RISC Architecture and Technology

Worst Case Execution Time Processing

Preliminary User's Guide

RAT-ALS-WP20-PUG-001
Issue 2 Revision 0

29th November, 1996

Prepared By:

Brian Dobbing, Technical Manager

Reviewed By:

Paul Warren, QA Authority

Authorised By:

Adrian Larkham, Project Manager

Document Status Sheet

Project Risc Architecture and Technology Worst Case Execution Time Processing			
Document Title Preliminary User's Guide			
Document Reference RAT-ALS-WP20-PUG-001			
Issue	***Revision***	***Date***	***Reason for Change***
1	0	05/05/95	Initial agreed version.
2	0	29/11/96	Updated to reflect subset of use defined by Prototype.

Distribution

Company	***Person(s)***	***Copies***
ESTEC	Tullio Vardanega	
Thomson Software Products	Adrian Larkham, Brian Dobbing, Paul Warren, Project File	4

Table of Contents

1 Introduction 1

1.1 Status 1

1.2 Purpose 1

1.3 Background 1

1.4 Definitions, Acronyms and Abbreviations 3

1.5 References 4

1.6 Overview of the User Guide 5

2 The Computational Model 6

2.1 Scheduling Theory 6

2.2 Critical Tasks 6

2.3 The Scheduling Model 7

2.4 Mapping Ada to the Scheduling Model 8

2.4.1 Decomposition 8

2.4.2 Application Program Restrictions 9

2.4.3 Task Templates 9

2.4.4 Critical Task Restrictions 9

2.5 WCETE operation 10

2.5.1 Actions Taken During Compilation 10

2.5.2 Actions Taken During Binding 10

2.5.3 Modifying WCETE Operation 11

2.5.4 Treatment of Exceptions 11

3 Coding Restrictions and Extensions 12

3.1 Additional Pragmas 12

3.1.1 Pragma Passive 12

3.1.2 Pragma Loop_Count 13

3.1.3 Pragma Exclude_WCET 14

3.2 The WCET Ada Restrictions 16

3.2.1 The WCET Ada Application Restrictions 16

3.2.2 The WCET Ada HRT Restrictions 16

3.2.3 Unbounded Execution Time Restrictions 17

4 Using the WCETE 18

4.1 Exploitation 18

4.1.1 Logical Model 18

4.1.2 Physical Model 18

4.1.3 Use during Development Lifecycle 19

4.1.4 Use with ATAC Tasking Co-Processor 21

4.2 Invocation 21

4.2.1 Compiler Window 22

4.2.2 Binder Window 24

5 Data Formats 26

Table of Contents (cont)

5.1 Input Files 26

5.2 Output Files 27

6 Warning and Error Messages 29

6.1 Compile-Time Messages 29
6.1.1 Compile-Time Errors 29
6.1.2 Compile-Time Warnings 29

6.2 Bind-Time Messages 30
6.2.1 Bind-Time Errors 30
6.2.2 Bind-Time Warnings 30

Annex A 31

Prototype Definition 31
A.1 Task Templates 31
A.2 User Pragmas 31
A.3 Restriction Enforcement 32
A.4 User Options 34
A.5 Input and Output Files 34
A.6 Error and Warning Messages 34

1 Introduction

1.1 Status

This is the Preliminary User's Guide for the Worst Case Execution Time Estimation (WCETE) capability, which forms an extension to the Ada Compilation System being developed as part of the ESA 32-bit Microprocessor and Computer System Development project (MCD - [R1]). Logically, the WCETE capability is a bridge between the Ada Compilation System and the Hard Real Time (HRT) toolset also being developed as part of MCD [R2, R3].

During Phase 1 of project MCD, the need was identified for a capability to support the generation of the timing data required by the HRT tools (the Schedulability Analyser and Scheduler Simulator). A User Requirements Specification for this capability was produced [R4] which led to the generation of a Contract Change Notice to specify the software requirements for the Worst Case Execution Time Estimator [R5].

A follow-on project [R6] was set up to consolidate these software requirements [R7], to consolidate the interface control definition [R8] and to generate an Architectural Design [R9]. This Preliminary User Guide document has also been produced as part of that work in order to provide early visibility of the user interface for the WCETE capability.

A further project [R13] was set up to generate a rapid *prototype* implementation of a subset of the full WCETE capability, which is defined in Annex A of this User Guide.

1.2 Purpose

This guide provides a description of the WCETE capability from the user's perspective. It includes details of Ada language restrictions, descriptions of all user-meaningful inputs and outputs, operating instructions and recommendations on its use in conjunction with the HRT analysis tools being developed as part of project MCD.

Annex A, which describes the subset of the WCETE capability supported by the prototype implementation, complements those sections of the User Guide which are relevant to the prototype. Other sections should be considered as preliminary and subject to change. The final User Guide will be produced as part of any later project to generate the full implementation of the WCETE capability.

1.3 Background

There is increasing recognition that the software components of critical real-time applications must be provably predictable. This is particularly so for a Hard Real-Time system, in which the failure of a component of the system to meet its timing deadlines can result in a catastrophic failure of the whole system. The choice of a suitable design and development methodology, in conjunction with support tools enabling the real-time performance of a system to be analysed and simulated, can lead to a high level of confidence that the final system meets its real-time constraints.

An earlier project, the ESA Hard Real-Time Operating System Kernel study (HRTOSK - [R10]), examined the design and development process used to produce complex critical applications. HRTOSK proposed a design methodology based on HRT-HOOD, and identified three specialised

tools to aid the development life-cycle. Two of these tools, the *Schedulability Analysis Tool* and the *Scheduler Simulator*, are being developed as part of the MCD project. The third, the *Worst-Case Execution Time Estimator*, is the capability described here.

HRTOSK found that the traditional methods used for the design and development of complex critical applications, which concentrate on functionality, are inadequate for hard real-time systems. This is because non-functional requirements such as dependability, timeliness and dynamic change management are left until too late in the development cycle. HRTOSK proposed a new design methodology which extends the software development life-cycle to explicitly include a *Physical Architecture Design* stage. This stage, which addresses timing and schedulability requirements, is carried out in parallel with the more traditional *Logical Architecture Design* stage, which addresses functional requirements.

The key to this approach is the use of fixed-priority process scheduling, and the provision of tools which allow the schedulability of a system to be analysed and simulated throughout the development life-cycle. The Logical Architecture Design phase identifies the processes within a system and their interaction, and the Physical Architecture Design Phase enables timings to be associated with these processes, and scheduling requirements to be set and investigated. Using this approach the real-time behaviour of systems can be analysed and simulated at a very early stage in their development, so avoiding the common problem of finding only during final system integration and testing that the design fails to meet its timing requirements.

The validity and accuracy of the results obtained from the schedulability analyser and simulator tools depend critically on the execution timing data available for the various components of the application code. To provide valid results the timings must be *worst-case* timings, i.e. for each section of code an upper bound on the execution time must be found. For the best accuracy (and to avoid the need to provide more processing power than necessary) the timings should not be too pessimistic.

During the design and early implementation phases of a system, the final code will not be available, and so in order to perform any schedulability analysis, approximate estimates of timing information will need to be provided. It is often possible to base initial estimates of the process timings on the developer's previous experience with similar systems. As an implementation progresses and more code is written it becomes possible to refine the timing data, and any schedulability problems will thus be flagged at a much earlier stage than with traditional design methods.

The Worst-Case Execution Time Estimator enables the developer to automatically generate the required timing data as the code is developed. It combines information from the source code with detailed knowledge of the instruction timings of the object code generated by the compilation system to produce valid worst-case times. An additional function of the WCETE is to check for coding restrictions which must be followed by the components of an application in order for it to be suitable for analysis by the HRT tools. The primary output from the WCETE is an *Execution Skeleton*, which is a textual file containing information about the tasks in an Ada application, including their worst-case timing details and their interaction, in a form suitable for direct input to the Schedulability Analyser and the Scheduling Simulator.

1.4 Definitions, Acronyms and Abbreviations

ACS	Ada Compilation and Debugging System
Ada83	The Ada language as defined by the Ada Reference Manual ANSI/MIL-STD 1815 A
Ada95	The Ada language as defined by the Ada95 Reference Manual ISO/IEC 8652-1995
ADS	Arbitrary Deadline Scheduling
ATAC	Ada Tasking Accelerator Circuit
Critical Task	A task whose failure to meet its deadline could cause system failure
Deadline	The maximum time allowed to a task to produce a response following its invocation
DMS	Deadline Monotonic Scheduling
EDF	Earliest Deadline First scheduling
Execution Skeleton	Description of the worst case timing characteristics of a multi-tasking Ada application program
HRT-HOOD	Hard Real Time - Hierarchical Object Oriented Design
HRT Toolset	The Worst Case Execution Time Estimator, Schedulability Analyser and Scheduling Simulator
Instruction Timings File	Vendor-provided file containing timing information for processor instructions
MCD	32-bit Microprocessor and Computer System Development project
Passive Task	Ada83 extension used to provide protected object functionality
PBPS	Priority-Based Pre-emptive Scheduling
PCE	Priority Ceiling Emulation
Profile	The definition of the execution of a subprogram, entry body, or main loop in a thread in two parts: (1) The worst case execution path in terms of its WCET, the count of the number of memory read and write operations, and the calls to passive tasks; (2) The set of called passive tasks on all paths.
Protected Object	A HOOD object which does not have arbitrary synchronization constraints and can be analysed for its blocking times.
RMS	Rate Monotonic Scheduling

TDF	Timing Data File: (a) Timing data recorded by the WCETE internally in the Ada program library; (b) Optional output file from WCETE which contains an ASCII dump of (a)
UCF	User Configuration File
WCET Ada Restrictions	Restrictions which must be followed by Ada application code, and by its critical tasks, to allow analysis using the HRT toolset.
WCETE	Worst Case Execution Time Estimation

1.5 References

[R1] 32-bit Microprocessor and Computer System Development, ESTEC Contract 9848/92/NL/FM, Saab Ericsson Space.

[R2] 32-bit Microprocessor and Computer System Development, Software Tools Overall Definition Document MCD-LOG-P1-TN-007, Issue 2, 11th July 1994.

[R3] Consolidated Hard Real Time Tools Interface Control Document, Ref 32B-SBI-0189-ICD-001, Issue 1 rev 1, 03/02/95.

[R4] Worst-Case Execution Time Analysis Requirements Specification, WDI/TV/1903 rev 1.0, June 25th 1993.

[R5] Software Requirements for the Worst Case Execution Time Estimator, DTI-32B-ST-004, Issue 1 Rev 3, July 1st 1994.

[R6] Statement of Work for Requirements Consolidation and Outline Architectural Design for Worst Case Execution Time Processing, WDI/TV/2222 Issue 1, October 3rd 1994.

[R7] RISC Architecture and Technology, Consolidated SRD for WCET Processing RAT-ALS-P2A-SRD-01, 27th February 1995, Issue 1, Revision 0.

[R8] RISC Architecture and Technology, Consolidated ICD for WCET Processing RAT-ALS-P2A-ICD-01, 27th February 1995, Issue 1, Revision 0.

[R9] RISC Architecture and Technology, Worst Case Execution Time processing Architectural Design Document, RAT-ALS-WP20-ADD-001 Issue 1 Revision 0.

[R10] Hard Real-Time Operating System Kernel Final Report ESTEC Contract 9198/90/NL/SF - Ref: TP8879 Feb. 1993.

[R11] A Performance Analysis of a Hard Real-time System, C.M. Bailey, A. Burns A.J. Wellings & C.H. Forsyth.

[R12] Alsys Ada Software Development Environment for Sparc based Workstations and Servers under Unix, version 5.5.2, March 1994.

[R13] RISC Architecture and Technology, Worst Case Execution Time Processing ESTEC contract No, 11231/94/NL/FM(SC) Contract Change Notice 1, April 1996.

1.6 Overview of the User Guide

1. Computational Model

   This section describes the computational model which provides the basis for the scheduling theory used by the HRT tools, and explains how the operation of the WCETE follows this model.

2. Coding Restrictions and Extensions

   This section describes the source-level restrictions which must be placed on the components of an application to ensure that it follows the computational model, and also the extensions provided to the Ada language (in the form of implementation-defined pragmas) to enable the user to control the WCETE operation.

3. Using the WCETE

   This section describes how to invoke the WCETE capability, and also discusses how the user can best exploit the facilities provided.

4. Data Formats

   This section describes the format of all user-supplied input data for the WCETE, and user-readable output from the WCETE.

5. Warning and Error Messages

   This section details the warnings and error messages which may be generated as part of the WCETE process, and where appropriate describes corrective action.

6. Prototype Implementation

   This annex defines the subset of the full WCETE capability that is supported by the prototype implementation.

2 The Computational Model

2.1 Scheduling Theory

The Hard Real-Time toolset provides support for the real-time analysis and simulation of systems implemented in Ada, in which processes are implemented as Ada tasks and their interactions are mapped to a form of the Ada rendezvous construct. The primary aim of analysing a system's real-time behaviour is to determine whether it can be scheduled in such a way that it is guaranteed to meet its timing constraints, which are normally defined in terms of deadlines which must be satisfied by the individual tasks.

The traditional approach has been to dispense entirely with separate processes each with their own independent thread of control, and to use a *cyclic executive* which calls each subtask in turn in a fully deterministic manner. Such a system becomes easy to analyse, but is typically difficult to design, inflexible, and not well suited to applications where sporadic activity may occur and where error recovery is important.

Recent research in scheduling theory has found that with a careful choice of scheduling method together with suitable restrictions on the interactions allowed between processes, useful analysis of real-time behaviour is possible. The Hard Real-Time Operating System Kernel work [R10] has identified a scheduling algorithm known as *Deadline Monotonic Scheduling* (DMS) which, when used with *Priority-Based Pre-emptive Scheduling* (PBPS) and *Priority Ceiling Emulation* (PCE) to avoid priority inversion and deadlocks, provides a model suitable for analysis of real-time embedded systems. The model supports cyclic and sporadic processes, the idea of *hard*, *soft* and *non-critical* components, and can also be used with distributed systems.

DMS is a refinement of an earlier algorithm known as *Rate Monotonic Scheduling* (RMS). Other scheduling algorithms which may be used are *Earliest Deadline First* (EDF) and *Arbitrary Deadline Scheduling* (ADS), both of which are enhancements to DMS. All these assume the scheduling method is PBPS. The HRT tools developed in project MCD provide support for analysis using either ADS or DMS.

2.2 Critical Tasks

The various tasks in an application will each have timing constraints. These are normally defined in terms of *deadlines*: the deadline is the maximum time within which a task must complete its response following its invocation.

Each task is classified into one of the following three levels of *criticality* according to the importance of meeting its deadlines:

- *Hard*
  A hard-deadline task is one which must meet its deadlines. The failure of such a task to meet its deadline may result in catastrophic system failure.

- *Soft*
  A soft-deadline task is one which should meet its deadlines most of the time. An occasional missed deadline can be tolerated without causing system failure (but may result in degraded system performance).

- *Non-critical*
  A non-critical task has no strict timing requirements. Such tasks are typically background tasks which perform activities such as system logging. Failure of a non-critical task does not endanger the performance of the system.

Hard and soft deadline tasks are known collectively as *critical* tasks. The HRT tools restrict their analysis to the application's critical tasks.

2.3 The Scheduling Model

At any moment in time, some processes may be *ready* or *runnable* (meaning that they are able to execute instructions if processor time is made available) while others are *suspended* (meaning they cannot execute until some event occurs). Suspended processes may become ready synchronously (as a result of an action taken by a currently running process) or asynchronously (as a result of an event, such as an interrupt or timeout, which is not directly connected with the current process). In addition, a process may be *blocked* (meaning that it awaits access to a shared resource which is currently exclusively owned by another process).

With priority-based pre-emptive scheduling on a single processor, a priority is assigned to each process and the scheduler ensures that the highest priority ready process is always executing. If a process with a priority higher than the currently running process becomes ready, the scheduler performs a *context switch* as soon as it is able to enable the higher-priority process to restart execution. The term pre-emptive indicates that this can occur as a result of an asynchronous event (i.e. beyond the control of the running process).

Processes will normally require to interact as a result of contention for shared resources and the need to synchronize their activities. Uncontrolled use of such interactions can lead to a number of problems:

- *Unbounded Priority Inversion / Blocking*, where a high-priority process is blocked awaiting a resource in use by a low-priority process, and as a result runnable processes of intermediate priority may effectively hold up the high priority process for an unbounded time as they will run in preference to the low priority process.

- *Deadlock*, where a group of processes (possibly the whole system) block one other permanently due to circularities in the ownership and contention for shared resources.

- *Missed Deadlines*, where the complexity of the interactions between processes makes it impossible to choose suitable priorities to give predictable system performance and as a result, a process fails to complete its response before its deadline has expired.

The scheduling model chosen for use by the HRT tools is designed to minimise the upper bound on blocking time, to prevent deadlocks, and to verify that there is sufficient processing power available to ensure that all critical tasks meet their deadlines. In this model, processes do not interact directly, but instead interact via shared resources known as *protected objects*. Each protected object provides either a resource-control function or a synchronization function.

A protected object used for resource control provides the facility commonly known as a *monitor* or *critical region*, where at most one process at a time can have possession of the object. During the

period that a process has possession, it must not perform any operation which could result in it becoming suspended.

A protected object used for synchronization provides a signalling facility, whereby processes can signal and/or wait on events. In the chosen model, the use of protected objects for synchronization by the critical tasks is constrained so that at most one task can wait on each protected object, and this wait must occur at the start of its outermost loop.

By assigning a priority to each protected object which is just greater than the highest priority of all its users, and using Priority Ceiling Emulation, whereby a process using a protected object has its priority raised dynamically to the priority of the object, deadlocks can be prevented. Furthermore the applicable time that a process may be blocked when attempting access to a protected object, as a result of access to the same object by lower-priority processes, is calculated as the maximum time that the object is in use by these lower-priority processes.

Both the non-critical and the critical tasks of an application must follow certain restrictions to ensure that the HRT tools can successfully predict the scheduling behaviour. There are fewer restrictions on the non-critical tasks, as the only requirement is to ensure that they do not perform operations which could cause the scheduling of the critical tasks to be affected in an unpredictable manner.

This guide does not aim to present the theory behind scheduling analysis in any further detail; the interested user is referred to [R10, R11].

2.4 Mapping Ada to the Scheduling Model

As indicated above, the analysis of an Ada application which makes unrestricted use of Ada runtime features including tasking rendezvous is not feasible. In addition the non-deterministic and potentially unbounded behaviour of many tasking and other runtime calls may make it impossible to provide the upper bounds on execution time which are required for schedulability analysis and simulation. In order to use the HRT tools with the above scheduling model, Ada coding restrictions must be followed both to ensure that all code within critical tasks is time-bounded (so that worst-case times can be found) and to ensure that task interaction can be modelled using protected objects (so that DMS or ADS analysis can be performed and blocking times are predictable).

The coding restrictions are collectively known as the *WCET Ada Restrictions*. They are briefly described below, and are presented in detail in section 3. The WCETE capability checks for compliance with this subset, and will only generate an Execution Skeleton if the application as a whole meets all the restrictions.

2.4.1 Decomposition

The application must be decomposed into a number of separate processes each with a single thread of control, with all interaction between these processes identified. Each process is implemented as a separate Ada task, and in addition a number of *passive tasks* must be introduced to implement resource control and synchronization (e.g. for data sharing and transfer). The passive tasks correspond to the protected objects discussed above. Non-passive tasks are known as *active* tasks.

The decomposition into active and passive tasks is normally the result of applying a design methodology such as HRT-HOOD. The passive tasks are exactly the protected objects identified by HRT-HOOD analysis.

Protected objects and passive tasks are not features of Ada83. They arise from the need to provide a mechanism for implementing constructs such as semaphores and critical regions in Ada without the overhead associated with a separate manager task with its own thread of control. The use of passive tasks also enables Priority Ceiling Emulation to be used. The Ada Compilation System developed for the MCD project is extended to enable a task to be explicitly declared as passive by means of a pragma. A passive task must meet certain restrictions (described in section 3) which in particular ensure that the only code associated with the task is the code of its accept bodies. Ada95 does introduce the concept of protected objects and types, and their semantics as defined in Ada95 have been closely followed in the Ada83 passive task extension.

2.4.2 Application Program Restrictions

In order to be suitable for HRT analysis, all the code of an application must meet the tasking restrictions known as the *WCET Ada Application Restrictions*, which are described in full in section 3.

The main restrictions are that all tasks must be declared in the elaboration part of library-level packages or their subpackages, and that active tasks can only rendezvous with passive tasks.

2.4.3 Task Templates

Critical tasks must in addition match one of the following task templates:

- *Cyclic*

  A cyclic task has an outermost infinite loop containing an unlabelled call to the Ada runtime procedure Delay_Until at the start or end of the loop.

- *Software Sporadic*

  A software sporadic task has an outermost infinite loop containing an unlabelled call to the guarded select alternative of a passive synchronization task at the start of the loop.

- *Interrupt Sporadic*

  An interrupt sporadic task has an outermost infinite loop containing an unlabelled accept statement for an interrupt entry at the start of the loop. The entry must be defined as an interrupt entry by a representation clause in the task specification.

- *Passive Resource Control*

  A passive resource control task is a passive task with no guarded select alternatives.

- *Passive Synchronization*

  A passive synchronization task is a passive task whose select statement has one guarded and one unguarded select alternative (and which consequently has exactly two entries).

2.4.4 Critical Task Restrictions

Critical tasks, and any subprograms they may call, must in addition conform to the restrictions known as the *WCET Ada HRT Restrictions*, which are described in full in section 3.

The main restrictions are that no operations which could suspend or potentially block a task, other than passive task entry calls and calls to Delay_Until, are permitted while a critical task is executing its main loop, and all code executed by a critical task while in its main loop must have bounded execution time.

2.5 WCETE operation

The WCETE capability is integrated with the Ada Compilation System, and is invoked by selecting extra options during the compilation and binding steps.

The WCETE may, during its operation, detect error conditions and potential errors. These are known as WCETE errors and WCETE warnings respectively. Detection of such warnings and/or errors does not affect the overall success of the Ada compilation and binding steps.

WCETE errors detected during the compilation process cause the WCETE data for that compilation unit to be invalid; otherwise the data is valid. During the bind process, the generation of the Execution Skeleton by the WCETE is suppressed if any units in the application contain invalid WCETE data, or if new errors are detected. However the detection of a WCETE error at any stage does not alter the validity of the WCETE data for other compilation units in the application.

2.5.1 Actions Taken During Compilation

- All code is checked for compliance with the WCET Ada Application Restrictions. Any non-compliant code is reported as a WCETE error, and the WCETE data for the unit is invalid.

- All tasks matching the allowable templates of section 2.4.3 are assumed to be critical tasks, unless explicitly indicated to the contrary by the user.

- A subprogram body which does not comply with the WCET Ada HRT Restrictions, but which could be called from a critical task, is recognised and reported as a WCETE warning, unless explicitly indicated to the contrary by the user. Note that at compile-time the WCETE cannot determine whether a subprogram is called from a critical task (in which case non-compliance of any of its code to the restrictions is a WCETE error) or is only called from non-critical tasks.

- Worst-case timing for all code sequences which meet the WCET Ada HRT Restrictions and which are not explicitly excluded by the user, are recorded in the program library.

2.5.2 Actions Taken During Binding

- All units included in the closure are checked to ensure they have been compiled with WCETE processing enabled, and have not been found to fail the WCET Ada Application Restrictions.

- All code which is callable from the main loop of a critical task is checked for compliance with the WCET Ada HRT Restrictions.

- If any of the above checks fail, WCETE errors are reported and no Execution Skeleton is generated. Otherwise, if no errors have been found, the critical task profile and timing information from the individual compilation units is combined to produce an Execution Skeleton for the whole application.

Note that if a WCETE error is detected at bind-time which relates to a set of compilation units in the application, only these units need to be re-compiled before another bind is attempted. In other words, the validity of the WCETE data for the other compilation units in the application is not affected.

Additionally, the WCETE report can be requested by the user even if bind-time WCETE errors have occurred, and the (valid) timing information relating to units not affected by the errors can be generated and selected for inclusion in the report along with the WCETE error or warning messages.

2.5.3 Modifying WCETE Operation

The user has the ability to modify the operation of the WCETE in a number of ways:

- Timing data may be supplied to the WCETE at bind time to override the timing data for particular subprograms. This is useful during prototyping and early development, where the final code may not be available.

- The WCETE may be directed by means of **pragma** Exclude_WCET to ignore sections of code for the purpose of calculating worst-case timings. The WCETE assumes that any code sequence for which this pragma applies will not be executed by a critical task.

- The WCETE may be provided with upper limits for the number of iterations of loops by means of **pragma** Loop_Count.

Full details of these pragmas are provided in the following section.

2.5.4 Treatment of Exceptions

Critical tasks are not permitted to raise exceptions, as the handling of exceptions is a potentially unbounded operation. WCETE processing cannot enforce this restriction in the case of exceptions raised by the failure of Ada-defined implicit runtime checks, and the approach taken is to assume that such predefined checks always succeed. For consistency with this approach, any sequential code sequence ending in an explicit **raise** statement is also treated as if it contained an implicit **pragma** Exclude_WCET. In this case, a warning is also issued to the user that this sequence of statements has been excluded from worst case execution time calculation.

Exception handlers are permitted within critical task code, but since exception raising is assumed not to be used as a programming tool for the normal case paths which are to be analysed, the handlers are ignored for the purpose of calculating worst-case timings (ie. they also contain an implicit **pragma** Exclude_WCET to exclude their sequence of statements from consideration by the WCETE). As above, a warning is issued to the user that the handlers have been excluded from worst case execution time calculation.

3 Coding Restrictions and Extensions

3.1 Additional Pragmas

3.1.1 Pragma Passive

The passive task extension enables more efficient use of memory and execution time for the implementation of Ada tasks whose sole purpose is to provide a mutual exclusion or synchronisation abstraction, such as shared resource control or event signalling. In addition, the passive task is implemented in such a way as to be compatible with the concept of a *protected object* in HRT-HOOD or Ada95. In order for an Ada task to be implemented as a passive task, it must meet a set of restrictions. These restrictions arise partly from efficiency of execution, partly from simplification of implementation, partly from HRT requirements, and partly from the Ada95 protected object model.

A passive task must be explicitly nominated as passive by including **pragma** Passive in its task specification, and must obey the following restrictions. The term *guard* is used below to indicate a **select** alternative which has a **when** condition - note that this is common usage but was not used in the Ada83 reference manual.

- The task must be declared in the elaboration part of a library package or sub-package, and must be declared as a single task object (i.e. not using a task type).
- The task specification must include at least one single entry declaration, and no entry families.
- Entry address clauses are not permitted.
- The statement part of the task body must consist of an infinite loop containing just a single **select** statement. The **select** statement must have exactly one select alternative for each entry, and an unguarded terminate alternative.
- The task body must not have exception handlers (although blocks declared within the accept bodies may have handlers). Evaluation of any guards in the select alternatives must not raise exceptions, and any exception raised in an accept body must be handled locally in the accept body.
- The guards in the select alternatives and the statements in the accept bodies are not permitted to contain potentially suspending Ada constructs. In particular this implies that the only tasking constructs permitted within accept bodies are calls to unguarded entries of other passive tasks.
- The task attribute *entry*'Count must not be used.

Further restrictions apply to any tasks which interact with a passive task:

- All entry calls to passive tasks must be simple (i.e. timed or conditional entries using a **select** statement are not permitted).
- A passive task entry must not be called from within an interrupt handler procedure or from within the accept statement of a task entry with an address clause (an interrupt entry).
- The task attributes 'Callable and 'Terminated must not be used with passive tasks.

- The **abort** statement must not be used with a passive task.

The exception Program_Error is raised in the following circumstances, which are considered as misuses of passive tasks but which cannot be detected by the compiler:

- If the current active priority of the caller of a passive task exceeds the passive task's priority.
- If a passive task entry is called before the passive task declaration is fully elaborated.

Example:

A typical passive task declaration will take the following form:

```
task name is
    pragma Passive;
    pragma Priority (integer);
    entry entry_1;                  -- Only simple entry declarations allowed
    entry entry_2;                  -- There must be at least one entry declaration
end name;

task body name is
    data-declarations               -- May be null
begin
    loop
        select
            accept entry_1 do       -- Example of unguarded entry
                ...
            end;
        or when condition =>            -- Example of guarded entry
            accept entry_2;
        or
            terminate;              -- Unguarded terminate is mandatory
        end select;
    end loop;
end name;
```

3.1.2 Pragma Loop_Count

This pragma is used anywhere in the outermost sequence of statements of a loop to indicate an upper bound for the number of iterations. For statements executed by a critical task, it is mandatory in all but **for** loops with static bounds, but it may also be used in such a **for** loop when the user wishes to indicate that the loop will always be exited before the full range is covered.

The syntax is:

```
pragma Loop_Count ( static_expression );
```

Example:

```
while Epsilon > Convergence loop
    pragma Loop_Count(5); -- We know convergence will be obtained within 5 iterations
    Calculate_New_Estimate;
end loop;
```

3.1.3 Pragma Exclude_WCET

This pragma is used to mark Ada constructs which are to be excluded from worst case execution time consideration because they do not form part of the critical task code which is to be subjected to schedulability analysis. The pragma can be placed immediately within a sequence of statements, or within the outermost declarative part of a program unit.

When the pragma is placed within a sequence of statements, the effect on worst case time estimation is to disregard the execution time of those statements, and to verify that at least one other alternative path through the code exists if this code can be executed during a critical task main loop.

When the pragma is placed within the outermost declarative part of a program unit, the effect on worst case time estimation is to assume that the entire unit does not contribute in any way to the execution of critical task code, and to generate a WCETE error if the unit or any contained program unit is found to be accessed as part of critical task execution.

The syntax is:

```
pragma Exclude_WCET;
```

This pragma has a number of applications:

- It can be used to indicate that a task which matches one of the templates for consideration as a critical task is not to be treated as critical (and so will not be included in the generated Execution Skeleton).

  Example:

```
task body Not_critical is
    pragma Exclude_WCET;      -- Prevent treatment as a critical sporadic task
begin
    loop
        passive.wait;         -- Await event trigger from passive task
        ...                   -- Processing performs time-unbounded input-output
    end loop;
end Not_critical;
```

- It can be used to inhibit WCETE warnings being generated during the compilation of a subprogram which does not obey the WCET Ada HRT Restrictions but which is known to be called only by non-critical tasks. A WCETE error is generated at compile-time or bind-time if a critical task calls such a subprogram.

  Example:

```
procedure Non_Critical is
    pragma Exclude_WCET;          -- Prevent warnings due to use of heap for access types
    type a_rec is access rec;
    a : a_rec := new a_rec'(...); -- Allocator execution time is not bounded
begin
    ...
end;
```

- It can be used to identify passive tasks whose entries are to be called only by non-critical tasks. A WCETE error is generated at compile-time or bind-time if a critical task calls an entry in such a passive task.

Example:

```
task body Not_Critical_PO is
    pragma Exclude_WCET;      -- Identify as callable only by non-critical tasks
begin
    loop
        select
        ...                   -- Accept statements for non-critical callers
        end select;
    end loop;
end Not_Critical_PO;
```

- It can be used to suppress consideration of a branch of code which is not expected to be executed during normal execution of a critical task. It is often convenient to include such code in critical tasks for example to cater for system testing, for erroneous behaviour (when the system may have already failed, but some recovery is possible) or for indicating the expected path of execution when a particular mode of system operation is being considered.

Example:

```
if status /= normal then
    pragma Exclude_WCET;      -- This branch will not be executed in normal operation
    ...
else
    ...
end if;
```

- It can be used to exclude the entire contents of a library package which is only used in non-critical context, ie. as a shorthand way of avoiding the inclusion of the pragma within every subprogram and passive task in a non-critical subsystem.

Example:

```
package Input_Output is
    pragma Exclude_WCET;      -- The entire package is to be excluded by the WCETE
    procedure Read (...);     -- Procedures called only by non-critical tasks
    procedure Write (...);
    task Non_Blocking_IO is-- Passive task called only by non-critical tasks
        pragma Passive;
        ...
    end Non_Blocking_IO;
end Input_Output;
```

Note that a side effect of excluding sequences of statements or program units from consideration when calculating the worst-case timings, is to inhibit compile-time warnings relating to possible violations of the WCET Ada HRT Restrictions by these statements or units. A WCETE error will be reported at bind time if a critical task does potentially access such statements or units.

There are a number of contexts where the WCETE implicitly inserts **pragma** Exclude_WCET:

- In an exception handler.
- In a sequence of statements terminating with an explicit **raise** statement.
- In code which cannot be executed as part of a critical task body, for example library package elaboration code, and (initialization) code in a critical task body before its main loop.
- In a task whose body does not conform to one of the critical task templates.

3.2 The WCET Ada Restrictions

The restrictions placed on Ada code which is to be used with the HRT toolset are collectively known as the *WCET Ada Restrictions*. In fact there are two levels of restriction; the *WCET Ada Application Restrictions*, which must be followed by all code to be included in an application, and the *WCET Ada HRT Restrictions*, which must in addition be followed by all code called within the main loop of critical tasks.

3.2.1 The WCET Ada Application Restrictions

These restrictions must be followed by all the code of an application.

- All tasks must be declared in the elaboration part of library-level packages or their subpackages.
- Tasks must be simple objects; they must not be declared as components of arrays or records, nor may they be created dynamically using task access types.
- Tasks must not be terminated, either as a result of normal exit of the task body or by use of the **abort** statement.
- Active tasks can only rendezvous with passive tasks.
- The **delay** statement must not be used. Critical tasks may use the Delay_Until runtime procedure, but only in the context described below.

As a consequence of not supporting task termination, all tasks must consist of an infinite loop which repeatedly performs its main actions, optionally preceded by initialization code.

Note that an active task entry is permitted, but its only allowable use is as an interrupt entry.

3.2.2 The WCET Ada HRT Restrictions

All code callable from the main (outermost) loop of a critical task must additionally meet the following restrictions:

- Each passive synchronization task must correspond with a single software sporadic task. The entry call in the outer loop of the sporadic task must be the only entry call to the guarded entry of the passive task.
- A passive task may be used solely by critical tasks or solely by non-critical tasks, but not by a mixture of the two. This restriction is necessary because the blocking time for a task accessing a

protected object depends on usage of the same object by lower priority tasks, but non-critical tasks are not defined to the Schedulability Analyser in the Execution Skeleton.

- No operations which may suspend the task, other than passive task guarded entry calls and calls to Delay_Until, are permitted while a critical task is executing its main loop. In addition, these calls may only be made in specific contexts.

- All code executed by a critical task while in its main loop must have bounded execution time. In particular, critical tasks may not make use of the Ada program heap, for example via allocators or dynamically-sized object creation, and may not perform input-output operations. All loops must have a statically known maximum number of iterations; use of the **return** statement within a loop, the **goto** statement, and the unconditional **exit** statement, is not allowed; recursive subprogram calls must not be made; and exceptions must not be raised.

Note that the WCETE assumes an implicit **pragma** Exclude_WCET in any sequence of statements ending with an explicit **raise** statement. As a result, a conditionally-executed sequence of statements including a **raise** statement is effectively ignored by the WCETE and an unconditionally-executed sequence with a **raise** statement will result in a WCETE error if on the execution path of a critical task.

3.2.3 Unbounded Execution Time Restrictions

Certain Ada constructs result in calls to runtime routines which either have potentially unbounded execution time or may cause the current task to suspend or be blocked for an unknown amount of time. Such constructs are not allowed to be executed by critical tasks although they can be executed by non-critical tasks. An exhaustive list of these constructs will not be available until the design and implementation phase of the full WCETE capability. However, the following are currently known:

- Any use of the I/O packages provided as part of the Ada runtime.
- Explicit uses of the runtime heap such as the **new** operator with access types.
- Implicit use of the heap which the compiler makes for storage of objects. This can occur in a number of situations, such as the use of unconstrained types for function return values and the use of records with defaulted discriminants.
- Operations on composite types such assignment, comparison, and relational and logical operators. These should be coded using an element-by-element loop around a scalar operation.
- Certain attributes such as 'Image, 'Value and 'Width which can have unbounded or very long execution times.

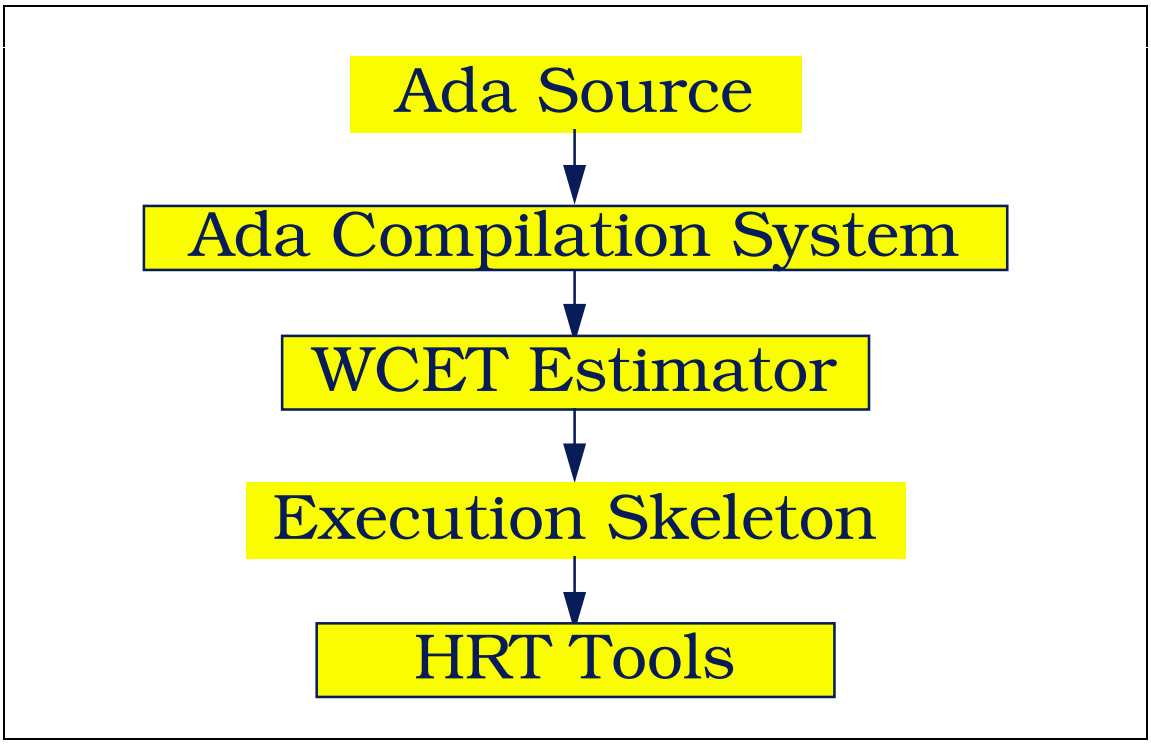
4 Using the WCETE

4.1 Exploitation

This section describes how to exploit the WCETE capability in the context of the software development of a Hard-Real Time application for the ERC32 processor. The WCETE is primarily intended to support the design, code and test stages of the software development lifecycle as defined by the PSS-05-0 standard; however its use is not precluded during other phases. In particular, it is also useful during the maintenance phase to ensure that changes to the software do not introduce regression in the meeting of deadlines.

4.1.1 Logical Model

The WCETE effectively forms a bridge between the Ada program development tools on the one hand and the hard real time tools used to verify the timing requirements on the other. The key data structure forming this bridge is the Execution Skeleton generated by the WCETE which contains the description of the application in terms of the worst case timing characteristics of its threads of control and their interactions. The logical model can be viewed diagrammatically as follows:

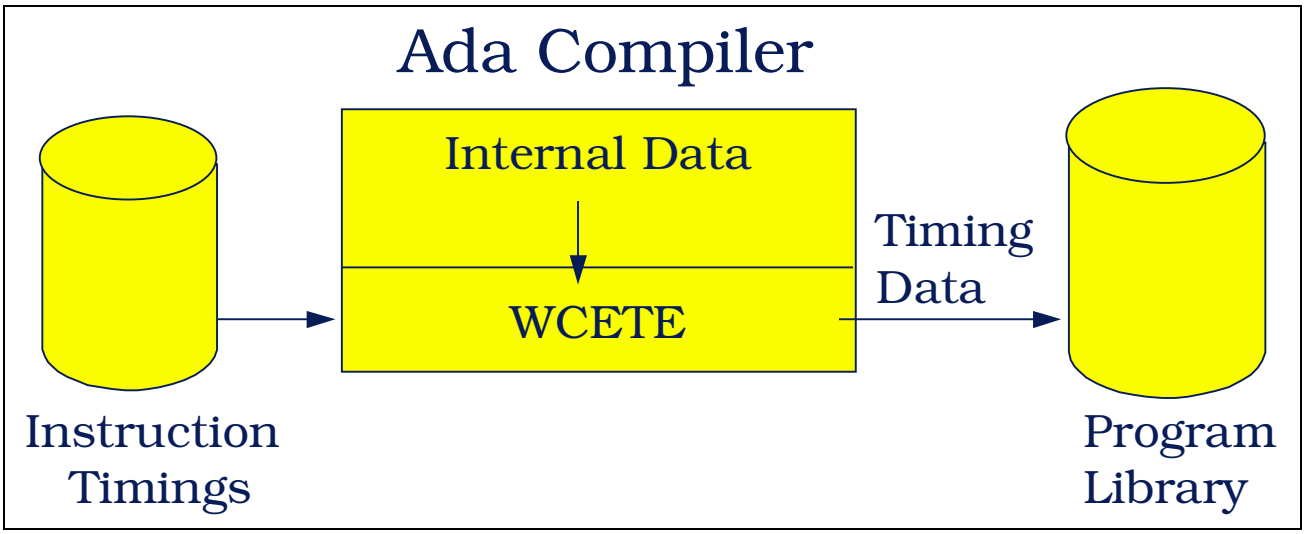


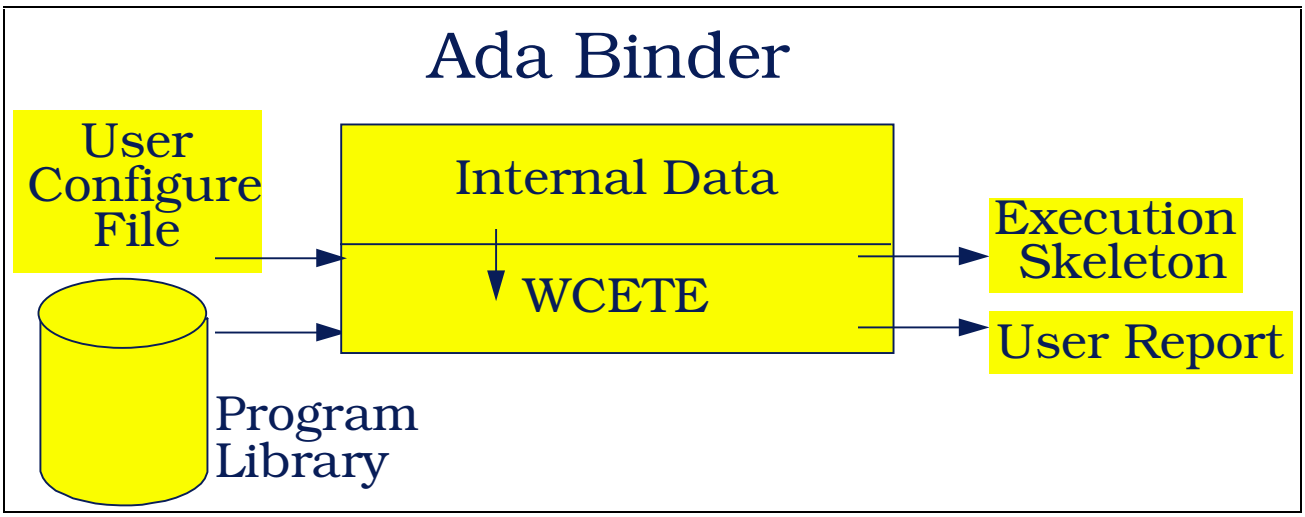
4.1.2 Physical Model

The diagram above shows the WCET Estimator as a separate entity from the Ada Compilation System (ACS). Although this is the case logically, WCETE has physically been integrated with the compilation system so that it can easily re-use and exploit the ACS internal data structures for maximum efficiency and quality. Part of the WCETE processing is done at compile time and part at bind time when the application as a whole is built. Thus from the user's perspective, the WCETE

processing is invoked as a sub-action of running both the ACS Ada Compiler and Binder tools. The user interface for invocation of the WCETE is presented below in section 4.2.

Thus the physical model of the WCETE processing can be viewed diagrammatically as follows:





4.1.3 Use during Development Lifecycle

Clearly the full Ada source code is not always present, and it is highly desirable to be able to build timing requirements into the design, and have them continually verified from the early phases of the development process onwards. Typically, the high-level design for the application will identify all the tasks, including their type, and also all the protected objects (modelled as passive tasks) to be used for inter-task communication. The corresponding Ada framework can then be constructed either manually or automatically (if tool support such as HRT-HOOD exists). If no code has yet been produced, timing estimates can be generated either empirically or based on prototyping, and these can be input to the WCETE to override the null Ada bodies. This allows a valid Execution

Skeleton to be generated so that timing analysis, verification and simulation can be performed even at this early stage.

As the detailed design and coding phases proceed and more code is written, the timing estimates can be refined and eventually eliminated altogether, allowing the values computed by the WCETE to be used. At this stage, the user needs assurance that the computed values are not grossly over-pessimistic and to assist this, the WCETE provides tuning facilities.

Excluding Error Paths

It may be the case that for certain subprograms, the worst case path is one which is only executed in exceptional or error situations. If the user wishes to exclude such a path from the scenario being analysed, then he introduces **pragma** *Exclude_WCET* into the source code sequence of statements forming the path. The effect is that the WCETE will ignore these statements when computing the worst case time.

Tuning Loop Iterations

Another important user action to exploit the WCETE capability fully is to tune carefully the use of **pragma** *Loop_Count,* which is required for any loop whose loop range is not static. Unless the WCETE can determine definitively that the maximum number of iterations of a loop is less than the value supplied by the user, the user's value in the pragma argument will be used in the computation of the worst case execution time for a loop. Thus if the user is overly pessimistic with the maximum upper bound, this will have an adverse effect on the accuracy of the timing estimates, and hence also on the results of the schedulability analysis. If the user is uncertain of the value to use, then if the coding is sufficiently mature to be used with realistic test data, the number of iterations for various test scenarios can be measured experimentally, either interactively using the Ada Debugger or the ERC32 Target Simulator, or else by printing out a count of the number of iterations in the code just before loop exit. On the other hand, if the data structures which control loop exit are not yet mature, an estimate needs to be produced. If the loop is a **for** loop then the range of the type of the loop variable can be used, as long as Ada's strong typing is used properly in the design (eg. avoiding over-use of type Integer); otherwise if the loop is a **while** loop or has no loop control and is exited via an **exit** statement, this often indicates that a linked list is being traversed and so the count should be based on the estimated maximum number of elements in the list.

Analysing and Overriding Timings

In order to assist the user in analysing how the worst case execution time for a thread was computed, the report generated by the WCETE includes timing details on a per-subprogram basis. If the user wishes to override an individual subprogram timing, for example to investigate the effect of a future design change on the schedulability of the system, then he can input such an override to the WCETE in the same way as was done for estimates during the high-level design phase, ie. via the User Configuration File (UCF). Similarly, if a time computed by the WCETE is known to be over-pessimistic and the user is able to determine accurately the precise worst case time, for example by measuring using the "CPU time between two points" facility of the ERC32 Target Simulator, then this time can be used as an override. In order to facilitate the construction of subprogram override definitions in the UCF, the format of the data is the same as that generated for the subprogram by the WCETE in its output report. Thus the user can edit the report file, extract the subprogram timing definition, modify it to reflect the required override, and finally place it into the WCETE-specific section in the UCF.

4.1.4 Use with ATAC Tasking Co-Processor

The Ada Tasking Accelerator Circuit (ATAC) is a co-processor present in the ERC32 chip-set which implements Ada83 tasking operations directly. The Ada Compilation System being developed in project MCD includes support for targeting this co-processor.

The ATAC contains a limited amount of internal on-chip memory for storage of its data structures such as task control blocks. Clearly access to this internal memory is very efficient. If the internal memory space is exhausted via creation of too many active tasks for example, then STORAGE_ERROR exception will be raised during program execution. One of the ways to circumvent this problem is to attach *external* memory to the ATAC co-processor which then can be large enough to accommodate the extra tasks. However the decision as to whether external memory is needed, and the resultant association of the memory area with the ATAC, are *static configuration* issues, rather than state changes that can occur dynamically during program execution. It is therefore necessary for the user to know whether the active tasks declared in the application will overflow the ATAC internal memory area or not.

The Ada Compilation System Binder tool contains a check as to whether the total number of tasks in the application together with the entry declarations that they specify will fit inside the ATAC internal memory or not. The check is based on user-supplied values for "total tasks" and "total entries" which are input to the Binder via invocation options (see *Maximum_Tasks* and *Maximum_Entries* in the figure in section 4.2.2). These options have default settings which are small enough for the check to succeed, but are in no way representative of the actual number of tasks in the application. (The defaults are present so that if the application is known to fit comfortably within the ATAC internal memory, for example because it declares only a few tasks, then the user is not burdened with having to supply actual totals - the check then passes automatically.)

The WCETE provides a facility to automate the count of total tasks and total entries in the application program so that the check does not need to be based on user-supplied estimates. If the WCETE compile-time processing is invoked, then a count of the number of library-level single task object declarations and their simple entry declarations in the compilation unit is performed, and the resultant counts are stored as part of the Timing Data File. At bind-time, if the WCETE compile-time processing has been invoked for every compilation unit in the program closure, then the WCETE bind-time processing sums the counts across all the compilation units and stores the resultant totals for use by the ACS Binder ATAC internal memory capacity check, effectively overriding the user-supplied or default settings, and providing potentially more accurate data for the check.

4.2 Invocation

The WCETE processing is invoked via new options and sub-options to the ACS Compiler and Binder, as described in section 4.1. Note that in order to execute the WCETE processing in the Ada Binder, it is necessary to have invoked the compile-time WCETE processing for every compilation unit which forms part of the program being bound. This is because the Binder needs the Timing Data File for every compilation unit in order to generate the entire Execution Skeleton. An error is generated by the Bind-time processing if the timing data for one of the included units is not found in the program library.

The actual user interface for the Compiler and Binder tools is based on the use of Motif style X windows as described in *AdaWorld for Motif* in *Alsys Ada Software Development Environment for*

Sparc based Workstations and Servers under Unix [R12]. The compiler and binder windows, including the new WCETE-specific options, are shown below.

4.2.1 Compiler Window

The following diagram shows the proposed updated Motif window for invoking the AdaWorld Compiler to incorporate the WCETE options.

HRT_DATA

The radio button marked *HRT_DATA* in the *KEEP* menu must be set to *ON* in order to enable the WCETE processing at compile-time. The default setting for this button is *OFF,* but the user can

override this by the use of AdaWorld for Motif *defaults* facility. The recommended practice when using AdaWorld for Motif is for the user to customise his own set of default option settings for the ACS tools. This avoids, for example, having to remember to set *HRT_DATA* every time the compiler is executed when building an application for timing analysis. AdaWorld for Motif includes a **Default** menu which displays the list of the ACS tools. By selecting *Compile* from this list, a window is opened which closely resembles the compiler invocation window shown above. It is then possible for the user to set all the tool options to the desired default settings, and then click on the **Apply** button to register them as the defaults to be used automatically.

The WCETE-specific section is near the bottom of the window, and is labelled *WCETE.* The section contains two boxes of text fields marked *Times* and *TDF.*

TIMES

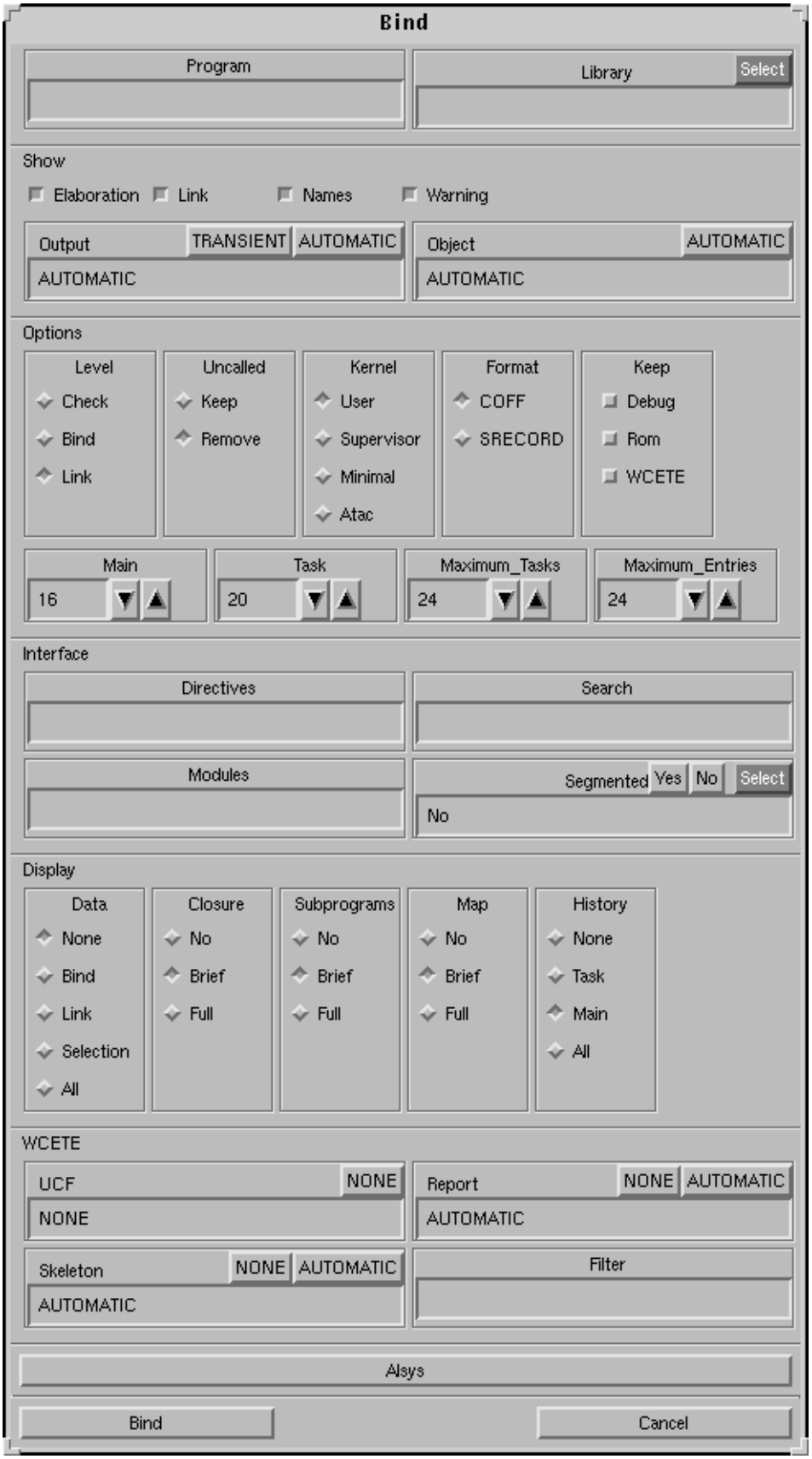
When the *HRT_DATA* button is set, the user is required to enter a filename in the box marked *TIMES.* This is the name of the file containing the times of all instructions on the ERC32 processor in clock cycles, and is supplied as part of the delivered development environment. The user should not normally need to edit this file at all since almost all the times are fixed. However a small subset of the ERC32 instructions have variable timings based on their arguments and in these cases, the worst case time is provided by default in the file. If these worst case times are found to be too pessimistic for a particular application, for example one which performs a large number of floating point operations, the user can take a copy of the predefined timing file and adjust the times for the variable length instructions. This modified file can then be named as the default setting for the *TIMES* option without affecting other users of the system. In order to facilitate the identification of the variable length ERC32 instructions, these are clearly marked and segregated from the fixed length ones in the timings file provided by default. Note that the user must preserve the correct syntax for entries in the timings file otherwise it will be rejected by the compile-time WCETE processing.

TDF

In addition to the text field box for the instruction timings filename, there is also a WCETE-specific text field box labelled *TDF* to enter the name of a file to contain an unstructured ASCII dump of the Timing Data File contents. This is not intended for general use by the WCETE user. It is there mainly for rapid prototyping, for WCETE debugging, and for use by the MCD project development team to assist them in creating the *RTS Characteristics* file, which contains the worst case times of several runtime system operations. However the option is retained in the user interface to allow the user to cause the internal WCETE data to be output, for example, to simplify WCETE maintenance. The option defaults to NONE since it is not intended for use in the normal case.

4.2.2 Binder Window

The following diagram shows the proposed updated Motif window for invoking the AdaWorld Binder to incorporate the WCETE options.



WCETE

The radio button marked *WCETE* in the *OPTIONS* menu must be set to *ON* in order to enable the WCETE processing at bind-time. The default setting for this button is *OFF,* but the user can override this by the use of the AdaWorld for Motif *defaults* facility, as described in section 4.2.1 above for the compiler option settings.

The WCETE-specific section is near the bottom of the window, and is labelled *WCETE.* The section contains four boxes of text fields marked *UCF, Report, Skeleton* and *Filter.*

UCF

The *UCF* text field is for naming the input User Configuration File which can contain user-supplied subprogram definitions to override the ones computed by the WCETE. The standard default setting for this is *NONE*, but the user can override this by the use of the AdaWorld for Motif *defaults* facility as noted above.

REPORT

The *Report* text field is for naming the file to be used for the output report. If no report is required, this should be set to *NONE*. The standard default is *AUTOMATIC* which names the output file as the name of the main subprogram in the Bind (see the *Program* box) with extension *.RPT*. The use of AUTOMATIC for all binder output file names is a convenient way to identify the consistent set which are generated from one run, since all will use the main subprogram name but with different file extensions.

SKELETON

The *Skeleton* text field is for naming the file to be used for the Execution Skeleton. If no report is required (ie. the user merely wants to verify that there are no WCETE-specific errors when building the application) this should be set to *NONE*. The standard default is *AUTOMATIC* which names the output file as the name of the main subprogram in the Bind (see the *Program* box) with extension *.ESF*.

FILTER

The *Filter* text field is used to control the amount of output generated in the report. It is ignored if the *Report* text field is set to *NONE*. If the field is empty, the report includes timing details of subprograms in all compilation units included in the bind of the application. If the application is very large, this would result in a substantial amount of output. The user can control this by naming specific Ada compilation units whose subprograms are to be included in the report. This name can be the name of a library unit or the fully-expanded name of a subunit. In both cases, all of the subunits of the named unit are also automatically included.

5 Data Formats

5.1 Input Files

There are two user-supplied input files to the WCETE processing:

- The Instruction Timings File
- The User Configuration File

Instruction Timings File

This is an ASCII file supplied by default as part of the delivery of the WCETE. The user should not normally need to edit this file, but there may be circumstances in which edits are needed (see section 4.2.1). The section which contains instructions which have variable length times is separated from those with fixed length times in order to encapsulate those instruction times which might need editing. The syntax used in the default delivered file must be maintained by the user.

User Configuration File

The overall definition of the User Configuration File can be found in the Consolidated HRT Tools Interface Control Document in project MCD ([R3]). The file contains a section specific to the WCETE whose definition is in the Consolidated ICD for WCETE Processing ([R8]). This section is primarily for the user to provide timing estimates for named subprograms to override those computed from the Ada source code by the WCETE (see section 4.1.3).

For example, if the user wishes to construct the profile of a subprogram called FOO in library package OUTER which merely consists of supplying a worst case execution time of 100 microseconds, he would create the following entry in the WCETE-specific section of the UCF:

```
SUBPROGRAM Outer.Foo
    WCET 100 m
END
```

Notes:

1. The subprogram name is the fully-expanded Ada name, ie. it includes the library package name as prefix.

2. The time has the letter 'm' appended. This stands for a time in microseconds. It is also possible for the user to supply subprogram times in processor clock cycles instead, in the format produced in the Execution Skeleton (including number of memory read and write accesses).

   Breaking a total time down into these components is useful since the user can specify in the UCF the number of memory wait states for read and write operations that are correct for the kind of memory present on the target board.

3. It is possible to create much more complex profiles which include calls to passive task entries and loop constructs (see section 5.2 for an example).

5.2 Output Files

There are two output files generated for the user by the WCETE processing:

- The Execution Skeleton
- The Report

Execution Skeleton

The Execution Skeleton is generated on request from the user via the WCETE-specific option to the Binder. It is only generated if no WCETE-specific errors (compile-time or bind-time) are encountered in all the compilation units which form the application.

The Execution Skeleton file is the primary output of the WCETE processing for input to the HRT tools developed in project MCD. The user should not in general edit the file since it defines the Ada application that has been built. However in cases where sub-systems of the same application are developed independently and have not yet been combined into a single Ada program, it may be desirable to verify the schedulability of the system as a whole. In this case, the thread and protected object definitions from one skeleton may, with care, be catenated with those of another, although final verification of the schedulability of the application must be done after the sub-systems are combined into a single Ada program, and a single skeleton representing the entire system is automatically generated.

The definition of the Execution Skeleton can be found in the Consolidated HRT Tools Interface Control Document in project MCD ([R3]).

WCETE Report

The WCETE report is generated on request from the user via the WCETE-specific option to the Binder. The report can be produced even if errors have been detected which prevent generation of the Execution Skeleton. The report contains the profile of every Ada subprogram called directly or indirectly from the main loop in any task body which conforms to the cyclic, software sporadic or interrupt sporadic thread template definitions, and every Ada subprogram called directly or indirectly from any passive task accept statement. Any other subprogram, for example those called only by non-critical tasks, is not included in the report.

In addition, the report contains all the WCETE-specific warning and error messages relating to the attempt to construct the data for the Execution Skeleton.

The subprogram profiles automatically generated by the WCETE and listed in the report contain more detail than just the overall worst case execution time. For example, they contain a definition of all calls to passive task entries that the subprogram makes and preserves the execution position of each call. This is for accurate simulation of the scheduling by the Scheduler Simulator tool in project MCD. In addition, the profiles can contain iterator constructs representing the loops in the Ada code.

As an example of a more complicated subprogram profile generated in the report, consider the following Ada subprogram:

```
procedure Foo is
```

```
begin
    Proc_1(...);                    -- A subprogram call
    while Element /= null loop
        pragma Loop_Count (40);     -- Max elements in the list
        Resource.Claim;             -- A passive task entry call
        ...;                        -- Process element indivisibly
        Resource.Release;           -- A passive task entry call
        Element := Element.Next;
    end loop;
end;
```

This subprogram would produce a profile in the report of the form:

```
SUBPROGRAM Outer.Foo
    WCET 100, 120, 20
    LOOP 40
        CALL_PO Resource Claim
        WCET 6, 8, 2
        CALL_PO Resource Release
        WCET 4, 6, 1
    END LOOP
END
```

Notes

1. The first WCET statement represents 100 CPU cycles, 120 memory reads and 20 memory writes corresponding to the body execution of subprogram Proc_1, which does not make any calls to passive task entries.
2. The LOOP statement includes the upper bound supplied by the user via pragma Loop_Count.
3. The CALL_PO statements identify both the passive task and the entry which is called. Their position in the execution order is preserved for accurate simulation of scheduling.
4. If the user were able to determine a more accurate time for the execution of subprogram Proc_1 than was produced by the WCETE, he would extract the entire subprogram profile from the report, edit the WCET statement for Proc_1 (eg. 90, 110, 15), and then insert the profile into the WCETE-specific section of the User Configuration File. This would have the effect of preserving all the other data in the profile which is not to be overridden.

   Note however that this technique should be used with care since any edits to the body of procedure Foo would cause the user-supplied profile data to become inaccurate.

6 Warning and Error Messages

6.1 Compile-Time Messages

The WCETE may generate the following errors, warnings, and informational messages during compile-time processing. The messages are merged with other messages from the compiler and will be output to the destination specified in the compiler's OUTPUT option (see figure in section 4.2.1). Messages relating to a specific construct in the Ada source are output accompanied by the relevant source line or lines with the construct clearly flagged. [Aside: The messages relating to WCET subset errors and warnings may also include further information about the illegal construct.]

Note that the detection of a WCETE error does not cause the compilation to fail, but does result in the compilation unit being flagged in the Ada library as being unsuitable for use with the HRT tools. If the WCETE is later enabled during the bind of an application including the unit, a WCETE bind-time error will result and no Execution Skeleton will be generated.

6.1.1 Compile-Time Errors

WCET subset error: This construct is not permitted in the WCET Ada Application Restrictions.

WCET subset error: This construct is not permitted in the WCET Ada HRT Restrictions, but has been used in the main body of a critical task.

WCET error: Critical task *T* makes a call to an entry in passive task *PO* which has been marked as being callable only by non-critical tasks.

WCET error: The Instruction Timings File is missing or corrupt.

WCET error: No upper bound for execution of this loop has been provided.

WCET error: All possible flows of control at this point have been excluded from WCET timing consideration.

6.1.2 Compile-Time Warnings

WCET subset warning: This construct is not permitted in the WCET Ada HRT Restrictions. The containing subprogram must not be called within the main body of a critical task.

WCET subset warning: No upper bound for execution of this loop has been provided. The containing subprogram must not be called within the main body of a critical task.

WCET warning: The upper bound specified by pragma for this loop is greater than the actual upper bound and has been ignored.

WCET warning: This sequence of statements terminates with a raise statement and has been excluded from worst case execution time calculation.

WCET warning: This set of exception handlers has been excluded from worst case execution time calculation.

6.2 Bind-Time Messages

The WCETE may generate the following errors, warnings, and informational messages during bind-time processing. The messages are output as part of the User Report file. Messages relating to the User Configuration File are output accompanied by the relevant source line or lines.

Note that the detection of a WCETE error does not cause the bind to fail, but does prevent the generation of the Execution Skeleton.

6.2.1 Bind-Time Errors

WCET error: Subprogram *S* in unit *U* was not compiled with WCETE processing enabled.

WCET error: WCET subset errors were detected in unit *U* at compile-time.

WCET subset error: WCET subset errors were detected in subprogram *S* in unit *U* at compile-time.

WCET error: Critical task *T* makes a call to an entry in passive task *PO* which has been marked as being callable only by non-critical tasks.

WCET subset error: Subprogram *S* in unit *U* may be called recursively.

WCET error: The User Configuration File cannot be read.

WCET error: The User Configuration File has invalid data.

WCET error: Subprogram *S* in the User Configuration File is overloaded and cannot be unambiguously identified.

WCET error: No timing data has been supplied for interfaced subprogram *S* in the User Configuration File.

6.2.2 Bind-Time Warnings

WCET warning: Subprogram *P* has a null body but no timing data has been supplied for it in the User Configuration File.

WCET warning: Subprogram *P* has been inlined and the override timing specified in the User Configuration File has been ignored.

Annex A

Prototype Definition

This section defines the subset of the full WCETE functionality which is supported by the rapid prototype version delivered from contract [R13].

A.1 Task Templates

The prototype implementation supports all three forms of critical task as defined in section 2.4.3 (*Cyclic*, *Software Sporadic* and *Interrupt Sporadic*), and both forms of passive task (*Resource Control* and *Synchronisation*).

A.2 User Pragmas

Of the three additional pragmas defined in section 3.1, only **pragma** Passive is supported by the prototype to generate Protected Objects.

Pragma Loop_Count is not supported, hence all source code loops executed by critical tasks must have a statically-known maximum number of iterations. This implies that **while** loops and infinite loops containing embedded **goto**, **exit** or **return** statements cannot be used in the execution of a critical task or protected object. The workaround is to use a **for** loop with a static iteration definition equal to the Loop_Count value, ie. which represents the worst case number of iterations eg.

Replace:
```
MAX_ITER : constant := 30;
while PTR /= null loop
    pragma Loop_Count (MAX_ITER);
    ...
    PTR := PTR.NEXT;
end loop;
```

By:
```
MAX_ITER : constant := 30;
for I in 1..MAX_ITER loop
    exit when PTR = null;
    ...
    PTR := PTR.NEXT;
end loop;
```

Note that unconditional **exit** statements are excluded via the WCET Ada HRT Restrictions. However use of conditional **exit when** <expression> is supported for *non-static* expressions except when it occurs immediately after an **end loop** for an inner loop. Thus the effect of an unconditional **exit** statement can usually be achieved for example via **exit when** Zero=0; where Zero is an integer variable initialised to 0.

Pragma Exclude_WCET is also not supported, and so the timing reflects the worst case path through the code. Implicit application of **pragma** Exclude_WCET, to exclude all paths which end with an unconditional exception raise, and all exception handlers, is performed by the prototype. For any critical task body main loop and any subprogram called by that loop body code, if all execution paths are thus implicitly excluded, then a WCET equal to zero time is generated. If the user wishes to exclude any execution path which does not end with an unconditional exception raise, then this should be temporarily commented out (eg. using a source pre-processor) whilst timing measurements are being obtained.

A.3 Restriction Enforcement

The WCET Ada Application Restrictions defined in section 3.2.1 are not enforced by the prototype, and so no warning or error messages relating to violations of these restrictions are generated.

The following WCET Ada HRT Restriction defined in section 3.2.2 is also not enforced by the prototype:

- A passive task may be used solely by critical tasks or solely by non-critical tasks, but not by a mixture of the two.

- The entry call in the outer loop of a software sporadic task must be the only entry call to the guarded entry of a passive synchronization task.

The following WCET Ada HRT Restrictions defined in section 3.2.2 **are** enforced by the prototype:

- No operations which may suspend the task, other than passive task guarded entry calls and calls to Real_Time.Delay_Until, are permitted while a critical task is executing its main loop. In addition these calls may only be made in specific contexts.

- All code executed by a critical task while in its main loop must have bounded execution time. In particular, critical tasks may not make use of the Ada program heap, for example via allocators or dynamically-sized object creation, and may not perform input-output operations. All loops must have a statically known maximum number of iterations, the use of the **goto** statement is not allowed, recursive subprogram calls must not be made and exceptions must not be raised other than in error paths.

The prototype implementation allows only the following runtime calls during the execution of the main loop in critical tasks:

- All operations in predefined package Real_Time are permitted except:

  - function *Clock*
  - functions *To_Duration* and *To_Time_Span*
  - procedure *Delay_Until* (*Calendar.Time*)

  [Aside: *Clock* is intended only to be called before the main loop starts, rather than during the main loop execution, and all timing is intended to be performed using types *Real_Time.Time* and *Real_Time.Time_Span*, hence no support for timing using *Calendar.Time* or *Duration*.]

- Calls to any passive resource control task (see section 2.4.3)

- An operation matching one of the critical task templates (see section 2.4.3) with the proviso that for an *Interrupt Sporadic* task, the **accept** statement for the interrupt entry must have a null body.

There are no such restrictions on non-critical code, for example that which is executed by a task in its task body prior to the infinite loop, and that which forms the elaboration code of the library units in the program. This allows total freedom during program start-up to create all the needed data structures and tasks prior to the execution of the critical code. Likewise any non-critical tasks and their called subprograms can perform any operation at any time, although the timing estimates will

be compromised if any violation of the WCET Ada Application Restrictions defined in section 3.2.1 occurs.

In addition to Ada operations relating to tasking (eg. creation, delay, abort, rendezvous and termination), heap management (eg. allocators and Unchecked_Deallocation), exception management (raising and handling), and input-output, runtime system support routines are also used to implement a number of other Ada operations. Since all calls to the runtime system in critical code are prohibited by the prototype, these operations are therefore excluded. The list of such excluded runtime operations follows:

- All exponentiation operators
- All block move operations, eg. whole array or record assignment
- All block logical operations (**and**, **or**, **xor**, **not** applied to an array of boolean components)
- All block relational operations (eg. "=", ">" applied to array or record objects)
- All bit-level operations (eg. load, store, "=", ">", **and, not** applied to components defined at a particular bit offset by a record representation clause)
- Fixed-point multiplication and division
- Integer multiplication, division, **mod** and **rem**

The workaround for exponentiation is to use repeated multiply in a loop, as long as the power is a static value. For a dynamic power, a static-bounded loop with an embedded **exit when** statement can be used to achieve the correct effect.

In general, the workaround for block operations on arrays is to explicitly code a loop operating on one element at a time. Since arrays used by critical code need to have static bounds, the loop iteration definition will also be static. Similarly for a record object, a block operation such as assignment can be worked-around by performing the operation explicitly on a component-by-component basis.

There is no easy work-around for bit-level operations. Record representation clauses placing components on bit boundaries should be avoided for record types which are manipulated by critical code.

For fixed-point and integer multiplication and division, a work-around is to convert the operands to floating point for the operation, and then convert back the result.

For integer **mod** and **rem**, the normal case of requiring a counter to cycle from 0 up to the required modulo can be worked-around by overloading the predefined **mod** operation, eg.

```
function "mod" (Left, Right : Integer) return Integer is
begin
    if Left = Right then return 0; else return Left; end if;
end "mod";
pragma Inline ("mod");
```

The function can then be called in the normal way, eg. Index := Index **mod** Limit + 1;

A.4 User Options

Regarding the WCETE-specific Compiler invocation options described in section 4.2.1, option *HRT_DATA* in the *KEEP* menu is supported by the prototype, and its setting to *YES* is the means by which the WCETE compile-time processing is obtained. Neither of the other two WCETE-specific Compiler options (*TIMES* and *TDF*) are supported by the prototype.

Regarding the WCETE-specific Binder invocation options described in section 4.2.2, option *WCETE* in the *OPTIONS* menu is supported by the prototype, and its setting to *YES* is the means by which the WCETE bind-time processing is obtained. Text field *SKELETON* in the WCETE section is also supported by the prototype to name the output file containing the Execution Skeleton. None of the other three text fields *UCF, REPORT,* and *FILTER* are supported by the prototype.

A.5 Input and Output Files

Neither of the two input files specified in section 5.1 (the Instruction Timings File and the User Configuration File) are supported by the prototype. The times in cycles of the ERC32 instruction set is built into the prototype data structures and cannot be modified by the user. There is no support for user-supplied overrides for subprogram and thread profiles via the UCF, and so this must be achieved by hand-editing the generated Execution Skeleton ASCII text file if necessary.

Regarding the two output files specified in section 5.2 (the Execution Skeleton and the Report File), only the Execution Skeleton is generated by the prototype. This is the primary output of the WCETE capability for input to the Schedulability Analyzer and Scheduler Simulator.

A.6 Error and Warning Messages

The following error and warning messages defined in sections 6.1.1, 6.1.2, 6.2.1 and 6.2.2 are generated by the prototype. For compiler messages, if the reason for the message is not obvious then the user should obtain an assembler listing for the compilation unit (using option ASSEMBLER => CODE) and should check the relevant section of generated code for calls to the run-time system (a **call** instruction to an entry point in ALSYS_ADA_RUNTIME). As indicated in section A.3, the only run-time calls which are allowed for critical tasks and their called subprograms are to MONITOR_LOCK, MONITOR_UNLOCK, MONITOR_WAIT and MONITOR_SIGNAL, plus most of the operations in package REAL_TIME.

Compiler

WCET subset error: This construct is not permitted in the WCET Ada HRT Restrictions, but has been used in the main body of a critical task.
[The compile-time processing has detected use of a restricted construct (see A.3) inside the infinite loop in a critical task body]

WCET subset error: No upper bound for execution of this loop has been provided.
[The compile-time processing has detected use of a loop with non-static bounds inside the infinite loop in a critical task body]

WCET subset warning: This construct is not permitted in the WCET Ada HRT Restrictions. The containing subprogram must not be called within the main body of a critical

task.
[The compile-time processing has detected use of a restricted construct (see A.3) inside a subprogram body which might be called by a critical task]

WCET subset warning: No upper bound for execution of this loop has been provided. The containing subprogram must not be called within the main body of a critical task.
[The compile-time processing has detected use of a loop with non-static bounds inside a subprogram body which might be called by a critical task]

Binder

WCET error: Subprogram *S* in unit *U* was not compiled with WCETE processing enabled.
[The bind-time processing has detected that a critical task calls a subprogram directly or indirectly which was not compiled using the *HRT_DATA* option]

WCET error: WCET subset errors were detected in unit *U* at compile-time.
[The bind-time processing has detected that a compilation unit exists in the closure of the program which generated WCET subset error(s) at compile-time]

WCET error: WCET subset errors were detected in subprogram *S* in unit *U* at compile-time.
[The bind-time processing has detected that a critical task calls a subprogram directly or indirectly which generated WCET subset warning(s) at compile-time]

WCET subset error: Subprogram *S* in unit *U* may be called recursively.
[The bind-time processing has detected that a critical task calls a subprogram directly or indirectly which may call itself recursively]