3.14 VADS_EXEC: V_NAMES Services

package V_NAMES provides name services. These services allow objects to be shared between programs or in conjunction with V_XTASKING.INTER_PROGRAM_CALL allow a procedure to be called from another program. V_NAMES has been layered on the Ada Kernel's name services.

A.3.15 VADS_EXEC: V_MUTEXES Services

package V_MUTEXES provides mutexes and condition variables. It is layered directly on top of the V_I_MUTEX package, which provides abort-safe mutexes and condition variables.

A.3.16 ADA_KRN_I: Interface To the Ada Kernel Services

In earlier releases, the interface to the low level kernel services was scattered across multiple v_i_* files. In Version 3.0, the interface is consolidated in one package, ADA_KRN_I. Most of the services are needed to support the Ada tasking semantics. The remaining services are needed to support VADS EXEC. The VADS EXEC services are optional and not supported by all versions of the OS Threads.

The interface to the services was designed from the viewpoint of the Ada tasking layer. For example, when a task is created, it returns the Ada TASK_ID and not the underlying kernel's TASK_ID.

The services are subdivided into the following groups.

Program
Kernel scheduling
Task management

Task masters synchronization
Task synchronization
Interrupt
Time
Allocation
Mutex
ISR mutex
Semaphore
Count semaphore
Mailbox
Callout
Task storage

Here is list of the services provided in each group.

Program services
- PROGRAM_INIT
- PROGRAM_EXIT
- PROGRAM_DIAGNOSTIC
- PANIC_EXIT
- PROGRAM_IS_ACTIVE
- PROGRAM_SELF

Program services (VADS EXEC augmentation)
- PROGRAM_GET
- PROGRAM_START
- PROGRAM_TERMINATE
- PROGRAM_GET_KEY
- PROGRAM_GET_ADA_ID
- PROGRAM_GET_KRN_ID

Kernel scheduling services (VADS EXEC augmentation)
- KERNEL_GET_TIME_SLICING_ENABLED
- KERNEL_SET_TIME_SLICING_ENABLED

Task management services
- TASK_SELF
- TASK_SET_PRIORITY
- TASK_GET_PRIORITY
- TASK_CREATE
- TASK_ACTIVATE
- TASK_STOP
- TASK_DESTROY
- TASK_STOP_SELF
- TASK_DESTROY_SELF

Task management services (VADS EXEC augmentation)
- TASK_DISABLE_PREEMPTION
- TASK_ENABLE_PREEMPTION
- TASK_GET_ADA_ID
- TASK_GET_KRN_ID
- TASK_SUSPEND
- TASK_RESUME
- TASK_GET_TIME_SLICE
- TASK_SET_TIME_SLICE
- TASK_GET_SUPERVISOR_STATE
- TASK_ENTER_SUPERVISOR_STATE
- TASK_LEAVE_SUPERVISOR_STATE

Task masters synchronization services
- MASTERS_LOCK
- MASTERS_TRYLOCK
- MASTERS_UNLOCK

Task synchronization services
- TASK_LOCK
- TASK_UNLOCK
- TASK_WAIT
- TASK_WAIT_LOCKED_MASTERS
- TASK_TIMED_WAIT
- TASK_SIGNAL
- TASK_WAIT_UNLOCK
- TASK_SIGNAL_UNLOCK
- TASK_SIGNAL_WAIT_UNLOCK

Sporadic task services (CIFO augmentation)
- TASK_IS_SPORADIC
- TASK_SET_FORCE_HIGH_PRIORITY

Interrupt services
- INTERRUPTS_GET_STATUS
- INTERRUPTS_SET_STATUS
- ISR_ATTACH
- ISR_DETACH
- ISR_IN_CHECK

Time services
- TIME_SET
- TIME_GET
- TIME_DELAY
- TIME_DELAY_UNTIL

Allocation services
- ALLOC
- FREE

Mutex services
- MUTEX_INIT
- MUTEX_DESTROY
- MUTEX_LOCK
- MUTEX_TRYLOCK
- MUTEX_UNLOCK
- COND_INIT
- COND_DESTROY
- COND_WAIT
- COND_TIMED_WAIT
- COND_SIGNAL
- COND_BROADCAST
- COND_SIGNAL_UNLOCK

ISR mutex services
- ISR_MUTEX_LOCKABLE
- ISR_MUTEX_LOCK
- ISR_MUTEX_UNLOCK
- ISR_COND_SIGNAL

Priority ceiling mutex services (CIFO augmentation)
- CEILING_MUTEX_INIT
- CEILING_MUTEX_SET_PRIORITY
- CEILING_MUTEX_GET_PRIORITY

Semaphore services
- SEMAPHORE_INIT
- SEMAPHORE_DESTROY
- SEMAPHORE_WAIT
- SEMAPHORE_TRYWAIT
- SEMAPHORE_TIMED_WAIT
- SEMAPHORE_SIGNAL
- SEMAPHORE_GET_IN_USE

Count semaphore services (VADS EXEC augmentation)
- COUNT_SEMAPHORE_INIT
- COUNT_SEMAPHORE_DESTROY
- COUNT_SEMAPHORE_WAIT
- COUNT_SEMAPHORE_SIGNAL
- COUNT_SEMAPHORE_GET_IN_USE

Mailbox services (VADS EXEC augmentation)

MAILBOX_INIT
MAILBOX_DESTROY
MAILBOX_READ
MAILBOX_WRITE
MAILBOX_GET_COUNT
MAILBOX_GET_IN_USE

CALLOUT Services

CALLOUT_INSTALL

Task storage services (VADS EXEC augmentation)

TASK_STORAGE_ALLOC
TASK_STORAGE_GET
TASK_STORAGE_GET2

A.3.17 ADA_KRN_DEFS: Ada Kernel Type Definitions

All type definitions used by the Ada tasking layer that are OS Threads specific are defined in ADA_KRN_DEFS. ADA_KRN_DEFS also contains numerous functions for allocating and initializing object attribute records.

ada_krn_defs.a contains type definitions for the following objects:

- mutex
- condition variable
- semaphore
- counting semaphore
- mailbox

The following types are of interest to anyone using Ada tasking:

KRN_TASK_ID

The type of the underlying thread. ADA_KRN_I has services for mapping between the Ada TASK_ID and KRN_TASK_ID.

KRN_PROGRAM_ID

The type of the underlying program/process. ADA_KRN_I has services for mapping between the Ada PROGRAM_ID and KRN_PROGRAM_ID.

CALLOUT_EVENT_T

All versions of the Ada Kernel / OS Threads are expected to support at least EXIT_EVENT and UNEXPECTED_EXIT_EVENT.

TASK_STORAGE_ID

Where supported, the ID or handle of a user defined object stored in every task.

INTR_VECTOR_ID_T

Signal number range.

INTR_STATUS_T

Signal mask.

DISABLE_INTR_STATUS

Constant for disabling all interrupts. For self, constant for disabling all asynchronous signals.

ENABLE_INTR_STATUS

Constant for enabling all interrupts. For self, enabling all signals.

INTR_ENTRY_T

The address of an `INTR_ENTRY_T` object is specified in an interrupt entry `for use at` clause. The `INTR_ENTRY_T` record contains two fields: interrupt vector and the task priority for executing the interrupt entry's accept body See the section "Interrupt Entry" for more details.

The `INTR_ENTRY_T` record can be initialized using one of the overloaded `INTR_ENTRY_INIT` subprograms:

```
procedure intr_entry_init
    intr_entry : a_intr_entry_t;
    intr_vector: intr_vector_id_t;
    prio       : priority := priority'last);
function intr_entry_init(
    intr_entry : a_intr_entry_t;
    intr_vector: intr_vector_id_t;
    prio       : priority := priority'last) return address;
function intr_entry_init(
    -- does an implicit "intr_entry: a_intr_entry_t :=
```

```
 (Continued)
    --                                  new intr_entry_t;”
    intr_vector: intr_vector_id_t;
    prio       : priority := priority’last) return address;
```

`intr_entry_init()` can be used as follows to define an interrupt entry:

```
with system;
with ada_krn_defs;
package one is
    -- Does an implicit allocation of the
    -- intr_entry_t record
    task a is
        entry ctrl_c;
        for ctrl_c use at ada_krn_defs.intr_entry_init(
                        intr_vector => 2,
                        prio => priority’last);
    end;
end one;
with system;
with ada_krn_defs;
package two is
    -- No implicit allocation is done
    ctrl_c_intr_entry_rec: ada_krn_defs.intr_entry_t;
    ctrl_c_intr_entry: system.address :=
        ada_krn_defs.intr_entry_init(
            intr_entry  => ada_krn_defs.to_a_intr_entry_t(
                            ctrl_c_intr_entry_rec’address),
            intr_vector => 2,
            prio        => priority’last);
    task a is
        entry ctrl_c;
        for ctrl_c use at ctrl_c_intr_entry;
        -- OR
        -- for ctrl_c use at ctrl_c_intr_entry_rec’address;
    end;
end two;
```

TASK_ATTR_T

The address of a TASK_ATTR_T record is the first argument of the TASK_ATTRIBUTES pragma and is passed to the underlying OS Threads at task create. The definition of the TASK_ATTR_T record is OS specific. However, all variations of the TASK_ATTR_T record contain at least the PRIO field. PRIO specifies the priority of the task. If the task also had a pragma PRIORITY(PRIO), then, the PRIO specified in the TASK_ATTR_T record takes precedence. See the section *Pragmas* for more details.

The TASK_ATTR_T record can be initialized using one of the overloaded TASK_ATTR_INIT subprograms:

```
procedure task_attr_init(
    task_attr   : a_task_attr_t;
    prio        : priority := priority'first;
    -- ...  OS Threads specific fields
);
function task_attr_init(
    task_attr   : a_task_attr_t;
    prio        : priority := priority'first;
    -- ...  OS Threads specific fields
) return address;
function task_attr_init(
    -- does an implicit "task_attr: a_task_attr_t :=
    --                           new task_attr_t;"
    prio        : priority := priority'first;
    -- ...  OS Threads specific fields
) return address;
```

task_attr_init() can be used as follows in a TASK_ATTRIBUTES pragma:

```
with system;
with ada_krn_defs;
package one is
   -- Does an implicit allocation of the task_attr_t record
    task a is
        pragma task_attributes(ada_krn_defs.task_attr_init(
                               prio => 20,
                               ... OS threads specific fields
```

```
 (Continued)
                              ));
    end;
    task type tt;
    b: tt;
        pragma task_attributes(b, ada_krn_defs.task_attr_init(
                                prio => 30,
                                ... OS threads specific fields
                              ));
end one;
with system;
with ada_krn_defs;
package two is
    -- No implicit allocation is done
    a_attr_rec: ada_krn_defs.task_attr_t;
    a_attr: system.address :=
        ada_krn_defs.task_attr_init(
            task_attr   => ada_krn_defs.to_a_task_attr_t(
                                a_attr_rec'address),
            prio        => 20,
            ... OS threads specific fields
        );
    b_attr_rec: ada_krn_defs.task_attr_t;
    b_attr: system.address :=
        ada_krn_defs.task_attr_init(
            task_attr   => ada_krn_defs.to_a_task_attr_t(
                                b_attr_rec'address),
            prio        => 30,
            ... OS threads specific fields
        );
    task a is
        pragma task_attributes(a_attr);
        -- or
        -- pragma task_attributes(a_attr_rec'address);
    end;
    task type tt;
    b: tt;
        pragma task_attributes(b, b_attr);
        -- or
        -- pragma task_attributes(b, b_attr_rec'address);
end two;
```

MUTEX_ATTR_T

The address of a MUTEX_ATTR_T record is the second argument of a PASSIVE pragma. The passive task's critical region is protected by locking and unlocking a mutex. The MUTEX_ATTR_T record is used to initialize the mutex. See the section *Pragmas* for more details on usage in passive tasks.

The mutex services in ADA_KRN_I can be used directly by the user program. The address of a MUTEX_ATTR_T record is passed directly to ADA_KRN_I.MUTEX_INIT.

The mutex attributes are OS Threads dependent. See ada_krn_defs.a in standard for the different options supported. (The VADS MICRO supports FIFO, priority or priority inheritance waiting when the mutex is locked by another task. In the CIFO add-on product, VADS MICRO also supports priority ceiling mutexes using the priority ceiling protocol emulation algorithm documented in the POSIX 1003.4a standard.)

The VADS MICRO has the following variant mutex attribute record type:

INTR_ATTR_T

disables interrupts (signals) to provide mutual exclusion

The function, DEFAULT_MUTEX_ATTR, is provided to select the default mutex attributes.

To provide mutual exclusion by disabling all interrupts, use DEFAULT_INTR_ATTR. If the underlying OS Threads doesn't support interrupt attributes, the PROGRAM_ERROR exception is raised.

The following subprograms are provided to initialize the MUTEX_ATTR_T record:

```
fifo_mutex_attr_init()
prio_mutex_attr_init()
prio_inherit_mutex_attr_init()
prio_ceiling_mutex_attr_init()
intr_attr_init()
```

If the underlying OS Threads doesn't support the type of mutex being initialized, the PROGRAM_ERROR exception is raised.

Here are the overloaded subprograms for initializing the MUTEX_ATTR_T record:

```
procedure fifo_mutex_attr_init(
    attr           : a_mutex_attr_t);
function fifo_mutex_attr_init(
    attr           : a_mutex_attr_t) return a_mutex_attr_t;
function fifo_mutex_attr_init(
    attr           : a_mutex_attr_t) return address;
function fifo_mutex_attr_init return a_mutex_attr_t;
    -- does an implicit
    --      "attr: a_mutex_attr_t := new mutex_attr_t;"
function fifo_mutex_attr_init return address;
    -- does an implicit
    --      "attr: a_mutex_attr_t := new mutex_attr_t;"
procedure prio_mutex_attr_init(
    attr           : a_mutex_attr_t);
function prio_mutex_attr_init(
    attr           : a_mutex_attr_t) return a_mutex_attr_t;
function prio_mutex_attr_init(
    attr           : a_mutex_attr_t) return address;
function prio_mutex_attr_init return a_mutex_attr_t;
    -- does an implicit
    -- "attr: a_mutex_attr_t := new mutex_attr_t;"
function prio_mutex_attr_init return address;
    -- does an implicit
    -- "attr: a_mutex_attr_t := new mutex_attr_t;"
procedure prio_inherit_mutex_attr_init(
    attr           : a_mutex_attr_t);
function prio_inherit_mutex_attr_init(
    attr           : a_mutex_attr_t) return a_mutex_attr_t;
function prio_inherit_mutex_attr_init(
    attr           : a_mutex_attr_t) return address;
function prio_inherit_mutex_attr_init return
              a_mutex_attr_t;
    -- does an implicit
    -- "attr: a_mutex_attr_t := new mutex_attr_t;"
function prio_inherit_mutex_attr_init return address;
    -- does an implicit
    --      "attr: a_mutex_attr_t := new mutex_attr_t;"
```

```
 (Continued)
procedure prio_ceiling_mutex_attr_init(
    attr    : a_mutex_attr_t;
    ceiling_prio: priority := priority'last);
function prio_ceiling_mutex_attr_init(
    attr    : a_mutex_attr_t;
    ceiling_prio: priority := priority'last) return a_mutex_attr_t;
function prio_ceiling_mutex_attr_init(
    attr    : a_mutex_attr_t;
    ceiling_prio: priority := priority'last) return address;
function prio_ceiling_mutex_attr_init(
    -- does an implicit "attr: a_mutex_attr_t := new mutex_attr_t;"
    ceiling_prio: priority := priority'last) return a_mutex_attr_t;
function prio_ceiling_mutex_attr_init(
    -- does an implicit "attr: a_mutex_attr_t := new mutex_attr_t;"
    ceiling_prio: priority := priority'last) return address;
procedure intr_attr_init(
    attr           : a_mutex_attr_t;
                disable_status  : intr_status_t :=
                DISABLE_INTR_STATUS);
function intr_attr_init(
    attr           : a_mutex_attr_t;
    disable_status : intr_status_t := DISABLE_INTR_STATUS)
                return a_mutex_attr_t;
function intr_attr_init(
    attr           : a_mutex_attr_t;
    disable_status : intr_status_t := DISABLE_INTR_STATUS)
return address;
function intr_attr_init(
    -- does an implicit "attr: a_mutex_attr_t :=
                new mutex_attr_t;"
    disable_status : intr_status_t := DISABLE_INTR_STATUS)
return a_mutex_attr_t;
function intr_attr_init(
    -- does an implicit "attr: a_mutex_attr_t :=
                new mutex_attr_t;"
    disable_status : intr_status_t := DISABLE_INTR_STATUS)
                                                return address;
```

The above init subprograms can be used as follows in a PASSIVE pragma:

```
with system;
with ada_krn_defs;
package one is
    task a is
        pragma passive(ABORT_SAFE,
               ada_krn_defs.fifo_mutex_attr_init);
    end;
    prio_mutex_attr_rec: ada_krn_defs.mutex_attr_t;
    prio_mutex_attr: system.address :=
        ada_krn_defs.prio_mutex_attr_init(
            ada_krn_defs.to_a_mutex_attr_t(
                prio_mutex_attr_rec'address));
    task b is
        pragma passive(ABORT_SAFE, prio_mutex_attr);
        -- or
        -- pragma passive(ABORT_SAFE,
               prio_mutex_attr_rec'address);
    end;
end one;
```

COND_ATTR_T

The address of a COND_ATTR_T record is passed to the Ada kernel service, cond_init().

The condition variable attributes are OS Threads dependent. See ada_krn_defs.a in standard for the different options supported. The VADS MICRO supports FIFO or priority waiting.

The function, DEFAULT_COND_ATTR, is provided to select the default condition variable attributes.

The following subprograms are provided to initialize the COND_ATTR_T record:

```
fifo_cond_attr_init()
prio_cond_attr_init()
```

If the underlying OS Threads doesn't support the type of condition variable being initialized, the PROGRAM_ERROR exception is raised.

Here are the overloaded subprograms for initializing the COND_ATTR_T record:

```
procedure fifo_cond_attr_init(
    attr            : a_cond_attr_t);
function fifo_cond_attr_init(
    attr            : a_cond_attr_t) return a_cond_attr_t;
function fifo_cond_attr_init return a_cond_attr_t;
    -- does an implicit
    --      "attr: a_cond_attr_t := new cond_attr_t;"
procedure prio_cond_attr_init(
    attr            : a_cond_attr_t);
function prio_cond_attr_init(
    attr            : a_cond_attr_t) return a_cond_attr_t;
function prio_cond_attr_init return a_cond_attr_t;
    -- does an implicit
    --      "attr: a_cond_attr_t := new cond_attr_t;"
```

SEMAPHORE_ATTR_T

The address of a SEMAPHORE_ATTR_T record is passed to the VADS EXEC service, v_semaphores.create_semaphore() which returns a BINARY_SEMAPHORE_ID or the Ada kernel service, ada_krn_i.semaphore_init().

The semaphore attributes are OS Threads dependent. See ada_krn_defs.a in standard for the different options supported. (The VADS MICRO only supports FIFO queuing when the task waits on a semaphore.)

The function, DEFAULT_SEMAPHORE_ATTR, is provided to select the default semaphore attributes.

COUNT_SEMAPHORE_ATTR_T

The address of a COUNT_SEMAPHORE_ATTR_T record is passed to the VADS EXEC service, v_semaphores.create_semaphore() which returns a COUNT_SEMAPHORE_ID or the Ada kernel service, ada_krn_i.count_semaphore_init().

The COUNT_SEMAPHORE attributes are OS Threads dependent. See ada_krn_defs.a in standard for the different options supported. (The VADS MICRO uses a mutex to protect the count. It waits on a condition

variable. The COUNT_SEMAPHORE_ATTR_T is a subtype of MUTEX_ATTR_T. The COND_ATTR_T is derived from the MUTEX_ATTR_T. A FIFO condition variable is used for a FIFO mutex. A priority condition variable is used for either a priority, priority inheritance or priority ceiling mutex.)

The VADS MICRO has the following variant COUNT_SEMAPHORE attribute record type:

```
count_intr_attr_t - interrupts (unix signals) are
disabled
                    when accessing the semaphore count
```

The function, DEFAULT_COUNT_SEMAPHORE_ATTR, is provided to select the default COUNT_SEMAPHORE attributes.

To protect COUNT_SEMAPHORE operations by disabling all interrupts, use DEFAULT_COUNT_INTR_ATTR. If the underlying OS Threads doesn't support interrupt attributes, the PROGRAM_ERROR exception is raised. However, if the DEFAULT_COUNT_SEMAPHORE_ATTR is interrupt (UNIX signal) safe, then, DEFAULT_COUNT_INTR_ATTR returns DEFAULT_COUNT_SEMAPHORE_ATTR and doesn't raise PROGRAM_ERROR.

Alternatively, the COUNT_SEMAPHORE attributes can be initialized to select the disable interrupts options by using one of the overloaded COUNT_INTR_ATTR_INIT subprograms:

```
procedure count_intr_attr_init(
    attr            : a_count_semaphore_attr_t;
    disable_status  : intr_status_t :=
              DISABLE_INTR_STATUS);
function count_intr_attr_init(
    attr            : a_count_semaphore_attr_t;
    disable_status  : intr_status_t := DISABLE_INTR_STATUS)
              return a_count_semaphore_attr_t;
function count_intr_attr_init(
    -- does an implicit
    --  "attr: a_count_semaphore_attr_t :=
    --                          new count_semaphore_attr_t;"
    disable_status  : intr_status_t := DISABLE_INTR_STATUS)
              return a_count_semaphore_attr_t;
```

MAILBOX_ATTR_T

The address of a MAILBOX_ATTR_T record is passed to the VADS EXEC service, v_mailboxes.create_mailbox() or the Ada kernel service, ada_krn_i.mailbox_init().

The mailbox attributes are OS Threads dependent. See ada_krn_defs.a in standard for the different options supported. (The VADS MICRO uses a mutex to protect the mailbox. It waits on a condition variable. The MAILBOX_ATTR_T is a subtype of MUTEX_ATTR_T. The COND_ATTR_T is derived from the MUTEX_ATTR_T. A FIFO condition variable is used for a FIFO mutex. A priority condition variable is used for either a priority, priority inheritance or priority ceiling mutex.)

The VADS MICRO has the following variant mailbox attribute record type:

```
mailbox_intr_attr_t - interrupts (unix signals) are
                      disabled when accessing the mailbox
```

The function, DEFAULT_MAILBOX_ATTR, is provided to select the default COUNT_SEMAPHORE attributes.

To protect mailbox operations by disabling all interrupts, use DEFAULT_MAILBOX_INTR_ATTR. If the underlying OS Threads doesn't support interrupt attributes, the PROGRAM_ERROR exception is raised. However, if the DEFAULT_MAILBOX_ATTR is interrupt (UNIX signal) safe, then, DEFAULT_MAILBOX_INTR_ATTR returns DEFAULT_MAILBOX_ATTR and doesn't raise PROGRAM_ERROR.

Alternatively, the mailbox attributes can be initialized to select the disable interrupts options by using one of the overloaded MAILBOX_INTR_ATTR_INIT subprograms:

```
procedure mailbox_intr_attr_init(
    attr           : a_mailbox_attr_t;
    disable_status : intr_status_t := DISABLE_INTR_STATUS);

function mailbox_intr_attr_init(
    attr           : a_mailbox_attr_t;
    disable_status : intr_status_t := DISABLE_INTR_STATUS)
                                  return a_mailbox_attr_t;
function mailbox_intr_attr_init(
    -- does an implicit
```

```
(Continued)
   --  "attr: a_mailbox_attr_t :=
   --                              new mailbox_attr_t;"
   disable_status : intr_status_t := DISABLE_INTR_STATUS)
                                   return a_mailbox_attr_t;
```

A.3.18 Files Added to Standard

The following files have been added to `standard`

ada_krn_i.a	interface to the Ada kernel services
ada_krn_defs.a	Ada kernel type definitions
krn_call_i.a	interface to the routines resident in the user program that call the kernel services
krn_cpu_defs.a	kernel program's type definitions that are CPU specific
krn_defs.a	kernel program's type definitions
krn_entries.a	kernel program's service entry IDs and arguments
simple_io.a	
simple_io_b.a	unprotected I/O subprograms that can be called from an ISR (UNIX signal handler)
usr_defs.a	type definitions for services resident in the user program
v_i_cifo.a	interface to the cifo type definitions used by Ada tasking
v_i_except.a	interface to the Ada exception services
v_i_mutex.a	interface to the ABORT_SAFE mutex services
v_usr_conf_i.a	interface for configuring the user program (moved from usr_conf)

A.3.19 New v_i_* Low Level Interfaces

The following v_i_* files were added:

A.3.19.1 v_i_cifo.a

This package contains the interface to the CIFO type definitions used by Ada tasking.

A.3.19.2 v_i_except.a

This package contains the interface to the Ada exception services for:

1. getting the ID and PC of the current Ada exception
2. returning the string name of an exception ID
3. installing a callout to be called whenever an exception is raised

A.3.19.3 v_i_mutex.a

This package interfaces to the mutex and condition variable services that are Ada tasking ABORT_SAFE. After locking a mutex the task is inhibited from being completed by an Ada abort until it unlocks the mutex.

However, if the task is aborted while waiting at a condition variable (after an implicit mutex unlock), it is allowed to complete. The V_I_MUTEX services also address the case where multiple ABORT_SAFE mutexes can be locked. A task is inhibited from being completed until all the mutexes are unlocked or it does a condition variable wait with only one mutex locked. There are services to init, destroy, lock, trylock and unlock an ABORT_SAFE mutex. There are services to init, destroy, wait on, timed wait on, signal and broadcast an ABORT_SAFE condition variable. In the CIFO add-on product, there are also services to init, set priority of and get priority of an ABORT_SAFE priority ceiling mutex.

A.3.20 Modified v_i_* Low Level Interfaces

Changes made to **v_i*** files that existed previously are summarized below.

A.3.20.1 v_i_alloc.a

Preserve backward compatibility by layering `KRN_AA_GLOBAL_NEW` on `ADA_KRN_I.ALLOC` and `KRN_AA_GLOBAL_FREE` on `ADA_KRN_I.FREE`.

A.3.20.2 v_i_callout.a

The interface to the low level callout kernel services is now provided in `ada_krn_i.a`. Types used by these services are defined in `ada_krn_defs.a`.

This package preserves backward compatibility by layering upon the callout data structures and subprograms found in `ada_krn_defs.a` and `ada_krn_i.a`.

Differences from earlier releases:

- Only the callout events, `EXIT_EVENT` and `UNEXPECTED_EXIT_EVENT` are supported by all the underlying RTS kernels. The VADS MICRO RTS continues to support the events: `PROGRAM_SWITCH_EVENT`, `TASK_CREATE_EVENT`, `TASK_SWITCH_EVENT` and `TASK_COMPLETE_EVENT`. The event, `IDLE_EVENT` has been added for the VADS MICRO RTS.
- For VADS MICRO: the task events are called with a pointer to the micro kernel's task control block, not the Ada `TASK_ID` as was done in earlier releases.
- For VADS MICRO: the program events are called with a pointer to the micro kernel's program control block, not the Ada `PROGRAM_ID` as was done in earlier releases.
- Added the service, `get_task_storage2()` to get the address of task storage using the underlying kernel's `TASK_ID` (not the Ada `TASK_ID` as is used for `get_task_storage()`). This was added because the task callouts are called with the kernel's `TASK_ID` and not the Ada `TASK_ID`.

A.3.20.3 v_i_csema.a

The interface to the low level counting semaphore services is now provided in `ada_krn_i.a`. Types used by these services is defined in `ada_krn_defs.a`.

This package preserves backward compatibility by layering upon the counting semaphore data structures and subprograms found in `ada_krn_defs.a` and `ada_krn_i.a`.

Differences from earlier releases:

- The `INTR_FLAG` and `INTR_STATUS` are only applicable to the VADS MICRO.
- For the `delete()` service, if the `CONDITIONAL_DELETE_FLAG` is `TRUE`, the semaphore might not be deleted even though no tasks are waiting on it. This caveat isn't applicable to the VADS MICRO.

A.3.20.4 v_i_intr.a

The interface to the low level interrupt services is now provided in `ada_krn_i.a`. Types used by these services is defined in `ada_krn_defs.a`.

This package preserves backward compatibility by layering upon the interrupt data structures and subprograms found in `ada_krn_defs.a` and `ada_krn_i.a`.

Differences from earlier releases:

- The following services aren't supported: `enter_isr()`, `complete_isr()`.

  If called, they raise the exception, `V_I_INTR_NOT_SUPPORTED`. Check `V_INTERRUPTS` in VADS EXEC, `V_PASSIVE_ISR` in `V_USR_CONF` or `V_SIGNAL_ISR` in `V_USR_CONF` for the routines to be called.

A.3.20.5 v_i_mbox.a

The interface to the low level mailbox services is now provided in `ada_krn_i.a`. Types used by these services is defined in `ada_krn_defs.a`.

This package preserves backward compatibility by layering upon the mailbox data structures and subprograms found in `ada_krn_defs.a` and `ada_krn_i.a`.

Differences from earlier releases:

- The INTR_FLAG and INTR_STATUS are only applicable to the VADS MICRO RTS.
- For the delete() service, if the CONDITIONAL_DELETE_FLAG is TRUE, the mailbox might not be deleted even though no tasks are waiting to read. This caveat isn't applicable to the VADS MICRO RTS.
- For mailbox write, only the DO_NOT_WAIT option is supported. Earlier releases also supported timed and WAIT_FOREVER write options.

A.3.20.6 *v_i_pass.a*

The passive task header record was changed to accommodate changes in the compiler and the RTS. Added ABORT_SAFE versions of the passive enter and leave services. In the CIFO add-on product, we also support priority ceiling server tasks.

A.3.20.7 *v_i_sema.a*

The interface to the low level semaphore services is now provided in ada_krn_i.a. Types used by these services is defined in ada_krn_defs.a.

This package preserves backward compatibility by layering upon the semaphore data structures and subprograms found in ada_krn_defs.a and ada_krn_i.a.

Differences from earlier releases:

- Semaphore must be initialized by calling the newly added routine, init_sema().
- For VADS MICRO, only FIFO queuing is supported. For priority queuing use the mutex and condition variable services provided in ada_krn_i.a.
- The following services aren't supported: suspend(), timed_suspend() or resume().

A.3.20.8 v_i_sig.a

The A_SIGNAL_T type was renamed to SIGNAL_ID. create_signal() is passed the address of an INTR_ENTRY_T record instead of the interrupt vector (SIGNAL_CODE). However, if the address is less than or equal to OLD_STYLE_MAX_INTR_ENTRY defined in V_USR_CONF, its still treated as the SIGNAL_CODE.

In addition, added ignore and unignore signal services for supporting POSIX_Signals (per 1003.5).

A.3.20.9 v_i_taskop.a

The subprograms were changed to use the types, SYSTEM.TASK_ID, SYSTEM.PROGRAM_ID, MASTER_ID and A_LIST_ID. Backward compatibility is preserved by using the following subtypes also defined here:

```
subtype a_task_t is system.task_id;
subtype a_program_t is system.program_id;
subtype a_master_t is MASTER_ID;
subtype a_alist_t is A_LIST_ID;
```

DAY_T type is now defined in system and not V_I_TYPES.

We have added two subprograms to support ABORT_SAFE operations:

```
ts_disable_complete
ts_enable_complete
```

The services, TS_INITIALIZE and TS_EXIT now have subprogram parameters. Two parameters, TASK_ATTR and HAS_PRAGMA_PRIO were added to the services, TS_CREATE_TASK and TS_CREATE_TASK_AND_LINK. The parameters passed to TS_ATTACH_INTERRUPT were redefined to be the ATTACHED_ENTRY and INTR_ENTRY.

The routine, TS_GET_STACK_LIMIT, was added to return the STACK_LIMIT of the current task. It handles the case where the caller task is executing code in a fast rendezvous using the acceptor's stack. It also handles the case or multiprocesssor or multithreaded Ada, where the compiler can't reference the stack limit directly from memory.

Added subprograms to get/put information on the last exception raised in the current task:

```
ts_get_last_exception
ts_put_last_exception
```

Added subprograms to support interrupt tasks:

```
ts_create_intr_task
ts_activate_intr_task
ts_complete_intr_task
ts_get_intr_task_vector
ts_get_intr_task_handler
ts_set_intr_task_state
```

Added subprograms to support CIFO:

```
ts_caller
ts_cifo_term_callout

ts_trivial_conditional_Call
ts_trivial_accept
ts_set_entry_criteria
ts_set_select_criteria
```

A.3.20.10 v_i_tasks.a

The subprograms were changed to use the types, `SYSTEM.TASK_ID`, `SYSTEM.PROGRAM_ID` and `INTEGER`. Backward compatibility is preserved by using the following subtypes defined in `V_I_TYPES`:

```
subtype v_i_types.a_task_t is system.task_id;
subtype v_i_types.a_program_t is system.program_id;
subtype v_i_types.user_field_t is integer;
```

Added subprograms to support CIFO:

```
task_has_pragma_priority
get_cifo_tcb
set_cifo_tcb
get_task_master

task_is_valid
in_passive_rendezvous
```

Added subprograms to support fast rendezvous optimization:

```
get_fast_rendezvous_enabled
set_fast_rendezvous_enabled
```

The following miscellaneous subprograms were added:

```
get_task_sequence_number
check_in_rendezvous
get_configuration_table
```

A.3.20.11 v_i_time.a

DAY_T is now defined in system. It was previously defined in V_I_TYPES.

The SET_TIME service was fixed so that it properly adjusts the time for both delta delays and delay until time events.

The XCALENDAR.SET_CLOCK routine was also fixed.

A.3.20.12 v_i_trap.a

It was removed from the self host standard.

A.3.20.13 v_i_types.a

Interrupt types moved to ada_krn_defs.a. DAY_T moved to system.a. EXCEPTION_STACK_SIZE moved to the configuration table in v_usr_conf_b.a.

Eliminated the different address types. Now only use SYSTEM.ADDRESS. Eliminated the miscellaneous types:

```
A_SIGNAL_T
UNIVERSAL_INTEGER_T
LONG_INTEGER
PHYSICAL_ADR_AS_INT.
```

A.3.21 User Program Configuration (v_usr_conf)

The following configuration parameters were deleted from the self host v_usr_conf:

```
KRN_STACK_SIZE
INTR_STACK_SIZE
PENDING_OVERFLOW_CALLOUT    -- the added routine,
                           -- V_PENDING_OVERFLOW_CALLOUT
                            -- is called directly
```

The following configuration parameters were deleted from the self host and cross target **v_usr_conf**:

```
EXIT_USER_PROGRAM_CALLOUT
            -- replaced by the ada_krn_i.callout_install
               -- service using either EXIT_EVENT or
               -- UNEXPECTED_EXIT_EVENT
```

The configuration parameter, `PRIORITY_INHERITANCE_ENABLED` was deleted. Priority inheritance is only supported in the CIFO add-on product.

The following configuration parameters were added to `v_usr_conf` for both self hosts and cross targets:

```
IDLE_STACK_SIZE
EXCEPTION_STACK_SIZE
SIGNAL_TASK_STACK_SIZE
FAST_RENDEZVOUS_ENABLED
WAIT_STACK_SIZE
FLOATING_POINT_SUPPORT
OLD_STYLE_MAX_INTR_ENTRY

DEFAULT_TASK_ATTRIBUTES
MAIN_TASK_ATTR_ADDRESS
SIGNAL_TASK_ATTR_ADDRESS
MASTERS_MUTEX_ATTR_ADDRESS
MEM_ALLOC_MUTEX_ATTR_ADDRESS
ADA_IO_MUTEX_ATTR_ADDRESS
```

The following configuration parameter was added to `V_USR_CONF` for UNIX self hosts with more than 32 signals:

```
DISABLE_SIGNALS_33_64_MASK
```

SUN_THREADS specific configuration parameters were added:

```
CONCURRENCY_LEVEL
DEFAULT_TASK_ATTRIBUTES
ENABLE_SIGNALS_MASK
ENABLE_SIGNALS_33_64_MASK
```

```
EXIT_SIGNALS_MASK
EXIT_SIGNALS_33_64_MASK
INTR_TASK_PRIO
INTR_TASK_STACK_SIZE
INTR_TASK_ATTR_ADDRESS
```

Check `v_usr_conf.a` for information on these newly added parameters.

`V_START_PROGRAM` was rewritten to call `TS_INITIALIZE`. Added `V_START_PROGRAM_CONTINUE`. Its address is passed to `TS_INITIALIZE` and is called when `TS_INITIALIZE` completes.

`V_PASSIVE_ISR` was rewritten to reflect compiler changes.

`V_CIFO_ISR` was added to support CIFO interrupt tasks.

For VADS MICRO self hosts, the routines, `V_PENDING_OVERFLOW_CALLOUT` and `V_KRN_ALLOC_CALLOUT` were added.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X