A Lace for Ada's* Corset

T.P. Baker[†]
K. Jeffay

TR-86-09-06
Department of Computer Science
University of Washington
Seattle, WA 98195

October 25, 1986

---

*Ada is a registered trademark of the U.S. Department of Defense (AJPO).

†Visiting from the Florida State University, Tallahassee, FL. This work supported in part by Boeing Aerospace Company and the Washington Technology Center.

Abstract

Lace, a Low-level Adaptable Common Executive, is a specification for an executive that implements a model of real-time, lightweight tasks. Lace is designed to string together Ada's Corset – a compact runtime support environment for tasking, described in a companion report [1].

Lace/I is a prototype Lace implementation designed for multiprocessor configurations of the MIL-STD-1750A instruction set architecture with shared memory. This is a decentralized "self-service" design in which each task executes its own run-time service calls, and each processor does its own dispatching out of a global task pool, thus automatically balancing the load between different processors.

Although specified in Ada, Lace's interface is the familiar procedure call interface, and therefore Lace could be tailored for use as a low-level real- time multiprogramming kernel with any contemporary sequential programming language.

1 Introduction

Lace is a specification for a low-level executive that implements a simple tasking paradigm. Its design was driven by the desire to provide an efficient executive that was rich enough to support a complex tasking paradigm, such as full Ada tasking, while remaining simple enough so that direct use of its primitives would result in a system with real-time performance suitable for embedded real-time applications.

Specific goals of the Lace interface as well as of the Lace/I prototype include the desire to experiment with providing:

- low overhead - in particular, avoid unnecessary saving and restoring of task state both during context switches and in interrupt handlers;
- localized cost - implement simple functions efficiently; their performance should not be degraded because of side effects of implementing more complex functions, even if this means slowing down already costly operations;
- predictable performance - use simple, deterministic algorithms, where possible (e.g. design the dispatcher with a minimum number of locks and so that interrupts never have to be disabled);
- limited capacities - restrict capacities (e.g. number of tasks, range of priorities) to obtain greater efficiency;
- adaptability - adhere to an interface that is implementable for various processor configurations, as well as modifications to the tasking paradigm.

Lace is not a complete runtime environment. It only provides for management of task execution, through allocation of tasks to processors. In particular, it does not provide directly for intertask communication or memory management. Such services are presumed to be layered on top of Lace, using the Lace operations in their implementation. Corset/I (a COmpact Runtime Support Environment for Tasking), described in a companion report [1], illustrates how a complete runtime support environment for Ada can be built using Lace.

Section 2 presents the Lace tasking paradigm of lightweight tasks. This is followed in Section 3 with a discussion of Lace's interface and usage. Section 4 describes Lace/I, a prototype implementation of Lace for a multiprocessor configuration of the MIL-STD-1750A processors with shared memory.

2 Tasking Paradigm

A Lace task is simply a "thread" of control, represented by a set of register values. It is presumed to have a body of code and a working storage area, but the management of storage for code and data is not a responsibility of Lace. An implementation of Lace provides a primitive set of operations for creating, synchronizing, executing, and recycling tasks. In addition, a Lace tasking environment includes interrupt handlers whose operations are transparent to Lace tasks. This environment is represented pictorially in Figure 1.

The Lace tasking paradigm can be understood as a finite state system where each state is characterized by a set of attributes. At any point in time, a Lace task has a set of boolean attributes that determine the allowable operations on the task. These attributes are organized hierarchically as shown in Figure 2. Conceptually, the leaves in this hierarchy are task states and the path from the root to a leaf (state) enumerates the attributes of the state. We assume that all tasks in the system are Lace tasks (the root attribute). The other attributes of interest are:

- allocated - the task is a valid Lace task;
- enabled - the task is eligible for execution on a processor;
- assigned - the task is executing on a processor;
- preemptible - task may be preempted on its processor by a higher priority Lace task.

Most Lace operations toggle task attribute values, as shown in Figure 2, and therefore cause state transitions, as shown in Figure 3. For example, a task in the contending state has the attributes that it is allocated, is enabled and is contending. Performing the HOLD operation on this task toggles the enabled attribute, hence revoking any other sub-attributes such as contending (see Figure 2) and causing a state transition from contending to a disabled state (see Figure 3). Similarly, when the DISPATCH operation selects a contending task for execution, that task toggles from contending to assigned and becomes preemptible.

Note that when an executing task becomes disabled, due to a HOLD operation, it continues to execute until it calls DISPATCH or is preempted. Thus a task A may wait for a reply from another task B by doing a HOLD on itself, posting a request for service to B in a buffer, and then calling DISPATCH. When B completes the service, it does a RELEASE on A. If, due to there being more than one processor, B should release A before A calls DISPATCH, A may return immediately from the call to DISPATCH, without needing to wait at all.

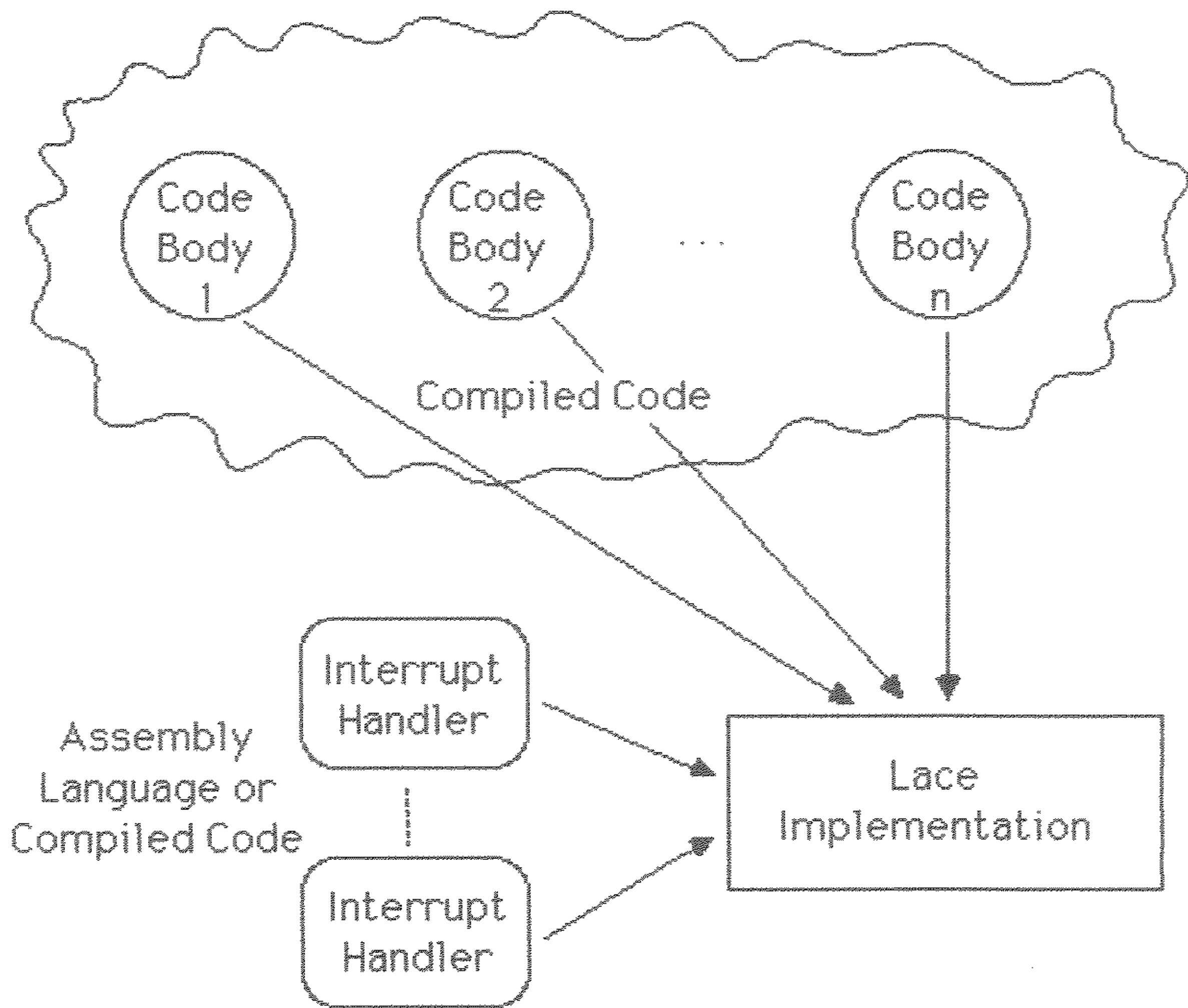

Figure 1: Lace Tasking Environment

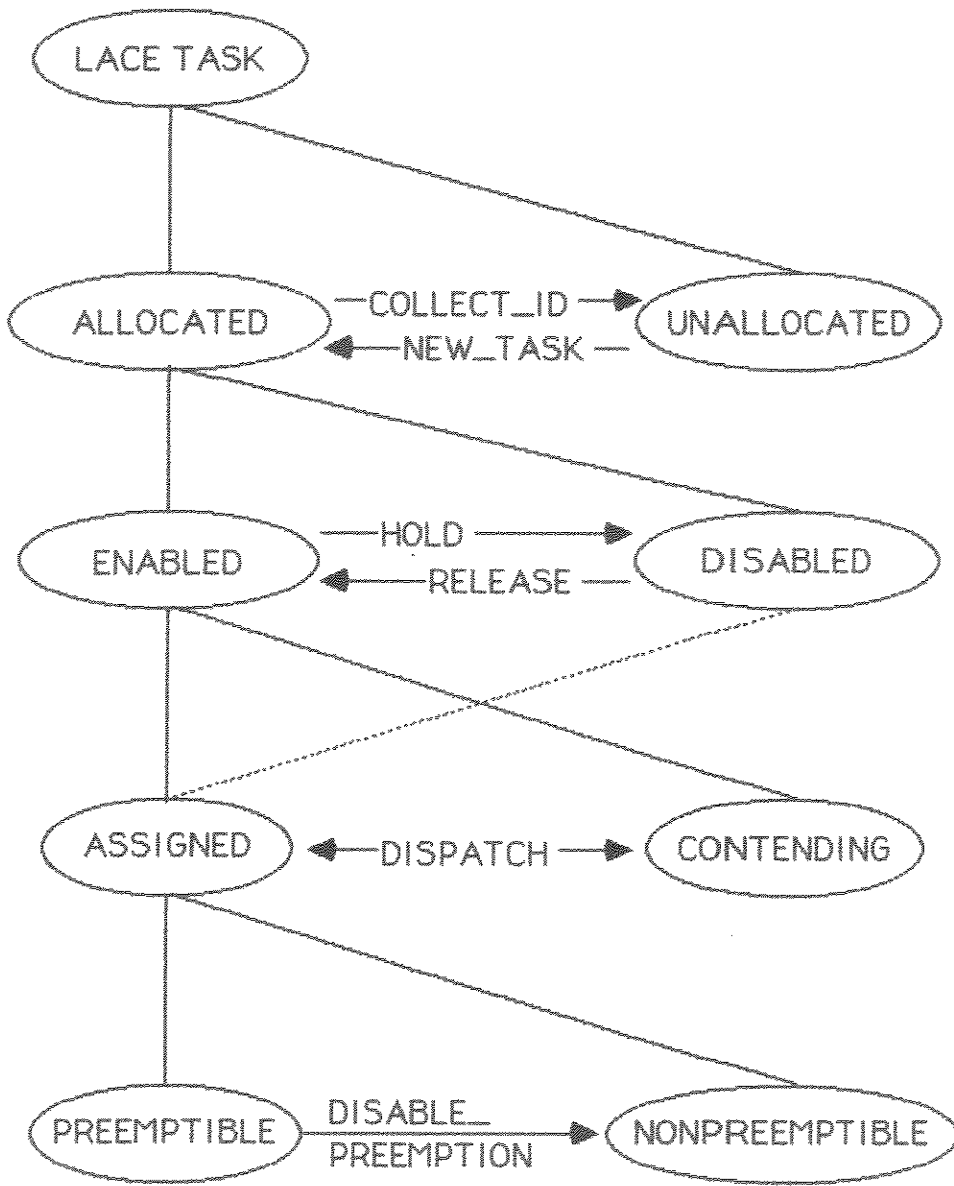

Figure 2: Lace Task Attributes

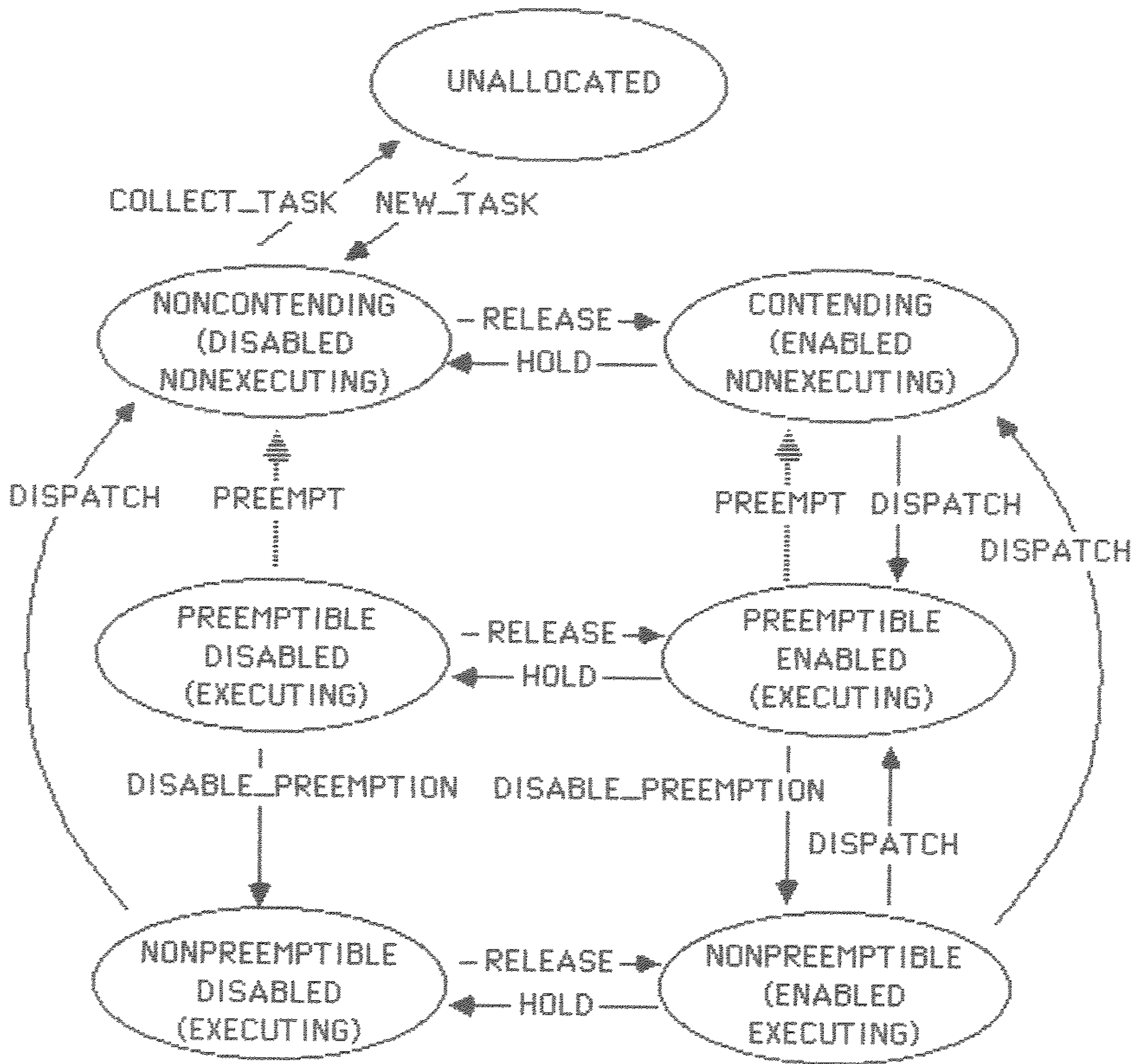

Figure 3: Lace Task States

Otherwise, A will wait until it is released. In either case, task A may be delayed in returning from DISPATCH if there is a higher priority task contending for the same processor.

Although technically they are not Lace tasks, interrupt handlers can also be responsible for state transitions, as indicated by the shaded arcs in Figure 3. Interrupt handlers can always interrupt Lace tasks. If, when an interrupt handler is invoked, the currently executing (assigned) task on the interrupted processor is in a preemptible state the handler can cause the task to be preempted, by calling the Lace entry PREEMPT. On the other hand, if the current task is not preemptible, the interrupt handler must restore the state and return control to that task. Thus interrupts are transparent to a nonpreemptible task, although they may degrade the rate of the task's progress.

The operation PREEMPT is only intended to be used by interrupt handlers. It has the effect of saving the state of the preempted task and invoking DISPATCH. Lace also provides operations for determining a task's identity (SELF), altering a task's priority (SET_PRIORITY), and for forcing a task to call a specified procedure (FORCE_CALL). This latter feature is used primarily for handling exceptional conditions. Its effect is also asynchronous.

Note that the operation DISPATCH should never be performed by a task in a preemptible state. Note also that there is a preemptible disabled state. If a task in this state is preempted it will not resume exection until explicity released by another task. This state is included because it might be of use in constructing interrupt-driven task schedulers, which could hold a task and preempt it, and later release it again.

Examples of the use of Lace operations to implement higher level task intercommunication operations can be found in [1].

3 Lace External Interfaces

The interface to Lace is a set of procedures that are invoked by user programs that wish to either execute as Lace tasks or manipulate other Lace tasks. Although not necessarily implemented as procedures, Lace entries have the property that, barring exceptional conditions such as abortion, they will eventually return to the user program at the point following the location of the call.

The interfaces between Lace and user code can be categorized into the following three views corresponding to the three major entities in Figure 1:

1. Lace as seen by the compiled user code;
2. the compiled code, as seen by Lace;
3. Lace, as seen by an interrupt handler.

We will consider each of these in turn.

3.1 The Compiler's View

To the compiler, a task is simply an integer of type TASK_ID. A task may also have an associated priority. Tasks are created by NEW_TASK, where the initial state of a task is given by specifying the initial machine state (register values - in particular the program counter and probably a stack pointer) for the processor when executing the task. The interface presented by Lace to a compiled user program is described ideally by the following packages.

The type TASK_ID is used by Lace and Corset, both internally and as part of their external interfaces. Because the package below is also part of the Lace and Corset internal interfaces, the full type declaration is given. However, for external interface purposes, all that should be assumed about this type is that it is an integer type of a certain size; that is, the range is not part of the interface.

```
with SYSTM; use SYSTM;
package TASK_IDS is
  type TASK_ID is new DOUBLE_WORD_BIT_POSITION;
  NULL_TASK: constant TASK_ID:= TASK_ID'first;
end TASK_IDS;

with TASK_IDS; use TASK_IDS;
with SYSTM; use SYSTM;
package LACE is
  MAX_PRIORITY, MIN_PRIORITY: INTEGER;
```

All procedures with arguments of type TASK_ID require that the id be "alive". That is, it must have been returned by NEW_TASK and COLLECT_ID must not have been called for it since then.

3.1.1 Preemptible Procedures

The following procedures are required to be implemented so that they do not require waiting for any locks, and so may be called from interrupt handlers, as well as from tasks.

```
function  SELF return TASK_ID;
procedure DISABLE_PREEMPTION;
procedure HOLD(T: TASK_ID);
procedure RELEASE(T: TASK_ID);
procedure COLLECT_ID(T: TASK_ID);
procedure SET_PRIORITY(T: TASK_ID; P: INTEGER);
```

Function SELF returns the id of the task which calls it, except if called from a hardware interrupt handler, in which case it returns the ID of the task which was executing when the interrupt occurred.

Procedure DISABLE_PREEMPTION makes the resident task not preemptible by the dispatcher. The effect is undone (only) by procedure DISPATCH.

Procedure HOLD disables dispatching of the task T. If it is running, the task will continue until it next calls dispatch, or is preempted by an interrupt.

Procedure RELEASE enables dispatching of task T. If T is already enabled, there is no change. How soon T gets to run depends on its priority and the activities of higher priority tasks.

Procedure COLLECT_ID deallocates the task ID T. It assumes that T is not executing or contending for a processor (i.e. it is the caller's responsibility to insure this).

Procedure SET_PRIORITY sets the dispatching priority of T to P, if P is a priority value supported by the Lace implementation. Otherwise, the priority is set to the nearest supported value. A task with a numerically larger priority number always has preference for dispatching.

3.1.2 Nonpreemptible Procedures

Because the implementation of the following procedures is anticipated to involve waiting for and holding locks, we require that before they are called the processor be made nonpreemptible. Failure to follow this rule will lead to system failure. They should never be called from an interrupt handler.

```
procedure DISPATCH;
procedure FORCE_CALL(T: TASK_ID; PROC, PARAM: ADDRESS);
function  NEW_TASK(STATE: FULL_STATE) return TASK_ID;
```

Procedure DISPATCH is called to choose the next task to execute on the processor on which it is called, and transfers control to this task (which may be the calling task, if it is enabled). If there is no task eligible for execution, the dispatcher idles. Though it may switch to another task, from the point of view of the caller this behaves like any other procedure. That is, it eventually returns to the instruction following the call (barring exceptions). Calling the dispatcher is the normal means of return from service routines of an RSE built on Lace.

Note that DISPATCH is the only way to undo the effect of DISABLE_PREEMPTION. This is because interrupts that arrive during nonpreemptible period may cause the release of tasks which should be executed as soon as the processor becomes preemptible.

Procedure FORCE_CALL forces task T to call a procedure at address PROC, with parameter value PARAM. This procedure can be used to raise exceptions in tasks and to abort them, by forcing them to execute appropriate procedures. The effect is asynchronous, and does not take effect until T is next dispatched. If FORCE_CALL is called again for the same task before a pending forced call has taken effect, the later call will override. The practice of overriding pending forced calls should be avoided, because how soon a call will take effect cannot in general be predicted. (Nested calls may also cause space problems, as noted below.)

Care must be taken that every task has enough excess work space to tolerate any forced calls. Of course this is already true of other RSE calls, but is especially critical here because of the interaction with exception recovery. If STORAGE_ERROR is raised and if exception recovery is through such a forced call, if the exception recovery procedure uses storage, cycling could result. The obvious solution is to write all such routines to use only statically allocated global data and registers, or at least to use a predictable bounded amount of stack space.

Function NEW_TASK allocates a new TASK_ID and returns it. It returns NULL_TASK iff it cannot allocate a new task id. The new task starts out disabled, and its state is initialized to S. The priority of a new task is MIN_PRIORITY. Note that in order for the new task to begin execution it must later be enabled (via RELEASE).

3.1.3 Interrupt Handler Support

The procedures below are provided to support writing of hardware interrupt handlers. An illustration of how such a handler would be structured is given by procedure HANDLER, in Section 3.3.

```
  function  DISPATCHING_PRIORITY(T: TASK_ID) return INTEGER;
  function  PREEMPTION_OK return BOOLEAN;
  procedure PREEMPT(S: FULL_STATE);
end LACE;
```

Function DISPATCHING_PRIORITY returns the current dispatching priority of task T.

Function PREEMPTION_OK returns TRUE iff the current task is preemptible.

Procedure PREEMPT preempts the current task, and calls the dispatcher. This should never be called unless PREEMPTION_OK returns TRUE, otherwise chaos will ensue. It should only be called from an interrupt handler, since it presumes that interrupts are already disabled, and does not return to the point of call. It should only be called at the conclusion of the handler (at the point of return) and assumes the handler has saved the state of the task T to be preempted in S. The processor will be preempted from the current task and the dispatcher will be invoked to determine which task executes next.

3.2 Lace's View

In order to implement Lace functions such as DISPATCH and FORCE_CALL, Lace needs to know how the compiler used to create user programs has implemented procedure calls. Such information is required to determine which registers must be saved in the event of a task switch and how to simulate a task calling a user-defined procedure. The following package encapsulates the dependencies of the procedure calling protocol. The requirement that this package imposes on the compiler is that there must be a way to implement the procedures it contains. In particular it must me possible to implement FAKE_CALL in such a way that no matter when a task may be preempted a fake call can be forced on it. This constrains the implementation of exceptions and procedure calls.

```
with SYSTM; use SYSTM;
package PROCEDURES is
   procedure CALL_PROCEDURE(PROC, PARAM: ADDRESS);
   procedure FAKE_CALL(PROC,PARAM,RET: ADDRESS);
   procedure SAVE_BASIC_STATE(STATE: out FULL_STATE);
   procedure RESTORE_BASIC_STATE(STATE: in FULL_STATE);
end PROCEDURES;
```

Procedure CALL_PROCEDURE calls the procedure with entry address PROC and one parameter, located at address PARAM. The semantics of the call are the same as those for an ordinary Ada procedure call.

Procedure FAKE_CALL is like CALL_PROCEDURE, except that RET is the return address (and point of exception propagation). We assume that procedure calls and activation records are implemented in such a way that a task may be interrupted at any time and forced to execute a faked procedure call.

Procedure SAVE_BASIC_STATE saves any registers that are expected to be preserved by an ordinary Ada procedure. We assume this is much less than the full set of registers. Procedure RESTORE_BASIC_STATE restores the values of the registers saved by SAVE_BASIC_STATE.

3.3 An Interrupt Handler's View

Facilities for interrupt handlers to synchronize with Lace tasks are provided by the functions DISPATCHING_-PRIORITY and PREEMPTION_OK, and the procedure PREEMPT. In addition, interrupt handlers may also invoke those other Lace entries that do not require waiting for locks. It is assumed that interrupt handlers execute with interrupts disabled.

Interrupt handlers for the MIL-STD-1750A are expected to take the form of the template "procedure" HANDLER below, which uses operations exported by Lace to preempt the task which it interrupts and to release other tasks. Note that this is not truly an Ada procedure, and is presented as one for exposition purposes only. We presume that in practice the would be generated by an Ada compiler for an accept body, or written in assembly language. As interrupt handlers are likely to be hand coded in machine language, an optimization to the following code is to save and restore only those registers actually altered by the individual handler.

Note that the declarations imported from packages CORSET and LOW_LEVEL_TASKING are also provided directly by the Lace interface.

```
with SYSTM; use SYSTM;
with CORSET; use CORSET;
with LOW_LEVEL_TASKING; use LOW_LEVEL_TASKING;
procedure HANDLER is
   T: TASK_ID;
   C_STATE: CORE_STATE;
   F_STATE: FULL_STATE;
   procedure RESTORE_FULL_STATE(S: FULL_STATE) is
   begin null; -- a dummy for a  machine-level operation.
   end RESTORE_FULL_STATE;
begin
```

At this point code would be included to do the following, if it has not already been done by the hardware before execution reaches this point. It is likely to be hand-coded and specific to the particular interrupt.
1.) Save CORE_STATE of SELF in parameter C_STATE.
2.) Save any registers used internally in this procedure in F_STATE.
3.) Do any interrupt-specific work.
4.) Determine which task(s)are to be enabled as a result of this interrupt. (The remaining code assumes either 0 or 1 tasks are enabled, and that the task to be enabled is T.)

```
      if T/=NULL_TASK
      then RELEASE(T);
           -- See if a local preemption is possible and worthwhile.
           if PREEMPTION_OK and then
              DISPATCHING_PRIORITY(T) > DISPATCHING_PRIORITY(INTERRUPTED_TASK)
           then PREEMPT(F_STATE);
                -- Will not return
           end if;
      end if;
      RESTORE_FULL_STATE(F_STATE);
end HANDLER;
```

Note that in theory the call to RESTORE_FULL_STATE above would only restore the CORE_STATE and those other registers. that were saved in (2) above, hence the effect of this statement may well vary from interrupt handler to interrupt handler. In any case, the desired result is return from interrupt to the interrupted task.

4 Lace/I - a Multiprocessor Implementation

This section describes a prototype implementation of Lace for a multiprocessor configuration of MIL-STD 1750A processors with shared memory. It is included here primarily to help explain, operationally, the intended functions of the executive and to demonstrate that it can be implemented efficiently for a multiprocessor shared-memory system. This is only one of many possible implementations of the Lace interface, which is designed to be adaptable to a variety of hardware configurations and dispatching policies (within priority classes).

The reader is warned that though the description is expressed in Ada it is not intended to be executable code. To produce an executable implementation some portions will need to be coded in assembly language or as machine-code insertions. Especially, some operations which can be performed atomically as machine operations would not be correct if implemented as shown in Ada (because they would not be atomic). An attempt has been made to isolate most of these operations into separate packages, but there remain some dependencies that may not be obvious. In particular, we assume that assignment to an element of a bit-vector can be done atomically, wherever this occurs.

Internally, Lace/I is composed of the following packages:

1. TASK_IDS (described above) exports type TASK_ID and constant NULL_TASK as well as other related types.
2. SYSTEM is the standard Ada predefined package, with some implementation-specific extensions as permitted by the Ada standard [3].
3. MISCELLANY contains a number of miscellaneous declarations that are not central to the purpose of this document, but are required to make it consistent and complete enough enough to be compiled. Most are operations that are expected to be implemented via machine code and which cannot be coded accurately in machine-independent Ada.

4. LACE contains the actual data structures and code of the Lace implementation.

4.1 System

This version of the standard package SYSTEM has been augmented (as permitted by the Ada standard) to include additional machine- and configuration-dependent declarations. Note that we call this package SYSTM here to permit syntax-checking the code included in this report with a non-cooperative compiler. The declarations of ADDRESS, and NULL_ADDRESS are commented out and replaced by dummies, using unchecked conversion, for the same reason.

```
with SYSTEM; with UNCHECKED_CONVERSION;
-- package SYSTEM is
package SYSTM is
  type NAME is (MIL_STD_1750A);
  SYSTEM_NAME: constant NAME:= MIL_STD_1750A;
  ----------MACHINE TYPES----------
  type BYTE is range -16#80# ..16#7F#;
       for BYTE'size use 8;
  type WORD is range -16#8000# .. 16#7FFF#;
       for WORD'size use 16;
  type DOUBLE_WORD is range -16#80000000# .. 16#7FFFFFFF#;
       for DOUBLE_WORD'size use 32;
  STORAGE_UNIT: constant:= 16;
  subtype BYTE_BIT_POSITION is BYTE range 0..7;
  subtype WORD_BIT_POSITION is BYTE range 0..15;
  subtype DOUBLE_WORD_BIT_POSITION is BYTE range 0..31;
  ----------ADDRESSING----------
--subtype ADDRESS is WORD;
--NULL_ADDRESS: constant ADDRESS:= 0;
  subtype ADDRESS is SYSTEM.ADDRESS;
  function COERCE is new UNCHECKED_CONVERSION(INTEGER,ADDRESS);
  NULL_ADDRESS: constant ADDRESS:= COERCE(0);
  MEMORY_SIZE: constant:= 16#10000#;
  subtype EXTENDED_ADDRESS is DOUBLE_WORD range 0..16#FFFFF#;
  EXTENDED_MEMORY_SIZE: constant:= 16#FFFFF#;
  ----------CONFIGURATION----------
  MAX_PROCESSORS: constant:= 1;
  type PROCESSOR_ID is range 0..MAX_PROCESSORS;
  NULL_PROCESSOR: constant PROCESSOR_ID:= 0;
  subtype PROCESSORS is PROCESSOR_ID range 1..MAX_PROCESSORS;
  function THIS_PROCESSOR return PROCESSORS;
  ----------REGISTER STRUCTURE----------
  type CORE_STATE is record IC,MK,SW: WORD; end record;
  type FULL_STATE is record CORE: CORE_STATE;
                            R0,R1,R2,R3,R4,R5,R6,R7,
                            R8,R9,R10,R12,R13,R14,R15: WORD;
                     end record;
  ----------SYSTEM-DEPENDENT NAMED NUMBERS----------
  MIN_INT: constant:= -16#80000000#; -- = DOUBLE_WORD'first
  MAX_INT: constant:= 16#7FFFFFFF#;  -- = DOUBLE_WORD'last
  MAX_DIGITS: constant:= 12;
  MAX_MANTISSA: constant:= 31;
  FINE_DELTA: constant:= 16#0.00000002#; -- = 2**-31.
  TICK: constant:= 0.0001;
  ----------OTHER SYSTEM-DEPENDENT DECLARATIONS----------
  MAX_PRIORITY, MIN_PRIORITY: INTEGER;
  subtype PRIORITY is INTEGER range MIN_PRIORITY..MAX_PRIORITY;
end SYSTM;
```

We assume 16-bit addressing, so that if extended memory is present, it is accessed by an application through

read and write operations made available via a package interface. We also assume extended floating point, with 40-bit mantissa, is supported.

Values of the type PROCESSORS represent the actual processors in the configuration. The constant NULL_PROCESSOR does not represent any actual processor. The function THIS_PROCESSOR returns the ID of the processor on which it is called. It would most likely be implemented as a constant in the processor's private memory, or as an I/O instruction. Type CORE_STATE represents the set of registers saved by the interrupt mechanism.

Since dynamically adjusting priorities for rendezvous is costly, we would prefer not to allow specification of Ada priorities. By making the range of priorities nonstatic, we defer this decision to the runtime support implementation, which may choose to limit this range to a single value, effectively ignoring priorities. The variables MAX_PRIORITY and MIN_PRIORITY are therefore set by the runtime support implementation.

4.2 Miscellany

Package MISCELLANY contains a number of miscellaneous declarations that are not central to the purpose of this document, but are required to make it consistent and complete enough to be compiled. Procedure ASSERT, which checks the validity of an assertion, is called in some places to document an important precondition. If implemented, it presumably would halt the program and report a fault. The other exports are operations which we expect would be implemented via machine code, and which cannot be coded accurately in machine-independent Ada.

```
with SYSTM; use SYSTM;
package MISCELLANY is
  procedure ASSERT(A: BOOLEAN);
  procedure ENABLE_INTERRUPTS;
  procedure DISABLE_INTERRUPTS;
  procedure RESTORE_FULL_STATE(S: FULL_STATE);
  procedure RESTORE_ALL_BUT_IC(S: FULL_STATE);
  generic
    type POSITION is range <>;
    type BIT_VECTOR is array (POSITION) of BOOLEAN;
  function CHOOSE_BIT(V: BIT_VECTOR) return POSITION;
  AVAILABLE: constant BOOLEAN:= FALSE;
  generic
    type POSITION is range <>;
    type BIT_VECTOR is array (POSITION) of BOOLEAN;
  procedure AWAIT_BIT(V: in out BIT_VECTOR; I: POSITION);
  generic
    type POSITION is range <>;
    type BIT_VECTOR is array (POSITION) of BOOLEAN;
  procedure TEST_AND_SET_BIT(V: in out BIT_VECTOR;
                             I: POSITION;
                             B: out BOOLEAN);
  generic
    type POSITION is range <>;
    type WORD is range <>;
  procedure SET_BIT(W: in out WORD;
                    I: POSITION;
                    B: BOOLEAN);
  procedure GOTO_VARIABLE(A: ADDRESS);
end MISCELLANY;
```

Procedure RESTORE_FULL_STATE restores the full (register) state of the machine to S, which implies a goto the given IC (instruction counter) value. It does not return. In contrast, procedure RESTORE_ALL_-BUT_IC restores all of the full state to S, except for the IC, so that it returns normally.

A number of operations use words as bit-vectors. Procedure CHOOSE_BIT returns the position of the first nonzero bit in V. This is efficiently implementable on the 1750A via the FLOAT operation, provided POSITION is a subrange of 1..31.

Other operations use bits of a bit vector as semaphores or signals. When a bit vector is used for such a purpose we usually call it a "right". The constant value AVAILABLE is used to denote a right that is available.

Procedure AWAIT_BIT is a busy-wait test-and-set semaphore operation on B(I). Procedure TEST_AND_SET performs B:=V(I); V(I):=TRUE as one atomic operation, with memory access locked during the operation. Procedure SET_BIT sets the Ith bit of W to B in one atomic operation. In addition to these operations, we assume that assignment to a single bit of an array of boolean can be done in one atomic operation, even if the array is packed as a bit-vector.

Procedure GOTO_VARIABLE sets the IC to A, effecting a transfer to the instruction at that address.

4.3 Lace

This design for a Lace implementation restricts the range of permissible task IDs, so that it can use bit-vector operations to speed processing. The range of permissible priorities is also restricted, for the same reason. Test-and-set operations on elements of bit vectors are used for synchronization, and to claim "rights" to shared resources.

```
with TASK_IDS; use TASK_IDS;
with MISCELLANY; use MISCELLANY;
with SYSTM; use SYSTM;
with PROCEDURES; use PROCEDURES;
with UNCHECKED_CONVERSION;
package body LACE is
  type TASK_BIT_VECTOR is array (TASK_ID) of BOOLEAN;
--     for TASK_BIT_VECTOR'size use 32;
  type TASK_TASK_VECTOR is array (TASK_ID) of TASK_ID;
  type TASK_INTEGER_VECTOR is array (TASK_ID) of INTEGER;
  procedure TEST_AND_SET is new TEST_AND_SET_BIT(TASK_ID,TASK_BIT_VECTOR);

  type TASK_PRIORITIES is range 0..31;
  NULL_PRIORITY: constant TASK_PRIORITIES:= 0;
  -- is not a priority to which a task can be assigned.

  type PROCESSOR_BIT_VECTOR is array (PROCESSORS) of BOOLEAN;
  procedure TEST_AND_SET is new TEST_AND_SET_BIT(PROCESSORS,PROCESSOR_BIT_VECTOR);

  PREEMPTIBLE,
  SKIP_DISPATCH: PROCESSOR_BIT_VECTOR:= (others =>FALSE);
  RIGHT_TO_DISPATCH: TASK_BIT_VECTOR:= (others=> AVAILABLE);
  ASSIGNED,
  ENABLED: TASK_BIT_VECTOR:= (others=> FALSE);
  -- Task state variables.
```

PREEMPTIBLE(P) is true if and only if processor P can safely be preempted from the task it is currently executing. It is unset (to false) by the resident task, buy calling DISABLE_PREEMPTION, but should only be set (to true) by the dispatcher executing on processor P when it is called (voluntarily) by the resident task.

SKIP_DISPATCH(P) is unset (to false) to force full execution of the dispatcher algorithm. It is used as part of an optimistic heuristic to avoid performing the entire dispatching algorithm. It is set after every dispatch and is unset only when a change in a task state may have made it necessary to choose a dif and only iferent task (on some processor).

RIGHT_TO_DISPATCH is used to ensure that the same task is not dispatched concurrently on more than one processor. It is also needed to insure tasks are dispatched in priority order. Only RIGHT_TO_-DISPATCH(1) is used; it is a bit vector only because we have only declared TEST_AND_SET for bit vectors. This right is used as a lock around the loop that selects the next task to execute. The value is true (not AVAILABLE) if and only if another processor is executing the selection loop.

ASSIGNED(T) is true if and only if T is assigned to a processor. It is set by the dispatcher (via TEST_-AND_SET) and is reset by the dispatcher and handlers (in the case of a preemption).

ENABLED(T) is true if and only if T is to be considered for dispatching, if not already assigned. It can be set and reset by anyone, but resetting is risky programming unless done to self.

```
  SAVED_STATE: array (TASK_ID) of FULL_STATE;
```

SAVED_STATE is the storage area for states of disabled and preempted tasks. SAVED_STATE(T) is the contents of the registers of T if T is not executing.

```
type REQUEST is record PROC, PARAM: ADDRESS; end record;

FORCED_CALL_REQUEST: array (TASK_ID) of REQUEST:=
                     (others =>(NULL_ADDRESS, NULL_ADDRESS));
FORCED_CALL_BUFFER: array (PROCESSORS) of REQUEST;
RIGHT_TO_FORCE_CALL: TASK_BIT_VECTOR:= (others=> AVAILABLE);
```

FORCED_CALL_REQUEST(T) is the current pending forced call request for task T, if any. We permit only one forced call to be pending for a given task at any time. A forced call request specifies the address PROC of a procedure to be called and the address PARAM of a parameter.

FORCED_CALL_BUFFER holds a copy of FORCED_CALL_REQUEST during the time the call is being forced. It is needed to preserve a request from being overwritten once we have decided to dispatch the forced call, during the time between the release of RIGHT_TO_FORCE_CALL and the actual forced call.

RIGHT_TO_FORCE_CALL(T) is a lock used to insure that FORCED_CALL_REQUEST(T) and FORCED_CALL_BUFFER(T) both remain consistent, since these each consist of two words and so cannot be updated atomically. The value is true (not AVAILABLE) if and only if data in FORCED_CALL_REQUEST(T) is busy.

```
PRIORITY: array (TASK_ID) of TASK_PRIORITIES:=
                             (others =>TASK_PRIORITIES'last);
HAS_PRIORITY: array (TASK_PRIORITIES) of TASK_BIT_VECTOR:=
              (TASK_PRIORITIES'first..TASK_PRIORITIES'last-1=>
                 (TASK_ID'first..TASK_ID'last=> FALSE),
                 TASK_PRIORITIES'last..TASK_PRIORITIES'last=>
                 (TASK_ID'first..TASK_ID'first=> FALSE,
                  TASK_ID'first+1..TASK_ID'last=> TRUE));
type PRIORITY_BIT_VECTOR is array (TASK_PRIORITIES) of BOOLEAN;
CONSIDER_PRIORITY: PRIORITY_BIT_VECTOR:= (FALSE, others=> TRUE);
```

PRIORITY is the internal priority assigned by the dispatcher to a task. It need not correspond exactly to the Ada priority of the task, so long as the relative ordering of tasks is consistent with relative ordering of Ada priorities. For 1750A implementation reasons, these priorities are stored inverted, so that 1 is the "highest" priority.

Since task priorities need not be unique, one can view the priority scheme as a two dimensional structure with rows being priority levels and columns being tasks. Thus HAS_PRIORITY(P) is the row in this structure that tells which tasks have priority P.

Initially, all tasks are in the lowest priority level, as per the initial value of PRIORITY.

Operations on variables of type PRIORITY_BIT_VECTOR are assumed to be bit operations (as for type TASK_BIT_VECTOR.)

CONSIDER_PRIORITY is used as a heuristic to speed up the task selection phase of DISPATCH. If bit i is true then we assume there are tasks at priority level i which need dispatching attention. If bit i is false then there are no priority i tasks ready for dispatching.

```
ACTIVE_TASK:  array (PROCESSORS) of TASK_ID:= (others =>NULL_TASK);
ALLOCATED: TASK_BIT_VECTOR:= (others=> FALSE);
```

ACTIVE_TASK(P) tells which task is executing (resident) on processor P. If the active task is the null task, it means procedure DISPATCH is executing and there is no resident task. It is incorrect to query ACTIVE_TASK for any processor other than THIS_PROCESSOR. This array is used only to allow the current executing task to determine its TASK_ID, and probably would be replaced in and actual implementation by a simple variable replicated in the private memory of each processor.

ALLOCATED(T) is true if and only if task ID T is allocated to a live task (in use).

```
type STATUSES is new BYTE range 0..7;
   RUNNING: constant:= 0;
   PREEMPTED: constant:= 1;
   FORCED_CALL: constant:= 2;
NORMAL_RUNNING:     constant STATUSES:= 2#001#;
NORMAL_PREEMPTED:   constant STATUSES:= 2#010#;
STATUS: array (TASK_ID) of STATUSES:= (others=> NORMAL_PREEMPTED);
```

STATUS(T) is a bit vector which expresses the state of task T. RUNNING is the position of the status bit telling whether T is running. PREEMPTED is the position of the status bit telling whether T was preempted;

if this bit is 0 the task was not preempted, and may be running or voluntarily switched out via a call to the dispatcher. FORCED_CALL is the position of status bit telling whether T has a pending forced call.

NORMAL_RUNNING is that status of a task that is running with no pending forced call. NORMAL_PREEMPTED is the status of a preempted task that is normal in the sense that there is no FORCED_CALL pending.

The following are local procedures used within the Lace implementation.

```
procedure AWAIT is new MISCELLANY.AWAIT_BIT(TASK_ID,TASK_BIT_VECTOR);
function  CHOOSE is new
          CHOOSE_BIT(TASK_PRIORITIES,PRIORITY_BIT_VECTOR);
function  CHOOSE is new CHOOSE_BIT(TASK_ID,TASK_BIT_VECTOR);
function  COERCE is new UNCHECKED_CONVERSION(WORD,ADDRESS);
procedure DO_FC(PROC, PARAM: ADDRESS);
procedure DO_FC_PREEMPTIVE;
procedure MAKE_PREEMPTIBLE_AND_GOTO(A: ADDRESS);
procedure SET is new
          SET_BIT(BYTE_BIT_POSITION,STATUSES);
```

The following procedures are exported from Lace.

```
procedure DISABLE_PREEMPTION is
begin PREEMPTIBLE(THIS_PROCESSOR):= FALSE;
end DISABLE_PREEMPTION;

procedure HOLD(T: TASK_ID) is
begin ENABLED(T):= FALSE;
      if T=SELF
      then SKIP_DISPATCH(THIS_PROCESSOR):= FALSE;
      else SKIP_DISPATCH:= (others=> FALSE);
      -- Without locking (i.e. possibly waiting here and in other
      -- more time-critical places) we cannot accurately
      -- determine where T is executing, hence we warn every
      -- processor to perform the full dispatch algorithm.
      end if;
end HOLD;

procedure RELEASE(T: TASK_ID) is
begin ENABLED(T):= TRUE;
      SKIP_DISPATCH:= (others=> FALSE);
      -- A new task is eligible for dispatching. By forcing everybody to
      -- perform a full dispatch, we ensure T executes ASAP.
      CONSIDER_PRIORITY(PRIORITY(T)):= TRUE;
end RELEASE;
```

Procedure DISPATCH (below) assumes the processor is not preemptible when it is called. When it returns the processor is preemptible.

This implementation assumes that any processor can execute any task. It makes use of the lock RIGHT_TO_DISPATCH(1) to insure that at most one processor is executing in the task selection loop at any one time. This version idles inside the selection loop, with the lock held, when there is no task to dispatch. This is safe, given the assumption that any processor can execute any task. If this assumption is not true, idling would need to be done without holding the lock, by testing SKIP_DISPATCH(THIS_PROCESSOR).

```
procedure DISPATCH is
   CHOICE: TASK_ID;
   T: TASK_ID;
   P: TASK_PRIORITIES;
   CAN_SKIP: BOOLEAN;
   CONTENDING: TASK_BIT_VECTOR;
   -- Will be all and only those tasks eligible for dispatching.
begin assert(not PREEMPTIBLE(THIS_PROCESSOR));
      TEST_AND_SET(SKIP_DISPATCH,THIS_PROCESSOR,CAN_SKIP);
      if CAN_SKIP
```

```
then PREEMPTIBLE(THIS_PROCESSOR):= TRUE; return; end if;
AWAIT(RIGHT_TO_DISPATCH,1);
```

The highest priority task is chosen via bit-vector operations, first by choosing the highest priority level, P, where there may be a task contending, and then by choosing a contending task at that level. The heuristic for considering priority classes is: unless you are certain there are no more tasks remaining at priority P, assume there are more. If no tasks are found at level P, the next lower level is tried. If P=0 the dispatcher is idling, waiting for some task to be released.

```
loop CONTENDING:= not ASSIGNED;
     CONTENDING(SELF):= TRUE;
     CONTENDING:= ENABLED and CONTENDING;
     P:=CHOOSE(CONSIDER_PRIORITY);
     if P/=0
     then CONSIDER_PRIORITY(P):= FALSE;
          CHOICE:= CHOOSE(CONTENDING and HAS_PRIORITY(P));
          if CHOICE/=NULL_TASK
          then CONSIDER_PRIORITY(P):= TRUE;
               ASSIGNED(CHOICE):= TRUE;
               RIGHT_TO_DISPATCH(1):= AVAILABLE;
               -- We have a task, release the lock.
               exit;
          end if;
     end if;
end loop;
```

Execution of the chosen task, CHOICE, is now resumed, based on its STATUS. A case-structure is suggested here, to speed up branching.

```
case STATUS(CHOICE) is
when 2#000#=> -- CHOICE had voluntarily called DISPATCH.
              if SELF/=NULL_TASK
              -- Put caller of DISPATCH to sleep.
              then SET(STATUS(SELF), RUNNING, FALSE);
                   SAVE_BASIC_STATE(SAVED_STATE(SELF));
                   ASSIGNED(SELF):= FALSE;
              end if;
              ACTIVE_TASK(THIS_PROCESSOR):= CHOICE;
              -- now SELF=CHOICE.
              SET(STATUS(SELF), RUNNING, TRUE);
              RESTORE_BASIC_STATE(SAVED_STATE(SELF));
              PREEMPTIBLE(THIS_PROCESSOR):= TRUE;
              return;
when 2#001#=> -- task that called DISPATCH is selected for execution.
              -- (i.e. CHOICE & SELF are the same.)
              PREEMPTIBLE(THIS_PROCESSOR):= TRUE;
              return;
when 2#010#=> -- CHOICE was preempted by an interrupt.
              if SELF/=NULL_TASK
              -- Put caller of DISPATCH to sleep.
              then SET(STATUS(SELF), RUNNING, FALSE);
                   SAVE_BASIC_STATE(SAVED_STATE(SELF));
                   ASSIGNED(SELF):= FALSE;
              end if;
              ACTIVE_TASK(THIS_PROCESSOR):= CHOICE;
              -- now SELF=CHOICE.
              SET(STATUS(SELF), PREEMPTED, FALSE);
              SET(STATUS(SELF), RUNNING, TRUE);
              -- Set bits separately to avoid erasing an incoming
              -- forced call.
              RESTORE_ALL_BUT_IC(SAVED_STATE(SELF));
              MAKE_PREEMPTIBLE_AND_GOTO(
```

```
                  COERCE(SAVED_STATE(SELF).CORE.IC));
                -- will not return.
                raise PROGRAM_ERROR; -- should never get here.
   when 2#011#=> -- is impossible, since a preempted task can't be running.
                raise PROGRAM_ERROR;
   when 2#100#=> -- CHOICE voluntarily called DISPATCH and while it
                -- was inactive someone has forced a call on it.
                if SELF/=NULL_TASK
                -- Put caller of DISPATCH to sleep.
                then SET(STATUS(SELF), RUNNING, FALSE);
                     SAVE_BASIC_STATE(SAVED_STATE(SELF));
                     ASSIGNED(SELF):= FALSE;
                end if;
                ACTIVE_TASK(THIS_PROCESSOR):= CHOICE;
                -- now SELF=CHOICE.
                AWAIT(RIGHT_TO_FORCE_CALL,SELF);
                STATUS(SELF):= NORMAL_RUNNING;
                -- is inside the locked region, to avoid
                -- missing further forced calls.
                FORCED_CALL_BUFFER(THIS_PROCESSOR):=
                               FORCED_CALL_REQUEST(SELF);
                -- Copy data to avoid race with incoming forced call.
                RIGHT_TO_FORCE_CALL(SELF):= AVAILABLE;
                RESTORE_BASIC_STATE(SAVED_STATE(SELF));
                DO_FC(
                   FORCED_CALL_BUFFER(THIS_PROCESSOR).PROC,
                   FORCED_CALL_BUFFER(THIS_PROCESSOR).PARAM);
                PREEMPTIBLE(THIS_PROCESSOR):= TRUE;
                return;
   when 2#110#=> -- CHOICE was preempted (by an interrupt) and while it
                -- was preempted someone has forced a call on it.
                if SELF/=NULL_TASK
                -- Put caller of DISPATCH to sleep.
                then SET(STATUS(SELF), RUNNING, FALSE);
                     SAVE_BASIC_STATE(SAVED_STATE(SELF));
                     ASSIGNED(SELF):= FALSE;
                end if;
                ACTIVE_TASK(THIS_PROCESSOR):= CHOICE;
                -- now SELF=CHOICE.
                AWAIT(RIGHT_TO_FORCE_CALL,SELF);
                STATUS(SELF):= NORMAL_RUNNING;
                FORCED_CALL_BUFFER(THIS_PROCESSOR):=
                               FORCED_CALL_REQUEST(SELF);
                -- Copy data to avoid race with incoming forced call.
                RIGHT_TO_FORCE_CALL(SELF):= AVAILABLE;
                RESTORE_BASIC_STATE(SAVED_STATE(SELF));
                FAKE_CALL(DO_FC_PREEMPTIVE'address,NULL_ADDRESS,
                          COERCE(INTEGER(SAVED_STATE(SELF).CORE.IC)));
                raise PROGRAM_ERROR; -- should never get here.
   when 2#101#=> -- Task that called DISPATCH is selected for execution,
                -- but at some point since its last dispatch
                -- someone has forced a call on it
                AWAIT(RIGHT_TO_FORCE_CALL,SELF);
                STATUS(SELF):= NORMAL_RUNNING;
                FORCED_CALL_BUFFER(THIS_PROCESSOR):=
                               FORCED_CALL_REQUEST(SELF);
                -- Copy data to avoid race with incoming forced call.
                RIGHT_TO_FORCE_CALL(SELF):= AVAILABLE;
                DO_FC(
```

```
                        FORCED_CALL_BUFFER(THIS_PROCESSOR).PROC,
                        FORCED_CALL_BUFFER(THIS_PROCESSOR).PARAM);
                     PREEMPTIBLE(THIS_PROCESSOR):= TRUE;
                     return;
      when 2#111#=>  raise PROGRAM_ERROR;
                     -- A preempted task can't be running!
      end case;
end DISPATCH;
```

Procedure FORCE_CALL posts a request for a forced call. Locking is used to prevent interleaved updating of the fields PROC and PARAM. Since it cannot tell where any other task is executing, unless the forced call is to the current task this procedure uses SKIP_DISPATCH to warn all other processors that they need to check for a forced call on the next execution of DISPATCH.

```
procedure FORCE_CALL(T: TASK_ID; PROC, PARAM: ADDRESS) is
begin assert(not PREEMPTIBLE(THIS_PROCESSOR));
      AWAIT(RIGHT_TO_FORCE_CALL,T);
      FORCED_CALL_REQUEST(T):= (PROC, PARAM);
      SET(STATUS(T), FORCED_CALL, TRUE);
      RIGHT_TO_FORCE_CALL(T):= AVAILABLE;
      if T=SELF then SKIP_DISPATCH(THIS_PROCESSOR):= FALSE;
      else SKIP_DISPATCH:= (others=> FALSE);
      end if;
end FORCE_CALL;
```

Procedure COLLECT_ID presumes that the ID is not assigned and not enabled. For efficiency, however, an implementation would not attempt to check these assumptions.

```
procedure COLLECT_ID(T: TASK_ID) is
begin assert(not ASSIGNED(T) and not ENABLED(T));
      ALLOCATED(T):= FALSE;
end COLLECT_ID;

function NEW_TASK(STATE: FULL_STATE) return TASK_ID is
  CHOICE: TASK_ID;
  BUSY: BOOLEAN;
begin CHOICE:= NULL_TASK;
      -- Find the first unclaimed TASK_ID
      for I in TASK_ID'first+1 .. TASK_ID'last
      -- we reserve TASK_ID'first for the null task.
      loop TEST_AND_SET(ALLOCATED,I,BUSY);
           if not BUSY then CHOICE:= I; exit; end if;
      end loop;
      if CHOICE/=NULL_TASK
      then SAVED_STATE(CHOICE):= STATE; end if;
      HAS_PRIORITY(PRIORITY(CHOICE))(CHOICE):= FALSE;
      PRIORITY(CHOICE):= TASK_PRIORITIES'last;
      HAS_PRIORITY(PRIORITY(CHOICE))(CHOICE):= TRUE;
      STATUS(CHOICE):= NORMAL_PREEMPTED;
      return CHOICE;
end NEW_TASK;

function SELF return TASK_ID is
begin return ACTIVE_TASK(THIS_PROCESSOR);
end SELF;
```

Procedure SET_PRIORITY assumes that the selection loop of the dispatcher is protected by a lock, so that only one processor may be looking at the priority tables at any time. It need not be executed without preemption, but to insure that the priority change is immediate the caller may wish to first disable preemption. Note that this procedure inverts the priority as specified by parameter P, in order to better fit the MIL-STD-1750A machine operations. After inversion, the 'highest' priority is 1 and the 'lowest' priority is 31.

```
procedure SET_PRIORITY(T: TASK_ID; P: INTEGER) is
  OLDP, NEWP: TASK_PRIORITIES;
begin OLDP:= PRIORITY(T);
      NEWP:= TASK_PRIORITIES'last+1-TASK_PRIORITIES(P);
      if NEWP>TASK_PRIORITIES'last
      then NEWP:=TASK_PRIORITIES'last;
      elsif NEWP<=TASK_PRIORITIES'first
      then NEWP:=TASK_PRIORITIES'first+1;
      end if;
      HAS_PRIORITY(NEWP)(T):= TRUE;
      PRIORITY(T):= NEWP;
      HAS_PRIORITY(OLDP)(T):= FALSE;
      CONSIDER_PRIORITY(NEWP):= TRUE;
      SKIP_DISPATCH:= (others=> FALSE);
end SET_PRIORITY;
```

The following operationsare exported to support interrupt handlers. Of these, procedure PREEMPT is peculiear. It assumes that interrupts are disabled and the processor is preemptible, and does not return in the sense of an ordinary procedure. Instead, it invokes DISPATCH to choose a task to resume execute.

```
function DISPATCHING_PRIORITY(T: TASK_ID) return INTEGER is
begin return INTEGER(TASK_PRIORITIES'last+1-PRIORITY(T));
      -- due to internal inversion of priorities.
end DISPATCHING_PRIORITY;

function PREEMPTION_OK return BOOLEAN is
begin return PREEMPTIBLE(THIS_PROCESSOR);
end PREEMPTION_OK;

procedure PREEMPT(S:  FULL_STATE) is
begin DISABLE_PREEMPTION;
      SAVED_STATE(SELF):= S;
      SET(STATUS(SELF), PREEMPTED, TRUE);
      SET(STATUS(SELF), RUNNING, FALSE);
      ASSIGNED(SELF):= FALSE;
      ACTIVE_TASK(THIS_PROCESSOR):= NULL_TASK;
      ENABLE_INTERRUPTS;
      DISPATCH;
end PREEMPT;
```

Note that in procedure PREEMPT the enabling of interrupts and the following call to DISPATCH must be inseparable. This is insured by the MIL-STD-1750A operation for enabling interrupts, which does not take effect until after the following instruction.

The following procedures are not exported, but are used internally within the Lace implementation.

Procedures DO_FC and DO_FC_PREEMPTIVE, below, call the procedure PROC with parameter at address PARAM. The extra level of indirect calling which this introduces buys the ability to force a call with interrupts enabled. The two versions are required to handle the difference between preempted tasks and tasks that are suspended during a call to DISPATCH. DO_FC_PREEMPTIVE assumes that the calling task, which is SELF at the time of call, was previously preempted and its state can be recovered from SAVED_-STATE(SELF). It assumes it is called via FAKE_CALL, for reasons explained further below.

Both procedures assume that the processor is not preemptible, and that all parameters are passed by value. Thus, parameters and state information can be safely copied into local storage before the processor is made preemptible again. Both procedures restore the state of the processor to preemptible.

```
procedure DO_FC(PROC, PARAM: ADDRESS) is
begin PREEMPTIBLE(THIS_PROCESSOR):= TRUE;
      CALL_PROCEDURE(PROC, PARAM);
end DO_FC;

procedure DO_FC_PREEMPTIVE is
  PROC:  ADDRESS:= FORCED_CALL_BUFFER(THIS_PROCESSOR).PROC;
  PARAM: ADDRESS:= FORCED_CALL_BUFFER(THIS_PROCESSOR).PARAM;
```

```
  S: FULL_STATE := SAVED_STATE(SELF);
begin PREEMPTIBLE(THIS_PROCESSOR):= TRUE;
      CALL_PROCEDURE(PROC, PARAM);
      RESTORE_FULL_STATE(S);
end DO_FC_PREEMPTIVE;
```

Note that in the procedure above restoring the full state effectively pops the activation record stack and resumes the preempted task, if the forced call to PROC returns normally. Otherwise, if an exception is propagated out, the state will not be restored. The exception will propagate further out, to the point where the current task was preempted, due to the action of FAKE_CALL, which should have arranged for this.

The following procedure provides a way to resume the execution of a preempted task without having to disable interrupts. At entry, the task's state is presumed to have been restored except for the IC, whose value us passed as a parameter. It presumes that the processor is not preemptible. There is no return; execution resumes at the parameter IC value, with preemption enabled.

```
procedure MAKE_PREEMPTIBLE_AND_GOTO(A: ADDRESS) is
begin PREEMPTIBLE(THIS_PROCESSOR):= TRUE;
      GOTO_VARIABLE(A);
end MAKE_PREEMPTIBLE_AND_GOTO;
```

The package body of Lace must initialize the variables which export the maximum and minimum priority values which it supports. Of course, these could be preinitialized, at load time.

```
begin MAX_PRIORITY:= INTEGER(TASK_PRIORITIES'last);
      MIN_PRIORITY:= INTEGER(TASK_PRIORITIES'first);
end LACE;
```

5 Summary

Lace is still under development, though there is presently no plan to change the interface. Additional operations have been considered, and may eventually be added if they can be implemented without harming the efficiency of the core Lace operations. For example, two such operations are KILL and MAKE_PREEMPTIBLE.

The effect of procedure KILL would be similar to that of HOLD, but permanent. That is, dispatching of the task T would be disabled and any subsequent calls to RELEASE would not reenable it. This capability is needed for Ada task abortion, and can be implemented more efficiently in Lace than at the next level above it. However, we have chosen not to do this, in order to keep Lace simple.

Work on a MIL-STD-1750A implementation of Lace is under way at the Boeing Aerospace Company in Kent, Washington. Comments (especially critical ones) on the Lace interface or Lace/I design are solicited, and should be directed to:

Professor T.P. Baker
Department of Computer Science
Florida State University
Tallahassee, FL 32306-4016

phone: (904) 644-5452
(if no answer) 644-2296

net mail: TBAKER@ADA20.ISI.EDU.ARPA

References

[1] T.P. Baker, "A Corset for Ada" TR 86-09-05, University of Washington Computer Science Department (September 1986).

[2] T.P. Baker, "An Improved Ada Runtime System Interface", TR 86-07-05, University of Washington Computer Science Department (July 1986).

[3] *Military Standard Ada Programming Language*, ANSI/MIL-STD-1815A, U.S. Department of Defense, Ada Joint Program Office (January 1983).