INTERRUPTS AND ADA

J R Hunt
Plessey Research and Technology
Romsey ENGLAND

Intel is a registered trademark of the Intel Corporation.

SUMMARY

This paper discusses the issues involved in servicing interrupts in Ada, with reference to the mechanisms in (Refs. 1 and 2), and concludes that it can be done efficiently and reliably, but that some form of asynchronous communication is needed, and that this is best done explicitly using an additional run time system interface.

CONTENTS

1 CONVENTIONAL MULTI_TASKING EXECUTIVES

With such an executive, the sequence of events initiated by an interrupt from a peripheral device typically includes the following:

(a) Set processor priority equal to interrupt priority

(b) Inform device that interrupt has been accepted

(c) Initiate next operation on device

(d) Request executive to unsuspend a particular task, for lower priority processing subsequent to the interrupt

(e) Restore processor to lower priority and return from interrupt code

Exactly what is involved in (a) and (e) depends on the type of processor; in some cases there are one or more separate interrupt controller chips which hold some of the information about the processor priority, and in this case (e) may include configuration-dependent code.

Depending on the design of the device, (b) and (c) might not both be required.

Depending on the complexity of the decisions required for (c), it may be omitted from the interrupt code and carried out elsewhere, e.g. in the task unsuspended as a result of (d).

The request (d) is performed asynchronously, that is control returns from the executive as soon as the information necessary for the unsuspending has been recorded; the task might not yet even have suspended itself, and there may in any case be higher priority tasks ready to run. Were the request not performed asynchronously, the interrupt code might have to be suspended; on most processors this is equivalent to a busy loop on the interupt level, which could of course prevent the task ever getting to a point at which it could be unsuspended; even on those processors which are technically capable of suspending an interrupt level it is usually avoided.

2 INTERRUPTS SERVICED BY ADA TASK ENTRIES

Interrupts may be associated with Ada task entries by the use of address clauses (implementation-dependent option). When an interrupt occurs, the corresponding Ada task entry is called, with a priority greater than that of any normal (i.e. non-interrupt) task. It is up to the compiler implementor to decide whether this is actually the interrupt level priority. He or she has three choices:

1. Execute the accept body at the interrupt priority level

2. Execute the accept body at a "software" priority (high, but not interrupt, priority)

3. Execute the accept body at the interrupt priority level until step (d) is reached, whereupon the priority drops to a software priority and step (e) is performed implicitly.

Choice (1) has the disadvantage that step (d) cannot be performed, because an Ada rendezvous is synchronous.

Choice (3) resolves this problem, but is only straightforward if step (e) contains no configuration-dependent code. It has the disadvantage that the programmer must recognise the points in his code where (d) may occur, so that any operation required to occur on the interrupt level can be positioned appropriately. Also, depending on the processor architecture, a succeeding interrupt may be delayed or lost.

Choice (2) means that the bulk of steps (a) to (e) are performed in some interrupt routine external to the Ada program; at step (d) this interrupt routine causes the executive to queue a (dummy) task to the task entry that has been associated with the interrupt; the interrupt routine will contain device-specific (and maybe configuration-specific) code, the task entry will process the data or event whose transfer is signalled by the interrupt.

3 SUGGESTED APPROACHES TO IMPLEMENTATION

3.1 Ada Interrupt Routine With Asynchronous Call To Task Entry

This is essentially choice (2), with the interrupt routine written in Ada. An example is the Esprit project ASTERIX (Ref. 1).

This is efficient, allows all the user-written code to be in Ada, and doesn't need any additions to the LRM. It is desirable for the unsuspend interface to the executive to allow the task (and entry) to be specified, thus avoiding the need for an address clause, and hence avoiding the problems with associating interrupts with entries of objects of task types.

3.2 Task Polls Flag Set By Interrupt Routine

The interrupt routine is written in Ada and, instead of step (d), merely sets a flag which is inspected at intervals (using a delay statement) by an Ada task.

Because of the overheads of this, it is obviously only useful where a fast response by the Ada task is not required. An example of its possible use would be in a user interface, where the Ada task might only be involved at the end of a line of input data, and delays of tens of milliseconds could be accepted.

Note that this approach is implicit in various of the suggestions for using pragmas to declare that an accept body can be executed on an interrupt level because it will not attempt a rendezvous (and hence can only communicate by setting flags). It has the advantage over the use of such pragmas, that it can be used (probably) with any compiler, even one that doesn't support attaching interrupts to task entries at all, and its meaning does not depend on the use of pragmas.

3.3 Standardised Interrupt Control Hardware

This would allow choice (3) above to be implemented. The author does not believe that this would be particularly difficult to achieve. What it involves is being able to configure the Ada executive to support any reasonable set of interrupt controllers and to make use of a table relating these to the interrupting devices. However, this requires a consensus between designers of hardware and designers of executives (or an imposed standard). It would have the advantage of removing more of the details of interrupt servicing from the programmer, and of increased portability.

3.4 Ada-specific Interrupt Control Hardware

This could be designed, with an interrupt priority level per interrupting device, such that a particular interrupt could be suspended, allowing other interrupts of both higher and lower priorities to be serviced without allowing further interrupts from the suspended device. This would allow an accept body, executing on an interrupt priority level, to perform an unconditional rendezvous safely, with the interrupt level being suspended by the executive whenever the "task" calling the accept body became ineligible for execution.

A conditional rendezvous (or alternatively some form of asynchronous communication) would still be needed however should timing constraints require that an interrupt be re-enabled quickly (e.g. to continue acquiring data into a circular set of buffers while informing a task that a particular buffer had been filled).

The Intel 8259A Programmable Interrupt Controller appears to have been designed with this purpose in mind, but the author is not aware of any attempts to use it for this purpose. This could be seen as a very special case of the suggestion in the preceding section. It shares with it the disadvantage that further interrupts from the same device cannot be serviced until any rendezvous by the interrupt task entry has been completed.

4 COMPARISON WITH THE ARTEWG PROPOSALS

This section compares the approaches discussed in section 3 with the fast interrupt pragmas discussed in (Ref. 2).

4.1 Pragma INTERRUPT_TASK with KIND = SIMPLE

This is essentially an Ada interrupt routine that communicates solely by setting (statically allocated) flags (as discussed in section 3.2).

4.2 Pragma INTERRUPT_TASK with KIND = SIGNALLING

This is essentially an Ada interrupt routine that may or may not succeed in unsuspending a task waiting at an accept. If the task is already waiting, the effect is the same as an asynchronous unsuspend (as discussed in section 3.1); if not (and it can never be guaranteed), the task has to use a flag and a time-out. This is less satisfactory than the solution presented in section 3.1.

4.3 Pragma MEDIUM_FAST_INTERRUPT_ENTRY

This is the same as the previous case, except that there is a partial context switch to allow access (e.g. via the task's stack frame) to any visible types or objects.

5 DISCUSSION

The ARTEWG pragmas in (Ref. 2) fail to solve the basic problem of servicing an interrupt with an Ada task entry, which is that Ada does not provide an efficient, reliable means to unsuspend a lower priority task that may not yet have suspended itself.

The approach exemplified in (Ref. 1), in which a (possibly Ada-written) interrupt routine can asynchronously unsuspend an Ada task, meets the requirements of the application designer for efficiency and reliability, but uses a mechanism outside of the current definition of Ada (and currently not in the suggested list of run-time interfaces in (Ref. 2)).

The options discussed in sections 3.3 and 3.4, in which the run time system drops off the hardware interrupt level when required, have the problem that there may be delays before further interrupts from the same device may be serviced, due to waiting to complete a rendezvous with a lower priority task.

The use of specialised interrupt control hardware, as discussed in section 3.4, could be generalised to run **all** tasks on appropriate "hardware" priority levels. This would allow the priority inheritance or priority ceiling protocols (Ref. 3) to be applied to all tasks. Some early real time computers (e.g. the IBM 1800) did in fact use hardware interrupt levels for pre-emptive scheduling of applications software. However, such machines had limited main memory and hence a limited number of tasks, so that quite straightforward interrupt circuitry could be used. For Ada programs on modern processors, one would probably need to map external interrupts into a subset of the internal hardware priorities.

6 REFERENCES

1. Project Asterix Final Report, Bayan, R, et al, TECSI, Paris, May 1987

2. A Catalog of Interface Features and Options for the Ada Run Time Environment, Ada Run Time Environment Working Group (ARTEWG), ACM SIGADA, October 1986

3. The Priority Ceiling Protocol: a Method for Minimising the Blocking of High Priority Ada Tasks, Goodenough, J B and Sha, L, 2nd International Workshop on Real Time Ada Issues, June 1988