32bit microprocessor and computer system development

Phase 2b

ERC32 Software Development Toolset User Manual

Compiled by:	:	.....
		Paramjit Singh, Andy Walter
Reviewed by:	:	.....
		Paramjit Singh
Authorised by:	:	.....
		Andy Walter
Logica ref:	:	505.21197
Document No:	:	MCD-LOG-P2-TUM

Logica UK Ltd, Space Division

Wyndham Court, 94 Portsmouth Road, Cobham, Surrey, KT11 3LG

DOCUMENT TITLE:
ERC32 Software Development Toolset User Manual
DOCUMENT NUMBER:
MCD-LOG-P2-TUM

DOCUMENT STATUS SHEET			
ISSUE	REV	DATE	REASON FOR CHANGE
0	-	17-01-97	Initial version - draft for review
1	0	15-03-97	First issue following review

DISTRIBUTION		
COPIES	NAME	COMPANY
	T Vardanega	ESTEC
	R Svenningsson	Saab Ericsson Space
	A Larkham	Alsys
	C Piron	Spacebel
	P David	MMS (F)
	A Walter	Logica
	File	Logica

DOCUMENT CHANGE LOG		
Section	issue/rev	REASON FOR CHANGE
All	1/0	Update following comments from ESTEC and Saab Ericsson

Table of Contents

1. INTRODUCTION 1
1.1 Purpose 1
1.2 Scope 1
1.3 Overview 1
1.4 Structure 2
1.5 Reference Documents 3
1.6 Definitions and Acronyms 5
1.6.1 Acronyms 5
2. THE HARD REAL-TIME (HRT) SOFTWARE DEVELOPMENT 7
2.1 Hard Real-Time Software Life Cycle 7
2.2 Support for The HRT Software Lifecycle 9
2.3 Hard Real-Time Software Development 10
2.3.1 Preparation for HRT Analysis 12
2.3.2 Generation of Worst Case Timing Information 13
2.3.3 Analysis of Application 13
2.3.4 Simulation of Application Scheduling 14
2.3.5 Generation of Application 14
2.3.6 Testing of Application by Simulation 14
2.3.7 Testing of Application on Target 15
2.4 Hard Real Time Software Elements 15
2.4.1 Cyclic Threads 16
2.4.2 Sporadic Threads 16
2.4.3 Protected Objects 16
2.4.4 Thread Attributes 17
2.4.4.1 Period 17
2.4.4.2 Inter-Arrival Time 17
2.4.4.3 Deadlines 17
2.4.4.4 Criticality 17
2.4.4.5 Offsets 18
3. PHYSICAL COMPONENTS OF THE 32 BIT TOOL SET 19
3.1 The 32 Bit Microprocessor Tool Set 19
3.1.1 The Ada Compilation System Tool 20
3.1.2 The Schedulability Analyser Tool 20
3.1.3 The Scheduler Simulator Tool 21
3.1.4 The ERC32 Simulator Tool 21
4. HRT APPLICATION IMPLEMENTATION 23
4.1 Cyclic (Periodic) Thread Templates 24
4.1.1 Cyclic (Periodic) Thread Ada Template 24
4.1.2 Cyclic Thread Execution Profile Template 26
4.1.3 Cyclic Thread Definition Template 26

4.1.3.1 Hard/Soft Cyclic Thread Definition Template: 27
4.1.3.2 Non-critical Cyclic Thread Definition Template 27
4.2 Sporadic Thread Templates 27
4.2.1 Sporadic Thread Ada Templates 27
4.2.1.1 Interrupt Sporadic Thread Ada Template 28
4.2.1.2 Software Sporadic Thread Ada Template 29
4.2.2 Sporadic Thread Execution Profile Template 30
4.2.3 Sporadic Thread Definition Template 30
4.2.3.1 Interrupt Sporadic Thread Definition Template 30
4.2.3.2 Software Sporadic Thread Definition Template 30
4.3 Protected Object Templates 30
4.3.1 Protected Object Ada Templates 31
4.3.1.1 Synchronisation Protected Object Ada Template 31
4.3.1.2 Resource Control Protected Object Ada Template 33
4.3.2 Protected Object Definition Templates 34
4.3.2.1 Synchronisation Protected Object Execution Profile Template 34
4.3.2.2 Resource Control Protected Object Execution Profile Template 35
4.3.3 Protected Object Definition Template 35
4.4 Application Level Templates 35
4.5 Run-Time System (RTS) Characteristics 36
4.6 Priority Determination 37
4.6.1 Priority Calculation 37
4.6.2 Priority Assignment 38

5. USING THE TOOL SET 40

5.1 Preparation for HRT Analysis 40
5.1.1 Thread and Protected type identification 40
5.1.2 Application definition in ERC32 tool set 44
5.2 Generation of Worst case Timing Information 45
5.3 Calculation of Thread Priorities 45
5.4 Analysis of Application 47
5.4.1 Period Transformation 55
5.4.2 Alternate Analysis 56
5.5 Simulation of Scheduling of Application 62
5.6 Generation of Application 64
5.7 Testing of Application by Simulation 64
5.8 Testing of Application on Target 65

APPENDIX A: EXAMPLE OF A TYPICAL UCF 1

A.1 Schedulable UCF 1
A.2 UCF Additions for a Sporadic Thread 1
A.3 UCF modifications for a Periodic Thread 2

APPENDIX B: EXAMPLE OF A TYPICAL ESF 1

B.1 Schedulable ESF 1
B.2 ESF Additions for a Sporadic Thread 2
B.3 ESF modifications for a Periodic Thread 2
B.4 ESF modifications for Alternate analysis 2

APPENDIX C: THE 32 BIT MICROPROCESSOR TOOL SET ENVIRONMENT 1
C.1 Environmental Requirements 1
C.2 Installation of the 32 Bit Microprocessor Tool Set 1

1. INTRODUCTION

1.1 Purpose

This document is the reference guide for users of the 32 Bit Microprocessor Software Tool Set. It presents a high level view of the usage of the tools developed to assist in the generation of embedded Hard Real-Time (HRT) software for on-board space applications. Its aim is to allow the user to understand the relationships between each of the tools and how they together, as a collective, can be employed to generate HRT software. It is the only user document that treats the tools as a collective and explicitly sets each individual tool within the context of the HRT software life cycle. As such it is essential reading for any potential user of the tool set and should be read prior to any of the supplementary user manuals of the individual tools.

The user is assumed to be familiar with the scheduling theories underpinning HRT software (see [RD2, RD3, RD24]) systems and to have a working knowledge of Ada 83. This manual also stresses the importance of using a software design method capable of producing analysable code (in the sense of timing analysis) from the outset and sets out the basic requirements the method must meet in order to achieve this goal. Such a method helps to ensure the most effective use of the toolset.

By its nature this user manual does not present the level of detail that is contained in the tool user manuals and these should be consulted by the user for a thorough understanding of how to use each individual tool. Hence the user does not require detailed knowledge of any of the individual tools in order to use this manual effectively.

It is emphasised that all the software tools constituting the tool set have been tailored to operate within the ERC32 processor environment (i.e. the ERC32 processor is the target processor for any HRT software developed using this tool set). Use on processors other than the ERC32 would require additional, but minimal, adaptation by the tool vendors.

1.2 Scope

This user manual restricts itself to a top level view of the software tool set; it is intended to complement the user manuals of the individual tools and therefore contains information that is not available in the user manuals of the individual tools. It does not attempt to replicate the level of detail contained within the user manuals of the individual tools.

So that the user may use the tool set most effectively, the logical functionality of the required tool set is first detailed. The intended usage of the tools in the development of embedded HRT software is reinforced by way of an ‘operational scenario’.

Suffice it to say, a description of the target processor, the ERC32, lies outside the scope of this user manual.

1.3 Overview

The support tools for the 32 bit microprocessor are aimed at supporting the development of embedded HRT software for on-board applications. Applications of this type require both

high dependability and good real-time performance, and the functionality and characteristics of the support tools are intended to support a software developer in achieving those objectives.

The conventional and widely adopted approach to implementing HRT embedded software has been to base the design on independent cyclically scheduled processes, with sufficient idle time being left in each minor cycle to allow for overruns and the handling of aperiodic events (e.g. interrupts). However, in the face of increasingly complex HRT software systems, the cyclic approach has a number of drawbacks. As the number of activities grows, the effort of defining the schedule executive or modifying it to accommodate design changes increases significantly. Also, the efficiency of the schedule drops as more idle time is introduced into each minor cycle to allow larger numbers of aperiodic events to be serviced. Moreover, the behaviour of an overrunning cyclic schedule does not degrade progressively with processor utilisation, and the execution of a critical activity may be cut by the end of a minor cycle. In essence the cyclic approach to scheduling is not well suited to non-deterministic HRT systems that invoke sporadic activities, mode changes or error recovery following a transient overload.

However, there have now been advances in hard real-time theory [RD2, RD3, RD24 and RD25] that allow the analysis of priority based pre-emptively scheduled software, and can provide guarantees of the schedulability of suitably designed software (i.e. that the software is guaranteed to always meet its deadlines). One of the primary aims of this toolset is to support the development of this type of software. The tool set has built upon the outputs from earlier ESA programmes, in particular the ESA "Hard Real-Time Operating System Kernel Study" project (see [RD2] and [RD3]), which defined a design method to support the generation of analysable software designs and refined the theory for analysing the behaviour of pre-emptively scheduled hard real-time software.

As one of the main objectives of the tool set is to provide improved support for the fuller use of Ada for on-board applications, the tool set also facilitates the use of the Ada Tasking Coprocessor (ATAC). The ATAC is a device, developed on behalf of ESA, that implements key elements of an Ada Run-Time system (RTS) in hardware, and thereby improves the performance of Ada tasking applications. The tool set functionality includes the addition of ATAC support into the Ada cross-compilation system, and provision of an ATAC simulator as part of the target simulation capabilities of the support environment.

1.4 Structure

Section 2 identifies the requirements of the hard real-time software life cycle with respect to the conventional software life cycle and takes the user, step by step, through the HRT lifecycle, pointing out what needs to be done at each step. It also introduces the essential elements of HRT software used by the toolset to provide for the analysis of source code.

Section 3 presents the functionality of the tool set and establishes the separate physical entities that constitute the tool set. It goes on to describe each of the tools and the relationships that exist between the individual tools.

Section 4 discusses in detail issues of implementation and presents a core set of templates that the user must familiarise himself/herself with in order to use the tool set.

Section 6 presents the development sequence to identify the usage of the individual tools in the development of HRT software.

1.5 Reference Documents

[RD1] ESA Software Engineering Standards
PSS-05-0 Issue 2 February 1991

[RD2] Hard Real-Time Operating System Kernel - Volume 1
TP1079 - Volume 1, February 1993
ESTEC Contract 9198/90/NL/SF

[RD3] Hard Real-Time Operating System Kernel - Volume 2
TP1079 - Volume 2, February 1993
ESTEC Contract 9198/90/NL/SF

[RD4] Schedulability Analyser and Scheduler Simulator Software User Manual
32B-SBI-SUM-0189-001/2, Issue 1, Rev 3, 20/12/96

[RD5] Target Simulator User's Manual
32B-SBI-0189-003, Issue 1, Rev 1, 08/10/96

[RD6] Ada World Development Environment for SPARC based Workstations under Solaris 2 to ERC32 Targets User's Guide
MCD-ALS-P2-DOC-003, Issue 1, Rev 0, 29/02/96

[RD7] AdaMake User's Guide
MCD-ALS-P2-DOC-012, Issue 1 revision 0

[RD8] AdaMath User's Guide
MCD-ALS-P2-DOC-016, Issue 1 revision 0

[RD9] AdaProbe User's Guide
MCD-ALS-P2-DOC-011, Issue 1 revision 0

[RD10] AdaReformat User's Guide
MCD-ALS-P2-DOC-014, Issue 1 revision 0

[RD11] AdaWorld for Motif User's Guide
MCD-ALS-P2-DOC-004, Issue 1 revision 0

[RD12] AdaXref User's Guide
MCD-ALS-P2-DOC-013, Issue 1 revision 0

[RD13] Appendix F
MCD-ALS-P2-DOC-008, Issue 1 revision 1

[RD14] Application Developer's Guide
MCD-ALS-P2-DOC-007, Issue 1 revision 1

[RD15] Cross-Development Guide
MCD-ALS-P2-DOC-015, Issue 1 revision 1

[RD16] Flexible License Manager User's Guide
MCD-ALS-P2-DOC-005, Issue 1 revision 0

[RD17] User Interface Guide
MCD-ALS-P2-DOC-009, Issue 1 revision 0

[RD18] User's Guide
MCD-ALS-P2-DOC-003, Issue 1 revision 0

[RD19] Viewing Guide
MCD-ALS-P2-DOC-010, Issue 1 revision 0

[RD20] Welcome to AdaWorld
MCD-ALS-P2-DOC-019, Issue 1 revision 0

[RD21] Installation Guide
MCD-ALS-P2-DOC-002, Issue 1 revision 0

[RD22] Project Development Guide
MCD-ALS-P2-DOC-006, issue 1 revision 1

[RD23] Worst Case Execution Time Processing Preliminary User's Guide
RAT-ALS-WP20-PUG-001, issue 2 revision 0

[RD24] HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems
Alan Burns and Andy Wellings, Elsevier, 1995, ISBN: 0-444-82164-3

[RD25] A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems
Mark Klein, Thomas Ralya et al
Kluwer Academic Publishers, 1993, ISBN: 0-7923-9361-9

1.6 Definitions and Acronyms

1.6.1 Acronyms

AD	Architectural Design (phase)
ADS	Arbitrary Deadline Scheduling
ATAC	Ada Tasking Coprocessor
BSP	Board Support Package
DMS	Deadline Monotonic Scheduling
ERC32	32bit Embedded Real-Time Computer
ESF	Execution Skeleton File
HOOD	Hierarchical Object Oriented Design (method)
HRT	Hard Real-Time
HRTOSK	HRT Operating System Kernel
ICD	Interface Control Document
INHIB	Inhibit Interrupt (priority inheritance)
IPCI	Immediate Priority Ceiling Inheritance
ISA	Instruction Set Architecture
ISR	Interrupt Service Routine
PBPS	Priority based Pre-emptive Scheduling
PO	Protected Object (equivalent to a Passive Task)
RMS	Rate Monotonic Scheduling
RTS	Run-Time System
SR	Software Requirements (phase)
UCF	User Configuration File
WCET	Worst Case Execution Time
WCETE	Worst Case Execution Time Estimator

This page intentionally left blank

2. THE HARD REAL-TIME (HRT) SOFTWARE DEVELOPMENT

The tool set supports the generation of embedded HRT software. As well as knowing how to actually use the tool set, it is important that the user has an understanding of the way in which the production of HRT software differs from the production of conventional software. This section provides a basis for this understanding by contrasting the conventional software development lifecycle, as defined by PSS-05 [RD1], against a HRT software lifecycle. It then goes on to identify the key activities to be undertaken for such a software development and identifies the key elements of a HRT software application.

2.1 Hard Real-Time Software Life Cycle

The user is assumed to be familiar with the PSS-05 conventional software lifecycle as identified in [RD1]. This lifecycle is essentially of a waterfall type, which although not precluding iteration within each phase, is sequential and unidirectional in nature, i.e. each phase is completed before proceeding to the next one in the sequence and once a phase has been completed it is not revisited again. For example, typically it is the case that once having entered the architectural phase, the user requirements phase is not re-visited.

From a HRT embedded software perspective, this waterfall approach has a number of draw backs, chief amongst which is that it does not view the software as an integral part of a larger overall system. Rather, it views the software as an end in itself and consequently does not focus explicitly on the problem of integrating the software with the hardware. For HRT embedded systems, however, the issue of integrating the software with the hardware (and with the rest of the larger overall system) is of primary importance: the operational performance of the hardware must be taken into account in producing HRT software from the outset of the design process.

In order to understand the HRT software lifecycle it is necessary to identify the features that any design method must have to support the production of HRT software. The best way to identify these features is to first consider the differences between the design processes of conventional software and HRT embedded software:

- From the outset, the design of HRT embedded software is heavily influenced by the limitations of the hardware on which it is to run.
- The design of HRT embedded software is driven by non-functional requirements and constraints originating from the external environment and/or the user (e.g. HRT requirements), which do not apply to the design of conventional software.
- For a HRT embedded system it is not always possible to derive a definitive set of requirements at an early stage of the design process whereas for conventional software it is (in particular the timing requirements in a HRT system may change due to external factors).
- When designing HRT embedded software the developer cannot be totally confident that all the real-time requirements will have been met until the software has been integrated with and tested upon the target hardware. This is in contrast to the design of conventional software where there are no such real-time requirements.

- In the design of HRT embedded system considerable effort is expended in assuring that the timing behaviour of the software is entirely predictable under all operational conditions, whereas this is of little or no concern in the design of conventional software.
- For a HRT embedded system, it is imperative that the timing requirements are verified continuously throughout the design and development process.

The above differences have a number of ramifications for any design process that is to be employed to produce HRT embedded software. Nevertheless, careful examination of the conventional software lifecycle shows that it can be adapted to meet the needs of HRT embedded software. The necessary adaptations to the conventional lifecycle are:

- The SR phase should include sufficiently accurate timing information and analysis to allow the feasibility of the requirements to be assessed and approved prior to completion of the architectural design.
- The SR phase should give consideration to timing aspects, environmental constraints and non-functional requirements.
- The SR phase should identify and define in sufficient detail (to allow timing analysis to be performed) all time-critical subsystems and processes.

These adaptations can be achieved by including specific timing analysis in the SR phase. This analysis can be supported by estimates based on software re-use, parallels drawn from the experience of the developers in producing similar systems, and the results of early prototyping activities. Prototyping is especially useful as it can both provide preliminary code timing data to help confirm the feasibility of the design and is a powerful way of for the developers to gain a close understanding of the likely problem areas. This in turn helps to improve both requirements and the design and can also speed up the process of coding during the DD phase.

The net result of the need to adapt the lifecycle is that there is a blurring of the distinction between the SR phase and the AD phase: these are effectively amalgamated into a single analysis stage during which requirements analysis and design are performed concurrently and iteratively. The emphasis moves gradually from requirements definition to refinement of the architectural design as confidence in the timing analysis increases.

The adapted conventional waterfall cycle appropriate to HRT embedded software is shown in Figure 2.1-1. At any stage in the development, iteration to the previous lifecycle phase may be required if a problem is highlighted in the feasibility of implementing any of the requirements (including timing constraints). The process is no longer unidirectional and sequential, and any phase may be revisited from any other succeeding phase.

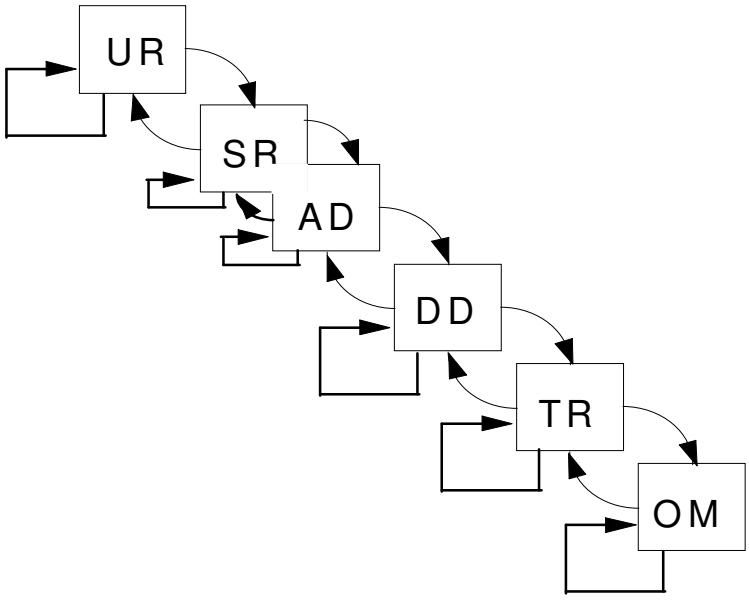

Figure 2.1-1: HRT Software Development Lifecycle

2.2 Support for The HRT Software Lifecycle

Advances in hard real-time theory [RD2, RD3, RD24 and RD25], now allow the analysis of priority based pre-emptively scheduled software, and can provide guarantees of the schedulability of suitably designed software (i.e. that the software is guaranteed to always meet its deadlines). In particular, these developments can be used to support the development at an early stage in the development i.e. SR/AD.

The tool set utilises these advances and allows the developer to satisfy the requirements of embedded HRT development by providing a mechanism for verifying the software both at a prototyping stage and during the latter stages of development.

In particular when the tool set is used in tandem with a design method which supports HRT design, such as HRT-HOOD [RD2, RD3 and RD24], a preliminary design can be produced and analysed with results fed back to the requirements analysis. In this case, the design identifies the basic software structure.

The following steps summarise the practical implementation of the HRT embedded software design process described above:

During the architectural design phase:

- a high level design for the application which is analysable via the use of the tool set is produced;
- estimates (either empirical or based on prototyping) are produced for the processing time of each analysable entity;

- the design is subject to an HRT analysis, which reports on the schedulability and the scheduling events to be expected for the application;
- the results are examined by the user to confirm the feasibility of the design, which is then refined by adding detail (e.g. subprogram calls, or making modifications to the threads);
- estimates (based on prototypes if required) are produced for the subprograms if required and the HRT analysis is repeated;
- the process of refining the design continues until the developer is sufficiently confident to proceed to detailed design;
- During the Design and Production Phase:
- the detailed design is produced and estimates/prototype values of the processing times are updated if required;
- the HRT analysis is repeated;
- the detailed design is refined and HRT analysis repeated until the developer is confident to proceed to coding;
- the application is coded and a worst case path execution time (WCET) is calculated for each thread and subprogram based on the object code produced by the compilation process;
- the HRT analysis is repeated using the calculated WCET times;
- the code is refined and HRT analysis repeated until the developer is confident to proceed to formal testing;
- a machine code version of the application is built and testing commences using a target processor simulator;
- as errors are detected, the code is corrected (the design also as required) and re-tested. This continues until the developer is confident to move to integration and testing on the actual target;
- testing continues on the target processor, including running the Acceptance Test at which the software is provisionally accepted;
- As part of the testing, timing data can be produced for comparison with the calculated time. The alternative timing data can be used to perform further HRT analysis if required.
- At any stage, the HRT analysis may reveal that significant changes to the application are needed. In this situation the development process will return to the architectural design, detailed design or coding as appropriate.

2.3 Hard Real-Time Software Development

Section 2.2 shows the overall process of development and analysis to be one of continual refinement, with iteration to previous stages in the development as required. The detailed process performed at each stage is similar and is summarised by the following procedures:

- Preparation for HRT analysis;
- Generation of worst case timing information;
- Analysis of application;
- Simulation of application scheduling.
- The sequence of events for testing are also to be split into a number of subtasks as follows:
  - Generation of application
  - Testing of application by simulation
  - Testing of application on target

This sequence is shown in Figure 2.3-1 below:

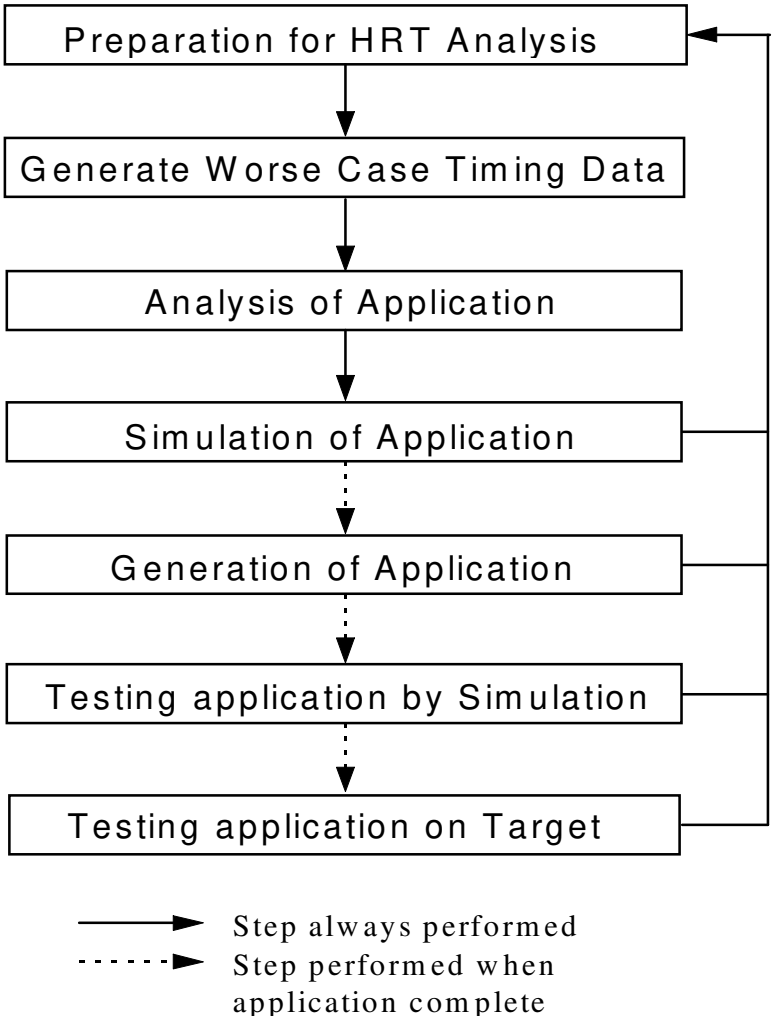

Figure 2.3-1: HRT analysis and simulation process

Each of the tasks identified above is described in more detail in the following sub-sections.

2.3.1 Preparation for HRT Analysis

In order to perform an HRT analysis, it is first necessary to define the application in terms its constituent threads of execution and the communication between those threads, which is performed through the use of protected objects.

The user defines the application by using methods and tools that are compatible with producing analysable HRT software (e.g. HRT-HOOD [RD24], which partitions active objects into sporadic and cyclic activities, and identifies any protected objects required for data sharing/transfer). The level of detail defined for any particular HRT analysis depends on the stage of development. Initially, the user only has to define the application in terms of a set of package and task specifications and an outline of the corresponding bodies. As the development progresses, the user refines this outline by including lower level procedures and functions, and he/she does this until all code is produced for the application.

When defining the application the user provides a specification of the scheduling characteristics (i.e. the timing constraints and criticality) of the threads. Before performing an HRT analysis, priorities, which depend on the scheduling algorithm and the thread characteristics, are assigned to the threads and protected objects. The scheduling algorithm to be used also dictates the characteristics which must be identified. For priority based pre-emptive scheduling (PBPS), there are a number of algorithms for priority assignment and HRT analysis. Of these Deadline Monotonic Scheduling (DMS) and Arbitrary Deadline Scheduling (ADS) are supported in the tool set. The user identifies the following timing constraints for DMS and ADS:

- for cyclic threads, the period of execution and deadline for completion of execution;
- for sporadic threads, the minimum inter-arrival time between successive occurrences and a deadline.

In addition each thread is assigned a criticality by the user. This is used to impose a relative ordering on the threads in terms of the importance of meeting the deadline, with a thread being identified as either 'hard', 'soft' or 'non-critical'. 'Hard' threads are considered the most important in scheduling terms and will always be given a higher priority than 'soft' threads, which in turn are assigned higher priorities than non-critical threads.

In a typical development, the scheduling algorithm and characteristics are known during the initial design phase (i.e. for an on-board application they are actually a requirement, for example servicing particular events). Consequently, the priorities can be calculated are assigned to the threads and protected objects. Indeed, it is possible that the priorities may not change through a development. However, there are times when the scheduling characteristics may change, for example:

- when the thread set is not schedulable and a thread needs to change to be 'soft';
- a refinement of a deadline is based on application knowledge;
- the worst case timing information could then be used to influence the deadline;
- decision to change scheduling algorithm.

In these cases, priority assignment is recalculated.

At any particular stage in the development, this activity is completed when the developer decides there is a sufficient definition (or change to the definition) of the application to justify performing an analysis.

2.3.2 Generation of Worst Case Timing Information

The next stage in the process is to determine the worst case execution timing information for the threads, procedures and functions which make up an application.

The developer either estimates the worst case times, based on either the past experience of the developer or via prototyping sections of the application, or they can be generated automatically by the toolset. For the calculation, a value for the worst case is determined from the actual object code produced for the application, based on the processor cycle times for each instruction in the longest path through each thread. Each thread in an application must have a worst case time in order that the HRT analysis be performed.

The developer uses the two mechanisms at different times in the development lifecycle. In the initial design phase estimates can be used, but as the development progresses and code becomes available, calculated values are used to give a more accurate representation. In certain situations, however the developer may wish to adapt or change the calculated values: for example, when certain paths are known to be unfeasible due to mutual exclusion, or when a developer wishes to investigate future design changes. In these cases, the calculated values are reviewed and either certain paths are discounted from the calculation. The user is able to do this by manually commenting out certain paths from the Ada source code.

As the development reaches the testing phase the developer is able to get timing data from the test facilities and can use this data to scale or verify the timing values used for the HRT analysis.

2.3.3 Analysis of Application

Analysis is performed using formulae developed from academia and previous projects [RD2 and RD3]. The formulae assume a critical instant (a time at which all threads are released) and combine the worst case timing and the priority information for an application with the run-time characteristics of the Ada RTS to provide an assessment of the schedulability of the algorithm (i.e. whether each thread can be performed before its deadline). The following information about the schedulability is provided:

- which threads, if any, fail to meet their deadline;
- the margin by which threads, if any, meet their deadline;
- the overall processor utilisation;
- the utilisation of each of the threads.

The results provide the developer with a statement on the schedulability of the application upon which to base the continued development. However, before returning to continue the development process (i.e. by continuing design or coding) the developer performs a simulation of the scheduling events for the application so as to gain a more detailed understanding of the application timing characteristics. However, this more detailed examination may is not necessary if the analysis is based on estimated timing data and shows the design to be easily schedulable.

2.3.4 Simulation of Application Scheduling

The analysis only provides the developer with a result that states whether or not an application is schedulable. Typically (especially in the case where there is a failure), the developer would like more detailed information about the exact scheduling decisions that occur when the application is running. The simulation provides this information by indicating each scheduling event and the threads involved. The operation of the Ada RTS and the scheduling algorithm used by the application are both simulated.

The output of a simulation is displayed either graphically through a Gantt chart representation or textually. The following types of event are indicated on both displays:

- start and completion of thread;
- suspension and resumption of thread;
- entry and exit of protected object;
- indication of thread's current priority;
- release of thread and associated deadline;
- missing of thread deadline;
- context switches;

Simulation is also used to investigate the behaviour of the application under conditions other than worst case (e.g. to establish the sensitivity of the application to the pattern of sporadic events).

Based on the results of the analysis and simulation, the developer makes an assessment on whether the current design for the application will meet its timing constraints.

In the case when the results are satisfactory, the development continues with further design, coding or testing as appropriate.

When the results are not satisfactory the developer reviews the design of the application to assess whether it can be re-designed or re-coded in a more optimal manner.

2.3.5 Generation of Application

Having completed the design and coding necessary for HRT analysis and schedule simulation of the application the next stage is to commence testing. In order for testing to start, the application is completed by coding any parts of the application not required for the HRT analysis (e.g. non-critical code) as well as allocating a priority level to each thread within the application in accordance with the results of the analysis and scheduling simulation. A machine code version of the application is then produced by compiling, binding and linking the code.

2.3.6 Testing of Application by Simulation

The testing of an application ideally takes place on the target hardware. However, this is not always possible for economic or technical reasons (e.g. cost of producing two pieces of hardware, or concurrent hardware and software development), especially at the beginning of the development. Additionally, debugging of the application on the hardware may be more difficult as access to all data required (e.g. registers) may not be possible. Consequently, it is

essential that a simulation of the target processor is available. It allows the target environment to be modelled and for an in-depth study of the application.

In a typical testing session, a developer loads the simulator with an application and configures the simulation environment to exactly match the target environment. Then, using either a pre-defined test script or by manual commanding, the application is executed. Access to the simulation status is provided through an interface which provides source level examination of the application data and machine level access to the target, with facilities for debugging such as tracing, break pointing and stepping through the application. All interactions are logged and can then be used by the user to verify the outcome of the testing activities.

If the developer detects any errors during testing he/she iterates back to the previous threads, with the nature of the error dictating the level and detail of changes. Potentially, a design change may be required which necessitates subsequent re-analysis. However, the change could be equally restricted to a minor code change which does not justify repeating the timing analysis.

Although all testing can be performed on a simulator, acceptance testing must usually be performed on the target. Therefore, at some stage in the development testing moves to the real target environment.

2.3.7 Testing of Application on Target

On-target testing is performed to ensure that the application software runs in its final environment. Typically, this is achieved by having a breadboard containing the processor (in this case a ERC32 configuration). Testing with the target uses the same machine code as testing on the simulator and the developer has broadly similar control and examination capabilities. Consequently, the functionality and the scenario for use are similar to that described for simulation described in section 3.6.

Upon completion of testing with the target, the application is ready to be used in a system environment or actual environment. For a typical embedded application, this may mean integration with other, higher level subsystems.

2.4 Hard Real Time Software Elements

In order to understand the HRT tool set and its operation it is first necessary to understand the underlying elements which are used to describe an application.

At the fundamental level the timing analysis of HRT software is concerned with generating estimates of the worst case (maximum) time taken to complete a given thread, with each thread corresponding to an independent thread of control (i.e. an independent processing activity). Therefore, within the toolset the structure of a HRT application is subdivided into a number of elements:

- **CYCLIC** threads;
- **SPORADIC** threads;
- **PROTECTED** objects.

Effectively any HRT system is described in terms of the above elements i.e. each software object within the HRT system must be classified as either CYCLIC or SPORADIC or PROTECTED.

Obviously it is imperative that this division of the HRT system into the abstractions given above is made at the earliest stage of the lifecycle, namely the architectural design phase. The design method HRT-HOOD shows how this can be done (see [RD2, RD3 and RD24], note however that the mapping from HRT-HOOD objects to ERC32 toolset elements is not necessarily one-to-one).

Before the user can designate a thread as being of a particular type, it is necessary to know the characteristics that define each type. These are set out below.

2.4.1 Cyclic Threads

A CYCLIC is periodic in nature and models those activities within an application which repeat at well defined fixed intervals. Each CYCLIC consists of a single thread of control which is scheduled at run-time.

2.4.2 Sporadic Threads

A SPORADIC is aperiodic in nature and models those activities whose occurrence is determined by events (either external or internal) within an application (note however for the HRT analysis, two occurrences of an aperiodic thread are considered to be separated by a minimum period of time: the inter arrival time). Each SPORADIC consists of a single thread of control at run-time.

Due to the differences in handling characteristics, these SPORADIC threads are subdivided into threads driven by external events and those driven by internal software threads:

1. **Interrupt Sporadic**: this models sporadic software threads that are triggered by external sources i.e. interrupts.
2. **Software Sporadic**: this models sporadic software threads that are triggered by other software threads i.e. represent synchronisation between two threads.

2.4.3 Protected Objects

A PROTECTED object is defined to be an executable region where only one thread is allowed at a time. It has no thread of control at run-time and guarantees protection from concurrent access by enforcing mutual exclusion.

PROTECTED objects provide the mechanism for inter-thread communication by removing the need for two threads to synchronise. This is a fundamental part of an HRT system as it removes the direct interdependency that exists within the Ada 83 rendezvous mechanism and allows the system to be analysable.

The PROTECTED object is implemented as a mutually exclusive region which may be accessed by two separate threads. This is used for two subclasses of PROTECTED object:

1. **Resource Control Protected Object:** this provides the facility commonly known as a monitor or critical region, where at most one thread of control can have possession of the object. During the period that a thread has possession, it must not perform any operation which could result in it being suspended: it provides shared access to a resource. Simply put, it allows a resource to be read and written, but not at the same time by any two separate threads.
2. **Synchronisation Protected Object:** this provides a signalling facility, whereby processes can signal and/or wait on events.

2.4.4 Thread Attributes

In order to support the timing analysis of HRT systems, each of the above HRT elements are characterised by time-domain features. These time-domain features are termed the *attributes* of the object. The *attributes* are set out below.

2.4.4.1 Period

This is simply period of execution of a an activity. Only CYCLIC threads have periods associated with them.

2.4.4.2 Inter-Arrival Time

The inter-arrival time is the minimum interval of time between consecutive executions of an aperiodic activity. This time is taken to be the one corresponding to the worst case arrival frequency of a thread. Only SPORADIC threads have an inter-arrival time associated with them.

Note, however, that the tool set provides for modelling pseudo random interrupt arrival.

2.4.4.3 Deadlines

The deadline is the time by which all activities within that thread must be completed. Deadlines are assigned to both CYCLIC and SPORADIC threads.

2.4.4.4 Criticality

In general, not all threads within the system will be at the same level of criticality. Broadly speaking, the more critical an object is the 'harder' it is said to be where the 'hardness' refers to the level to which overruns of thread deadlines can be tolerated. Thus the criticality of a thread provides a measure of the 'hardness' (importance) of a thread.

Four levels of criticality are supported:

- Interrupt: This is the highest level of criticality and is reserved for threads that are classified as Interrupt Sporadic objects. The deadlines for this type of thread cannot be missed and therefore they are awarded the highest priority within the system (i.e. considered to be the most important in terms of meeting their deadlines).
- Hard: This is the second highest level of criticality and is assigned to threads for which an overrun of deadlines cannot be tolerated. Both SPORADIC and PERIODIC threads can be given a criticality level of 'hard'.

- Soft: This is the third highest level of criticality and is assigned to threads for which occasional overruns of deadlines can be tolerated, but is nevertheless undesirable. Both SPORADIC and PERIODIC threads can be assigned a criticality level of 'soft'.
- Non-critical: This is the lowest level of criticality and is assigned to threads that typically have no strict timing requirements. Only PERIODIC threads can be given a 'non-critical' level of criticality.

2.4.4.5 Offsets

An offset allows the execution of a thread to be delayed relative to other threads which have the same period (i.e. thread execution is offset from the nominal period expiry). For example, the user may wish to deliberately delay the execution of a thread due to an inter-dependence on completion of processing by another thread. In this case the dependent thread should have an offset.

It is however important to note that although the tool set supports the notion of offset times, these are only used for simulation purposes; for HRT analysis the thread is assumed to execute at a given period with no offset. This ensures that if the application is schedulable when there is no offset, then it will always be schedulable when the offset is non-zero.

3. PHYSICAL COMPONENTS OF THE 32 BIT TOOL SET

3.1 The 32 Bit Microprocessor Tool Set

The tool set consists of:

1. Ada Compilation System Tool;
2. Schedulability Analyser Tool;
3. Scheduler Simulator Tool;
4. ERC32 Simulator Tool.

In addition, through the Ada Compilation System, the user may invoke the Worse Case Execution Time Estimator (WCETE) functionality.

The toolset architecture is provided in Figure 3.1-1 below:

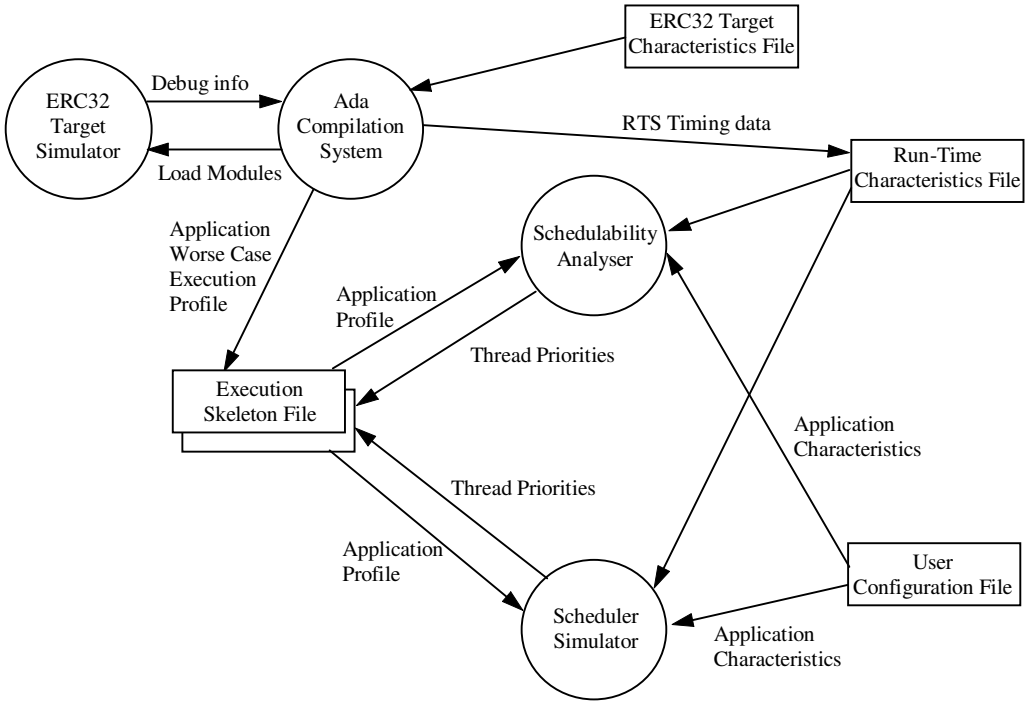

Figure 3.1-1: ERC32 Toolset Architecture

A number of files are used to provide the information transfer between the tools and also to allow user input to the toolset. The purpose of these are as follows:

- Target Characteristics File: provides the timing characteristics of the SPARC V7 instruction set in terms of clock cycles. This is used in the calculation and generation of worse case timing. Typically, this would be produced by the chip manufacturer.
- Run Time System File: provides the timing information for the Ada RTS operations. This is produced by the ACS vendor.

- User Configuration File: provides the input for the application and actual target characteristics, e.g. number of read and write wait states, clock frequency, thread period or minimum inter-arrival time. This file is provided by the user, who may generate the data from the application design, if HRT-HOOD is used.
- Execution Skeleton File: provides the application profile for schedulability analysis and simulation. It represents the worse case execution path for each thread and is generated by the WCETE functionality or input directly by the user. Note, this file appears as a multiple instance as the HRT tools automatically calculates and outputs the thread priorities.

A summary of the functionality of each of the tools follows:

3.1.1 The Ada Compilation System Tool

The ACS allows for compilation, binding and linking of the application source code. The compiler supports Ada 83, with specific modification and extensions introduced to support HRT analysis. These include pragma passive for the implementation of protected objects which correspond to Ada 95 protected types and the tailoring of the RTS to ensure time bounding of all RTS operations.

The tool also provides full debugging capability via an interactive debugger, which may run on either the ERC32 Simulator Tool, developed as part of the tool set, or on the target ERC32 processor. The ACS is fully configurable for any ERC32 configuration through a BSP which may be tailored by the user.

The ACS also contains the WCETE functionality. This tool calculates the maximum time taken for execution of each individual thread (Ada task) within the application, with the calculation itself being based upon the analysis of the source code provided by the user. The output of the WCETE functionality is the Execution Skeleton of the application, which is contained within the Execution Skeleton File i.e. the Execution Skeleton File is generated automatically whenever the WCETE functionality is invoked. The user may invoke this functionality, in isolation, without having to generate object code, and this is particularly useful during the early stages of the development as there is typically little detailed source code available at this time. Note if the user chooses not to invoke the WCETE functionality, the Execution Skeleton File (ESF) will have to be generated manually.

3.1.2 The Schedulability Analyser Tool

The Schedulability Analyser tool provides functionality to assign priorities to the threads and to indicate the schedulability of an application. Two scheduling algorithms are supported by this tool:

1. Deadline Monotonic Scheduling (DMS);
2. Arbitrary Deadline Scheduling (ADS).

Each of the above scheduling schemes must be augmented by a blocking protocol and therefore the Schedulability Analyser tool supports two types of blocking protocol:

1. Inhibit Interrupt (INHIB);
2. Immediate Priority Ceiling Inheritance (IPCI).

The user selects the scheduling approach (DMS or ADS) and the blocking protocol (IPCI or INHIB) against which the application is to be analysed along with the thread scheduling characteristics (i.e. period, deadline, minimum inter-arrival time, offset and criticality). From this input the tool calculates the priority for each thread and updates the execution skeleton for the application being analysed.

The tool calculates the schedulability of an application. The calculation uses characteristics such as the clock and the task management timings of the Ada RTS. This information is input from the RTS characteristics file produced by the compiler vendor as part of the 32 bit microprocessor tool development. A report on the schedulability of the application and each thread is output for the user.

3.1.3 The Scheduler Simulator Tool

The Scheduler Simulator tool provides functionality to allow the user to perform a detailed examination of the scheduling events of an application. If the schedulability analyser tool has been run, the Scheduler Simulator tool uses the previously calculated priorities, otherwise the priorities are calculated. The user selects the scheduling algorithm against which the application is simulated.

It should be noted that whereas the Schedulability Analyser assumes that a software sporadic has a minimum inter-arrival time, the Scheduler Simulator models its actual behaviour. Thus the Scheduler Simulator supports a more realistic analysis of the operation. In addition, to allow detailed investigation of the application behaviour, the Scheduler Simulator also provides the user with the opportunity to supersede the minimum inter-arrival times specified for interrupt sporadics by one of a number of distribution models.

The simulation is then performed according to the user inputs and also according to the characteristics of the Ada RTS which is part of the Ada Compilation System. Each scheduling event is output so that the user may both monitor the simulation and subsequently view the output either graphically or textually. In addition, the user is able to suspend and resume the simulation at any stage in order to examine any particular events which have occurred (e.g. an unexpected missed deadline).

3.1.4 The ERC32 Simulator Tool

The ERC32 Simulator tool provides the functionality to simulate on-board computer based on the ERC32 system target processor at register level allowing the execution of an application. The tool has two modes of operation, attached to the debugger and stand-alone. In the debugger mode the tool runs an application under the control of the Ada Compilation System debugger and interacts through the target monitor used when target debugging. In stand-alone mode the user has a direct command line interface. The simulation models the behaviour of the ERC32 system core and the peripherals on the ERC32 hardware, including an option to simulate the ATAC device. It is expected to provide a performance capability in terms of processing time of a minimum of 1/100 (although 1/10 is within reach of modern workstations).

This page intentionally left blank

4. HRT APPLICATION IMPLEMENTATION

Central to the concept of the 32 Bit Microprocessor Software Support Tool Set is HRT theory. HRT theory, however, lies beyond the scope of this document and is well documented elsewhere [RD2 and RD3]. Nevertheless an understanding of the implementation of key parts of HRT theory within the tool set is essential for effective use of the tool set. This section details that implementation and in so doing stresses the crucial role of Ada *tasking* in implementing the HRT abstractions.

Section 2.4 introduced the fundamental elements used within HRT systems; these are revisited below and their implementation within the tool set is described via three types of templates:

- The Ada Template. This is a template of the implementation of a particular HRT element using Ada 83, augmented by any necessary modifications/extensions. All Ada Templates are semantically fully upward-compatible (i.e Ada 95 semantics are used for the constructs, however for Passive Tasks will need to be re-coded as Ada 95 Protected Types). All HRT abstractions have an Ada template associated with them. Note that Ada *tasking* is employed in all Ada Templates.
- The Thread Definition Template. This is a template, specific to each thread, which is used to define the thread characteristics for the HRT tools. These definitions are contained in the UCF. Note that not all HRT elements have a Thread Definition Template. They are not required for Resource Control Protected Objects and Synchronisation Protected Objects as these have no specific timing or criticality characteristics.
- The Execution Profile Template. This is a template, which defines the execution profiles of each of the HRT elements for the Schedulability Analyser and Scheduler Simulator. The execution profiles are contained in the ESF and are generated either automatically by the WCETE functionality or manually by the user. If manual generation of the profiles is required it is incumbent upon the user to define the execution profile of each thread using the appropriate Execution Profile Template.

  In the case of threads (i.e. SPORADIC or CYCLIC), the execution profile of each thread is described as a sequence of worst case execution times and calls to protected objects which can potentially cause a pre-emption of the thread (i.e. calls to the RTS resulting in re-scheduling or entry calls). The worst case execution times represent a concatenation of the processing times of the object code preceding any call to a protected object. They are calculated according to the worst case path for a thread, which in turn is based on the worst case paths of the operations (i.e. Ada functions, procedures and entry calls) which make up the thread.

  As well as identifying all protected objects lying on the worst case execution paths, all protected objects lying on all other execution paths (i.e. not worst case paths) are identified. This is required in order to automatically establish the relationships between the protected objects and threads and thus determine the priority for both, one of the features of the HRT tools. In addition, it also allows the user to verify, if he/she so desires, the timing analysis of ‘non-worst case’ scenarios (i.e. the *alternative* analysis).

It is stressed that the user must adhere fully to all the templates presented in the ensuing sections if an analysable HRT system compatible with the tool set is to be produced. The templates allow the tool set to extract the relevant information automatically. Consequently, any deviations from the templates will result in a system that cannot be guaranteed to be analysable and will therefore invalidate the results from the tool set.

The templates do not however impose onerous constraints on the user. Rather the templates represent the natural structure that would be expected from the various thread types. For example, cyclic threads are coded with a delay statement as the first statement in the outer loop of the task body and sporadic threads use a barrier on a protected object to control the execution of the outer loop. In keeping with the general approach to embedded systems all threads are constructed as infinite loops;

For reasons of clarity the formal generalised syntactical representation of the templates is not given below; this is to be found in the Schedulability Analyser and Scheduler Simulator Software User Manual [RD4] and the Ada manuals [RD13, RD14 and RD15]. Rather, the templates are presented in a form closer to what would be found in a practical HRT application. The templates do, of course, strictly observe the syntax presented in the manual [RD4].

For the templates it is assumed that the user is familiar with Ada and in particular its task and package structures. Note, that no guidance is provided on the overall structure of the application and this is left to the discretion of the user, who may use a structure as identified by the HRT-HOOD method [RD2, RD3 and RD24].

4.1 Cyclic (Periodic) Thread Templates

This section provides the templates along with annotations used in the toolset for the representation of cyclic threads.

4.1.1 Cyclic (Periodic) Thread Ada Template

The key feature of the Ada thread template is the main loop which is used to encapsulate the periodic processing which is performed by the thread. Prior to the actual main loop is the prologue processing where the task will initialise any thread specific configuration and will calculate the start time for the main processing.

The HRT analysis and simulation use the concept of a critical instant, a point at which all threads are assumed to be released at the same time. As a consequence any two threads which share the same frequency of execution will always be considered to be released at the same instant (in fact, any two threads will be considered to have a common release at a rate equal to the lowest common multiple of their periods).

Therefore, in order to ensure that the analysis and simulation reflects the actual application scheduling event sequence, the user should adopt this common release point in the Ada task implementations. Failure to take this approach does not invalidate the HRT analysis, which by adopting this approach ensures a worse case assessment. Rather, it produces a higher degree of pessimism in the HRT analysis and simulation results and means that there will be a discrepancy in the scheduling events generated by the Scheduler Simulator when compared against the actual application.

The critical instant can be easily reflected in the application structure by using a common timestamp for the initial release for all cyclic threads. The timestamp should ideally correspond to a system timing datum. For example, the Huygens On-Board Software requirements include a 15 second period during which the software is expected to perform some initialisation and incorporate any patches before the main processing is started.

The priority of the task is set according to the report by the Schedulability Analyser/Scheduler Simulator. It should be included in the Ada code following an initial HRT assessment.

The cyclic thread template is as follows:

```
-- specification of the task including the priority
with SYSTEM;
with GLOBAL;
TASK Cyclic_Identifier IS
   PRAGMA PRIORITY (Priority_Integer);
END Cyclic_Identifier;

-- definition of the body of the task
with REAL_TIME;
TASK BODY Cyclic_Identifier is
   [declare local variables]
   -- variables for controlling the period of execution
   T : REAL_TIME.TIME;
   TASK_PERIOD : REAL_TIME.TIMESPAN :=
                    REAL_TIME.MILLISECONDS(task_period);
BEGIN
   [perform any initial actions]
   -- set up start time T to global start time set up elsewhere
   T := GLOBAL.START_TIME;
   Period_Action:
      LOOP
         DELAY_UNTIL (T);
         [execute periodic actions]
         T : = T + TASK_PERIOD;
      END LOOP Period_Action;
END Cyclic_Identifier;
```

where *Priority_Integer* is an integer representing the priority of the task and can be stored as part of an array of constants corresponding to the priorities of all cyclic tasks;

task_period is an integer representing the period of the task (i.e. a multiple of the basic clock period) and can be stored as part of an array of constants corresponding to the periods of all cyclic tasks.

The above assumed that a global start time is used for all cyclic threads. This should be set up as part of the general application initialisation and can be performed as follows:

```
with REAL_TIME;
PACKAGE GLOBAL IS
   START_TIME : REAL_TIME.TIME
                    := REAL_TIME.CLOCK +
                         REAL_TIME.MILLISECONDS(start_time);
END GLOBAL;
```

where *start_time* is an integer representing the desired global start time for cyclic tasks.

Note, this approach will not provide an exact time from power-on, rather it provides a time reference from the moment that the Ada RTS initialises its clock mechanism. This is because for any Ada application, there will be a time delay between processor power-on and

initialisation of the Ada environment (e.g. the processor will typically perform some self-test and will require some configuration before the RTS is initialised and run). If a precise start-up from power-on is required then the application should be timed (e.g. using a Logic Analyser) from power-on until the RTS clock initialisation and this should then be subtracted from the *start_time* value.

4.1.2 Cyclic Thread Execution Profile Template

The execution profile for the cyclic threads is automatically generated by the WCETE functionality when the template in section 4.1 is used. In situations where this functionality can not generate the profile the user should use the following template:

```
THREAD Cyclic_Identifier
   TYPE CYCLIC          -- identifies the thread type

-- this section corresponds to the main loop, the ordering should
-- represent the worst case path and be a sequence of the following
-- statements.
   WCET Cp, Cmr, Cmw      -- statement sequence unbroken by entry calls
   CALL_PO POName EntryName     -- call to a passive task entry
   LOOP Integer           -- construct to provide for loops within the
      LoopBody            -- main processing loop
   END

-- this section provides list of all other passive task entry
-- entry calls (i.e. those not on the worse case path
   PO POName EntryName    -- passive task entry call
END Cyclic_Identifier
```

where C_p , C_{mr} , C_{mw} are the execution time in processor clock cycles i.e. an integer; the count of memory reads and the count of memory writes;

CALL_PO signifies a call to a protected object on the worst case execution path for the thread, where *POName* is the identifier (name) of the protected object being called and *EntryName* is the identifier (name) of the entry used in the protected object being called;

LoopBody is a sequence of WCETs, CALL_POs and LOOPs;

PO signifies a call to a protected objected contained on a path different from the worst case execution path.

Points to note are :

- the sequence of all calls to all protected objects on the worst case execution path are provided.
- the loop body may contain WCETs , calls to protected objects on the worst case execution path and inner loops.
- a complete list of all calls to all protected objects on all possible execution paths different from the worst case execution path is provided.

4.1.3 Cyclic Thread Definition Template

In the cyclic thread templates given below, the following definitions apply:

- *Duration*: time measured in the number of clock cycles of the processor and is therefore always an integer.
- OFFSET: the time between the first release of a cyclic thread and the actual time at which it starts executing. Note that this offset is optional and is ignored by the Schedulability Analyser Tool and is only defined for use by the Scheduler Simulator Tool

For the definition three types of cyclic threads exist: hard, soft and non-critical. The hard and soft threads have the same definition template, while non-critical threads do not have a deadline reflecting the fact that this type of thread represents a background activity for which the execution constraints are not time critical for the HRT analysis.

4.1.3.1 Hard/Soft Cyclic Thread Definition Template:

```
THREAD Cyclic_Identifier
   CRITICALITY HARD or SOFT
   PERIOD Duration
   DEADLINE Duration
   [OFFSET Duration]
END Cyclic_Identifier
```

4.1.3.2 Non-critical Cyclic Thread Definition Template

```
THREAD Cyclic_Identifier
   CRITICALITY NON-CRITICAL
   PERIOD Duration
   [OFFSET Duration]
   [WCET Duration]
END Cyclic_Identifier
```

Points to note are:

- threads classified as non-critical have no deadline associated with them and are not analysed by the Schedulability Analyser Tool.
- the entry for the WCET is optional and is only ever used by the Scheduler Simulator Tool.

4.2 Sporadic Thread Templates

This section provides the templates along with annotations used in the toolset for the representation of sporadic threads. It includes both types of sporadic, interrupt sporadic and software sporadic, as separate threads.

Within the ensuing templates of Interrupt and Software Sporadic threads, the concept of inter arrival time is used: this is simply the minimum time between two successive invocations of a particular sporadic thread.

4.2.1 Sporadic Thread Ada Templates

The tool set is able to support an ATAC and non-ATAC board solution. In the case where an ATAC is used, various RTS operations are performed using the ATAC hardware. One of the functions it performs is the handling of interrupts i.e. the releasing of an appropriate sporadic thread in response to a particular interrupt.

The implementation of the interrupt handling within the ATAC requires the use of the Ada83 interrupt entry. In this case a task must declare an entry which is allocated to the interrupt, with the user providing the mapping of each interrupt to the appropriate sporadic task entry. For reasons of consistency and compatibility, the toolset supports the same mechanism in the non-ATAC case also.

In contrast, software sporadics are triggered by other software threads and hence they can be handled in a different manner. In this case the sporadic thread will call a barriered entry in a Synchronisation Protected Objects (see §4.3.1.1) and will be blocked until the barrier is lowered by the releasing software task.

As a consequence, there are specific Ada templates for each type of sporadic. However, there are some common parts. As for the cyclic threads the key feature of the template is the main loop which is used to encapsulate the main processing performed by the thread. There is also a prologue to the main loop, however, this is only required if there is any specific thread configuration and is not shown in the template.

4.2.1.1 Interrupt Sporadic Thread Ada Template

The distinguishing features of the interrupt sporadic Ada thread template are the assignment of the task entry to an interrupt, the null accept body and the definition of a procedure as an Interrupt Service Routine (ISR), which has a single statement calling the task entry. The constraints on a single statement entry call within the procedure and a null accept body are necessary to ensure compatibility between the ATAC and non-ATAC RTSs. For the ATAC RTS, these particular code sections are implemented within the hardware and consequently any additional statements would be ignored, see [RD15].

Also of note for the interrupt sporadic is the use of a specific interrupt priority range. This range occupies the highest 15 priority levels and is used to ensure the prioritisation of interrupt handling by the RTS. The HRT tools assign priorities in this range to interrupt sporadics and these assigned values should be used to ensure consistency with the HRT analysis and simulation.

The Ada template is as follows:

```
TASK IT_Sporadic_Identifier is
   -- the priority and the trap number defined here should
   -- be the same as for the ATTACH_HANDLER call below
   PRAGMA PRIORITY (Interrupt_Priority_Integer);
   ENTRY ISR;
      FOR ISR'ADDRESS USE  trap_number;
END IT_Sporadic_Identifier;

TASK BODY IT_Sporadic_Identifier IS
   [declare local variables]
BEGIN
   Interrupt_Service_Processing:
      LOOP
         ACCEPT ISR DO
            -- must be null to ensure ATAC and non-ATAC RTS consistency
            NULL;
         END ISR;
         [interrupt service processing]
      END LOOP Interrupt_Service_Processing ;
END IT_Sporadic_Identifier;
```

```
PROCEDURE ISR_Identifier is
BEGIN
   -- must only consist of call to interrupt sporadic entry.
   IT_Sporadic_Identifier.ISR ;
END;

-- interrupt priority and trap number must be the same as
-- those used for the task entry definition.
-- the last two parameters default to the values shown.
INTERRUPT_MANAGER.ATTACH_HANDLER
   (HANDLER => ISR_Identifier'ADDRESS;
   INTERRUPT=> Trap_Number;
   PRIORITY=> Interrupt_Priority_Integer;
   NUMBER_OF_BUFFERS => 1;
   MAX_PARAM_AREA_SIZE => 0);
```

Points to note are:

- When the ATAC is used, the interrupt service routines and accept statements are handled by the ATAC. Consequently, for compatibility, the only code is the call to the entry which must be contained in the ISR procedure.
- the priority level of the ISR procedure that makes the call to the entry of the interrupt sporadic task must be and can only be set by the ATTACH_HANDLER procedure.

4.2.1.2 Software Sporadic Thread Ada Template

The distinguishing feature of the software sporadic Ada thread template is the call to the synchronisation PO entry which is used to control the execution of the main processing loop. A software sporadic must be released by another thread, which is achieved through the synchronisation protected objects (see §2.4.3 and §4.3.1.1). The software sporadic must contain a single call to a barriered entry and requires that another thread (which may be a cyclic, interrupt sporadic or another software sporadic) calls the corresponding non-barriered entry for the protected object.

N.B. A software sporadic is the only thread type which may contain a call to a barriered entry and must only contain a single call. This is a constraint necessary to perform HRT analysis.

The Ada template for the software sporadic thread is as follows:

```
TASK SW_Sporadic_Identifier IS
   PRAGMA PRIORITY (Priority_Integer);
END SW_Sporadic_Identifier;


TASK BODY SW_Sporadic_Identifier IS
   [declare local variables]
BEGIN
   Software_Sporadic_Processing :
      LOOP
         -- call to synchronisation protected object entry to control
         -- execution of the main processing loop
         Sync_PO_Identifier.Wait_EntryName ([parameters]);
         [software sporadic processing];
      END LOOP Software_Sporadic_Processing;
END SW_Sporadic_Identifier;
```

4.2.2 Sporadic Thread Execution Profile Template

The templates for the two types of sporadic thread differ at one point only, namely the TYPE line where the appropriate nomenclature is used. The general structure and layout is the same as for the cyclic thread (see §4.1.2) and is as follows:

```
THREAD Sporadic_Identifier           -- either IT_Sporadic_Identifier
                                     -- or SW_Sporadic_Identifier
   TYPE INTERRUPT SPORADIC | SPORADIC      -- identifies thread type
   PRIORITY Integer

-- this section corresponds to the main loop, the ordering should
-- represent the worst case path and be a sequence of the following
-- statements.
   WCET Cp, Cmr, Cmw      -- statement sequence unbroken by entry calls
   CALL_PO POName EntryName     -- call to a passive task entry
   LOOP Integer           -- construct to provide for loops within the
      LoopBody            -- main processing loop
   END

-- this section provides list of all other passive task entry
-- entry calls (i.e. those not on the worse case path)
   PO POName EntryName    -- passive task entry call

END Sporadic_Identifier
```

4.2.3 Sporadic Thread Definition Template

The two types of sporadic thread, interrupt sporadic and software sporadic have slightly different definition requirements. Interrupt sporadics are considered to be of the highest priority for HRT analysis purposes and therefore do not have a `CRITICALITY` associated.

For both types, MINIMUM corresponds to the minimum inter-arrival time between two executions of the thread, while the specification of DEADLINE is optional; if one is not specified then the deadline is assumed to be the inter-arrival time.

4.2.3.1 Interrupt Sporadic Thread Definition Template

```
THREAD IT_Sporadic_Identifier
   MINIMUM Duration
   [DEADLINE Duration]
END IT_Sporadic_Identifier
```

4.2.3.2 Software Sporadic Thread Definition Template

```
THREAD SW_Sporadic_Identifier
   CRITICALITY HARD or SOFT
   MINIMUM Duration
   [DEADLINE Duration]
END SW_Sporadic_Identifier
```

4.3 Protected Object Templates

Two types of protected object are supported by the tool set:

1. Resource Control Protected Object used to ensure non-concurrent access to shared resources and to provide for data exchange between independent threads. There should be a resource protected object for each resource accessed by two or more threads and for each data communication between threads;

2. Synchronisation Protected Object used to enforce synchronisation between independent threads (i.e. the release of one thread is dependent upon another thread), for example the release of Software Sporadic Threads. There should be one synchronisation protected object for each software sporadic thread.

4.3.1 Protected Object Ada Templates

Both types of protected object are implemented via the PASSIVE task construct, which is based on the Ada 95 Protected Object semantics (and therefore does not incur the overhead of an independent thread of control). PASSIVE tasks, and hence Protected Objects, are identified via the use of pragma PASSIVE and are implemented using the IPCI protocol (also known as Priority Ceiling Emulation).

Otherwise the structure follows the normal Ada task syntax, including the inclusion of a priority pragma. In this case the priority represents the priority that any calling task assumes when calling the protected objects. This should be set to the value specified by the HRT analysis tools.

4.3.1.1 Synchronisation Protected Object Ada Template

The key feature of the sporadic protected object is that it contains exactly two entry calls, one of which must have a barrier (i.e. a “when” clause). The barriered entry is used for a sporadic task to await release, while the other entry is used by the releasing (or signalling) task.

The semantics of the accept bodies are such that:

- the release (or signal) operation must lower the barrier, by setting the “when” clause boolean true;
- the barrier operation should allow the software sporadic to execute and then should raise the barrier, by setting the “when” clause boolean false.

The Ada template is as follows:

```
TASK SPO_Identifier IS
   -- priority set according to the HRT tools
   PRAGMA PRIORITY (Priority_Integer);
   PRAGMA PASSIVE;                    -- enforces passive attribute
   -- exactly two entries, one for the wait and one for the signal
   ENTRY Wait_EntryName ([parameters]);
   ENTRY Signal_EntryName ([parameters]);
END SPO_Identifier;

TASK BODY SPO_Identifier IS
   [declare local variables]
BEGIN
   Forever:
      -- an infinite loop to ensure continuous operation
      LOOP
         SELECT
            -- entry with boolean expression to hold the SW sporadic
            WHEN (Barrier_logical_expression) =>
               ACCEPT Wait_EntryName ([parameters]) DO
               BEGIN
                  [perform release actions]
                  -- reset the barrier to ensure that the SW sporadic
                  -- is held upon completion of its processing
```

```
                  [reset barrier logical expression];
               END;
         OR
            -- entry to allow the release of the SW sporadic
            ACCEPT Signal_EntryName ([parameters]) DO
            BEGIN
               -- lower the barrier to allow the SW sporadic to execute
               [Set barrier logical expression];
            END;
         OR
            -- to ensure the correct close-out of the passive task
            TERMINATE;
         END SELECT;
      END LOOP Forever;
END SPO_Identifier;
```

The implementation of the barrier is an implementation issue

For example a *barrier_logical_expression* can be a simple logical variable:

```
TASK BODY Simple_SPO IS
   Barrier : BOOLEAN := FALSE;
BEGIN
   Forever:
      LOOP
         SELECT
            WHEN Barrier =>
               ACCEPT Wait DO
                  Barrier := FALSE;
         OR
            ACCEPT Signal DO
                  Barrier := TRUE;
         OR
            TERMINATE;
         END SELECT;
      END LOOP Forever;
END Simple_SPO;
```

It should be noted however, that in this case multiple calls of the `Signal` entry will only provide for one release as the release resets the barrier.

However, it is possible that the releasing thread will execute a number of times before one completion of the software sporadic it releases (this is particularly the case if the releasing thread has a short period/minimum inter-arrival time and deadline and the sporadic has a long deadline). This is particularly the case for ADS scheduling, where it is necessary to accumulate the releases within the *barrier_logical_expression*, in order to avoid losing them.

Therefore, if multiple releases are required the following should be used:

```
TASK BODY Complex_SPO IS
   Barrier_Count : INTEGER := 0;
BEGIN
   Forever:
      LOOP
         SELECT
            WHEN Barrier_Count > 0 =>
               ACCEPT Wait DO
                  Barrier_Count := Barrier_Count - 1;
         OR
            ACCEPT Signal DO
                  Barrier_Count := Barrier_Count + 1;
         OR
```

```
            TERMINATE;
         END SELECT;
      END LOOP Forever;
END Complex_SPO;
```

In this case the number of releases is counted. Note, that the protected object mutual exclusion mechanism ensures consistency of the `Barrier_Count` manipulation.

Note also, that for the purposes of the HRT analysis it is assumed that a barrier count mechanism is in place. In the event of the Scheduler Simulator determining that the signal entry is called for multiple releases before the wait entry is called, the number of calls are recorded and used to serve further wait entry calls (i.e. it is possible to have a signal, signal, wait, wait sequence). This is the case even if the barrier mechanism is implemented as a boolean flag (which allows only a single signal-wait sequence).

4.3.1.2 Resource Control Protected Object Ada Template

The key features of resource protected objects are that the object encapsulates a resource and that there are one or more entries, none of which have a barrier.

Encapsulation of the resource takes one of two forms. In the case where the resource is a data structure, then the data structure is defined within the task body and the entries provide the only access. Alternatively, the resource is an external entity (e.g. multiplex bus) and the entries encapsulate the access to the resource.

The constraint disabling barriers on entries is a necessity in order to perform the HRT analysis as it bounds the time that a thread must wait to perform a particular entry (the thread is guaranteed to perform the entry unless another lower priority thread is already accessing the protected object, in which case the thread is blocked until the accessing thread completes its entry).

The exact number of entry calls is dependent on the purpose of the protected object and the operations necessary on the object. If the object is just providing mutual exclusion to a resource (e.g. access to a processor port) then a single entry, which is called by all threads, is appropriate. However, if the actual access to the resource varies according to the thread, then the number of entries should reflect the number of different operations to be performed by the threads. For example, if the resource is a data structure which may be read and written then two entries, one read and one write are appropriate.

The Ada template is as follows:

```
TASK RPO_Identifier IS
   PRAGMA PRIORITY (Priority_Integer);
   PRAGMA PASSIVE;
   -- one or more entry calls, one per object operation
   ENTRY Entry_Name_1 ([parameters]);
   ENTRY Entry_Name_2 ([parameters]);
   .......
END RPO_Identifier;

TASK BODY RPO_Identifier IS
   -- if the resource is a data structure provide definition
   -- in the local variable section
   [declare local variables]
BEGIN
   Forever:
      -- an infinite loop to ensure continuous operation
```

```
      LOOP
         SELECT
            ACCEPT Entry_Name_1 ([Parameters]) DO
            BEGIN
               -- code for operation to be performed
               [protected code statements];
            END;
         OR
            ACCEPT Entry_Name_2 ([Parameters]) DO
            BEGIN
               [protected code statements];
            END;
         OR
            .........
         OR
            -- to ensure the correct close-out of the passive task
            TERMINATE;
         END SELECT;
      END LOOP Forever;
END RPO_Identifier;
```

4.3.2 Protected Object Definition Templates

The protected object definition templates consist of an identification section, including a field for the type of the object and a section for each entry in the protected object. For every entry there is a sequence which represents the worse case execution path for the entry and a section containing the list of other (i.e. not on worse case path) protected object entry calls made within the entry itself.

4.3.2.1 Synchronisation Protected Object Execution Profile Template

For the synchronisation protected object the entry which is barriered will have a BARRIER WCET statement to represent the execution time taken to evaluate the barrier itself.

```
PROTECTED SPO_Identifier
   -- a type field can be one of either SYNCHRO or SPO
   TYPE SYNCHRO | SPO

   -- a worse case time for the evaluation of the barrier
   BARRIER WCET Cp, Cmr, Cmw
   ENTRY Wait_EntryName
      -- Worst case execution path sequence for the wait entry.
      WCET Cp, Cmr, Cmw
      CALL _PO POName EntryName
      LOOP Integer
         LoopBody
      END
      -- this section provides list of all other protected object
      -- calls (i.e. those not on the worse case path) for this entry
      PO POName  EntryName

   ENTRY Signal_EntryName
      -- Worst case execution path sequence for the signal entry
      WCET Cp, Cmr, Cmw
      CALL_PO POName EntryName
      LOOP Integer
         LoopBody
      END
      -- list of other protected object calls
```

```
      PO POName  EntryName

END SPO_Identifier
```

4.3.2.2 Resource Control Protected Object Execution Profile Template

```
PROTECTED RPO_Identifier
   -- a type field can be one of either RESOURCE or RPO or PROTECTED
   TYPE RESOURCE | RPO | PROTECTED

   ENTRY Entry_Name_1
      -- Worst case execution path sequence for the entry
      WCET Cp, Cmr, Cmw
      CALL _PO POName EntryName
      LOOP Integer
         LoopBody
      END
      -- this section provides list of all other protected object
      -- calls (i.e. those not on the worse case path) for this entry
      PO POName  EntryName

   ENTRY Entry_Name_2
      WCET Time(Cp, Cm, Cr)
      CALL _PO POName EntryName
      LOOP Integer
         LoopBody
      END
      PO POName  EntryName
   . . .
END RPO_Identifier
```

4.3.3 Protected Object Definition Template

There is no definition of the protected objects within the UCF, since all the characteristics are found in the execution profile, or are derived by the HRT tools.

4.4 Application Level Templates

In addition to providing the thread HRT characteristics, the user must provide a description of the application level HRT characteristics (e.g. blocking protocol) and the target characteristics (e.g. clock frequency). As has already been established, the User Configuration File is used to contain this information. Therefore at the start of the UCF the following are defined:

- the scheduling algorithm to be employed;
- the blocking protocol to be employed (IPCI or INHIBIT);
- whether or not the ATAC is to be used;
- the minimum and the maximum priorities that can be assigned;
- the CPU clock frequency;
- the wait states;

The template for this section of the file is as follows:

```
-- HRT analysis preferences
PREFERENCES
   -- select scheduling approach either deadline monotonic or
   -- arbitrary deadline scheduling
```

```
   DMS | ADS
   -- select blocking protocol
   BLOCKING PROTOCOL IPCI | INHIBIT
END
-- definition of the characteristics of the hardware
TARGET CHARACTERISTICS
   -- ATAC or non-ATAC environment
   [NO] ATAC
   -- range of priorities maximum is 1 to 63
   PRIORITY LOW Low_value PRIORITY HIGH High_value
   CPU CLOCK FREQUENCY Frequency MHZ
   -- read and write wait states for memory
   WAIT STATES READ Read_wait_states WRITE Write_wait_states
END
```

The complete and formal syntax for the UCF is presented the Schedulability Analyser Tool and the Scheduler Simulator Tool User Manual [RD4].

4.5 Run-Time System (RTS) Characteristics

In order to complete the information to allow analysis to be performed the characteristics of the RTS need to be provided to the ERC32 tool set. The RTS file contains this information. It is provided by the compilation vendor and contains the execution timing data for the RTS routines used to implement the HRT application.

The values are used by the Schedulability Analyser and Scheduler Simulator tools to estimate the worst case computation time of all threads within the application.

The values provided within the file are independent of the application. they represent the logical abstraction of the RTS operations which are required to support an HRT application and as such are dependent only on the RTS. Consequently, the values are only expected to change if the RTS is changed (i.e. if the compiler changes). Within the ERC32 software tool set a two RTS sets are provided corresponding to the ATAC and non-ATAC RTSs which are contained within the Ada Compilation Tool.

The only user interaction is to specify the file name as input to the Schedulability Analyser and Scheduler Simulator.

For information the following are the RTS values for the ERC32 tool set Ada compilation system:

```
-- RTS Characteristics for DEM32 at 10 MHz (non-ATAC runtime)
-- Characteristics are in Processor Cycles
ENTER PASSIVE TASK 80,0,0
EXIT PASSIVE TASK 110,0,0
ENTER SOFTWARE SPORADIC WAIT (70,0,0)
EXIT SOFTWARE SPORADIC WAIT 30,0,0
ENTRY QUEUE SERVICING (60,0,0)
CONTEXT SWITCH TIME 340,0,0
SCHEDULER SELECT TIME (50,0,0)
ENTER DELAY UNTIL OVERHEAD AT HEAD (390,0,0)
ENTER DELAY UNTIL OVERHEAD NOT AT HEAD (220,0,0) + (30,0,0) * C
EXIT DELAY UNTIL OVERHEAD 80,0,0
TIMER INTERRUPT OVERHEAD 210,0,0
READY AFTER DELAY 120,0,0
INTERRUPT HANDLING OVERHEAD 670,0,0
ENTER INTERRUPT SPORADIC WAIT (30,0,0)
```

```
EXIT INTERRUPT SPORADIC WAIT 30,0,0
MAXIMUM NON PREEMPTION 1300,0,0


-- RTS Characteristics for DEM32 at 10 MHz (ATAC runtime)
-- Characteristics are in Processor Cycles
ENTER PASSIVE TASK 140,0,0
EXIT PASSIVE TASK 120,0,0
ENTER SOFTWARE SPORADIC WAIT (70,0,0)
EXIT SOFTWARE SPORADIC WAIT 40,0,0
ENTRY QUEUE SERVICING (80,0,0)
CONTEXT SWITCH TIME 410,0,0
SCHEDULER SELECT TIME (0,0,0)
ENTER DELAY UNTIL OVERHEAD AT HEAD (230,0,0)
ENTER DELAY UNTIL OVERHEAD NOT AT HEAD (240,0,0)
EXIT DELAY UNTIL OVERHEAD 80,0,0
TIMER INTERRUPT OVERHEAD 0,0,0
READY AFTER DELAY 0,0,0
INTERRUPT HANDLING OVERHEAD 0,0,0
ENTER INTERRUPT SPORADIC WAIT (30,0,0)
EXIT INTERRUPT SPORADIC WAIT 0,0,0
MAXIMUM NON PREEMPTION 650,0,0
```

4.6 Priority Determination

The final part to the implementation of an HRT application is the calculation of the priorities and the incorporation of these into the Ada code. As noted previously, the priorities for each thread and Protected Object is determined by either the Schedulability Analyser Tool or, if that is not used by the Scheduler Simulator Tool. These two tools use the definition of the application provided in the ESF and UCF. This implies that the user must first either generate the ESF by hand or must partially code the application and use the WCETE functionality to generate the ESF. The calculated priorities should then be incorporated into the Ada source code.

4.6.1 Priority Calculation

The priorities are determined in accordance with the scheduling algorithm specified in the UCF. The user is informed of the priorities that have been determined for all threads and Protected Objects within the application via both the ESF and the Analysis Report generated by the Schedulability Analyser Tool.

Details of how the priority levels are determined can be found in [RD4].

The range of permissible priority levels to be used by the Schedulability Analyser and Scheduler Simulator tools is specified by the user in the UCF. Although the user is free to specify whatever priority range he/she wishes, it should noted that the ATAC and the RTS do not support priority levels greater than 63. It is, therefore, strongly recommended that, for all practicable purposes, the maximum priority level stipulated by the user in the UCF should not exceed 63.

The following points pertaining to priority assignment are worthy of note:

- Interrupts sporadics and any Protected Objects accessed by the interrupt sporadics are assigned the highest priority levels in the range 49 to 63. 15 levels have been specifically identified as this corresponds to the number of traps which can be served on the ERC32. In the event of their being more that 15 Interrupt Sporadics and Protected Objects accessed by Interrupt Sporadics then the range is extended downwards eroding the allocation for other thread types.
- For the special case of two or more threads having the same deadline and same criticality type, priorities are assigned in accordance with the order of thread specification in the UCF.
- When two or more Protected Objects share the same highest priority thread, then an arbitrary monotonic order of priorities is imposed upon the Protected Objects: each Protected Object is given a unique priority level where the interval between adjacent priority levels is one.
- When the aggregate sum of threads and Protected Objects exceeds the maximum permissible priority level (as specified in the UCF), then the non-critical threads are grouped together and assigned the same priority level. If necessary (i.e. the maximum priority level is still exceeded) the same is done for soft threads and the analysis is allowed to proceed. This is an acceptable approach since all it means is that the grouped threads within any particular criticality level will be scheduled arbitrarily, and this does not pose a problem since overruns of non-critical and soft threads can be tolerated.
- If, however, the thread set consists of entirely of hard threads and interrupts, then the analysis is halted and the user informed of the situation since overruns of hard threads cannot be tolerated (the system is unschedulable).

In summary, no overlapping is permitted:

- between threads pertaining to different criticality classes;
- between protected objects and threads;
- between interrupt sporadics;
- between hard threads.

4.6.2 Priority Assignment

As noted above once the priorities have been calculated the user must incorporate these into the Ada source code. This is achieved by explicitly assigning the priority to a thread by one of two mechanisms, either through the use of pragma PRIORITY, or through the use of the system procedure SET_DYNAMIC_PRIORITY. Note, however, that for PASSIVE tasks the pragma mechanism must be used.

There are advantages to both mechanisms. Pragma PRIORITY means that the priority definition is contained with the definition of the task, which will ease understandability. In addition this mechanism is simpler to implement as the pragma will appear with the task definition. The SET_DYNAMIC_PRIORITY approach allows a single procedure to assign all priorities to the tasks, except the PASSIVE tasks, and means that there is a reduction in the re-compilation effort in the event that the priorities change.

For the ERC32 software tool set, the pragma mechanism is recommended. This is illustrated for each type of task below:

Cyclic Task:

```
TASK Cyclic_Identifier IS
   PRAGMA PRIORITY (Priority_Integer);
END Cyclic_Identifier;
```

Interrupt Sporadic Task and procedure:

```
TASK IT_Sporadic_Identifier is
   PRAGMA PRIORITY (Interrupt_Priority_Integer);
   ENTRY ISR;
      FOR ISR'ADDRESS USE  trap_number;
END IT_Sporadic_Identifier;

PROCEDURE ISR_Identifier is
BEGIN
-- Make call to interrupt sporadic entry.
   IT_Sporadic_Identifier.ISR;
END;

INTERRUPT_MANAGER.ATTACH_HANDLER
   (HANDLER => ISR_Identifier'ADDRESS;
   INTERRUPT=> Trap_Number;
   PRIORITY=> Interrupt_Priority_Integer;
   NUMBER_OF_BUFFERS => 1;
   MAX_PARAM_AREA_SIZE => 0);
```

Software Sporadic Task:

```
TASK SW_Sporadic_Identifier IS
   PRAGMA PRIORITY (Priority_Integer);
END SW_Sporadic_Identifier;
```

Synchronisation Protected Object:

```
TASK SPO_Identifier IS
   PRAGMA PRIORITY (Priority_Integer);
   PRAGMA PASSIVE;
   ENTRY Wait_EntryName ([parameters]);
   ENTRY Signal_EntryName ([parameters]);
END SPO_Identifier
```

Resource Protected Object:

```
TASK RPO_Identifier IS
   PRAGMA PRIORITY (Priority_Integer);
   PRAGMA PASSIVE;
   ENTRY Entry_Name_1 ([parameters]);
   ENTRY Entry_Name_2 ([parameters]);
   . . .
END RPO_Identifier;
```

5. USING THE TOOL SET

Having described in the previous section how to create an HRT application with the tool set this section now goes on to show the overall development process to be followed for a complete HRT application development lifecycle.

The process for developing an HRT application was broken into seven steps in section 2.3. In this section each of the steps is re-visited and the interaction between the developer and each tool explained.

5.1 Preparation for HRT Analysis

The preparation activity consists of defining the application in terms of the threads involved and the communication between the threads. The actual action of the developer depends on the phase of the development and is repeated as the software becomes more refined (e.g. identification of detailed procedures and functions).

There are two stages to this process: the identification of the threads and the implementation of these within the ERC32 tool set.

Throughout this section, an example application is used. For reference, the UCF and ESF along with the modifications discussed are provided in Appendices A and B.

5.1.1 Thread and Protected type identification

The tool set does not provide direct support for any design method, rather the user is expected to use an appropriate technique and/or tool to identify the HRT entities (i.e. cyclic threads, sporadic threads and protected objects). The HRT-HOOD method ([RD2], [RD3] and [RD24]) has been produced to fulfil this role and it is recommended that such an approach is employed.

Regardless of the exact method used in order to produce or represent the design, the entities identified must conform to the ERC32 tool set HRT entities. Therefore guidelines are provided below for the identification of the thread types and protected objects.

Cyclic threads:

> Typically there are a number of activities within an space application which are required to be performed at regular time intervals. For example, satellite attitude and orbit control is performed based on a control law which is time dependent. Each such activity should be identified as a unique cyclic thread with the time dependency providing the frequency of occurrence.
>
> Note, however, that it is important to ensure that threads represent a logically complete activity and do not internally perform pseudo-scheduling. For example, some control law implementations might collect sensor inputs every 10 milliseconds, but generate outputs every 5 milliseconds based on a control loop value. In this case the acquisition and generation activities should separate.

Interrupt Sporadic threads:

Identification of these type of thread are simple. Each external interrupt requiring handling should have a corresponding interrupt sporadic thread.

Note however, it is important to decide the amount of processing to be performed within the interrupt sporadic itself, which will have a high priority (interrupt sporadics have a priority range greater than all other threads). For certain interrupts it may be more practical to limit the processing done at the occurrence of the interrupt and to complete processing at a lower priority, by deferring it to another thread. As example, consider transient data from a device which causes an interrupt to the processor. This must be read immediately, but processing can be delayed. In such a situation an interrupt sporadic thread should be created to read the data, but the processing should be performed by a separate software sporadic thread, initiated by the interrupt sporadic.

Sporadic Threads:

This type of thread is used for those activities which are required to be performed on an irregular basis or as a sub-activity of another thread. In addition, this thread could be used in the case where an activity is required as a sub-activity of a number of threads, but for which only one instance can occur at one time. This is analogous to a sub-program structure but where the activity can be performed as an independent thread of control and must be mutually exclusive of another instance.

The initiation of the activity must be performed by another thread, either cyclic, interrupt sporadic, or software sporadic and must be performed using a Synchronisation Protected Object.

Examples of this type of thread are:

- as an irregular thread: as a thread initiated to change Attitude and Orbit Control modes, which performs the necessary reconfiguration required;
- as a sub-activity of a single thread: as a thread initiated to complete the processing associated with the occurrence of an interrupt;
- as a sub-activity of a number of threads: a bus request activity which requests data and collects the response;

Resource Protected Objects:

The Resource Protected Object provides a mechanism for mutual exclusive access to a particular operation and consequently access to data. Typically, this is used when data is passed between two threads, without the threads themselves requiring synchronisation. This represents the buffering of data by one thread for subsequent access by one or more other threads.

An instance of this type of object should be used in all cases where data is accessed or shared between threads and where referential consistency is required.

A simple example is the use a Resource Protected Object for a buffer between a reader thread and writer thread:

```
TASK Buffer IS
   PRAGMA PRIORITY (30);
   PRAGMA PASSIVE;
   ENTRY Read (Item : out Data_Item);
   ENTRY Write (Item : in Data_Item);
END Buffer;

TASK BODY Buffer IS
   Item_Store : Data_Item;
BEGIN
   Forever:
   LOOP
      SELECT
         ACCEPT Read (Item : out Data_Item) DO
            Item:= Item_Store;
      OR
         ACCEPT Write (Item : in Data_Item) DO
            Item_Store := Item;
      OR
         TERMINATE;
      END SELECT;
   END LOOP Forever;
END Buffer;

TASK Reader IS
   PRAGMA PRIORITY (25);
END Reader;

TASK BODY Reader is
   Read_Item : Data_Item;
   Period : REAL_TIME.TIME_SPAN := REAL_TIME.MILLISECONDS (10);
   Reader_Time : REAL_TIME.TIME;
BEGIN
   Reader_Time := REAL_TIME.CLOCK + Period;
   Period_Action:
   LOOP
      DELAY_UNTIL (Reader_Time);
      Buffer.Read (Read_Item);
      Reader_Time := Reader_Time + Period;
   END LOOP Period_Action;
END Reader;

TASK Writer IS
   PRAGMA PRIORITY (26);
END Writer;

TASK BODY Writer is
   Write_Item : Data_Item := 0;
   Period : REAL_TIME.TIME_SPAN := REAL_TIME.MILLISECONDS (10);
   Writer_Time : REAL_TIME.TIME;
BEGIN
   Writer_Time := REAL_TIME.CLOCK + Period;
   Period_Action:
   LOOP
      DELAY_UNTIL (Writer_Time);
      Buffer.Write (Write_Item);
      Write_item := Write_Item + 1;
      Writer_Time := Writer_Time + Period;
   END LOOP Period_Action;
END Writer;
```

Synchronisation Protected Objects:

The Synchronisation Protected Object provides a mechanism for providing synchronisation between two threads. It differs from the Ada83 rendezvous in that only one thread will wait, the other thread only recording that it has passed through before continuing. The mechanism is used by one to initiate a software sporadic, the different initiation situations are described above for the software sporadic thread.

Typically this mechanism would be used where an activity is only required to be performed when the data it operates on becomes available. In this situation the data provider (another thread) will cause the initiation through a Synchronisation Protected Object.

For example the reader/writer example when the read will only occur when data is available. Below the reader is initiated only when data has been written:

```
TASK Controller IS
   PRAGMA PRIORITY (30);
   PRAGMA PASSIVE;
   ENTRY Wait (Data_Item : out Data_Item_Type);
   ENTRY Signal (Data_Item : in Data_Item_Type);
END Controller;

TASK BODY Controller IS
   Data_Available    : BOOLEAN := FALSE;
   Passed_ Data_Item : Data_Item_Type;
BEGIN
   Forever:
   LOOP
      SELECT
         WHEN Data_Available =>
            ACCEPT Wait (Data_Item : out Data_Item_Type) DO
            begin
               Data_Available := FALSE;
               Data_Item := Passed_Data_Item;
            end;
      OR
         ACCEPT Signal (Data_Item : in Data_Item_Type) DO
         begin
            Data_Available := TRUE;
            Passed_Data_Item := Data_Item;
         end;
      OR
         TERMINATE;
      END SELECT;
   END LOOP Forever;
END Controller;

TASK Reader IS
   PRAGMA PRIORITY (25);
END Reader;

TASK BODY Reader is
   Read_Item : Data_Item_Type;
BEGIN
   Aperiodic_Action:
   LOOP
      Controller.Wait (Read_Item);
   END LOOP Aperiodic_Action;
END Reader;
```

```
TASK Writer IS
   PRAGMA PRIORITY (26);
END Writer;

TASK BODY Writer is
   Write_Item : Data_Item_Type := 0;
   Period : REAL_TIME.TIME_SPAN := REAL_TIME.MILLISECONDS (10);
   Writer_Time : REAL_TIME.TIME;
BEGIN
   Writer_Time := REAL_TIME.CLOCK + Period;
   Period_Action:
   LOOP
      DELAY_UNTIL (Writer_Time);
      Write_Item := Write_Item + 1;
      Controller.Signal (Write_Item);
      Writer_Time := Writer_Time + Period;
   END LOOP Period_Action;
END Writer;
```

Thread Criticality:

Thread criticality requires the user to select the most appropriate category for the thread, as follows:

- Hard: this category is for threads which must meet their deadlines;
- Soft: this is for threads for which it is desirable that they meet their deadlines, but failure may be tolerable;
- Non-Critical: this category is for threads that have no deadline (except their period) and which can effectively be considered as background threads;

5.1.2 Application definition in ERC32 tool set

Having defined the design structure, it is then recorded in the Ada compiler system by defining the necessary package and task specifications and bodies such that a complete and consistent Ada program is known to the compiler (i.e. compilation of the program does not result in any closure errors). The level of detail recorded depends on the stage of the development. Initially, only an outline of the task bodies and the appropriate package specifications and bodies could be described (note, in this case the user will need to produce the ESF). Subsequently, as the design progressed procedures and functions would be added, and finally during coding the Ada code is added.

In the HRT software development scenario described in §2.3, preparation also includes the definition of the thread type and the scheduling characteristics. When using the tool set this is achieved in two stages. The type (e.g. cyclic) is identified from the application code by the constructs used in the source (templates described in section 5.1). The imposition of the structure allow the thread type's to be extracted either automatically (by the WCETE) or manually by examination of the application code. The scheduling characteristics (criticality, period, deadline and minimum inter-arrival time) along with the scheduling algorithm are defined by the user in the UCF (or if HRT-HOOD is used, they can be extracted from the design).

The use of code structure to denote thread type means that the Execution Skeleton can provide the thread type and this means that the user must define an application outline as a

minimum in order to identify the threads' priorities. The impact on the recompilation effort is minimised by allowing the initial activity to be carried out at an early stage in the development when the application definition is minimal.

This activity is completed when a complete and consistent application has been identified to the compiler. The user is responsible for ensuring that all threads for an application are defined (although any discrepancies will be flagged during the analysis process). On completion of this definition the developer moves on to the generation of worst case timings.

5.2 Generation of Worst case Timing Information

The purpose of this activity is to generate the WCET information for the threads, procedures and functions which make up an application.

The imposition of the various templates for the HRT application construction means that a WCETE functionality is able to generate the Execution Skeleton. Generation of an Execution Skeleton with this method is a two stage process as follows:

- during the compilation the user must request that detailed timing data is generated for each compilation unit which is part of the HRT analysis;
- during the bind and link process the user must request that an Execution Skeleton is produced. At this stage checks are made to ensure that all units provide timing information and that all units within the Ada closure are available;

Note, that with such a tool, if the user wishes to perform analysis prior to all units being coded then dummy units can be introduced with loop constructs which consume processing time equivalent to the users estimate.

If WCETE functionality is not available the user must produce the Execution Skeleton manually. This process may be made easier through the use of the HRT-HOOD for the design method.

Using the templates identified in the §4, the user should construct the application. In this case, the Ada-like syntax allows the user to enter the threads and protected objects in a similar manner to coding the application. However, the user will have to enter estimates for the processing time.

5.3 Calculation of Thread Priorities

The calculation of the thread priorities is performed automatically by the Schedulability Analyser and Scheduler Simulator. The priority levels are based on the scheduling characteristics of the threads and their interaction through protected objects. The automatic calculation therefore requires the definition of the thread characteristics in the UCF and the definition of the application provided in the ESF. Once calculated, the priorities are automatically added to the ESF.

The user is able to influence the priority assignment by understanding the logical algorithm for calculation. The assignment for threads is fundamentally based on the thread criticality type and the deadlines assigned to the threads within a type, while the protected objects priority is a function of the user threads. The detailed definition is contained in [RD4] and described in §4.6.1. It can however be summarised as follows:

- firstly threads are ordered by thread type as: interrupt sporadic; criticality hard; criticality soft; and non-critical;
- within each thread type, threads are ordered by deadline (or period for non-critical threads);
- protected objects are assigned a priority 1 higher than the maximum of its user threads and protected objects;

Therefore, the user may change the thread criticality type and the deadline to change the priority level assigned to a particular thread.

The allocation approach can be seen from the following segments of UCF and ESF:

UCF:

```
THREAD CYCLIC_1
   CRITICALITY HARD
   PERIOD 10000
   DEADLINE 10000
END CYCLIC_1

THREAD CYCLIC_2
   CRITICALITY HARD
   PERIOD 10000
   DEADLINE 10000
END CYCLIC_2

THREAD CYCLIC_LOW
   CRITICALITY HARD
   PERIOD 15000
   DEADLINE 15000
END CYCLIC_LOW

THREAD IT_SPORADIC
   MINIMUM 10000
   DEADLINE 10000
END IT_SPORADIC



THREAD SW_SPORADIC
   CRITICALITY HARD
   MINIMUM 10000
   DEADLINE 10000
END SW_SPORADIC
```

ESF:

```
THREAD CYCLIC_1
   TYPE CYCLIC
   WCET 500,0,0
   CALL_PO RESOURCE_PO ACQUIRE
   WCET 500,0,0
END CYCLIC_1
THREAD CYCLIC_2
   TYPE CYCLIC
   WCET 500,0,0
   CALL_PO SW_SYNCH_PO SIGNAL
   WCET 500,0,0
END CYCLIC_2
THREAD CYCLIC_LOW
   TYPE CYCLIC
   WCET 800,0,0
   CALL_PO RESOURCE_PO ACQUIRE
   WCET 1000,0,0
END CYCLIC_LOW
THREAD IT_SPORADIC
   TYPE INTERRUPT SPORADIC
   WCET 1000,0,0
   WCET 1000,0,0
END IT_SPORADIC



THREAD SW_SPORADIC
   TYPE SPORADIC
   CALL_PO SW_SYNCH_PO WAIT
   WCET 1000,0,0
END SW_SPORADIC
PROTECTED SW_SYNCH_PO
   TYPE SPORADIC
   ENTRY SIGNAL
      WCET 500,0,0
   BARRIER WCET 100,0,0
   ENTRY WAIT
      WCET 500,0,0
END SW_SYNCH_PO
PROTECTED RESOURCE_PO
   TYPE PROTECTED
   ENTRY ACQUIRE
      WCET 1000,0,0
END RESOURCE_PO
```

In this example the interrupt sporadic thread will be assigned the highest priority, 63, as it is the only thread of that type. Of the remaining threads, priorities are assigned according to deadline. However, three threads CYCLIC_1, CYCLIC_2 and SW_SPORADIC share the same details. Therefore, the allocation is arbitrary, but will occur in order of definition. Thread CYCLIC_LOW will be assigned the lowest priority as it has the longest deadline. For the protected objects, RESOURCE_PO is assigned a priority one greater than CYCLIC_1, while SW_SYNCH_PO will be assigned a priority one greater than its two accessing threads, CYCLIC_2 and SW_SPORADIC.

Note, that the Ada RTS supports 64 priority levels, however the schedulability analyser and scheduler simulator are to able to assign across a range from 0 to 127. The user is recommended to restrict the range in the UCF to 1 to 63 for reasons of compatibility. It should also be noted that in the event that there are greater than 64 threads and protected objects the schedulability analyser and scheduler simulator will share priorities starting from the less critical threads. The user is however warned that this will take place.

5.4 Analysis of Application

The analysis provides the assessment of schedulability. The calculations are performed by the Schedulability Analyser based on a critical instant, which assumes all threads (both sporadic and periodic) are ready for release.

In order to perform the analysis the definition of the application and threads are required in the ESF and UCF. These are provided as filenames to the tool, along with the Ada RTS characteristics that are contained in the RTS Characteristics File. Note that it is the responsibility of the user to ensure that the correct RTS characteristics are provided (i.e. ATAC or non-ATAC).

As part of the processing the Schedulability Analyser reports on the consistency of the data in the input files (e.g. that there is a thread definition for each thread defined in the ESF). The precise error reports are contained in the user manual [RD4]. Any inconsistencies should be corrected.

Upon completion of the analysis a report summarising the outcome is provided, it identifies the following:

- worst case response time of each thread;
- which threads, if any, fail to meet their deadlines;
- the margin by which threads, if any, meet their deadline;
- the overall processor utilisation;
- the utilisation of each of the threads.

These results provide the developer with a statement on the schedulability of the application upon which the user is able to assess the feasibility of the design and/or code.

Following is the schedulability analysis resulting from the example provided in §5.3 above. In this case the thread set is schedulable and has an overall utilisation of 90%.

```
Schedulability Analyser
***********************

Beginning of the Offline Analyser Report

Current date and time : 14/3/1997::18:35:35

Name of the input files :
UCF Files : /export/home/awalter/hrt_tools/testing/um_eg.ucf
ESF Files : /export/home/awalter/hrt_tools/testing/um_eg.esf
RTS Files : /export/home/awalter/hrt_tools/testing/demo1.rts

Scheduling Policy : ADS   Blocking Protocol : IPCI   ATAC Mode : Not selected

List of defined threads
_______________________

In the following report, all threads are defined by a numeric identifier :

   Identifier   Thread
   1............CYCLIC_1
   2............CYCLIC_2
   3............CYCLIC_LOW
   4............IT_SPORADIC
   5............SW_SPORADIC

List of defined Protected Objects
_________________________________

In the following report, all Protected Objects are defined by a numeric identifier :

   Identifier   Protected Object
   1............RESOURCE_PO
   2............SW_SYNCH_PO
```

List of all schedulable and non schedulable threads :
****

Sch	Th#	Criticality	Deadline	Prio	WCCT FR	WCCT SR	Period MIAT	Response Time	Blocking Time	Blocking Origin	Utilisation Factor	Margin Analysis
yes	4	INTERRUPT	10000	63	2000	2000	10000	2000	0	RTS	2.00000E-01	0.00E+00
yes	1	HARD	10000	5	2000	2000	10000	5000	1000	PO # 1	4.00000E-01	0.00E+00
yes	2	HARD	10000	3	1500	1500	10000	6500	1000	PO # 1	5.50000E-01	0.00E+00
yes	5	HARD	10000	2	1700	1600	10000	8200	1000	PO # 1	7.20000E-01	0.00E+00
yes	3	HARD	15000	1	2800	2800	15000	10000	0	RTS	9.06667E-01	0.00E+00

List of all protected objects
****

PO #	Prio	Protected Type	PO User List
1	6	RESOURCE	T#1, T#3,
2	4	SYNCHRONISATION	T#2, T#5,

Thread Set Utilisation Factor
****

The Utilisation factor for the whole thread set is : 9.06667E-01

In the cases where threads fail to meet their deadlines the user should firstly use the Scheduler Simulator to attempt to establish the precise point of failure. However, in order to use the Scheduler Simulator for this purpose the user must ensure that the Scheduler Simulator configuration is consistent with the analysis. This requires the following to be set up on the Scheduler Simulator:

- all barriers for software sporadic threads should be set closed;
- the pattern for all interrupt occurrences should be set to at the minimum inter-arrival time.

The user is referenced to the Schedulability Analyser and Scheduler Simulator User Manual [RD4] for guidance on how to perform these configurations. In addition, it is important to examine the simulation from the correct point in time. Due to limitations within the tools implementation, the first occurrence of an interrupt only happens after the minimum inter-arrival time has expired. This means that at the start of the simulation, the no interrupt sporadics are released. Therefore, it is necessary for the user to calculate the lowest common multiple for all threads using the periods and minimum inter-arrival times. The user should then examine the simulation from this point in time.

With this configuration, the user should perform a simulation and examine the output and establish the point of failure. This can be performed by examining the Gantt chart of the simulation and following the thread indicated by the analysis as failing to meet its deadline. The simulation should indicate a missed deadline on the trace of the thread. Working back from this time it is possible to establish the threads which have previously executed and the CPU time taken.

Using the example provided in §5.3, if the first WCET for CYCLIC_LOW is changed to 900, then the thread set becomes unschedulable, see schedulability analysis below in which the thread is indicated. In this case the Scheduler Simulator is used to examine the schedule from the 30 millisecond time point, this being the lowest common multiple for the thread set. In the Gantt chart the missed deadline can be clearly seen, being marked as a vertical bar on the time graph for the thread (alternatively, the Scheduler Simulator textual report and the simulation execution report identify the thread and the time at which the deadline is missed).

```
Schedulability Analyser
***********************

Beginning of the Offline Analyser Report

Current date and time : 14/3/1997::18:36:22

Name of the input files :
UCF Files : /export/home/awalter/hrt_tools/testing/um_eg1.ucf
ESF Files : /export/home/awalter/hrt_tools/testing/um_eg1.esf
RTS Files : /export/home/awalter/hrt_tools/testing/demo1.rts

Scheduling Policy : ADS   Blocking Protocol : IPCI   ATAC Mode : Not selected

List of defined threads
_______________________

In the following report, all threads are defined by a numeric identifier :

   Identifier   Thread
   1............CYCLIC_1
   2............CYCLIC_2
   3............CYCLIC_LOW
   4............IT_SPORADIC
   5............SW_SPORADIC

List of defined Protected Objects
_________________________________

In the following report, all Protected Objects are defined by a numeric identifier :

   Identifier   Protected Object
   1............RESOURCE_PO
   2............SW_SYNCH_PO
```

List of all schedulable and non schedulable threads :
****

Sch	Th#	Criticality	Deadline	Prio	WCCT FR	WCCT SR	Period MIAT	Response Time	Blocking Time	Blocking Origin	Utilisation Factor	Margin Analysis
yes	4	INTERRUPT	10000	63	2000	2000	10000	2000	0	RTS	2.00000E-01	-5.00E+00
yes	1	HARD	10000	5	2000	2000	10000	5000	1000	PO # 1	4.00000E-01	-5.00E+00
yes	2	HARD	10000	3	1500	1500	10000	6500	1000	PO # 1	5.50000E-01	-6.67E+00
yes	5	HARD	10000	2	1700	1600	10000	8200	1000	PO # 1	7.20000E-01	-5.88E+00
-NO	3	HARD	15000	1	2900	2900	15000	17300	0	RTS	9.13333E-01	-3.45E+00

List of all protected objects
****

PO #	Prio	Protected Type	PO User List
1	6	RESOURCE	T#1, T#3,
2	4	SYNCHRONISATION	T#2, T#5,

Thread Set Utilisation Factor
****

The Utilisation factor for the whole thread set is : 9.13333E-01

Scheduler Simulator Gantt chart showing missed deadline for CYCLIC_LOW

Note that, in exceptional circumstances, it is possible for the simulation not to flag the missed deadline and to indicate that the application is schedulable. This can occur because of the pessimism built into the analysis. For example, the analysis must assume that each thread is subject to blocking from a lower priority thread accessing a protected object. However, it may be impossible for the blocking to occur, due to the actual execution sequence (e.g. the sequence is such that the higher priority thread will only ever run when the lower priority thread has completed and hence blocking is not possible). If this situation arises the user is recommended to assume that a scheduling failure exists, especially since the conflicting results between the Schedulability Analyser and Scheduler Simulator indicate that the timing constraints are probably very tight for the application.

Having established the scheduling failure or the potential failure, the application should be modified. This can be achieved by modifying the thread set priority profile as follows:

- as the deadline of a thread will dictate its priority within its criticality group, examine the thread and all higher priority threads to see if the deadlines may be changed to raise the priority of the failing thread;
- examine the higher priority threads to see if any of the activities can be reduced by splitting the activity into a high and low priority thread, thus reducing the processing time requirements of the high priority thread;
- re-examine the thread partitioning to ensure that all threads are categorised hard and soft appropriately;
- examine the interrupt sporadic threads to see if the part of the activity can be split into a software sporadic thread with a lower priority;

Using the example from §5.3 modified as identified above for CYCLIC_LOW, the higher priority threads should be examined. Consider thread CYCLIC_1:

```
THREAD CYCLIC_1
   TYPE CYCLIC
   WCET 500,0,0
   CALL_PO RESOURCE_PO ACQUIRE
   WCET 500,0,0
END CYCLIC_1
```

It consists of a prologue WCET statement, followed by acquisition of the RESOURCE_PO and is completed by an epilogue WCET statement. It may be possible to split this epilogue code from the thread (e.g. it may not be necessary to complete this within the deadline, but rather had originally been added to the thread for design reasons). If this is the case the epilogue code could be performed by a new thread with a longer deadline, which could be either, a cyclic, or, a software sporadic, which is initiated by CYCLIC_1 through a synchronised protected object. These are illustrated below:

Cyclic solution:

UCF:

```
THREAD CYCLIC_1_COMPLETE
   CRITICALITY SOFT
   PERIOD 10000
   DEADLINE 20000
END CYCLIC_1_COMPLETE
```

ESF:

```
THREAD CYCLIC_1_COMPLETE
   TYPE CYCLIC
   WCET 500,0,0
END CYCLIC_1_COMPLETE
```

Sporadic Solution:

UCF:

```
THREAD CYCLIC_1
   CRITICALITY HARD
   PERIOD 10000
   DEADLINE 10000
END CYCLIC_1

THREAD CYCLIC_1_COMPLETE
   CRITICALITY HARD
   MINIMUM 10000
   DEADLINE 20000
END CYCLIC_1_COMPLETE
```

ESF:

```
THREAD CYCLIC_1
   TYPE CYCLIC
   WCET 500,0,0
   CALL_PO RESOURCE_PO ACQUIRE
   CALL_PO SW_SYNCH_COMPLETE SIGNAL
END CYCLIC_1
THREAD CYCLIC_1_COMPLETE
   TYPE SPORADIC
   CALL_PO SW_SYNCH_COMPLETE WAIT
   WCET 500,0,0
END CYCLIC_1_COMPLETE
PROTECTED SW_SYNCH_COMPLETE
   TYPE SPORADIC
   ENTRY SIGNAL
      WCET 50,0,0
   BARRIER WCET 100,0,0
   ENTRY WAIT
      WCET 50,0,0
END SW_SYNCH_COMPLETE
```

Note, the software sporadic example also illustrates the case where the analysis indicates that the thread set is not schedulable, but the simulation does not show a failure. In this case the reason is because the deadline for completion of the new software sporadic CYCLIC_1_COMPLETE is assumed to start at the critical instant for the analysis, while for the simulation, the deadline expires from the point of initiation of the thread (i.e. only after it is released by CYCLIC_1).

5.4.1 Period Transformation

In cases where the scheduling constraints are exceeded by only a minimal amount (e.g. one or two threads only), an alternative mechanism is available for assuring the thread set schedulability. This mechanism is known as period transformation and can be applied to any cyclic thread, where the following typical situations arises:

- where 'hard' threads which have a long deadline are causing 'soft' threads with shorter deadlines to overrun (since hard threads always have higher priority than soft threads);
- when the time of certain activity needs to be known or controlled precisely (e.g. output of data), known as jitter control [RD2 and RD3].

In these cases, a thread can be split into a number of sub-activities with precedence constraints (i.e. running order), such that the sub-activities run as a sequence which performs the same activity as the original thread. Each sub-activity becomes a thread with the same period of execution as the original thread and has precedence constraint imposed by introducing an offset to the period of execution for the thread (see §2.4.4.5). This offset must be calculated by the user and be such that previous sub-activity thread will have completed before the offset expires. In addition, the deadlines of the sub-activities must be modified to reflect the sub-division. Further guidance on the use of this technique can be found in the HRTOSK documentation [RD2 and RD3].

It should be noted however, the toolset provides no automated support for this functionality. Further, since the Schedulability Analyser does not take into account the offset and so the

results of any analysis should be carefully examined in this case (the Scheduler Simulator will however reflect the offset in the simulation).

5.4.2 Alternate Analysis

The Schedulability Analyser performs two distinct types of analysis: the *nominal* analysis and the *alternate* analysis. The two types of analysis differ in their treatment of blocking.

In the case of *nominal* analysis, the calculation of blocking time for a thread is made using only those Protected Objects that are on a thread's worst case path in the ESF (i.e. only protected objects which appear in the ESF for any thread with a `CALL_PO` prefix). The premise for this analysis approach is that the respective worst case execution paths yield the worst case response time for all threads.

This is not correct as the calculation of the worst case path for a thread is done on a per thread basis (i.e. independent of other threads in an application) and is not able to consider the potential blocking introduced by a call to a protected object. It is conceivable that a thread, in executing an alternative path, would be blocked on access to a Protected Object which lies on this alternative path (due to the Protected Object being held by a lower priority thread), such that the resultant blocking increases the thread response time to be greater than for the calculated worst case path.

One way of overcoming this uncertainty in the *nominal* analysis is to consider the effects of blocking associated with all Protected Objects contained within the application (i.e. take a global view of the application), rather than just those lying on the worst case paths: the *alternate* analysis. The *alternate* analysis is in effect an independent cross check of the *nominal* analysis.

Should the *alternate* analysis yield a result different (i.e. the response times of threads being increased) from the nominal analysis, then an additional report is automatically generated in which the Protected Object causing the longer blocking is clearly identified.

If this situation arises the user must carefully examine the application source code and analyse the result to establish where the blocking protected object is accessed. Having established this, the user must then consider whether the path upon which the offending protected object lies is realistic. If the path is considered realistic, then the user is recommended to examine the alternate path, by modifying the ESF to include this path as the worst case. This is achieved by either hand editing the ESF, or editing the source code to comment out the calculated worst case path and regenerating the ESF, in the case where the WCETE functionality has been used.

Using the example in §5.3, the reason for alternative analysis can be seen. Consider the addition of another protected object BUFFER, which is used by both CYCLIC_1 and CYCLIC_LOW:

```
THREAD CYCLIC_1
TYPE CYCLIC
   WCET 500,0,0
   CALL_PO RESOURCE_PO ACQUIRE
   WCET 500,0,0

   PO BUFFER READ
END CYCLIC_1

THREAD CYCLIC_LOW
TYPE CYCLIC
   WCET 800,0,0
   CALL_PO RESOURCE_PO ACQUIRE
   WCET 1000,0,0

   PO BUFFER WRITE
END CYCLIC_LOW

PROTECTED BUFFER
TYPE PROTECTED
   ENTRY READ
      WCET 50,0,0
   ENTRY WRITE
      WCET 3000,0,0
END BUFFER
```

Note, calls to BUFFER are not on the worse case path of either thread. However, in this case additional blocking will be introduced for all higher priority threads since CYCLIC_LOW is the lowest priority thread (i.e. has the longest deadline in this case). The Schedulability Analyser detects this situation and because it results in longer response times for the threads provides an additional report, see below:

```
Schedulability Analyser
***********************

Beginning of the Offline Analyser Report

Current date and time : 17/3/1997::14:45:19

Name of the input files :
UCF Files : /export/home/awalter/hrt_tools/testing/um_eg.ucf
ESF Files : /export/home/awalter/hrt_tools/testing/um_eg4.esf
RTS Files : /export/home/awalter/hrt_tools/testing/demo1.rts

Scheduling Policy : ADS   Blocking Protocol : IPCI   ATAC Mode : Not selected

List of defined threads
_______________________

In the following report, all threads are defined by a numeric identifier :

   Identifier   Thread
   1............CYCLIC_1
   2............CYCLIC_2
   3............CYCLIC_LOW
   4............IT_SPORADIC
   5............SW_SPORADIC

List of defined Protected Objects
_________________________________

In the following report, all Protected Objects are defined by a numeric identifier :

   Identifier   Protected Object
   1............RESOURCE_PO
   2............BUFFER
   3............SW_SYNCH_PO
```

List of all schedulable and non schedulable threads :
****

Sch	Th#	Criticality	Deadline	Prio	WCCT FR	WCCT SR	Period MIAT	Response Time	Blocking Time	Blocking Origin	Utilisation Factor	Margin Analysis
yes	4	INTERRUPT	10000	63	2000	2000	10000	2000	0	RTS	2.00000E-01	0.00E+00
yes	1	HARD	10000	5	2000	2000	10000	5000	1000	PO # 1	4.00000E-01	0.00E+00
yes	2	HARD	10000	3	1500	1500	10000	6500	1000	PO # 1	5.50000E-01	0.00E+00
yes	5	HARD	10000	2	1700	1600	10000	8200	1000	PO # 1	7.20000E-01	0.00E+00
yes	3	HARD	15000	1	2800	2800	15000	10000	0	RTS	9.06667E-01	0.00E+00

List of all protected objects
****

PO #	Prio	Protected Type	PO User List
1	7	RESOURCE	T#1, T#3,
2	6	RESOURCE	T#1, T#3,
3	4	SYNCHRONISATION	T#2, T#5,

Thread Set Utilisation Factor
****

The Utilisation factor for the whole thread set is : 9.06667E-01

Displaying Alternate Analysis Report :
****

Schedulability Analyser
****

Beginning of the Offline Analyser Report

Current date and time : 17/3/1997::14:45:20

Name of the input files :
UCF Files : /export/home/awalter/hrt_tools/testing/um_eg.ucf
ESF Files : /export/home/awalter/hrt_tools/testing/um_eg4.esf
RTS Files : /export/home/awalter/hrt_tools/testing/demo1.rts

Scheduling Policy : ADS Blocking Protocol : IPCI ATAC Mode : Not selected

List of defined threads
****

In the following report, all threads are defined by a numeric identifier :

```
   Identifier   Thread
   1............CYCLIC_1
   2............CYCLIC_2
   3............CYCLIC_LOW
   4............IT_SPORADIC
   5............SW_SPORADIC
```

List of defined Protected Objects

In the following report, all Protected Objects are defined by a numeric identifier :

```
   Identifier   Protected Object
   1............RESOURCE_PO
   2............BUFFER
   3............SW_SYNCH_PO
```

List of all schedulable and non schedulable threads :

Sch	Th#	Criticality	Deadline	Prio	WCCT FR	WCCT SR	Period MIAT	Response Time	Blocking Time	Blocking Origin	Utilisation Factor	Margin Analysis
yes	4	INTERRUPT	10000	63	2000	2000	10000	2000	0	RTS	2.00000E-01	N/A
yes	1	HARD	10000	5	2000	2000	10000	7000	3000	PO # 2	4.00000E-01	N/A
yes	2	HARD	10000	3	1500	1500	10000	8500	3000	PO # 2	5.50000E-01	N/A
-NO	5	HARD	10000	2	1700	1600	10000	15700	3000	PO # 2	7.20000E-01	N/A
yes	3	HARD	15000	1	2800	2800	15000	10000	0	RTS	9.06667E-01	N/A

```
List of all protected objects
*****************************
```

PO #	Prio	Protected Type	PO User List
1	7	RESOURCE	T#1, T#3,
2	6	RESOURCE	T#1, T#3,
3	4	SYNCHRONISATION	T#2, T#5,

```
Thread Set Utilisation Factor
*****************************

The Utilisation factor for the whole thread set is :  9.06667E-01
```

5.5 Simulation of Scheduling of Application

The Scheduler Simulator provides the developer with a detailed understanding of the scheduling behaviour of an application. The behaviour is presented to the developer through recording and display of the scheduling events which occur when the application is run.

In addition, the user is able to select an inter-arrival distribution function for interrupt sporadics through the Scheduler Simulator user interface.

The developer can view the results of the simulation either graphically through a Gantt chart representation or textually. The graphical information displayed at any one time depends on the number of threads in view and the time scale, with the user being able to zoom in and out as required. The following types of event are indicated on both displays:

- start and completion of thread;
- suspension and resumption of thread;
- entry and exit of protected object;
- indication of thread's current priority;
- release of thread and associated deadline;
- missing of thread deadline;
- context switches;

The developer is able to repeat the simulation changing the modelling for the interrupt sporadics as required until sufficient simulation data has been produced.

The Scheduler Simulator should always be used in tandem with the Schedulability Analyser to assess the application. The Scheduler Simulator is particularly useful in being able to establish when the deadline is missed and what effect the changing of inter-arrival times for interrupts would have.

The Scheduler Simulator provides a good visual representation of the percentage of time used within each thread for the RTS operations. In addition, it provides a statistical report containing the CPU and RTS utilisation, on a per thread and whole application basis, over the course of the simulation. This report also provides timing data on the maximum and minimum margins by which a thread meets or exceeds its deadline. An example of the statistic report output is provided in below.

Following the production of the simulation the changes outlined in §5.4 should be applied.

THREAD	TIME				ACHIEVED DEADLINE		MISSED DEADLINE	
	CPU		RTS		min	max	min	max
	SUM	%	SUM	%				
CYCLIC_1	38750	15.50%	0	0.00%	6300	8450	----	----
CYCLIC_2	37500	15.00%	0	0.00%	4200	6950	----	----
CYCLIC_LOW	49300	19.72%	0	0.00%	5200	10350	----	----
IT_SPORADIC	48000	19.20%	0	0.00%	8000	8000	----	----
SW_SPORADIC	42500	17.00%	0	0.00%	5250	7800	----	----
CYCLIC_1_COMPLETE	16850	6.74%	0	0.00%	5750	11600	----	----

```
Simulation_Time :  250000
Processing_Time :  232900
Overall CPU     :  93.16
Overall RTS     :   0.00
```

Scheduler Simulator Statistics showing CPU, RTS and Overall Processing Data (in microseconds and percentages)

5.6 Generation of Application

The generation of the application consists of binding and linking in order to produce a machine code executable version of the application.

The bind and link facilities of the Ada Compilation System are used to produce a load module of the application, see [RD6]. The user would also be expected to use the management and consistency checking facilities provided by the compiler system to ensure that a complete and consistent set of code is used.

Note, in the case of automatic generation of WCET information as described in §5.2, a sufficient amount of the application will have to be generated in order to provide the Execution Skeleton used for the analysis. This sufficient amount corresponds to precisely an Ada closure, in that a subsystem can be analysed and simulated in isolation providing their closure is complete.

5.7 Testing of Application by Simulation

The ERC32 Simulator and the debugger from the Ada Compilation System allow the user to perform in-depth testing of an application. The user can either test through the debugger or by direct access to the simulator.

The user must configure the ERC32 Simulator to be consistent with the actual board configuration (e.g. setting memory sizes). Full details of these requirements are provided in the ERC32 Simulator User Manual [RD5]. In the case of using the Ada debugger, additional minor configuration is required. This is also addressed in the ERC32 Simulator User Manual.

Both the Ada Debugger and ERC32 Simulator user interfaces provide commanding to allow for the load and execution of an application and both provide scripting facilities to automate testing and result collection. It should be noted that it is only possible to have Ada source code interfaces to variables and statement through the debugger, however, assembly level code windows are possible through the ERC32 Simulator interface.

The ERC32 Simulator provides accurate timing facilities that allow the user to time between the execution of two assembler level instructions. This allows the developer of HRT applications to verify the WCET data generated for the threads. Examination of the source code map will allow the user to establish the start and end instruction addresses for a particular thread. The user can then set up these addresses as timing marks and execute the application to generate the exact timing for particular threads. Note, however that the user must account for any interference in the execution of the thread by higher priority threads.

The Ada RTS also supports the tracing of scheduling events within the RTS. This facility is described in detail in the Ada Cross Development Guide [RD15]. The following events and times are recorded:

- The *Release* of a task, i.e. when it becomes ready to run
- Context Switch
- The start of a delay for the current task
- The expiry of a delay for a task

- Entry to a *Protected Object* (passive task)
- Exit from a Protected Object
- Suspension of current task on a closed *barrier* in a Protected Object
- Release of a task from its Protected Object barrier due to the barrier becoming open

The user should generate this information to compare to the similar information provided by the Scheduler Simulator. The correspondence should take into account the occurrence of the interrupts and will provide a verification of the analysis.

5.8 Testing of Application on Target

Finally following the availability of the actual target breadboard, the application can be tested in its real environment. The ERC32 software tool set Ada Debugger can be attached to the breadboard providing an appropriate link is provided (see [RD9]).

An executable image containing the AlsyMonitor must be generated to allow cross-debugging, and detailed testing to be performed.

The user is also able to generate schedule traces in this configuration and it is recommended that this be adopted to provide further verification of the analysis.

This page intentionally left blank

APPENDIX A: EXAMPLE OF A TYPICAL UCF

A.1 Schedulable UCF

```
PREFERENCES
   ADS
   BLOCKING PROTOCOL IPCI
END

TARGET CHARACTERISTICS
   NO ATAC
   PRIORITY LOW 1
   PRIORITY HIGH 63
   CPU CLOCK FREQUENCY 1 MHZ
   WAIT STATES READ 0 WRITE 0
END

THREAD DEFINITION
   THREAD CYCLIC_1
      CRITICALITY HARD
      PERIOD 10000
      DEADLINE 10000
   END CYCLIC_1

   THREAD CYCLIC_2
      CRITICALITY HARD
      PERIOD 10000
      DEADLINE 10000
   END CYCLIC_2

   THREAD CYCLIC_LOW
      CRITICALITY HARD
      PERIOD 15000
      DEADLINE 15000
   END CYCLIC_LOW

   THREAD IT_SPORADIC
      MINIMUM 10000
      DEADLINE 10000
   END IT_SPORADIC


   THREAD SW_SPORADIC
      CRITICALITY HARD
      MINIMUM 10000
      DEADLINE 10000
   END SW_SPORADIC

END

WCET DATA
-- Text for WCET
END
```

A.2 UCF Additions for a Sporadic Thread

```
   THREAD CYCLIC_1_COMPLETE
      CRITICALITY HARD
      MINIMUM 10000
      DEADLINE 20000
   END CYCLIC_1_COMPLETE
```

A.3 UCF modifications for a Periodic Thread

```
THREAD CYCLIC_1_COMPLETE
   CRITICALITY SOFT
   PERIOD 10000
   DEADLINE 20000
END CYCLIC_1_COMPLETE
```

APPENDIX B: EXAMPLE OF A TYPICAL ESF

B.1 Schedulable ESF

```
PROGRAM USERMANEG

THREAD CYCLIC_1
   TYPE CYCLIC
   PRIORITY 5
   WCET 500,0,0
   CALL_PO RESOURCE_PO ACQUIRE
   WCET 500,0,0
END CYCLIC_1

THREAD CYCLIC_2
   TYPE CYCLIC
   PRIORITY 3
   WCET 500,0,0
   CALL_PO SW_SYNCH_PO SIGNAL
   WCET 500,0,0
END CYCLIC_2

THREAD CYCLIC_LOW
   TYPE CYCLIC
   PRIORITY 1
   WCET 800,0,0
   CALL_PO RESOURCE_PO ACQUIRE
   WCET 1000,0,0
END CYCLIC_LOW

THREAD IT_SPORADIC
   TYPE INTERRUPT SPORADIC
   PRIORITY 63
   WCET 1000,0,0
   WCET 1000,0,0
END IT_SPORADIC

THREAD SW_SPORADIC
   TYPE SPORADIC
   PRIORITY 2
   CALL_PO SW_SYNCH_PO WAIT
   WCET 1000,0,0
END SW_SPORADIC

PROTECTED SW_SYNCH_PO
   TYPE SPORADIC
   PRIORITY 4
   ENTRY SIGNAL
      WCET 500,0,0
   BARRIER WCET 100,0,0
   ENTRY WAIT
      WCET 500,0,0
END SW_SYNCH_PO
```

```
PROTECTED RESOURCE_PO
   TYPE PROTECTED
   PRIORITY 6
   ENTRY ACQUIRE
      WCET 1000,0,0
END RESOURCE_PO

END
```

B.2 ESF Additions for a Sporadic Thread

```
THREAD CYCLIC_1
   TYPE CYCLIC
   PRIORITY 6
   WCET 500,0,0
   CALL_PO RESOURCE_PO ACQUIRE
   CALL_PO SW_SYNCH_COMPLETE SIGNAL
END CYCLIC_1

THREAD CYCLIC_1_COMPLETE
   TYPE SPORADIC
   PRIORITY 1
   CALL_PO SW_SYNCH_COMPLETE WAIT
   WCET 500,0,0
END CYCLIC_1_COMPLETE

PROTECTED SW_SYNCH_COMPLETE
   TYPE SPORADIC
   PRIORITY 7
   ENTRY SIGNAL
      WCET 50,0,0
   BARRIER WCET 100,0,0
   ENTRY WAIT
      WCET 50,0,0
END SW_SYNCH_COMPLETE
```

B.3 ESF modifications for a Periodic Thread

```
THREAD CYCLIC_1_COMPLETE
   TYPE CYCLIC
   PRIORITY 1
   WCET 500,0,0
END CYCLIC_1_COMPLETE
```

B.4 ESF modifications for Alternate analysis

```
THREAD CYCLIC_1
   TYPE CYCLIC
   WCET 500,0,0
   CALL_PO RESOURCE_PO ACQUIRE
   WCET 500,0,0
   PO BUFFER READ
END CYCLIC_1
```

```
THREAD CYCLIC_LOW
   TYPE CYCLIC
   WCET 800,0,0
   CALL_PO RESOURCE_PO ACQUIRE
   WCET 1000,0,0
   PO BUFFER WRITE
END CYCLIC_LOW

PROTECTED BUFFER
   TYPE PROTECTED
   ENTRY READ
      WCET 50,0,0
   ENTRY WRITE
      WCET 3000,0,0
END BUFFER
```

This page intentionally left blank

APPENDIX C: THE 32 BIT MICROPROCESSOR TOOL SET ENVIRONMENT

C.1 Environmental Requirements

The ERC32 Software Tool Set has been developed to run on any Sun-4/SPARC configuration under Solaris version 2.3 or later, and all user interfaces comply with the X/11 MOTIF standard.

The following Disk resources are required:

- At least 54 MB of disk space on a single file system for the Ada Compilation System;
- At least 23 MB of disk space for the ERC32 Target Simulator;
- At least 0.9 MB of disk space for the Schedulability Analyser;
- At least 2 MB of disk space for the Scheduler Simulator;
- At least 0.9 MB of disk space for the tools shared by the Schedulability Analyser and Scheduler Simulator;

C.2 Installation of the 32 Bit Microprocessor Tool Set

Instructions for the installation of each of the tools comprising the tool set are given in the following installation manuals:

AdaWorld Development Environment for SPARC based Workstations under Solaris 2: Installation Guide
MCD-ALS-P2-DOC-002, Issue 2 Revision 0, 29/11/96

Target Simulator V2.1 Delivery
32B-SBI-IM-0189-0003, V2.1, 20/12/96

Schedulability Analyser V2.1 Delivery
32B-SBI-IM-0189-0001, V2.1, 20/12/96

Scheduler Simulator V2.1 Delivery
32B-SBI-IM-0189-0002, V2.1, 19/12/96

This page intentionally left blank

This page contains document configuration data and is to be thrown away prior to issue:

bm_cover_issue: **1**
bm_cover_rev: **0**
bm_cover_date: **15-March-1997**

bm_header_issue: 1
bm_header_rev: 0
bm_header_date: 15/03/97