A Non-Intrusive Fault Tolerant Framework For Mission Critical Real-Time Systems

by

Sébastien Gorelov

B.Eng, Electrical Engineering
McGill University, 2003

SUBMITTED TO THE DEPARTMENT OF AERONAUTICS AND ASTRONAUTICS IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE IN AERONAUTICS AND ASTRONAUTICS
AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUNE 2005

Signature of Author: ____________ [illegible]
Department of Aeronautics and Astronautics
May 20, 2005

Certified by: ____________________
I. Kristina Lundqvist
Charles S. Draper Assistant Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by: ____________________
Jaime Peraire
Professor of Aeronautics and Astronautics
Chair, Committee on Graduate Students

A Non-Intrusive Fault Tolerant Framework For Mission Critical Real-Time Systems

by

Sébastien Gorelov

Submitted to the Department of Aeronautics and Astronautics
on May 20, 2005 in Partial Fulfillment of the
Requirements for the Degree of Master of Science in
Aeronautics and Astronautics

ABSTRACT

The need for dependable real-time systems for embedded application is growing, and, at the same time, so does the amount of functionality required from these systems. As testing can only show the presence of errors, not their absence, higher levels of system dependability may be provided by the implementation of mechanisms that can protect the system from faults.

We present a framework for the development of fault tolerant mission critical real-time systems that provides a structure for flexible, efficient and deterministic design. The framework leverages three key knowledge domains: firstly, a software concurrency model, the Ada Ravenscar Profile, which guarantees deterministic behavior; secondly, the design of a hardware scheduler, the RavenHaRT kernel, which further provides deadlock free inter-task communication management; and finally, the design of a hardware execution time monitor, the Monitoring Chip, which provides non-intrusive error detection.

To increase service dependability, we propose a fault tolerance strategy that uses multiple operating modes to provide system-level handling of timing errors. The hierarchical set of operating modes offers different gracefully degraded levels of guaranteed service. This approach relies on the elements of the framework discussed above and is illustrated through a sample case study of a generic navigation system.

Thesis Supervisor: I. Kristina Lundqvist
Title: Charles S. Draper Assistant Professor of Aeronautics and Astronautics

À mes parents.

Acknowledgements

I wish to recognize the people I owe the most. First, I want to thank the members of my family, my parents Bernadette et Simon, my Babouchka, Evgenia, my Sister Valérie and her husband Sven, my Brother Cyril and his wife Pascale, for their unconditional love, patience and support.

Secondly, I want to thank my advisor, Professor Kristina Lundqvist, who, starting with her famous first words “I know you, but you don’t know me yet”, helped me go through my studies at MIT with the relevance of her guidance, the support of her understanding, the freedom of her trust and the warmth of her friendship. I gratefully acknowledge her financial support.

I wish to thank Jayakanth Srinivasan for his reliably contagious energy and passionate enthusiasm. Thanks to the camaraderie of the larger ESL team, Kathryn, Martin, Gaston, Pee, Carl, and Anna, and especially the Swedish delegation, Johan Furunäs for his friendship, expertise and ironmanship, and Lars Asplund for his good humor and hospitality during my stay in Vasterås.

Finally I want to thank my second family, to whom I owe so many good times. Yves, you should be thanked as a friend, a roommate, a desert raider, a Montréal host (mmmgggnhehe), a coworker and a thesis proofreader. Jacomo, thanks for having invented the Jacomo-way-o-life and having taught me its guidelines. Fluff, it was an honor to enjoy your company all those times. Nakaitso, your love of the land is inspiring. Eugene, you’ve been keeping it real since the days of Stanislas. Special thoughts go to all my remote family members and friends.

Finally, I’d like to thank the Whitehead Institute cafeteria crew for working hard to provide us with the best moment of the day, every day.

Avwayons d'l'avant nos gens !

Table of Contents

ABSTRACT **2**

Acknowledgements 4

TABLE OF CONTENTS **5**

FIGURES AND TABLES **8**

CHAPTER 1 .... **9**

INTRODUCTION **9**

1.1 Mission Critical Real-Time Systems 9
1.2 Motivation 9
1.3 Approach 10
1.4 Fault Tolerance and Non-Intrusive Error Detection 10
1.5 Elements of the Gurkh Project 11
1.6 Thesis Organization 12

CHAPTER 2 .... **13**

BACKGROUND **13**

2.1 Scheduling in Real-Time Systems 13
2.2 Gurkh Project Enabling Technologies 14
2.2.1 Model Checkers *15*
2.2.2 System-On-Chip Solutions *16*
2.2.3 The Ravenscar Profile of Ada95 .... *16*
2.3 Overview of the proposed approach 17
2.4 Relevant Work 18
2.4.1 Model-Integrated Computing *18*
2.4.2 Execution Timekeeping *20*
2.4.3 Concurrent Hardware Monitoring *22*

CHAPTER 3 .... **24**

3.1 The Profile 24
3.2 The Hardware Kernel 26

CHAPTER 4 .... **30**

4.1 Design 31
4.2 Implementation 33
4.2.1 VHDL Implementation Files Added: po_ntrpt.vhd *33*
4.2.2 VHDL Implementation Files Added: ntrpt_manager.vhd *35*
4.3 VHDL Implementation Files Modified *36*
4.3.1 Arbitrating RQ RAM accesses *36*
4.3.2 Arbitrating the Processor *38*
4.4 Development and Testing 38
4.5 System Developer's User Guide 39
4.6 Future Improvements 42

CHAPTER 5 .... **43**

5.1 Monitoring Chip As A Component Of RavenHaRT 43
5.1.1 High Level Hierarchy *43*
5.1.2. Inputs To The MC *44*
5.1.3 Outputs Of The MC *45*
5.1.4 Input Drivers *45*
5.1.4.2 Signals driven by independent logic *46*
5.1.4.3 Signals driven by BISM *46*

5.1.5 Output Destination *46*
5.2 MONITORING CHIP ARCHITECTURE 47
5.2.1 The sc_top Component *47*
5.2.2 The sc_x Component *49*
5.2.3 Possible Sources of MC Detected Error *53*
5.3 TASK GRAPH CODING 54
SPECIAL ACKNOWLEDGMENT 59

CHAPTER 6 60

6.1 FAULT TOLERANCE THEORY 60
6.1.1 Fault Tolerance Phases *60*
6.1.2 Issues in Recovery for Cooperative Concurrent Processes *61*
6.2 STARTING WITH ERROR DETECTION 62
6.2.1 Transient Fault Assumption *62*
6.2.2 Concurrent error detection *63*
6.2.3 The Importance of the Application Context *63*
6.3 THE GURKH GENERIC NAVIGATION SYSTEM 63
6.3.1 The Gurkh Generic Navigation System as an Application Testbed *63*
6.3.2 GGNS Task Model *64*
6.4 FAULT TOLERANT STRATEGY 65
6.4.1 Creating The Fault Modes *65*
6.4.2 Making Design-Time Fault Models Available at Run-Time *68*
6.4.3 The Monitoring Chip Interrupt Handling Task *68*
6.4.4 Isolating the Erroneous Task *69*
6.4.5 Monitoring Chip Task Graph Fault Mode Synchronization *72*
6.4.6 Switchback *72*
6.5 DISCUSSION 73
6.5.1 Damage Confinement *74*
6.5.2 Damage Assessment *74*
6.5.3 Error Recovery *75*
6.5.4 Fault Treatment and Continued Service *75*
6.5.5 Mode Change Transition *75*
6.5.6 Limitations *76*
6.6 IMPLEMENTATION ISSUES 77
6.6.1 Checkpointing and Rollback *77*
6.6.2 Switchback *77*
6.6.3 Mode Change Coordination *78*
6.6.4 Building the MCIHT Lookup Table *79*
6.6.5 Coordination with RavenHaRT PO Circuits *80*

CHAPTER 7 81

7.1 MAJOR CONTRIBUTIONS 81
7.2 DIRECTIONS FOR FUTURE WORK 82
7.2.1 A Full-scale Application *82*
7.2.2 Refinement of Modeling Techniques *82*
7.2.3 Automated Monitoring Chip Hardware Description Generation *82*
7.2.4 Transfer of Fault Tolerant Software Functionality to Hardware *83*
7.2.5 RavenHaRT Kernel Upgraded for System Interrupts *83*
7.2.6 Reversing the role of the Monitoring Chip *84*

APPENDIX 88

FPGA SPACE SYNTHESIS REQUIREMENTS FOR FULL-SCALE APPLICATIONS 88

THE OLYMPUS ATTITUDE AND ORBITAL CONTROL SYSTEM 88
ASSUMPTIONS 88
SYNTHESIS PLATFORM 89

RAVENHART 89
RAVENHART AND THE MC 90

Figures and Tables

FIGURE 1: SYSTEM ARCHITECTURE 17
FIGURE 2: TASK TIMING PARAMETERS 20
FIGURE 3: ARCHITECTURE OF KERNEL AND APPLICATION 26
FIGURE 4: RAVENHART HARDWARE ARCHITECTURE DIAGRAM. NEW COMPONENTS ARE MARKED WITH A (*). 28
FIGURE 5: SYSTEM INTERRUPT VS. PROCESSOR INTERRUPT 30
FIGURE 6: COMPARING SOFTWARE AND HARDWARE ENTRY RELEASE MECHANISMS 32
FIGURE 7: THE PO_NTRPT STATE MACHINE 33
FIGURE 8: THE NTRPT_MANAGER STATE MACHINE 36
FIGURE 9: RECEIVING SENSOR DATA FROM HARDWARE 40
FIGURE 10: MC AS AN ELEMENT OF RAVENHART: ARCHITECTURE DIAGRAM 44
FIGURE 11: THE ERRPROC STATE MACHINE 48
FIGURE 12: THE SC STATE MACHINE 51
FIGURE 13: LIST OF SOFTWARE CALLS TO RAVENHART FOR A PO PROCEDURE 55
FIGURE 14: THE TASK GRAPH OF GGNS'S LOW RATE NAV TASK 57
FIGURE 15: EXAMPLE ROM TABLE TASK TIMING CODING 58
FIGURE 16: TASK MODEL OF GGNS 64
FIGURE 17: GGNS FAULT SCENARIOS, FAULT CLASSES, AND FAULT MODES A TO E 66
FIGURE 18: SWITCHING MODES 68
FIGURE 19: MCIHT ACTIVITY 69
FIGURE 20: FAULT MODE TRANSITION AND SWITCHBACK 71
FIGURE 21: XILINX 2VP30 CHIP UTILIZATION AND TIMING SUMMARY FOR RAVENHART SYNTHESIS OF AN APPLICATION REQUIRING UP TO 15 TASKS AND 31 POS 90
FIGURE 22: XILINX 2VP30 CHIP UTILIZATION AND TIMING SUMMARY FOR RAVENHART AND MC SYNTHESIS OF AN APPLICATION REQUIRING UP TO 15 TASKS AND 31 POS 91

TABLE 1: CONFIGURATION OF THE DEVELOPMENT VERSIONS 2.3 AND ABOVE OF RAVENHART 39
TABLE 2: DECODING THE ERROR_CODE SIGNAL 53
TABLE 3: ROM TABLE OF TASK CALLING A PO PROCEDURE 56
TABLE 4: GGNS TASK BEHAVIOR FOR EACH FAULT MODE 67
TABLE 5: MCIHT LOOKUP TABLE 80

Chapter 1
Introduction

The information revolution of the past few decades concerning computing technology continues to drive the changes in our societies at a formidable rate. Software enabled systems become omnipresent in domains as varied as manufacture, communication, finance, transport, exploration, and health.

1.1 Mission Critical Real-Time Systems

This thesis is concerned with a specific class of software enabled systems called *real-time systems*. Real-time systems are those computing machines in which the correctness of the system depends not only on the logical result of the computation but also on the time at which the result is produced [BW01]. Also called *embedded systems*, real-time systems typically occupy a privileged commanding position in the control loop of the larger systems they are embedded in [Liu00]. Real-time systems act as the brain of a wide class of engineering systems, from dishwashers and portable phones to nuclear power plants and deep space probes. The impact that a failure of such engineering systems can cause to their user or to the external world is what distinguishes *mission critical engineering systems*. This impact could involve monetary loss or physical threat to human lives. Space-based assets and modern jet aircrafts are examples of such systems. Laundry machines and media players, in contrast, are not mission critical engineering systems. When the embedded system provides a critical capability to its associated mission critical engineering system, it is, in turn, referred to as *a mission critical real-time system.*

Note that safety requirements can be a subset of the engineering system's mission requirements, and thus we consider here that the mission critical systems class includes safety critical systems.

1.2 Motivation

High technology industries (like aeronautics, astronautics, automotive, power, and health technologies) that use mission critical real-time systems face unprecedented commercial pressure to optimize their real-time systems design process. This translates practically into an urgent need for design processes that are shorter in time, less expensive, and that can yield real-time systems that provide more functionality while continuing to offer, over extended periods of time, the high levels of dependability required by their mission [VAR99].

In parallel, the trust which society places in mission critical systems grows ever higher. In reaction, industry and governments have established different standards and certification processes to guarantee that given mission critical real-time systems meet their required level of dependability.

Consequently, design processes that can easily prove the dependability of their systems are sought.

1.3 Approach

The Gurkh Project was created as a first step towards the development of a new design process which answers the needs of mission critical real-time system industries. The Gurkh Project aims at providing an integrated environment for the design, development, formal verification and validation, and maintenance of mission critical real-time systems [AL03]. To do so, tool support is combined to System-On-Chip methodologies. Starting with existing requirement specifications or existing code, the resultant process is meant to minimize design time and cost while maximizing functionality and dependability.

The authors of [ALL+04] identify four means of attaining dependability: fault prevention, fault removal, fault tolerance, and fault forecasting. While different elements of the Gurkh Project are designed to enable fault prevention and fault removal, the work presented in this thesis is concerned with the increase of dependability by the provision of a framework for fault tolerance to the design process of the Gurkh Project.

1.4 Fault Tolerance and Non-Intrusive Error Detection

The literature [ALL+04], [LA90] considers a *fault* to be defined by its ability to manifest itself and cause the internal state of a system to transition to an erroneous state which could lead to a *failure*. A failure is a deviation of a system's behavior from that required by its specification, or from the intended system behavior inadequately described by the specification. The *error* is the element of the erroneous state which makes it different from a valid state.

The practical impossibility to eliminate all development faults of complex software and hardware systems is widely recognized in the computing community [LA90]. As opposed to fault prevention and removal, fault tolerance aims at providing the means for a real-time system to cope with faults to avoid failures *after* the system comes into service.

The intuitive principle behind the implementation of fault-tolerance in a system is to provide redundant resources and the mechanisms necessary to make use of them when needed. These mechanisms typically act in four phases [LA90]:

- error detection: the starting point of fault tolerance techniques is the detection of an erroneous state, which is the consequence of a fault
- damage confinement and assessment: the extent to which the system state has been damaged should be evaluated based on the damage confinement structure of the system.
- error recovery: techniques for error recovery allow the transfer from the current erroneous system state to a predetermined error-free state from which predictable operation can continue.
- fault treatment and continued service: if the fault can be identified, fault treatment will repair it to prevent it from reoccurring.

Provision of error detection can either change the target system's fault-free behavior, in which case it is called *intrusive* error detection, or it can work without interfering with any of the nominal, fault-free activities of the system being monitored, in which case it is called *non-intrusive* error detection. Non-intrusive schemes have the advantage of making the system design process less error-prone by decoupling the functionality of the different elements in the system.

1.5 Elements of the Gurkh Project

The Gurkh Project environment consists of four main components:

1. A set of tools translating software and hardware description languages to a formal notation (timed state machines) that can be used for the verification of the target application and the run-time environment infrastructure [Neh04].
2. A formally verified and predictable hardware run-time kernel called RavenHaRT [Sil04].
3. An Ada Ravenscar to PowerPC and RavenHaRT kernel cross-compiler called pGNAT [See04].

4. An on-line hardware monitoring device, called the Monitoring Chip, created using a model of the target system application.

The work presented in this thesis concerns the integration of the Monitoring Chip in the Gurkh Project environment to provide a non-intrusive framework for fault tolerant designs of mission critical real-time systems.

1.6 Thesis Organization

This thesis is organized as follows. Chapter 2 provides an overview of the approach proposed and provides background information that motivates the approach chosen: enabling technologies and relevant work will be presented. Chapter 3 is dedicated to Ravenscar, the deterministic and concurrent subset of the Ada 95 software language, and to the RavenHaRT run-time kernel, designed for Ravenscar compliant software applications. Chapter 4 presents the modifications brought to RavenHaRT to give the run-time system interrupt capability. Chapter 5 describes the implementation of the Monitoring Chip, the error detection component enabling the fault tolerant solution of the system. Chapter 6 then discusses the strategy used to implement the fault tolerant framework and the existing integration issues. Finally Chapter 7 presents the conclusions of this work and suggests future avenues of work worth considering. The Appendix reports the FPGA gate-space requirements for the synthesis of the RavenHaRT and Monitoring Chip hardware needed to support a typical full-scale application.

Chapter 2
Background

The Gurkh Project uses preemptive priority based scheduling as a central paradigm to its process. In this Chapter, we provide background information on scheduling methods to justify this choice. To further motivate the general approach followed, enabling technologies and relevant work are presented, and an overview of the proposed approach is presented.

2.1 Scheduling in Real-Time Systems

Real-Time Systems have traditionally been built using cyclic executive approaches. For such systems, the application software consists of a fixed set of tasks executing periodically. The cyclic executive schedule is a timeline of task execution arranged such that each task runs at its correct frequency. The major cycle is the time period of execution of the complete timeline. The major cycle is composed of minor cycles, one for each task's own higher frequency execution requirements. If different tasks have different frequency requirements, the major cycle must have a frequency set to the least common multiple of all required frequency [VAR99]. Since the schedule is carefully elaborated in design time, tasks can share a common, unprotected address space.

The guarantee of deterministic behavior that is provided by the cyclic executive makes its usage attractive in mission critical real-time systems. For simple periodic systems, the cyclic scheduler approach is very effective. However, the scheduling problem is known to be computationally infeasible for large systems and its quickly growing complexity requires suboptimal schemes to be used in practice [BW01]. Furthermore, the necessity for tasks to share a harmonic relationship imposes artificial timing requirements [VAR99] that can be wasteful in processor bandwidth [Pun97]. Cyclic executives are difficult to construct, thus making them very inflexible to changes. For instance, adding a new task with large computation time or very low frequency requirements in the application requires major changes and rework of the complete system's schedule. While this method is very rigid, it presents the advantage of being deterministic.

An alternative approach to cyclic executive is concurrent tasking. This more flexible approach uses task attributes to determine which task should run at any time. These scheduling attributes, which can include the task's deadline, its worst-case execution time (WCET), its period, or its worst-case response time, are usually used to define the priority of a task. In effect, the task's priority is thus based on its timing requirements. The use of fixed-priority tasks is a popular approach to the scheduling problem; schedules composed of such tasks can be computed at design time and lend themselves well to analysis. In fixed priority preemptive schemes, a higher priority task becoming runnable while a lower priority task is running will be immediately granted the processor. In non-preemptive schemes, the running task will be completed before the runnable task starts executing. Tasks can therefore be in the following states: running, runnable, suspended on a timing event (periodic tasks), or suspended on a non-timing event (event-triggered tasks) [BW01].

An important advantage of concurrent methods is that minor design changes result in small implementation modifications, which is not the case for cyclic executive approaches. Although a lot more flexible, traditional concurrent tasking does not typically reach a level of determinism appropriate for certification [ISO99]. The motivation behind the creation of the restrictive Ravenscar Ada 95 language subset is precisely to allow for enough determinism in concurrent designs to satisfy the certification requirements of safety critical real-time systems [BAJL99].

As mentioned in the introduction, the need for better design processes contributes to the emergence of concurrency as the preferred approach [PZ03], even in traditionally very conservative mission critical application domains like spacecraft engineering [BW+93] [VAR99].

By using the tasking model of the Ravenscar profile as one of the pillars of the Gurkh Project, we aim to combine the benefits of the flexibility of concurrent designs with the benefits of the dependability of deterministic system implementations.

2.2 Gurkh Project Enabling Technologies

An embedded system's functionality can be implemented using both hardware and software components. The share of responsibility of functionality implemented using hardware and software components can vary between the two extremes of pure software functionality and pure hardware functionality.

Current practice usually defines the appropriate standard of distribution of functionality between software and hardware in a system. This standard is challenged with the emergence of new technologies.

In particular, our research has been enabled by the emergence of three technologies at the basis of our framework: model checkers, System-on-Chip implementation methodologies, and the Ravenscar profile of Ada 95.

These emerging sets of technical options are a consequence of the advances of computer and software engineering research. Increasing computing power is made available. Better algorithms and more computing power allow more efficient and powerful model checkers. Advances in semiconductor technology miniaturization yield Field Programmable Gate Arrays (FPGA) that can contain more gates. System-on-Chip methodologies make hardware design and its desirable features of parallelism, robustness and speed more accessible than ever.

2.2.1 Model Checkers

Model checking is an integral part of the Gurkh Project process. The Uppaal tool suite offers an integrated environment to model, validate and verify real-time systems modeled as networks of timed-automata [PL00]. The use of the networks of timed-automata modeling technique and the Uppaal model-checking tool in particular have been deemed an effective method to verify concurrent real-time systems models by both academia [Bur03] and industry [DY00]. System modeling can be done in different ways. Modeling techniques for mission critical real-time systems must allow the representation of concurrency, delays, and deadlines [Bur03]. Automatic verification then checks the feasibility and safety of the system model.

The Gurkh Project development process uses the Uppaal tool to verify application designs at different phases of their development lifecycle: the tool can be used at the beginning of the process or to incorporate existing components into the system model.

The use of Uppaal in the Gurkh Project process offers other advantages than formal verification of application system models. The ability for the Uppaal user to simulate a model's timing behavior has an ancillary benefit in that the user incidentally gains a valuable in-depth understanding of the bounded sequence of interactions that take place within the system. Furthermore, hardware/software integrated modeling showed that Uppaal can be used to help define the boundary between hardware and software components. The advantage of this is that hardware/software co-design decisions can be delayed in the design process to the moment where the optimal breakdown solution is found, and a well defined interface between hardware and software is obtained.

Finally, the application system model is used to derive the timing model used by the monitoring device, the Monitoring Chip, to verify the timely and correct execution flow of the system tasks.

The Uppaal tool was used to show the feasibility of the design of a Ravenscar compliant Run-Time kernel in [AL03b]. This formally verified design was used to implement the RavenHaRT hardware kernel.

2.2.2 System-On-Chip Solutions

A System-On-Chip (SoC) is an implementation technology that typically contains one or more processors, a processor bus, a peripheral bus, a component bridging the two buses and several peripheral devices [Mos00]. In the implementation technology we use for our research, the SoC consists of one or more processors and FPGA fields that can be easily configured to present all the hardware mentioned above and more [Xil04]. The benefit of this technology lies in the great flexibility we have in defining the hardware/software interface. As mentioned above, this allows us to challenge the current approach to solving real-time system problems.

The Gurkh Project development platform uses this implementation technology. The software running on the SoC processor is scheduled by the RavenHaRT runtime kernel which is synthesized as a peripheral device. The Monitoring Chip is synthesized as another peripheral device of the chip's processor. Figure 1 shows the system architecture of the Gurkh Project. The SoC development platform used at MIT's Aero/Astro Embedded System Lab consists of Xilinx's Virtex-2 Pro hardware. The ML310 boards used for project development have 2 PowerPC processors each, along with over 30,000 FPGA fabric logic cells and over 2400 kb of block RAM. Xilinx's ISE foundation version 6.3, with Xilinx's Embedded Development Kit software version 6.3, along with Mentor Graphics' ModelSim SE Plus 5.8e were the tools used for design entry, simulation, synthesis, implementation, configuration and verification.

2.2.3 The Ravenscar Profile of Ada 95

Real-time software languages need to provide support for concurrency and predictability [Shaw01]. As discussed in [Sil04], the Ada programming language is highly suitable for the Gurkh Project by virtue of the build-in safety and real-time support features the language offers.

In the context of a particular application domain, restricted profiles of Ada give the possibility to keep subsets of the complete language for increased optimality. The Ravenscar profile in particular offers a concurrent tasking subset to meet the dependability, analysis, and certifiability requirements of mission critical real-time system design [BDV03]. Because of the central importance of the role of the Ravenscar profile in the Gurkh Project, Chapter 3 is dedicated to its description and that of RavenHaRT.

2.3 Overview of the proposed approach

The Gurkh Project, well described in [AL03] and [Sil04], provides the overarching context for our non-intrusive fault tolerant real-time systems development framework. The focus of the work presented in this thesis is the provision of fault tolerant capabilities in this context, and we will only summarize here the salient points of the Gurkh Project relevant to that focus.

Gurkh framework application software can be developed in Ada 95 Ravenscar. A model of the application is extracted from the code, and different model checking tools verify the necessary timeliness and correctness properties. The application software is cross-compiled to run on the target processor and to communicate with RavenHaRT, the hardware real-time kernel developed in-house for Gurkh. An integrated model of both RavenHaRT and the application can be checked by the different tools and the implementation can be corrected for the next design iteration. The timed model of the application is also used to synthesize the Monitoring Chip (MC), which provides the non-intrusive error detection capabilities by monitoring the communication exchanges between the processor and RavenHaRT. The system architecture is shown in Figure 1.

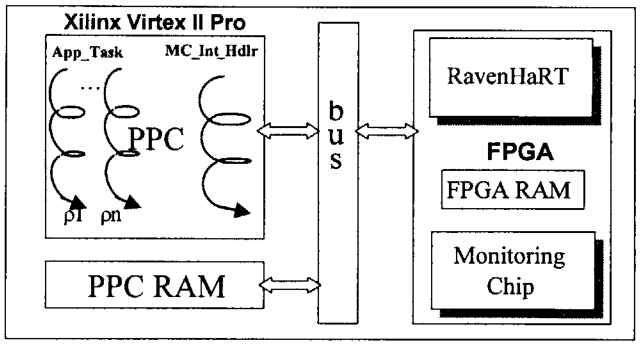


Figure 1: System Architecture

The fault tolerance strategy adopted is based on the types of errors that can be detected. The error detection capabilities offered by the MC concern two types of errors:

- illegal control flow from the software application
- tasks breaking their execution time bounds

Assuming the software faults are transients, they are tolerated using system reconfiguration by allowing the application to switch to fault tolerant modes of operation. Although the implementation details of this approach are application specific, the concept is generic and presented in detail in Chapter 6. Fault tolerant modes of operation in the presence of task errors are designed for the real-time system to offer a predictable and gracefully degraded service while waiting for nominal operations resumption.

Once the MC signals the error to RavenHaRT, the appropriate handler task is activated by the hardware kernel to switch the mode of operation of the application to the mode determined by the error information provided by the MC. Before terminating, the handler forces backward error recovery of the faulty task. The task is thus reset to the state it was in at the last checkpoint. When the handler exits, the application system runs in the appropriate mode. If the faulty task has a hard deadline, the whole mode changing operation is designed such that the faulty task is given another chance at meeting its deadline in the new mode of operation, thus exploiting time redundancy. The MC is designed to eventually restore continued nominal service.

2.4 Relevant Work

2.4.1 Model-Integrated Computing

The classical approach in computer science is to make a very clear separation between computer system's software and hardware components. The bridge between the two is usually clearly defined and standardized, allowing computing problems to abstract some of the total complexity of the system. However, embedded systems are among the computer systems in which software and physical systems exhibit the tightest coupling characteristics. Mastering such complex component interactions presents new challenges. Model-Integrated Computing answers the need existing for embedded system software and its associated physical system to evolve together in order for the embedded system to fulfill its control responsibilities [SK97].

The concept behind Model-Integrated Computing is to make design-time models of an application system available at run-time to benefit its dynamic behavior. The use of models, which first consists of capturing the computational processes required by the application, is extended to capture information about the environment the software executes in and the dynamic behavior of the application. Such a design paradigm eases the definition of functional decomposition boundaries helpful for fault prevention. It furthermore enables the application to be reflective: it can give it the ability to supervise its own operation. The Multigraph Architecture is a Model-Integrated Computing framework developed at Vanderbilt University to support domain specific Model-Integrated Program Synthesis (MIPS) environments [LBM00].

The ideas of Model-Integrated Computing are directly relevant to our work. The Gurkh Project could be seen as implementing a special case inside the generic context, an already domain-specific subset of the Multigraph Architecture. The Gurkh process aims at offering the possibility to generate models and code automatically from formal specifications. Those models are used for formal verification and model checking, as well as to implement the fault tolerant solution. They give the system the ability for the application to be reflective, or to self-reconfigure in the presence of faults. Models are also used to represent and manage resource limitations. For real-time systems, the limited resource is time. Key to the Gurkh Project is the ability for the system to capture timing relations in its models and to provide timing information to the supervisory device: the Monitoring Chip. Reconfiguration then consists of using spare time to cope with faults.

In [LBM00], Ledeczi discusses a wide range of different possible levels of self-adaptability. The paper recognizes a shortcoming in the general approach they present: mission critical applications can only accept a strictly limited and controlled form of self-adaptability. The Multigraph Architecture aims to be widely applicable and some of its components are simply not compatible with the predictability and dependability requirements of mission critical applications. The ideas of monitoring and configuration modification are present in Gurkh. The concept of having generative capability for dynamic architectures is interesting, but the Gurkh approach uses pre-designed functionality models in order to meet the predictability requirements mentioned above. Finally one of the main contributions of our work is to implement the concept of reflectivity by exploiting the advantages of hardware monitoring based on the models of the application. Again, this is the role of the Monitoring Chip.

2.4.2 Execution Timekeeping

The predictable timing requirements of real-time systems impose strict timing analysis constraints on the design process. Modern processors benefit from architectural features like caches and pipelines designed to speed up the execution time of individual instructions, which consequently varies as a function of many factors. For real-time applications, the necessary timing analysis is thus rendered very complex or inaccurate, and the use of latest generation processors not a privileged option [WA02]. Timing analysis accuracy is important; whereas overestimation of the worse case execution time (WCET) of a block of code results in poor resource utilization, underestimation invalidates the analysis required for timeliness certification, which is a much worse problem. In [WA02], Ward and Audsley have pushed the idea of using VHDL technology to synthesize Ada Ravenscar code into hardware circuits with the stated purpose to get the most accurate timing analysis possible from software. This effort proves the importance of having accurate timing analysis.

Hard real-time system design incorporates the task's deadline as a constraint, and uses the estimated Best-Case Execution Time (BCET) and WCET as design parameters to accomplish the objective of satisfying the deadline constraint. The BCET and WCET are contained inside the time period between task activation and its deadline.

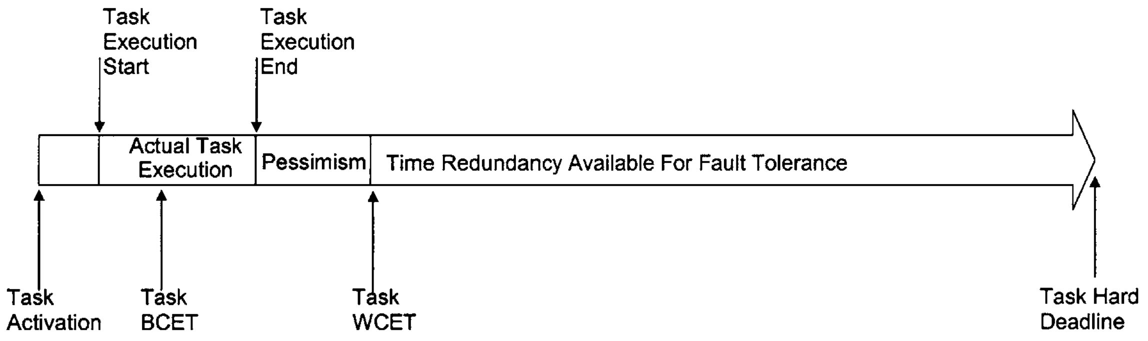


Figure 2: Task Timing Parameters

Although reduced to a minimum, the inaccuracy of the timing bounds estimate for practical application is never eliminated completely. The difference between the actual execution time and the worst-case estimate is referred to as pessimism in the literature, and is illustrated in Figure 2. On the one hand, the WCET is chosen to feature some pessimism to guarantee a safe upper bound on task execution time. On the other hand, too much pessimism leads to inefficient resource usage. System design relies on the guarantee offered by the WCET. Hence, breaks in timing bound estimates are dangerous, and should be detected and handled.

Typical temporal redundancy schemes repeat the execution of a faulty task using the time remaining after the fault is detected and before the deadline occurs [MM01]. This extra time between a task's WCET and its deadline is the spare time that is exploited to carryout the fault tolerance approach we propose.

Harbour mentions the importance for hard real-time systems to monitor the execution time of all tasks [HA03]. While monitoring of task execution times is routinely done for cyclic executives, the constructs required to support such capability for concurrent systems are not as easily implemented. Previous work by Harbour et Al. [HR+98], proposes the addition of an execution time clock library for Ada 95 that can be used to monitor the timing behavior of the executing application, and in [HA03], Harbour and Aldea Rivas suggest an extension to that library allowing a single task to manage the execution-time budgets of several other tasks. However, while the execution clock mechanism themselves are found compatible with the Ravenscar profile, the overrun management schemes proposed in [HR+98] use techniques such as dynamic priorities and aborts, which are not Ravenscar compliant. To solve this problem, de la Puente and Zamorano proposed a Ravenscar compliant scheme that allows a supervisory task to detect overruns and preempt the faulty task [PZ03].

The main advantage of the Gurkh Project hardware monitoring scheme is that the software application is completely independent from its monitoring device. The monitored tasks are not required to arm and disarm software timers. By being active concurrently and functionally decoupled from the monitored task, the hardware monitoring scheme is a lot simpler than its software counterparts. The system design process proposed in the Gurkh Project is thus less error-prone. As for some software solutions [PZ03], the scheduling of the system tasks is perturbed only after the overrun is detected, as soon as it occurs. The overall concept is similar to the supervised overrun detection scheme described in [PZ03].

Task overruns caused by a violation of the BCET or WCET may have system level repercussions. For instance, a misbalance in the appropriate share of processor time between tasks may develop. Additionally tasks interacting with the faulty task may miss their deadlines. One of the conclusions reached by de la Puente and Zamorano is that action to tolerate task overruns has to be taken at the system level:

"(...) there is a need to provide a system-level service for reconfiguration after an overrun has been detected" [PZ03].

Following this recommendation, the approach we adopt in Gurkh implements predictable system-level reconfiguration, in which the system is given the ability to enter fault handling modes of operation that allows it to manage the specific task overrun detected. The approach clearly targets the tolerance of unanticipated state-dependent faults that reveal their existence by violating their timing bounds or by transitioning to unexpected software application states. It is important to underline that lack of intervention by the MC is not a guarantee of correct system functionality. Other types of faults in the system might go unnoticed to the MC. These can be handled by other error detection techniques, in conjunction with the MC.

2.4.3 Concurrent Hardware Monitoring

Beyond processor spares, hardware voting techniques, and general replication as a fault tolerance measure, the idea of using redundant hardware that does not implement application functionality, but which is designed expressly for monitoring, has been around for some time. A good example of a successful full scale implementation of this concept on a mission critical real-time system is The ELEKTRA Railway Signaling System presented in [KK95]. The implementation is based on a two channel system. While the first channel provides the main interlocking control services, the second monitors the outcome of the control functions against safety rules. Actions are taken by the first channel only if the second agrees that no safety conditions are violated.

Mahmood and McCluskey present in [MM88] a survey of concurrent system-level error detection techniques using a watchdog processor: a simple coprocessor monitoring the behavior of the system. The coprocessor compares concurrently information about the system's behavior with behavior information provided before runtime. Such an error detection scheme does not depend on a particular and simplistic fault model. No particular faults are checked for, only the general presence of errors (possibly triggered by different types of faults) in the memory access behavior, the control flow, the control signals or the result reasonableness. Less hardware is required to implement this approach than for replication approaches. The watchdog is a self-contained circuit, designed independently from the monitored processor. It is also integrated or removed from the system without major changes, and does not exclude the use of other error detection techniques. These findings are directly relevant to our approach, and the Monitoring Chip can be described using the framework of watchdog processors.

In [MM88], experimental studies in a test with fault injection on external microprocessor pins find that invalid program flow error detection technique is the most efficient error detection technique. Two schemes are discussed: assigned- and derived-node signature control flow checking. The Gurkh approach is closer to the first approach. Our concept of node comes from the application / kernel integrated model verified in the Gurkh process. Nodes are basically bounded by two consecutive accesses to the kernel from software. The system architecture of our approach helps us a lot in this case, since the application running on the processor needs to communicate with the hardware kernel, the MC can simply tap in on the exchanged information. Other architectures must use elaborate and complex schemes for the watchdog processor to track the behavior of the system. Derived-signature (derived from the nodes) control flow checking is one such intricate scheme. Our MC verifies not only the execution timing of tasks, but also the correct and timely traversal of the control flow graph inside each task, as it is done for the assigned-signature control flow checking scheme. In addition to proving the viability of the concept of having hardware monitoring of the system behavior, the work presented explicitly identifies the opportunity represented by the possibility of automated watchdog program generation from a high-level language program. This is one of the elements identified in the Gurkh design process: the generation of the MC circuit based on the model of the application.

Now that the key ideas behind our framework have been presented, we explore the role of the Ravenscar profile in the Gurkh project in Chapter 3.

Chapter 3
Ravenscar and RavenHaRT

This Chapter presents the key concepts behind the concurrent tasking model of the Ravenscar profile of Ada 95. Those concepts play an important role in the Gurkh Project.

First and most visibly, the concurrency model imposes rules on the production of the application software. Second, the model dictates the architecture of the implementation of the Ravenscar supporting kernel, including the interrupt mechanism. Third, the Ravenscar Kernel and software interaction in turn constrain what the Monitoring Chip can monitor. Fourth, adherence to the principles of the Ravenscar profile also imposes constraints on the design of the fault tolerant infrastructure. For the sake of completeness and because these principles are omnipresent throughout our framework, we will present the important characteristics of these principles here even though the literature is rich in references to the Ravenscar profile definition. In particular, the interested reader could refer to [BDR98] for a formal definition of the profile and [BDV03] for a practical user guide.

3.1 The Profile

The core of the Ada language is mandatory for all language implementations. A set of annexes is defined to extend the language in order to fulfill special implementation needs. In particular, the real-time annex is mandatory for Ravenscar. Specific compiler directives, known as pragma `Restricitons`, are used to denote the forbidden features of Ada 95 that need to be specified to implement the Ravenscar profile [AL03b]. They are listed below [AL03b]:

```
No_Task_Hierarchy                          Max_Entry_Queue_Depth => 1
No_Abort_Statements                        No_Calendar
No_Task_Allocators                         No_Relative_Delay
No_Dynamic_Priorities                      No_Protected_Type_Allocators
No_Asynchronous_Control                    No_Local_Protected_Objects
Max_Task_Entries => 0                      No_Requeue
Max_Protected_Entries => 1                 No_Select_Statements
Max_Tasks => N                             No_Task_Attributes
Simple_Barrier_Variables                   No_Task_Termination
```

In addition, the pragma `Locking_Policy` restricts the policy to `Ceiling_Locking`.

The profile prevents tasks to be dynamically allocated, and allows only for a fixed number of tasks. None can terminate, so each consists of an infinite loop. Tasks have a single invocation event. That event can be called an infinite number of times. The invocation event can be time-triggered or event-triggered. Time-triggered tasks make use of a *delay until* statement. Tasks can only interact through shared data in a synchronized way, through the use of *Protected Objects* (POs). POs may contain three different types of constructs, the *Protected Function*, the *Protected Procedure* and the *Protected Entry.* A Protected Function is a read-only mechanism, while the Procedure is a read-write mechanism. The Protected Entry is associated to a boolean *Barrier* variable and both implement a mechanism used for event-triggered invocation of tasks. A task suspends itself by calling the Entry of a PO that has a closed Barrier. The suspended task is then put on the PO's *Entry Queue*, which can only have one task. The same PO can possess multiple Protected Functions or Procedures, but only one Protected Entry. To change the value of a PO's Barrier, only a Procedure belonging to that PO can be used. When a Procedure of a PO is about to exit, the Entry Queue of the PO is checked. If the Entry Queue has a suspended task waiting on the PO's Barrier to open, the value of the Barrier is checked to see if the Procedure has altered it. If the Procedure has indeed altered it, and thus the Barrier is now open, the remainder of the PO's Entry code is executed in the context of the task that called the Procedure. The task suspended on the Entry queue is then made runnable again. Since a Protected Function is read-only, it cannot change the value of its PO's Barrier [BDR98], [BDV03].

Priority-based preemptive scheduling is used in the RavenHaRT kernel, with *FIFO within priority* used to distinguish tasks with equal priorities. The *Ceiling Locking* mechanism is used to synchronize access to POs and prevent deadlocks and priority inversion problems. This mechanism works as follows: all tasks are given fixed priorities in the system. Each PO is also associated to a priority, called *Ceiling Priority*: it has the priority of the highest priority task that can access it. There is only one way a task can change priority and it is by accessing a PO. When a task accesses a PO, its priority is raised to the PO's Ceiling Priority. Ceiling Locking prevents other tasks potentially accessing the same PO of preempting the task that just saw its priority raised.

The profile defines a restricted subset of the complete Ada 95 language. As mentioned before, these restrictions enable the design of real-time systems that can meet hard real-time and mission critical certification requirements. Furthermore, they enable the profile to be mapped to a small run-time kernel that need only support the restricted behavior of the Ravenscar profile [Sil04]. The RavenHaRT hardware kernel was designed with the purpose of exploiting this possibility.

3.2 The Hardware Kernel

RavenHaRT was developed by Lundqvist and Asplund in [AL03b] and implemented in hardware by Silbovitz in [Sil04] using VHDL. The kernel design and implementation characteristics influence parts of the framework: it puts constraints on the design of the interrupt mechanism and it specifies how the Monitoring Chip can track task timing and task execution flow.

The RavenHaRT kernel is designed for application software written in Ravenscar compliant Ada 95 and compiled with a special purpose cross compiler to run on a single processor system. The compiler used in the Gurkh Project is called pGNAT, and is presented by Seeumpornroj in [See04]. Based on the open-source GNAT compiler, pGNAT uses a modified version of the GNARL run-time library and the GNAT's GCC back end to generate RavenHaRT compatible and Ravenscar compliant PowerPC object code.

Figure 3 reproduced from [Sil04] shows the architecture of the kernel and the application.

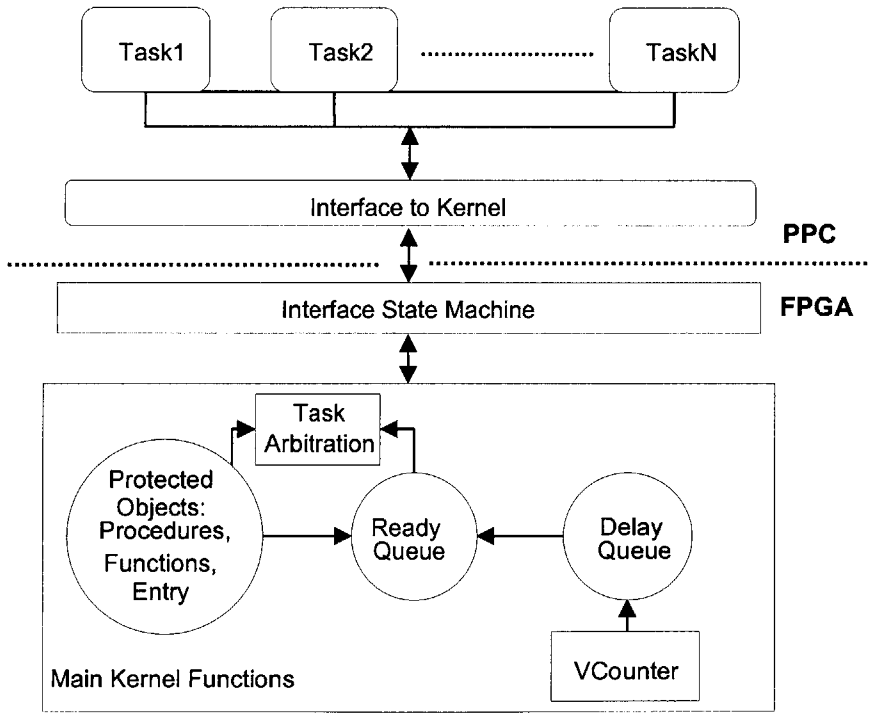


Figure 3: Architecture of Kernel and Application

The purpose of the kernel is to determine what task should be running on the processor at any time. To do so, RavenHaRT keeps track of task priorities and software service calls. These calls can be classified as either elaboration-time calls or run-time calls. The elaboration-time services the kernel provides are the creation of tasks (Create), the setting of the frequency of the kernel clock used to keep track of the task delay mechanism (SetFreq), the command to start the scheduling (FindTask), and provision of the system time to the software (GetTime). The run-time services the kernel provides includes the delay until command (DelayUntil) that suspends the calling task and all the PO servicing commands that enforce the Ceiling Locking mechanism (change of task priorities) and the event-triggering mechanism (Entry queue and Barrier checks).

To make sure the highest priority active task is always the one running, RavenHaRT manages a Ready Queue (RQ) holding each task's ID and current status (active or inactive) and current priority. Another construct, the Delay Queue (DQ), constantly finds the next task to be awakened. When the time to awake the task comes, the Ready Queue makes the task runnable. The hardware kernel also has PO managing circuits. Each PO declared in the code has its own PO circuit (po_one), and each PO circuit is composed of a Protected Entry Circuit (entry) and a Procedure / Function circuit (proc_func). The PO circuits can alter the task priorities of the RQ, suspend a task blocked on an Entry depending on the PO's Barrier value, and release the task blocked on the Entry. It is important to state that once the Procedure called by a task opens the Barrier of the PO, the Entry called by the suspended task finishes executing in the context of the task calling the Procedure, and only after does the Procedure exits the PO. At this point in time, both "suspended, Entry-calling" task and "releasing, Procedure-calling" task are active. Both are now outside the PO and have their original priorities restored. The Ready Queue will schedule the highest priority task to run first. Following the FIFO within priority rule, the "releasing Procedure-calling" task will run if both have the same priority.

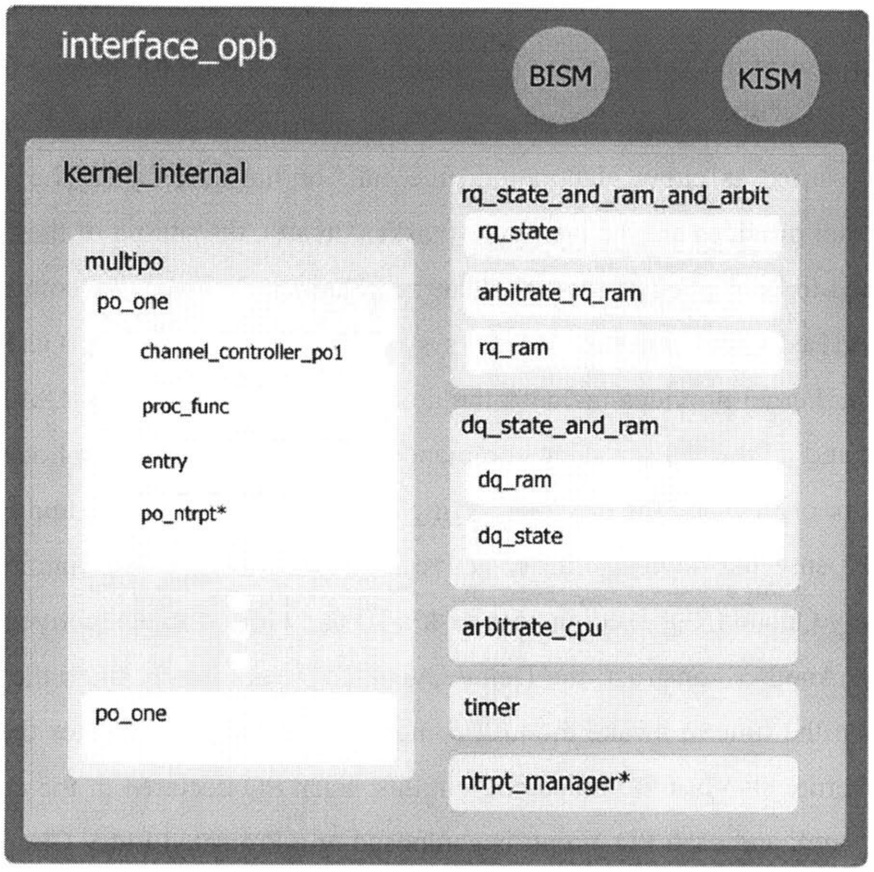


Figure 4: RavenHaRT Hardware Architecture Diagram. New components are marked with a (*).

Figure 4 shows the hardware architecture of RavenHaRT. Each circuit is identified and the hierarchy is represented by placement of the circuit areas and the different tones of gray. The external circuit is called interface_opb because it is the circuit as seen from the bus interface. OPB stands for On-Chip Peripheral Bus. The interface_opb contains three circuits, the Bus Interface State Machine (BISM), the Kernel Interface State Machine (KISM) and kernel_internal, which contains the rest of RavenHaRT's logic. The RQ entity is composed of rq_ram, arbitrate_rq_ram, and rq_state circuits, while the DQ entity is composed of dq_ram and dq_state. The PO logic is inside the multipo circuit, which is composed of one po_one circuit for each PO in the system. As mentioned above, each po_one circuit contains proc_func and entry, and also the channel_controller_po1 circuit and the po_ntrpt circuit. Inside the proc_func and entry state machines, two paths can be taken. They correspond to the nominal and exception paths. The kernel has therefore the ability to react differently to exception code from the application.

At the same level as multipo, arbitrate_cpu resolves potential conflicts caused by sudden priority changes of tasks involved with POs. The timer circuit provides the timekeeping and kernel clock frequency setting services, while the ntrpt_manager arbitrates potential simultaneous system interrupt signals.

The implementation described in [Sil04] does not include system interrupts servicing functionality. A mechanism implementing such functionality is presented in Chapter [redacted]. The components marked with an asterisk in Figure 3 were designed, implemented and integrated in the original RavenHaRT by the author to enable system interrupt servicing.

Chapter 4
System Interrupts

An embedded system provides the computing abilities necessary to control some physical process. Embedded systems therefore need to communicate with hardware, reading input via sensors and providing outputs via actuator commands. In our fault tolerant framework, error detection relies on the Monitoring Chip which must interrupt the normal operation of the system to handle timing or control flow errors.

The term system interrupt is employed to distinguish the interrupt corresponding to the necessity for hardware external to the RavenHaRT kernel to intervene in the execution of the application, from the lower-level processor interrupt. The processor interrupt is a low-level technical mechanism already used in the processor-RavenHaRT system. Using the precise vocabulary of the PowerPC405 User Manual [PPC], RavenHaRT preempts a task by signaling a *noncritical*, *asynchronous* exception to the PowerPC, triggering a *precise*, interrupt referred to as *external*. The term external is chosen by opposition to an internal interrupt such as the *Machine Check* or *Data Storage* interrupts signaled by the processor itself.

The system interrupt we are concerned with in this Chapter is thus a higher-level type of interrupt, external to the PPC-RavenHaRT system. Figure 5 below depicts this idea. The dashed arrow represents the conceptual system-level interrupt which is implemented in practice by the physical signals represented by the regular arrows.

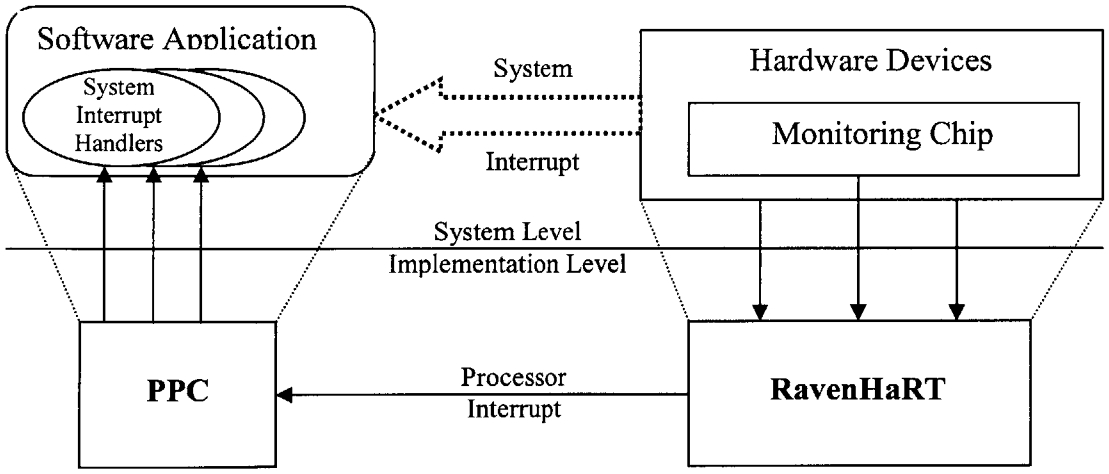


Figure 5: System Interrupt vs. Processor Interrupt

The design solution found to implement system-level interrupts in Gurkh is called the Hardware Interrupt PO-Entry Release Mechanism. The design and implementation issues are discussed in [Gor04].

The relevant details of the mechanism are provided below.

4.1 Design

The design of the Hardware Interrupt PO-Entry Release Mechanism proved to be an interesting exercise in hardware/software co-design. Our solution emulates the normal software Entry release mechanism used by RavenHaRT to implement event-triggering of tasks, as per the Ravenscar profile. The Hardware Entry Release Mechanism is illustratively compared to the "normal" software-activated entry mechanism in Figure 6. Our solution necessitates the creation of very high priority Ada Ravenscar tasks that act as interrupt handlers. According to the Ravenscar profile, tasks cannot be created nor destroyed. They can however be event-triggered via a Protected Object's Entry construct. In our hardware interrupt design, an interrupt handler PO and its Entry are declared in software, and the Entry code is empty. We associate one such unique PO for each hardware interrupt handling Ravenscar task. As the very high priority interrupt handling tasks starts running after system initialization, the tasks at once call their PO's Entry. The POs have by design a Barrier that is always closed, and thus the tasks subsequently suspends. Only a hardware signal can trigger the opening of the Barrier of the Entries in the hardware kernel, leading to the scheduling of the interrupt handling task. In RavenHaRT, the status and current priority of each task is held in the RQ RAM. Once the Entry state machines are activated by their Barrier becoming opened, they make the appropriate request to the RQ RAM to activate the interrupt handling task on the Entry Queue, exactly like "normal" software-activated Entries do.

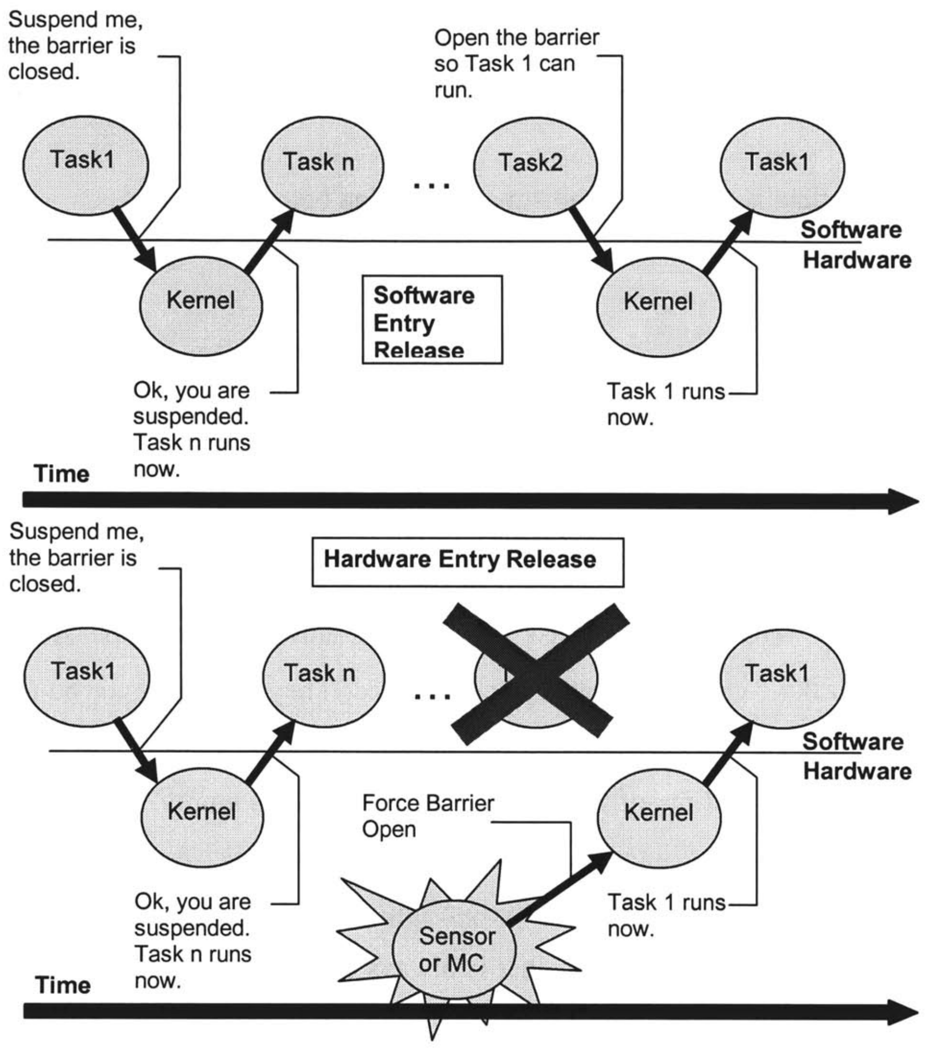


Figure 6: Comparing Software and Hardware Entry Release Mechanisms

Another approach to the problem of enabling hardware interrupts could have been to access the RQ RAM directly upon receiving an interrupt, without resorting to Entries and POs. This other approach was found to be less convenient. By using a PO's Entry we have a visible construct in software which already exists, and that carries out the task's suspension and activation process. It makes the work of the application developer simpler to consider the classical Interrupt Service Routine as just another task with its own PO, with the small difference that this PO has only one, empty, hardware-triggered Entry. Furthermore, the RQ RAM access problem is delicate, and it is preferred to use the entry state-machine, with its already well developed and tested interface, to interact with the RQ RAM.

4.2 Implementation

4.2.1 VHDL Implementation Files Added: po_ntrpt.vhd

The mechanism is easily configurable for synthesis of systems with different numbers of distinct system interrupts. Our solution takes the form of a hardware state-machine present in every PO circuit, even the ones not associated to interrupt handlers. This state machine, the "po_ntrpt.vhd", is inactive in those POs. When activated by an interrupt signal, it is able to emulate the appropriate signals that would be issued by software to trigger the entry release of the PO it is in. This implementation solution is somewhat inefficient in terms of FPGA gate space, but is conceptually easy to understand. The po_ntrpt state machine is shown in Figure 7.

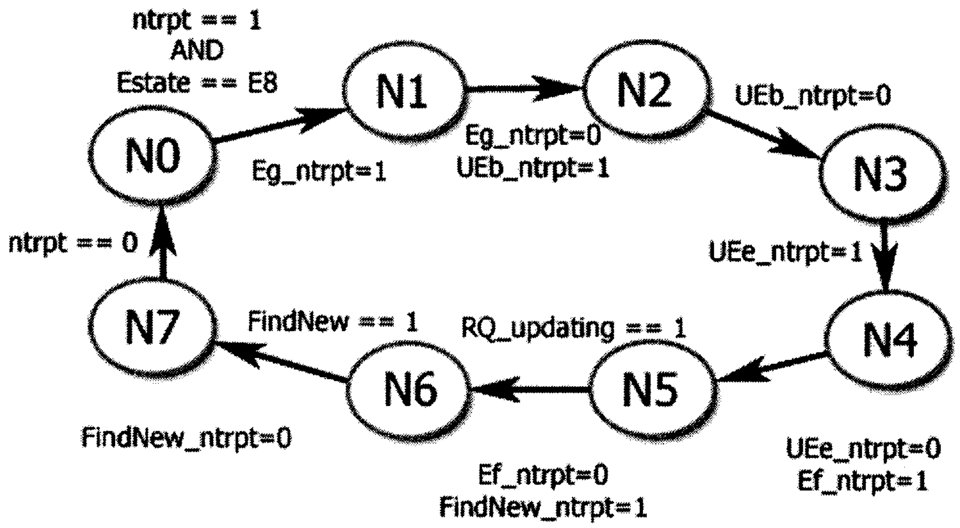


Figure 7: The po_ntrpt State Machine

To understand the implementation, it is worth noting the following points concerning the normal software Entry release mechanism that involves the PO entry, PO proc_func and channel_controller_po1 circuits. A Ravenscar task calls a PO's Entry code in software. This triggers the activation of the corresponding PO entry circuit in hardware. The kernel requests from the code the value of the Barrier (RavenHaRT does not store this value) and one of two paths are taken in the entry state machine. Either the Barrier is opened and the entry-continues-execution path in the state machine is taken, or the Barrier is closed and the entry-suspends-the-calling-task path in the state machine is taken. The latter path involves the entry circuit signaling a task in the Entry Queue hardware (a simple hardware signal) and the RQ circuits being instructed to suspend the calling task. Later, a Ravenscar task calls the same PO's Procedure code in software with the purpose of opening the Barrier. The proc_func circuit is then activated. One of its purpose is to check the PO's Barrier value if the Entry Queue signal indicates a task waiting. If the Barrier just opened, the proc_func circuit signals the entry circuit to proceed. Just before coming back to its initial state, the entry state machine commands a very important signal to the RQ RAM: the valid bit of the task status field in the RAM is set to one, thus activating the task that called the Entry. In this case, the priority of the task is also set back to the value it was before calling the PO's Entry. Finally, the last state transition of the entry circuit signals back the proc_func circuit so itself can come back to its initial state. Doing so, the proc_func state machine resets the priority of the Procedure calling task to the value it was before calling the PO's Procedure and indirectly (through channel_controller_po1) sends the crucial FindNew signal. The FindNew signal is the signal making RavenHaRT check if the currently running task is the highest priority active task in the RQ RAM. If it is not, then RavenHaRT initiates a task switch, pre-empting the currently running task. The FindNew signal is necessary each time tasks become active from DQ wake-up or from PO Entry release, and each time the PO related Priority Ceiling mechanisms change the priorities of active tasks.

Design of the po_ntrpt state machine is essentially an interfacing problem. The appropriate signals are emulated so that the entry circuit is tricked into "thinking" that it is a software Procedure that is opening the Barrier.

At the hardware hierarchical level of the PO, there are five major differences worth noting explicitly between the normal software and the proposed hardware Entry release scheme:

1. In the hardware release scheme, there is no software Procedure, and consequently no hardware activity of the proc_func state machine. This state machine is completely bypassed. Signals sent by it are emulated, and we make sure that signals that could activate it do not accidentally do so.
2. In the hardware release scheme, the entry state machine of the PO is exploited. The software Entry code however, being empty, will not lead to an exception. Therefore in our scheme, we assume that the entry state machine path taken is always the nominal path.

3. The PO which is associated to the interrupt handling task contains only one software entity, the Entry. The ceiling priority of the PO will thus be the same as the priority of the task. Therefore the priority of the interrupt handling task never changes and the last transition of the entry state machine will be useful only because the valid bit of the suspended task will be set in the RQ RAM.
4. FindNew is sent by the channel_controller_po1 circuit, triggered by the last transition of the procedure state machine. In the hardware release scheme, we make channel_controller_po1 trigger FindNew by using a signal called FindNew_ntrpt coming from the po_ntrpt state machine.
5. Similarly, the signal making the last transition of the entry state machine is sent by channel_controller_po1. As above, the signal will be triggered by using a signal called Ef_ntrpt from the po_ntrpt state machine.

The most important input of the po_ntrpt state machine is the hardware interrupt input. This is where the RavenHaRT circuit links a hardware signal interrupt to a particular PO-Ravenscar Handler Task combination. Aside from the obvious reset, clock and interrupt inputs, the po_ntrpt state machine will also input other signals to guarantee synchronicity with the rest of the system. These are control signals and signals indicating the state of the various circuits the po_ntrpt is interacting with.

4.2.2 VHDL Implementation Files Added: ntrpt_manager.vhd

To accommodate several different interrupting sources, the Hardware Interrupt PO-Entry Release Mechanism handles their simultaneous activation using the new ntrpt_manager circuit located inside kernel_internal. Thus the need exists for a circuit able to regulate interrupt requests. The "ntrpt_manager.vhd" state machine does precisely this. It was build for three interrupt lines, but can be extended to any number of interrupts. The ntrpt_manager allows the system to order the processing of the different possible interrupts. The system designer must thus prioritize the multiple system interrupts. The state machine gives a token to the highest priority interrupt signal which is activated first. Once an interrupt has acquired the token, its signal propagates to the output and no other interrupt can get the token. When the corresponding interrupt is reset, the token is released and the highest priority interrupt signal available at that moment acquires it. The ntrpt_manager state machine diagram is represented in Figure 8.

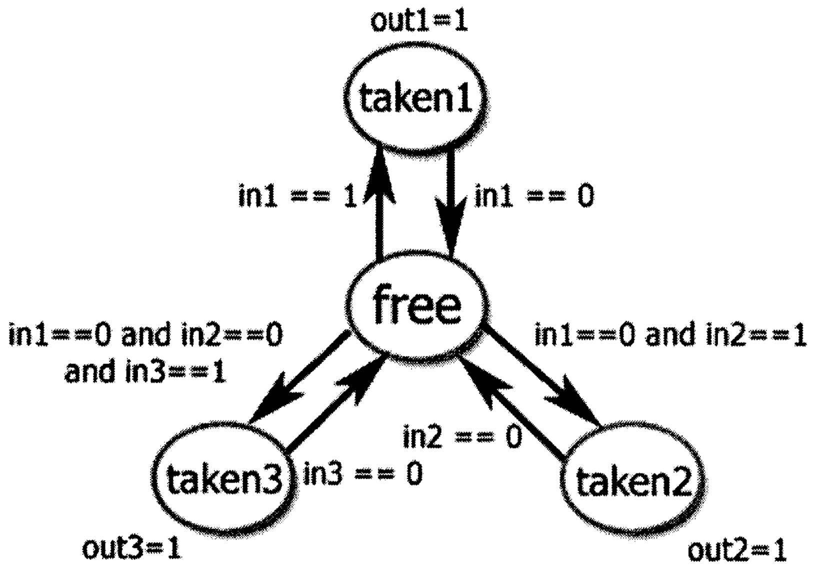


Figure 8: The ntrpt_manager State Machine

4.3 VHDL Implementation Files Modified

The following VHDL files were modified to accommodate the integration of po_ntrpt in the system: channel_controller_po1, po_one, multipo, RQ_state_and_ram_and_arbit, arbitrate_rq_ram, kernel_internal, arbitrate_cpu and interface_opb. All the changes are straightforward, except for two, in arbitrate_rq_ram and arbitrate_cpu.

4.3.1 Arbitrating RQ RAM accesses

For each hardware PO circuits, both entry and proc_func components have the possibility to write to the RQ RAM. The communication between the two PO components and the RQ RAM is realized at the hierarchical level of kernel_internal, by the interconnections between the multipo component and the RQ_state_and_ram_and_arbit component. By design, the original kernel of [Sil04] used the POId signal to select and validate the appropriate communication lines between multipo and RQ_state_and_ram_and_arbit. POId identifies uniquely the PO currently operating. These communication lines consist of the RQ RAM access lines and the FindNew signal lines. For each PO, there is one set of the following RQ RAM control and data signals:

- ram_en, to enable a RAM access, is a 1 bit signal.

- ram_we, to specify a read or write, is a 1 bit signal. Note that POs only write to the RAM, they never read.
- ram_addr, to specify which task should be read or written. The Ready Queue state machine is the one reading the RAM.
- ram_assign, the actual data to be written, includes the priority and status bit of the task.

In addition, FindNew is a 1 bit command for the RQ State machine. Each PO can generate this signal.

We call *Interrupt PO* the PO assigned to the interrupt servicing task. Under circumstances when hardware interrupt may occur, the interrupt PO must be able to carry out RQ RAM access and signal FindNew information to the RQ RAM and RQ state machine. In this case however, the POId signal is not available to select and validate the appropriate lines. Logic must therefore be provided at the level of kernel_internal to ensure that information from the interrupt PO is correctly processed.

The problem of RQ RAM access is probably the most challenging issue of the design of the Hardware Entry Release mechanism. The components that can access the RQ RAM are the RQ_state state machine (read/write) and every PO's proc_func or entry state machines (writes only). In the version of RavenHaRT which does not support hardware interrupts, only the PO validated by POId can be active at any time. The only possible collisions that can happen in accessing RQ_RAM in this case are therefore between PO writes and RQ_state reads or writes. The purpose of arbitrate_rq_ram is precisely to resolve the possible conflict. In the version of RavenHaRT which supports hardware interrupts there is a need to protect the RQ_RAM from a possible three way conflict between the previous RQ_state and PO components, and the new interrupting PO. At the heart of the problem is that we need to maintain synchronicity between the software possibly requesting PO commands to be executed and the hardware that needs to execute these while simultaneously processing the hardware interrupt. Two different approaches were implemented and tested to achieve this three-way arbitration of the RQ RAM. The first approach keeps the RQ_state_and_ram_and_arbit entity and its component unaltered, and uses an extra circuit. The second approach completely modifies the "arbitrate_rq_ram" circuit and requires a small change in the rq_state_and_ram_and_arbit circuit. The first approach was abandoned because of its complexity. The complexity arises from the fact that the two arbitration circuits are cascading the information and that they are on two different levels of hierarchy. This makes the arbitration problem spread in time and space. We therefore favored the approach that concentrated the arbitration into one component, the completely redesigned arbitrate_rq_ram. That circuit basically sequences and carries out all the possible combinations of requests: coming from the RQ State Machine (RQ), from software-triggered PO (PO_soft) and from interrupt PO (PO_hard). What complicates the problem significantly is that requests can follow one another in time. If arbitrate_rq_ram uses an arbiter that saves a request temporarily to output it a clock cycle later, the corresponding new state in the state machine must be able to listen and store another potential incoming request, and so on. Fortunately, the number of possible combinations is restricted to only a few by the logic of the state machines initiating those requests.

4.3.2 Arbitrating the Processor

The circuit of arbitrate_cpu was slightly modified to "filter out" any rapid changes, or transients, in the identity of the new task that is about to preempt the currently running one. Changes appear at the output of arbitrate_cpu only after the inputs have stabilized for a time of ARBITRATE_DELAY clock cycles. That ARBITRATE_DELAY consists of 18 clock cycles. This fix decreases the response performance of RavenHaRT but is necessary, considering the new possibility of having an Interrupt Task becoming active anytime. The justification and the analysis leading to the choice of ARBITRATE_DELAY are documented in [Gor04].

4.4 Development and Testing

The redesign and testing of arbitrate_rq_ram is documented in detail in [Gor04].

The results of the testing reported in [Gor04] identify a bug in the implementation of the Hardware Entry Release mechanism. The document mentions different possible reasons for the presence of the bug, without positively identifying any. Since the document was written, further testing and simulation revealed that the actual testing procedure was flawed and at the source of the problem. Searching for that cause took time but allowed us to gain confidence in our implementation.

Referring to table 2 of [Gor04] called "Features of the Different Development Versions of RavenHaRT" (which stops at version v2.2.5), we give below the exact configuration of versions v2.3 and above, according to the format of this table.

version	po_ntrpt	arbitrate_cpu LP filter	arbitrate_ntrpt	centralized arbitrate_rq_ram	ntrpt_manager	software inhibiting interface_plb	delayed po_ntrpt
v2.3 and above	Yes	Yes	No	Yes	Yes	No	No

Table 1: Configuration of the Development Versions 2.3 and above of RavenHaRT

The features that were not kept in versions v2.3 and above are not discussed here.

4.5 System Developer's User Guide

With the current implementation, system developers must change the hardware: they must manually ensure the correct synthesis of the particular system interrupt requirements of their application in RavenHaRT. This involves mainly updating the ntrpt_manager circuit to handle the appropriate number of interrupts (the current implementation can handle three), prioritize them, account for the appropriate number of PO in the system (the sum of application POs and interrupt POs) and finally assign each interrupt to a particular PO and PO circuit. This last step can be confusing if the developer forgets that the PO(0) is reserved as a null PO [Sil04]. The system interrupt solution presented here also makes assumptions on the Ravenscar programming style and pGNAT compiler.

Figure 9 illustrates the software implementation of a task, GPS_receiver, which waits for incoming GPS messages from hardware. GPS_receiver hangs on a PO called GPS_RCV_EVENT. The Barrier of that PO is opened when Procedure Signal is called from the Interrupt_Handler task. The Interrupt_Handler task is activated by the Hardware Interrupt PO-Entry Release Mechanism.

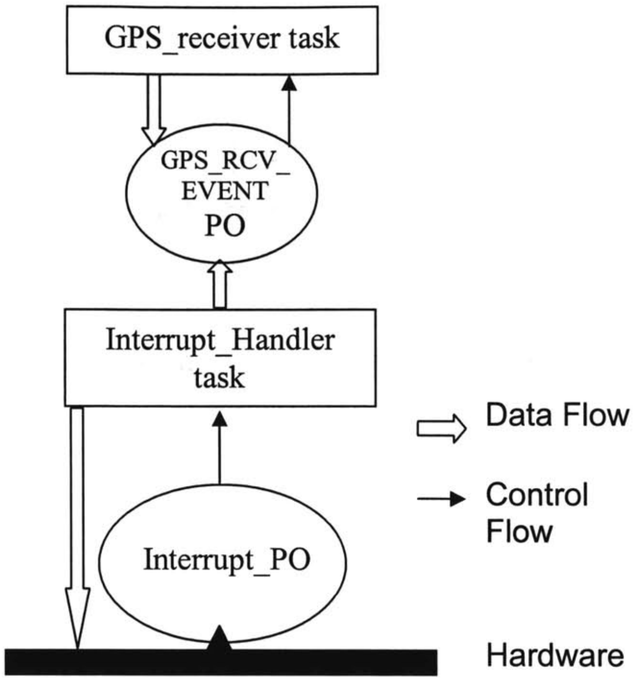


Figure 9: Receiving Sensor Data from Hardware

The code below shows the corresponding Ravenscar software:

- For the GPS_RCV_EVENT PO:

```
-- Event PO declaration ----------------------
protected type Event_Object(Ceiling : System.Priority) is
      entry Wait (D : out gps_message_datatype);
procedure Signal (D : in gps_message_datatype);
   private
      pragma Priority(Ceiling);
      Current : gps_message_datatype;
      Signalled : Boolean := False;
   end Event_Object;

   protected body Event_Object is
      entry Wait(D: out gps_message_datatype) when Signalled is
      begin
         D := Current;
         Signalled := False;
      end Wait;
      procedure Signal(D: in gps_message_datatype) is
      begin
         Current := D;
         Signalled := True;
      end Signal;
end Event_Object;
-- Event PO instantiation ---------------------
GPS_RCV_EVENT : Event_Object(4);
```

- For the Interrupt_PO:

```
protected Interrupt_PO is
```

```
      entry WaitforInterrupt; -- parameterless
      -- no software procedure to open the Barrier
   private
      pragma Attach_Handler(WaitforInterrupt,<interrupt_PO_id>);
      pragma Interrupt_Priority(<interrupt_PO_priority>);
      Activated : Boolean := False;
   end Interrupt_PO;
   protected body Interrupt_PO is
      entry WaitforInterrupt when Activated is
      begin
         -- empty
      end WaitforInterrupt;
end Interrupt_PO;
```

- For the GPS_receiver Task:

```
-- Event-Triggered GPS Task -------------------
task GPS_receiver is
      pragma Priority(3);
   end GPS_receiver;

   task body GPS_receiver is
   begin
      loop
         GPS_RCV_EVENT.Wait(gps_data);
         -- GPS message processing goes here
      end loop;
end GPS_receiver;
```

- For the Interrupt_Handler task:

```
task Interrupt_Handler is
      pragma Priority(<interrupt_PO_priority>);
   end Interrupt_Handler;

   task body Interrupt_Handler is
      hw_gps: gps_message_datatype;
     hw_data: hw_datatype;
   begin
      loop
         Interrupt_PO.WaitforInterrupt;
         -- the rest of the code is the interrupt handling
          hw_data := <read_register>;
          hw_gps := Gps_Message_Datatype(hw_data);
          GPS_RCV_EVENT.Signal(hw_gps);
          -- ack the interrupting device
          <device_register> := <acknowledge_code>;
      end loop;
end Interrupt_Handler;
```

The pGNAT compiler should react to the Attach_Handler pragma inside the Interrupt_PO declaration by connecting the PO ID <interrupt_PO_id> to that particular Interrupt_PO and its entry. The Interrupt handling task simply calls the Entry and gets suspended until hardware interrupts. It then reads the data from the device, calls the GPS_RCV_EVENT PO Procedure, delivers the data, acknowledges the interrupting device, and loops to suspend again.

4.6 Future Improvements

Although appropriate for the development stages of the Gurkh Project, the system interrupt solution presented here has a few limitations. The concept of scheduling the interrupt handler is easy to understand, but the Interrupt PO-Entry Release Mechanism is complex because it involves many entities at different levels of the hardware hierarchy. This complexity contributes to make the kernel harder to model and to verify. Furthermore, the design solution is not very flexible for system developers needing to add system interrupts. Although straightforward, the design of the ntrpt_manager entity must be modified to accommodate different numbers of interrupts.

A future redesign of RavenHaRT should include a revision of the system interrupt mechanism solution. In particular, the use of a PO and its Entry can be completely abandoned. The design of the system interrupt solution should focus on the central element of interest, the Ready Queue. A cautious design should give the capability of altering the RQ RAM directly to two types of external elements. The interrupts, the first type, should alter the status bit of their handler task in the RQ RAM directly, and subsequently call FindNew to activate their task. The software interrupt handling tasks, the second type, should be given kernel support to alter their status bit in the RQ RAM directly, in order to suspend themselves.

Such an implementation would make the hardware design simpler and easier to verify, but would require coordination with the pGNAT compiler design.

Chapter 5
The Monitoring Chip

This Chapter describes the architecture of the Monitoring Chip (MC). The Monitoring Chip forms the central element of our fault tolerance framework by providing non-intrusive error detection. Using a model of the application software, it monitors the execution of the application system and listens to the calls the software system makes to the hardware kernel. Its functionality allows it to time the execution times of the different task and alert the rest of the system when the timing bounds are violated. The Monitoring Chip can also alert the rest of the system when the sequence of interactions between software application and hardware kernel is different from the expected sequence of interactions. The MC is in hardware and therefore offers its services in a non-intrusive manner to the system. As mentioned in Chapter 2, the hardware monitoring scheme presents the advantages of being active concurrently and being functionally decoupled from the monitored task.

The paragraphs that follow describe in detail the technical aspects of the Monitoring Chip. This information is especially relevant to anyone interested in getting familiar with the VHDL code.

5.1 Monitoring Chip As A Component Of RavenHaRT

In the description below, capitalized names refer to constants defined in the file myvariables.vhd

5.1.1 High Level Hierarchy

The Monitoring Chip hardware is integrated inside the RavenHaRT top-level circuit to take advantage of the functionality of the Bus Interface State Machine (BISM) and Kernel Interface State Machine (KISM) processor interface circuits of the kernel. This implementation is more efficient in terms of gate-space needed as processor interfacing functionality does not need to be duplicated.

The Monitoring Chip ipcore itself consists of the following files:

- sc_top_parallel.vhd, the MC's top level hardware description
- sc_x.vhd, the actual task timer, each task being monitored in the system must have its own sc_x.vhd file

- serialcom.vhd, for serial transmission of the alert and the diagnosis message to an external entity

Hierarchically, the sc_x components are contained in sc_top, which also contains the serialcom component. In turn, sc_top is a component inside the interface_opb.vhd. The signals driving the Monitoring Chip are produced by the Kernel Interface State Machine, Bus Interface State Machines or some other logic of the interface_opb component. Figure 10 illustrates this hierarchy.

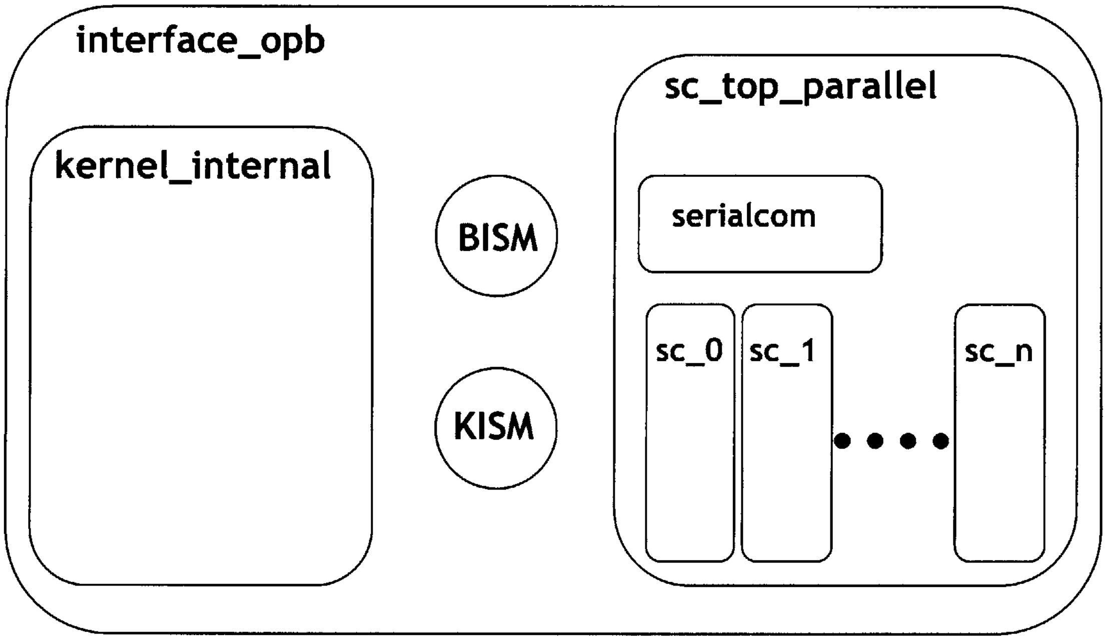


Figure 10: MC as an element of RavenHaRT: Architecture Diagram

The input and output signals are described and their origin and destination detailed in the next paragraphs.

5.1.2. Inputs To The MC

In addition to the clock and reset_n signals, sc_top requires the following inputs:

sc_cs	is the signal used to tell the monitoring chip that a service call is being issued to the kernel and that the instr and arg signals are valid and should be read. The signal is active when positive.
instr	is the instruction that the software system is sending to the kernel. It contains CMDREG bits.

arg is the argument of the instruction. This is used for kernel commands that concern a particular PO. It therefore contains BITPO bits.

tsw_irq is to inform the monitoring chip that the kernel has interrupted the software system in order to perform a task switch.

next_task keeps the MC informed on what is the next task the software system is switching to. It has BITTASKS bits.

errclr is a signal used to acknowledge the receipt of the alerting signal from the MC.

sTime of size MAXTIME bits is the system time used by the kernel.

5.1.3 Outputs Of The MC

The outputs of the MC consist of the alert signal and the diagnostic information, in parallel or serial form.

sc_irq is the alert signal which is used as a direct interrupt to the kernel.

Tx_sc is the serial line that outputs the task that raised the error, the diagnosis of the error and the timestamp when the error was detected.

The same information (not including the timestamp) is available in parallel.

errtask is the signal for the task identity with BITTASKS bits.

errcode is the signal for the diagnosis of the error with 4 bits.

5.1.4 Input Drivers

5.1.4.1 Signals driven by KISM

sc_cs, instr and arg are driven by the KISM state machine of the interface_opb component. All are initialized to zero when KISM resets. For the following commands received by the kernel, when KISM is in state i4, sc_cs is set to one:

```
DELAYUNTILi
FPSi
UPEi
UFEi
FPEi
UFPXi
UPXEi
UFXEi
FPXi
ESi
UEBi
UEEi
```

```
UEXi
EXi
RST_E
RST_FP
```

This list corresponds to the commands the MC reacts to; these are all the commands except CREATE, GETTIME, SETFREQ and FINDTASK. When sc_cs is activated, instr is set to the corresponding command. The KISM is simply relaying the command information received by the kernel to the MC. For all instructions on that list except DELAYUNTILi, KISM also relays the information concerning which PO the command is addressed to, by setting the arg signal.

States i6 and i7 of KISM reset sc_cs to zero. In state i9, corresponding to the software system providing the barrier value to the kernel, sc_cs is set to 1, instr to "barr" and arg to the correct POId. As the KISM goes through i6 again, sc_cs is reset to 0.

5.1.4.2 Signals driven by independent logic

tsw_irq and next_task are both altered whenever the state of KISM is i1 and the interrupt_ack is set to one. In this case, tsw_irq is set to one and next_task is set to NTcpu. Otherwise tsw_irq is zero and next_task bits are on high impedance ('Z').

sTime is an output of the kernel component kernel_internal.

5.1.4.3 Signals driven by BISM

errclr is driven by the BISM state machine. It is initialized to zero when BISM resets, and set to one in BISM's state bus_state whenever the software system reads from the addr_reg9 register containing the error information: errtask and errcode. errclr is zero again when BISM goes to state ack_state.

5.1.5 Output Destination

The current implementation sees sc_irq connected to the signal ntrpt1, the highest priority interrupt input of the kernel. This allows the MC to alert the kernel, which will then schedule the MC Interrupt Handling Task (MCIHT) to run on the processor.

Tx_sc is routed to the outside of the interface_opb component and then outside opb_adartk_topk to the external interface of the system.

errtask and errcode are destined to be read by the software system. Whenever software reads the addr_reg9, the hardware concatenates errtask and errcode so that the error code and faulty task identity are accessible by the software.

5.2 Monitoring Chip Architecture

This section describes the logic of the sc_top and sc_x components of the MC. The functionality of each process or state machine is detailed.

5.2.1 The sc_top Component

The sc_top component, declared in the sc_top_parallel file, contains six logic groupings: the timer process, the callproc process, the tswproc process, the serialcom component, the errproc process and the collection of sc_x components for the task being monitored.

<u>5.2.1.1 timer</u>

The timer generates clock ticks at a frequency equal to (clk frequency)/(system_frequency). The resulting output is labeled tick.

<u>5.2.1.2 callproc</u>

This process has the responsibility to present valid call_sc signals to the sc_x components. It selects the particular sc_x component pointed by the cpureg_array variable only if sc_cs is one and tsw_irq is zero. This is akin to a "smart" multiplexer designed to filter out sc_cs signals that occur at the same time a task switch is operating. Exactly as it is the case at the level of kernel communication, the software could send a command as the KISM transitions to state i1, interrupt. Here also, if a task switch is taking place, the call_sc signaling will not occur. The call_sc signal is like tsw_irq and sc_cs in that it stays high only for one clock cycle..

<u>5.2.1.3 tswproc</u>

This process uses the information from tsw_irq and next_task to determine which task is running. This information comes in two forms: the cpureg_array signal, used to select the appropriate task for the call_sc signal, and the running signal, which similarly signals which sc_x components should have its counter running.

<u>5.2.1.4 serialcom</u>

This component has two inputs: an 8 bit port labeled TxData, and a LoadA signal to validate the bits present at TxData. The TxBusy output signals that the Tx, one-bit-output port is sequencing the currently latched TxData bits out through the port. The rate is controlled by BRDIVISOR.

5.2.1.5 errproc

This component is responsible for retrieving and centralizing the task error information. It searches the tasks one after the other until an error is found. The diagnostic information is gathered and the process drives the errtask, errcode and sc_irq signals that are outputs of the MC. The process also drives the serialcom component and loads the diagnostic information byte after byte. Figure 11 shows the errproc state machine.

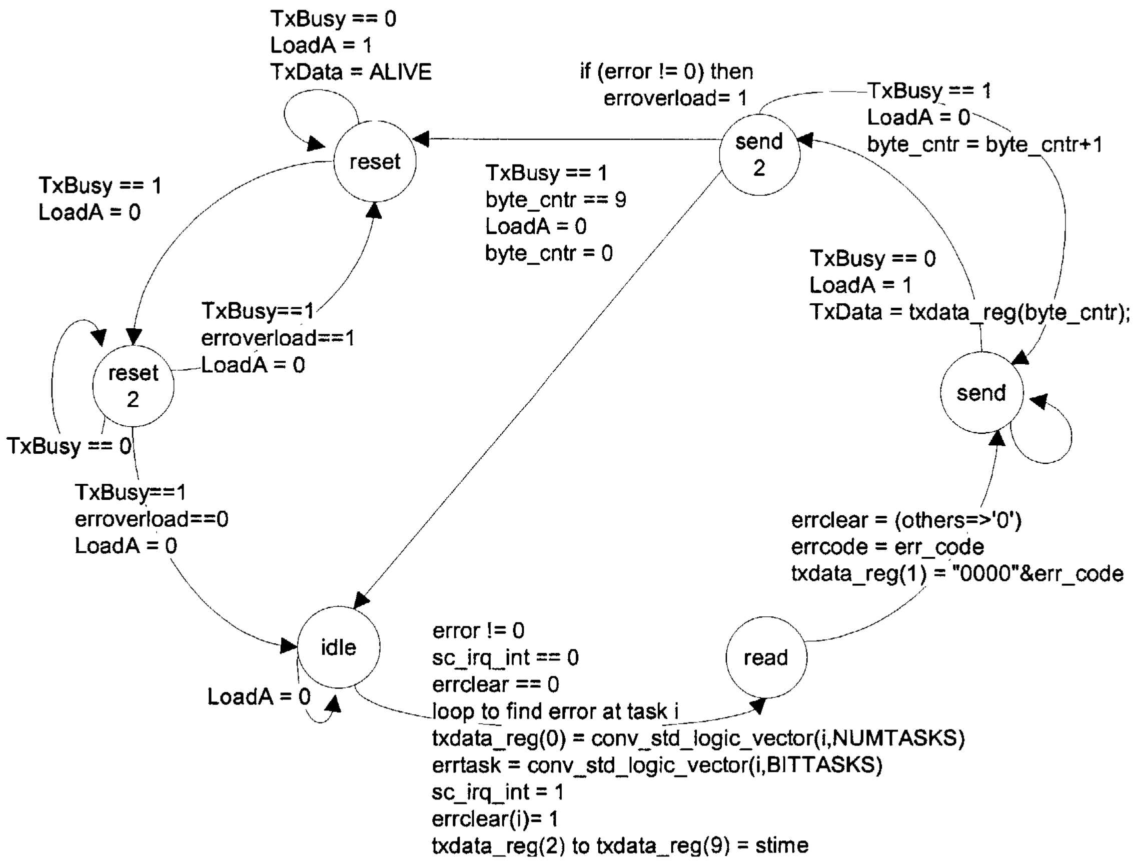


Figure 11: The errproc State Machine

The state machine initializes in the reset state and goes to idle after transmitting an "ALIVE" signal to the serialcom component. If an error is detected while in the idle state, all sc_x components will be polled by increasing index number order and the first to have an active error bit will have its ID registered by the errtask signal and the first txdata_reg signal. The third to the tenth txdata_reg signal contain the timeStamp of the present instant. The state machine then moves to read state, propagates the error to the output of sc_top and acknowledges the error of sc_x. Read state sets the second txdata_reg signal to the errcode and resets errclear, proceeding to send state. Send and send2 states loop and transmit all txdata_reg to the serialcom component. Once the transmission is done, there are two design possibilities: either the state machine resets, or it transitions to idle state. The functionality is present to detect more than one error as the first error is being processed through the read and send states using the erroverload signal. The current implementation resets and will process the second error once it receives the errclr input to sc_top from the BISM state machine. However, the timestamp that will be transmitted once the second error is processed will correspond to the time the second error starts to be processed, not the time it actually occurred. If this is problematic, several design ideas can solve the problem at the cost of extra logic space. A buffer could be added to each task to latch the timestamp when the error occurs, or a single FIFO buffer could store more than one error information set at once.

5.2.2 The sc_x Component

5.2.2.1 interface signals

The sc_x components have six inputs:

tick	The tick signal comes from the timer process of sc_top.
instr	The signal comes from the KISM state machine of the interface_opb component.
arg	Like instr, arg comes from the KISM.
running	The running signal comes from the tswproc process of sc_top.
call_sc	The call_sc signal comes from the callproc process of sc_top.
errclear	The errclear input is an ouput of the errproc process of sc_top.

sc_x outputs consist simply of:

error	The error signal is fed into the errproc process of sc_top.
error_code	The error_code signal is fed into the errproc process mentioned before.

The sc_x components contain five logic groupings: the errclr process, the wcet_proc process, the romifx component, the verification logic and the sc process. These are explored below.

5.2.2.2 errclr process

The purpose of the errclr process is to delay the transmission of the errclear signal by one clock cycle. The output is called clrerr.

5.2.2.3 wcet_proc process

This process checks that the WCET time of the current running task at the current Task Graph location is not broken. The process holds the value of WCET in wcetreg and counts down. If the task breaks its WCET, the missed_tot_wcet signal is raised. This happens when wcetreg, tick, and running are all equal to '1' and update_wcet is equal to zero. missed_tot_wcet is reset when clrerr is 1. The wcetreg counter is loaded with the value of tgr_WCET when update_wcet is 1. When update_wcet is zero, tick and running are '1' and wcetreg is bigger or equal to 1, wcetreg is decremented by one. This is the hardware counter that gives the WCET monitoring capability to the sc_x components.

5.2.2.4 romifx component

The x at the end of romifx is replaced by the task number corresponding to the above sc_x. The romifx component consists of a ROM data structure. It has a clock input, an address input of address_depth number of bits, and an enable signal. Its output is a simple data port of size data_width bits. The romifx components store the Task Graph information of each task. This information contains the expected behavior of the software task. It records the order of expected software calls to the hardware kernel including the different branches potentially followed, the arguments of these instructions, and the best-case and worst-case execution timing information. For a description of the rules used to encode this information, see the section below on Task Graph coding. The following signals contain the task_graph information whenever romifx is read:

`tgr_branch` of size `code_depth` bits
`tgr_call` of size `CMDREG` bits
`tgr_arg` of size `BITPO` bits
`tgr_WCET` of size `WCET_width` bits
`tgr_BCET` of size `BCET_width` bits

<u>5.2.2.5 verification logic</u>

Logic outside of the romifx component constantly verifies these signals. BCET_OK is 1 whenever wcetreg is smaller or equal to tgr_BCET. WCET_OK is 1 whenever wcetreg is greater or equal to tgr_WCET. To understand why the inequalities are directed as such, the reader should remember that the counter counts down. INSTR_OK is 1 whenever instr_int equals tgr_call. ARG_OK is 1 whenever arg_int equals tgr_arg. IS_NODE is 1 when tgr_call is a node. The signals that are verified against the Task Graph information come from the sc process, which we describe below.

<u>5.2.2.6 sc process</u>

This process is akin to a processor inside the sc hardware. It reads instructions from the romifx component and checks that transitions occurring are legal and that their BCET and WCET are met. Figure 12 shows the sc state machine.

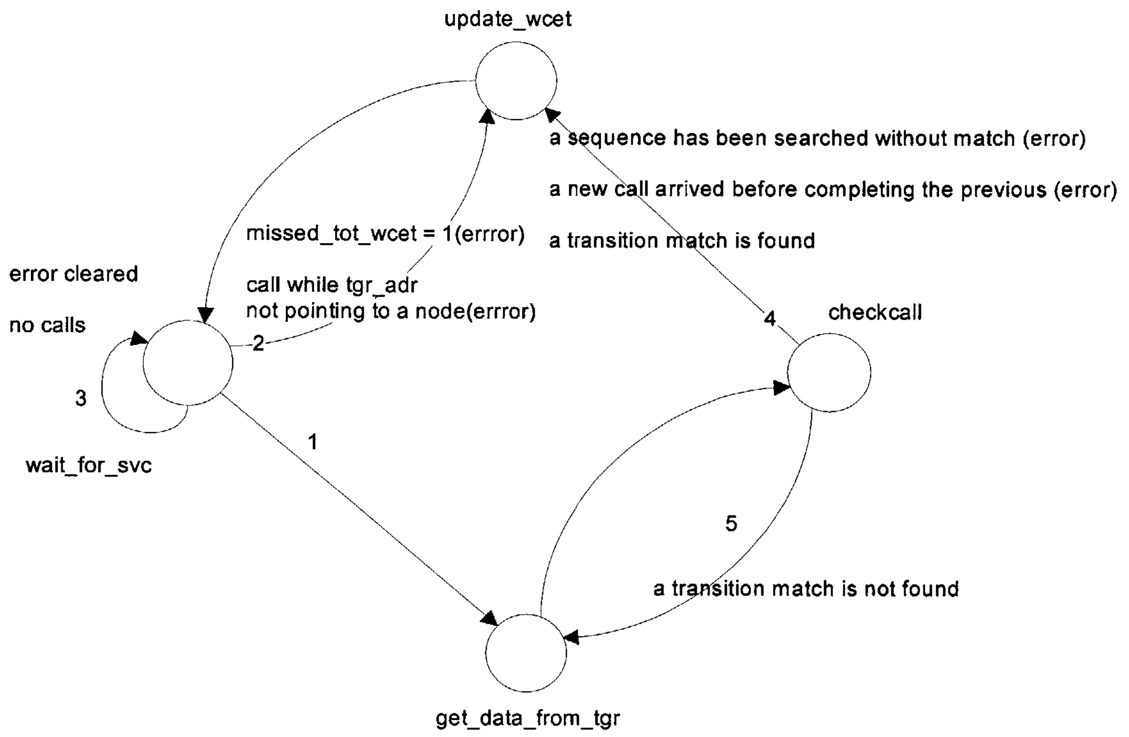

update_wcet
a sequence has been searched without match (error)
a new call arrived before completing the previous (error)
a transition match is found
missed_tot_wcet = 1(errror)
call while tgr_adr not pointing to a node(errror)
error cleared
no calls
2
3
wait_for_svc
1
4
checkcall
5
a transition match is not found
get_data_from_tgr

Figure 12: The sc State Machine

The two active states in sc are states wait_for_svc and checkcall. States update_wcet and get_data_from_tgr transition immediately to wait_for_svc and get_data_from_tgr respectively. Their purpose is to wait one clock cycle, in the first case to let the wcetreg latch tgr_WCET inside the wcet_proc, and in the second case to let romifx be read and all the checking logic be valid.

The tgr_addr signal is somewhat equivalent to a processor's program counter. In state wait_for_svc, the tgr_addr variable should always point towards a data line in romifx that contains a NODE in its tgr_call field. In other words, NODEs are the locations in the Task Graph where the sc process awaits service. This is why, when in this state a new call is received and the BCET is OK, the process checks that the present tgr_call field is a NODE. Before making that check, missed_tot_wcet equal to one signals an error and sets update_wcet and tgr_adr to the error_handler_address. call_sc equal to one signals that the instr and arg inputs to sc are valid, and these are latched into arg_int and instr_int, the verification logic signals. If indeed IS_NODE is one, the sc starts fetching the next entry in the ROM Table by incrementing tgr_adr and transitioning to state get_data_from_tgr, illustrated by path 1 in the graph of Figure 12. If IS_NODE is not one, then an error occurred and sc_error and update_wcet are raised, the tgr_addr is set to the error handler address of the romifx component and sc transitions to update_wcet state, which is path 2. If a new call is received and the BCET is not OK then update_wcet is raised and the process follows path 2 as well. Here however, diagnosis is accomplished by setting missed_bcet to NOT(BCET_OK) and sc_error to NOT(IS_NODE). Note that if IS_NODE is one then the error_code output of sc_x will contain only one active bit: the missed_bcet. If BCET is missed and IS_NODE is zero then error_code will contain two active bits: sc_error and missed_bcet. Finally, in wait_for_svc, if errclear is one, then sc_error is set to zero and path 3 is used to remain in wait_for_svc. Path 3 is taken by default.

In state check_call, we first check if a sequence has been searched completely or if a new service called has been received before the search could complete. If either is true then an error is reported by raising sc_error and update_wcet and transitioning to state update_wcet. tgr_addr is updated to the error handler address of the romifx component. Transition 1 is taken in the graph of Figure 12. Otherwise, state check call verifies if the current accessed romifx data line corresponds to the received call. If INSTR_OK, BCET_OK and WCET_OK are all 1, then a transition match is found. If the instruction is a DELAYUNTILi or if the argument of the instruction matches, update_wcet is raised and tgr_addr is updated with the content of tgr_branch, making the sc process branch to the next appropriate NODE. If the argument of the instruction does not match (ARG_OK = 0) then several similar instructions to different PO are possible and sc process simply transitions to get_data_from_tgr after incrementing tgr_addr, thus continuing the search. This is path 5. Finally if INSTR_OK is not 1, then similarly the search continues by taking path 5 and incrementing tgr_add.

5.2.3 Possible Sources of MC Detected Error

In summary, the sc process can get into the error_handler_address section following these circumstances:

- A new node was reached after searching a sequence of transitions, and the transition received from software was not found.
- A new call was received from software before the previous one could be checked.
- The call was received too early, and BCET was broken.
- In state wait_for_svc, tgr_addr not pointing to a node, which is a symptom of a Task Graph coding error.
- missed_tot_wcet is signaled

Concerning the second item, a particular application having many possible transitions will make it more likely that an error is generated because the sequence was not checked completely as a new command was received. One way to solve this problem could be to overclock the sc_process.

Another way to present this information is to look at the different error codes that the error_code of a sc_x circuit can output to the sc_top circuit. Table 2 lists the different error codes.

wcet	bcet	Sc_error	missed_tot_wcet	decimal	diagnostic
0	0	0	0	0	tgr coding error
0	0	0	1	1	wcet of sequence is broken
0	0	1	0	2	transition not in sequence
0	1	0	0	4	bcet of call is broken
0	1	1	0	6	bcet of call is broken AND tgr coding error

Table 2: Decoding the error_code signal

Note that the wcet field is not set to one.

A MC returning the Error Code 0000 reveals a Task Graph coding error. To protect against such a possibility from blocking a software task after a system's entry into service, a simple mechanism could be implemented. This mechanism would disable the monitoring of the task that returned a Task Graph coding error. Task Graph coding error should be automatically and systematically verified using tool support.

5.3 Task Graph Coding

Until a reliable tool is build to generate Task Graphs of application tasks from Uppaal diagrams automatically, the system developer must be familiar with the manual process of Task Graph Coding.

Task Graphs constitute the "software" that is executed by the Monitoring Chip. They are the model of the application task that the Monitoring Chip uses to enforce its monitoring.

We shall demonstrate Task Graph coding with an example from the GGNS system. GGNS is presented in Chapter 6. The Task Graph coding below corresponds to the GGNS PO Procedure code.

Figure 13 shows an ordered, sequential list of all the calls to the kernel in the source code for that Procedure (preceded by a `'$'` character), along with the control statements.

```
START
$FPs FUNCTION PROCEDURE START
IF NO EXCEPTION
     $UPe PROCEDURE END
ELSE EXCEPTION
     $UFPx FUNCTION PROCEDURE EXCEPTION
     $UPxe PROCEDURE EXCEPTION END
END IF THEN ELSE EXCEPTION
IF BARRIER REQUEST
     IF BARRIER CLOSED
     ELSE BARRIER OPEN
          $UEb ENTRY BEGIN
          IF NO EXCEPTION
               $UEe ENTRY END
          ELSE EXCEPTION
               $UEx ENTRY EXCEPTION
          END IF THEN ELSE EXCEPTION
     END IF THEN ELSE BARRIER CLOSED
END IF BARRIER REQUEST
IF NO EXCEPTION
     $FPe FUNCTION PROCEDURE END
ELSE EXCEPTION
     $FPx FUNCTION PROCEDURE
EXCEPTION END
```

Figure 13: List of Software calls to RavenHaRT for a PO Procedure

This list is one description of the Task Graph corresponding to a task calling only one PO Procedure. We use this to code the ROM Table, which is shown in Table 3 below.

Address	branch	call	arg	WCET	BCET	*Comment*
10011	01010	Exi	100	101	000	
10010	00000	node	000	101	000	
10001	10010	UExi	100	101	000	
10000	01010	UEei	100	101	000	
01111	00000	node	000	101	000	*two paths possible*
01110	01111	UEbi	100	101	000	
01101	00000	node	000	101	000	
01100	00000	FPxi	100	101	000	*next node is first node*
01011	00000	FPei	100	101	000	*next node is first node*
01010	00000	node	000	101	000	*two paths possible*
01001	01010	BARR	000	101	000	
01000	01101	BARR	001	101	000	
00111	00000	node	000	101	000	*two paths possible*
00110	00111	UPxei	100	101	000	
00101	00000	node	000	101	000	
00100	00101	UFPxi	100	101	000	
00011	00111	UPEi	100	101	000	
00010	00000	node	000	101	000	*two paths possible*
00001	00010	FPSi	100	101	000	
00000	00000	node	000	101	000	

Table 3: ROM Table of Task Calling a PO Procedure

The ROM Table should be read from the bottom up. We call *sequence* of a node the possible transitions that can emerge from the node. In the ROM Table, the sequence of a node contains the node and all commands above it until the next node, which is not included in the sequence.

When coding the ROM Table, the branch address of each call that is not a node must point to the address of a node, or else an error will be signaled with error code 0000 or 0110 (see Table 2).

The same information in graph form is shown in Figure 14, with the node address displayed in decimal:

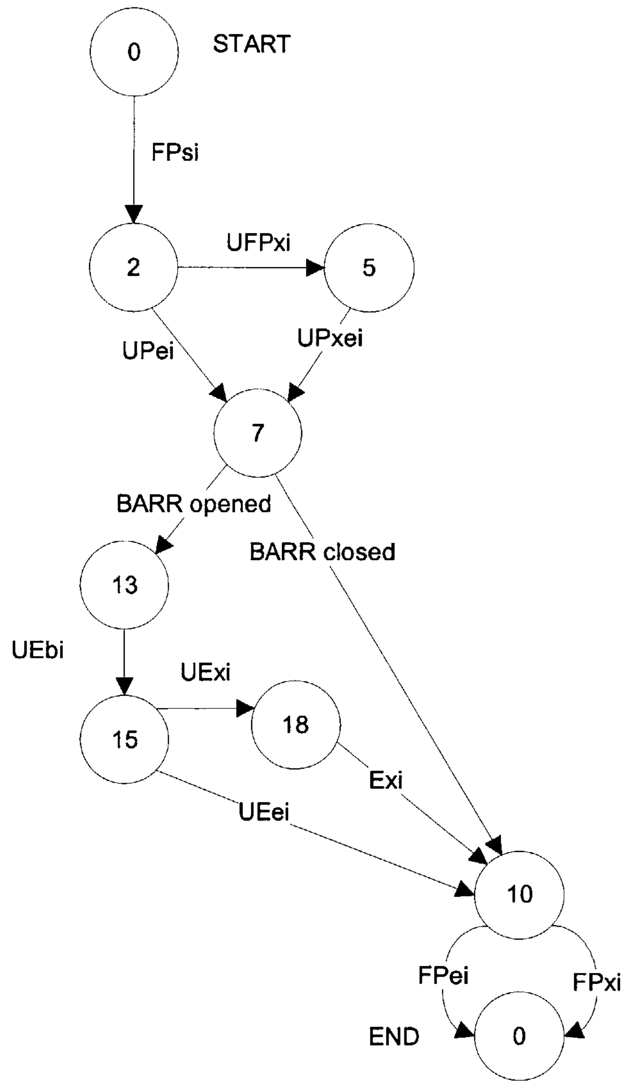


Figure 14: The Task Graph of GGNS's Low Rate Nav Task

One important design parameter is the counter resolution and range. These are determined by the variables BCET_width, WCET_width and system_frequency. For example, if the clock frequency is 100 MHz and the system_frequency is set to 1000 then 1 tick is generated out of the timer process in the sc_top component every ((10^3) / 10^8 = 10^-5) 10 microsecond. BCET and WCET can therefore represent a range of value of (2^BCET_width * 10 us) and

(2^WCET_width * 10 us) respectively with a 10 us resolution. These numbers should be carefully chosen to match the application's timing as efficiently as possible.

For the Task Graph coding of nodes, the corresponding BCET should always be zero. The corresponding WCET is the "tot_wcet", or total WCET. This number is the greatest WCET value of all the possible commands of the present sequence plus one. This is the number checked by the wcet_proc process. For all other commands, the wcet and bcet numbers to be entered are computed by finding the difference between tot_wcet and the particular transition's actual bcet and wcet. Considering the following example:

N+4	node	6	
N+3	Esi	2	4
N+2	Esi	1	3
N+1	Esi	3	5
N	node	5	0
addr	call	WCET	BCET

Figure 15: Example ROM Table Task Timing Coding

At address N, 5 represents the time after which the countdown timer of the sequence signals a missed_tot_wcet error. At address N+3, 2 here represents the time after which the countdown timer will make WCET_OK equal to zero if the particular path is followed. Since tot_wcet = 5, then the actual WCET of the N+3 call is 5 – 2 = 3. Similarly, N+3's actual BCET is 5 – 4 = 1. The verification logic will yield a BCET_OK of 1 once the countdown timer, starting at 5, decreases to 4. Address N+2 has a WCET of 1, which means it is a transition with *the* longest WCET in the sequence. Its actual wcet is 4. Filling the timing information of the Task Graph thus requires knowing the bcet and wcet values for all transitions, finding the longest wcet, adding one to it and setting it as the sequence's tot_wcet, and then performing the complementary algebra to find the wcet and bcet numbers to enter in the table. There should be no zeros in the WCET column, except if tot_wcet is zero as well. This disables WCET monitoring capability.

The error_handler_address is meant to point towards a different section of the ROM Table. This allows monitoring to continue if the task's behavior is changed by the system reacting to the error. The capability to switch to different parts of the ROM Table depending on the error source (missed BCET, unrecognized software transition, etc) is used for system fault-mode configuration synchronization as explained in Chapter 6.

Special Acknowledgment

Special thanks to Johan Furunäs from Mälardalen University for providing the design and implementation of the Monitoring Chip.

Chapter 6
Implementing Fault Tolerance

The inability to produce error free software is widely recognized [TP00], [LA90]. As seen previously, fault tolerance is the technique adopted to cope with faults that remain in the system after the design phase is complete.

The elements presented in the previous Chapters have built a base than can support non-intrusive fault tolerant designs of mission critical real-time systems. Together, RavenHaRT and the Monitoring Chip (MC) provide the run-time context and error detection functionality allowing the real-time system to gain awareness of the correct and timely execution time of its tasks. Now that the real-time system is alerted of the appearance of faults in its application tasks, the question emerges on how to find a practical and efficient way to take corrective action. This Chapter studies this question. Principles of fault tolerance are reviewed in section 6.1. We then examine our options: Section 6.2 looks at what can be done using the MC services as a starting point. A mission-critical real-time system application example, the Gurkh Generic Navigation System (GGNS), is presented in Section 6.3 with the purpose of illustrating the fault tolerant solution ultimately proposed. This solution is presented in Section 6.4, discussed in Section 6.5, and the issues related to its implementation in the context of the Gurkh Project are presented in Section 6.6.

6.1 Fault Tolerance Theory

Classical fault tolerance theory identifies three constituent phases in a fault tolerant system once errors are detected: damage confinement and assessment, error recovery, and fault treatment and continued service [LA90]. We present these generalities below, along with a discussion about relevant issues related to backward error recovery of concurrent processes.

6.1.1 Fault Tolerance Phases

- Damage confinement and assessment: during the damage confinement and assessment step, the extent of the unwanted consequences of a fault is assessed. Unwanted consequences can spread with any flow of information during the time between the occurrence of the fault and its detection. The structure of the system constrains these flows of information and will be used for the confinement of the damage by the use of protection mechanisms [LA90].

- Error Recovery: error recovery is the first active step, and consists of making the changes necessary to eliminate the error. If the damage of an error detected can be sufficiently well predicted, forward error recovery schemes, where selective adjustments to the system state are made, can be applied. Conversely, if the damage is unanticipated, backward error recovery schemes are more suitable. In backward error recovery, the reversal of time is simulated by restoring a previous system state [LA90]. The saved system state is called the recovery point, and the restoration of that state is referred to as rollback.
- Fault Treatment and Continued Service: the last step consists of making sure the fault that created the error does not reoccur. Repetition of the fault can cause system failure as the fault tolerance mechanism tries and fails to cope with the errors. Eradication of the fault from the system is called fault treatment, and involves localizing the fault and repairing the system. This step is very difficult to implement at runtime for software systems, and his hardly ever attempted in such conditions. If the fault is transient, the system can do without fault treatment. Continued Service concerns the resumption of normal system operations [LA90].

6.1.2 Issues in Recovery for Cooperative Concurrent Processes

A problem particular to concurrent processes which exchange information arises when individual processes undergo backward error recovery. This problem is called the *Domino Effect* because of the chain of rollbacks that can occur among processes than communicate with each other between their respective recovery points. If a recovery point precedes a communication exchange, the process that rolled back will not be able to communicate with its correspondent unless the correspondent rolls back as well, and so on. The strategy to work around this problem is to identify which, of the subsets of processes interacting with the process causing the error, will need to be altered to maintain the consistency of the system [LA90].

6.2 Starting with Error Detection

The MC error detection service could be used only as an instrument for the enforcement of predetermined WCET of tasks. A system could be developed relying on the assumption that whatever WCET number is preset for a task in the MC, that number will never be broken, whether the task completes execution or not. The infrastructure would in this case provide an interrupt service to the application tasks, much as cyclic-executive systems do, but in a concurrent tasking context. The erroneous task's priority could then be altered to let other tasks meet their deadlines. However, this would directly go against the Ravenscar profile's task model requirements specifying "`No_Dynamic_Priorities`".

There are many ways for a system to react to task overruns. In [PZ03], the authors quickly overview overrun handling methods present in the literature. Their conclusion is that recovery action must be taken at the system level rather than at the individual task level.

In [BW01], Burns and Wellings discuss mode changes and event-based reconfiguration mechanism as a possible solution to deadline overruns, WCET overruns, or unacceptable high frequency of sporadic processes. These ideas are interesting, but the requirement to follow the Ravenscar profile restricts in our case the options suggested.

A possible overrun handling approach is to simply force the erroneous task to restart, exactly as before. This simple scheme is the most basic form of deployment of time redundancy as a fault tolerant mechanism. Still, this scheme provides little guarantees of recovery success. Mission-Critical systems could not afford relying on such a weak level of protection. Another handling approach would be to force the erroneous task to stop executing. The system could be designed with this possibility in mind. The challenge would then be to solve the question of the enforcement of Ravenscar constructs and the guarantee of integrity.

6.2.1 Transient Fault Assumption

The decision concerning what fault tolerant strategy to use in the Gurkh Project is made by considering what assumptions are placed on the faults that are to be tolerated. Citing a NASA Technical Report:

> "A large number of the faults in operational software are state-dependent faults activated by particular input sequences [TP00]."

It is therefore arguable that such faults can be considered as being transient. As we shall see below, our fault tolerant strategy relies in part on this assumption.

6.2.2 Concurrent error detection

It is important to note that while the system is based on the principles of concurrency, we assume that it runs only on one processor. This implies that while the action of monitoring is concurrent, multiple erroneous tasks cannot be detected simultaneously. The erroneous task is treated before any other task in the system execute. Our implementation is not faced with the issues of concurrent error handling and can therefore afford to handle errors sequentially, one after the other. Concurrent error handling issues will have to be addressed in the Gurkh Project on multiprocessor implementations. [XRR00] provides a good discussion of the issues posed by concurrent error handling.

6.2.3 The Importance of the Application Context

Fault tolerant theory recognizes the importance of the real-time application characteristics in determining a particular fault tolerant solution that can carry out efficiently the error detection, the damage confinement and assessment, the error recovery, and the fault treatment for continued service.

6.3 The Gurkh Generic Navigation System

To give substance to the issue of providing fault tolerance in the Gurkh Project, we developed an elementary application testbed, the Gurkh Generic Navigation System (GGNS). An overview of the GGNS application testbed is presented in subsection 6.3.1 and subsection 6.3.2 presents the task model of the application.

6.3.1 The Gurkh Generic Navigation System as an Application Testbed

To motivate the GGNS application testbed, we briefly describe the system on which the GGNS design is based. The system that inspired the GGNS application computes navigation information, such as position, velocity and acceleration, based on two sensors: Inertial Measurement Unit (IMU) measurements and the Global Positioning System (GPS) data. GPS data enters the system in the form of messages that are processed to yield Line-Of-Sight (LOS) data, which is fed into a Kalman Filter (KF). The KF estimates present and future navigation information and corrects these estimates according to incoming LOS navigation data. These estimates are fed to a high rate Sequencer task. The Sequencer gathers all information and performs the actual navigation computations. Control laws are then applied to the derived navigation information so that actuators are appropriately exploited and navigation of the satellite system is achieved.

We are less concerned by the data-processing details of the existing system than by the organization of the different tasks, the data flow between them and their associated timing constraints. GGNS therefore models the core real-time software architecture of the guidance and navigation system described above. The model provides enough functional complexity to provide a representative subset of a Ravenscar-compliant target application and is small enough for the MC and RavenHaRT to be synthesized on the FPGA.

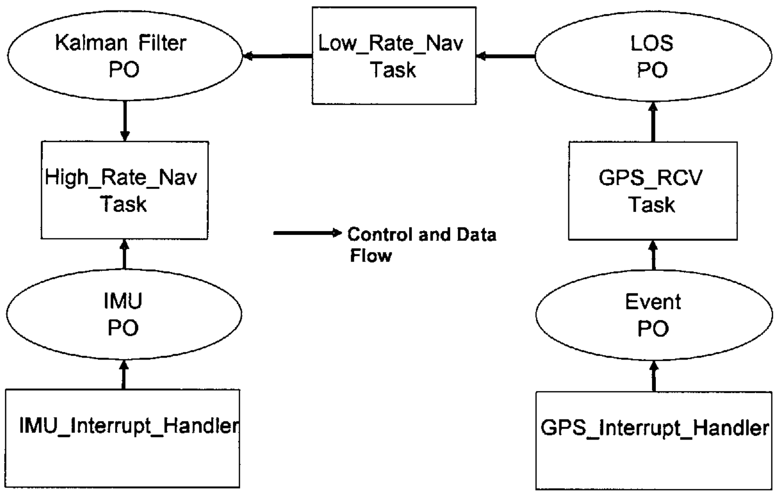


Figure 16: Task Model of GGNS

6.3.2 GGNS Task Model

The GGNS task model is shown in Figure 16. The system consists of five tasks; three are internal to GGNS and two interrupt handler tasks provide input data. The High_Rate_Nav task, acts as the overall sequencer, which provides actuator input to the higher level system. The Low_Rate_Nav task acts as the Kalman filter, carrying out estimation of navigation information by integrating information from both the GPS and IMU. The GPS_RCV task is an event-triggered task that gathers LOS data from the GPS_Interrupt_Handler.

The High_Rate_Nav task collects IMU data and KF data. The Low_Rate_Nav task collects LOS data and sends KF data. The GPS_Receive task is triggered by the arrival of a GPS message and has two activities: collect the GPS message, and send LOS data.

The data communications between these three main tasks are implemented with POs. Three of the POs are the buffers containing LOS data, KF navigation data, and IMU data. Event PO implements the event-triggering capability of the GPS Receive Task. The GPS_Interrupt_Handler receives GPS messages and interacts with Event PO. IMU_Interrupt_Handler receives incoming IMU data and interacts with the IMU data buffer.

While being relatively simple, the GGNS uses the key techniques of concurrency and data exchange used in Ravenscar-compliant applications. Cyclic tasks High_Rate_Nav and Low_Rate_Nav run with different periods and priorities and interact through two POs that include multiple Procedures and Functions. GPS_Receive is event-triggered, and communicates with a cyclic task through a protected buffer. This is an instance of Ravenscar software whose purpose is to process incoming hardware information.

6.4 Fault Tolerant Strategy

The fault tolerant strategy adopted for the GGNS application example is based on the concepts of Model-Integrated Computing concerning the provision of system-level reconfiguration capability to the system. The strategy defines different modes of operation destined to tolerate different application task faults. A particular fault mode of operation implements a consistent system-wide service level which yields a gracefully degraded version of the nominal service. In this sense, the strategy implements forward error recovery at the system-level. In parallel, backward error recovery is used at the task level to ensure inter-task operating mode consistency and to prepare for possible restoration of nominal operation.

6.4.1 Creating The Fault Modes

To use fault models at run-time, the system designer must go through a four step process which identifies system faults, builds the fault models, selects the ones to make available at run-time and finally assembles a fault mode lookup table to be used at run-time.

The first step identifies each possible combination of task errors (which we refer to as fault scenarios) and organizes these hierarchically into classes. The first class corresponds to the nominal case where all tasks execute correctly within their execution time bounds. The second class regroups the possible cases of only one task being erroneous. The third class regroups the possible cases of exactly two tasks being erroneous, and so on. This hierarchical organization serves the purpose of illustrating the different classes of degraded service to be provided in increasing order of criticality.

For GGNS, we are concerned with the three main tasks illustrated in Figure 16. High_Rate_Nav, Low_Rate_Nav and GPS_Receive. There are therefore only eight fault scenarios to consider; they are represented by circles and ellipses in Figure 17.

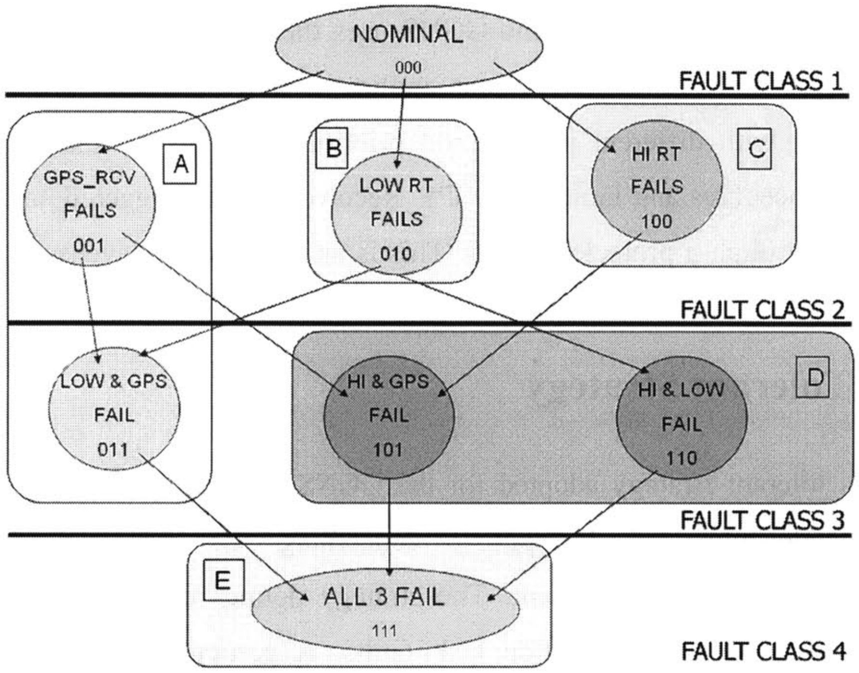


Figure 17: GGNS Fault Scenarios, Fault Classes, and Fault Modes A to E

The second step builds the fault modes; to each fault scenario we associate a fault mode designed to protect the system against the fault. As the number of tasks in the application system increases, the number of possible fault scenarios in the hierarchy increases exponentially. This can quickly become problematic if a fault model is to be built for each fault scenario. At this point, the application's formal model becomes helpful because it provides an abstraction of system behavior that captures the critical interaction between tasks. Using this information, the number of fault modes constructed can be made smaller than the number of different fault scenarios. This requires the functional relationships and dependencies between the tasks to be exploited wisely.

For GGNS, High_Rate_Nav (Hi) needs information issued by GPS_Receive (GPS) which is intermediately processed by Low_Rate_Nav (Lo). If GPS is faulty or Lo is faulty, or both fail simultaneously, Hi will not have access to reliable GPS information. In the three cases the same fault mode can thus be entered to handle the situation. If only Lo is faulty, the designer might decide that some raw information is better than none, and the system bypasses Lo temporarily. The latter choice would yield two fault modes (A and B in Figure 17) for the three fault scenarios (**001**,**010**,**011**). Similarly at the third class, if both Hi and Lo fail or if both Hi and GPS fail, then we would need to insure that Hi can fulfill its essential responsibilities without reliable GPS information. This is fault mode D accounting for **101** and **110**. The resultant system has 5 fault models and 8 fault scenarios, as seen in Figure 17.

The actual work of constructing the fault models for each task involves a substantial design effort which is application dependent. What should be the behavior of Hi if Lo is erroneous? In this case, what should be the behavior of Lo, or GPS? Table 4 details the specific behavior of each task corresponding to each fault mode of GGNS.

Fault Mode	**High_Rate_Nav**	**Low_Rate_Nav**	**GPS_ReveiveTask (sporadic)**
A	Degraded GPS Mode	Holding Error States	Quarantined
B	Degraded LOS Mode	Quarantined	Nominal
C	Basic Mode and Quarantined	Nominal	Nominal
D	Basic IMU only Mode	Quarantined	Quarantined
E	Survival Mode	Survival Mode	Survival Mode

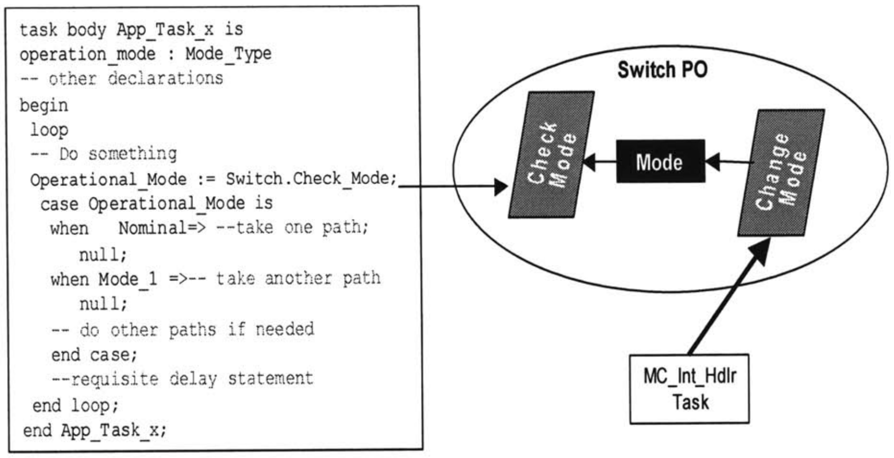
Table 4: GGNS task behavior for each Fault Mode

The third step requires the system designers to select and implement only the fault modes required for the target level of fault tolerance given the limited resources of time and gate space available to them. The hierarchy of fault scenarios assists the designers in this decision.

Finally, the fourth step constructs the Fault Model Lookup Table that will be used to retrieve the fault mode to be switched to given the current mode, the identity of the erroneous task, and the type of error detected. Note that the reconfiguration mechanism can only switch to a mode of the upper or lower class adjacent to the class of the current fault mode.

6.4.2 Making Design-Time Fault Models Available at Run-Time

To make the chosen design-time fault modes available at run-time, each application task follows the same template as shown in Figure 18, which was reproduced from [LSG05].



```
task body App_Task_x is
operation_mode : Mode_Type
-- other declarations
begin
 loop
 -- Do something
 Operational_Mode := Switch.Check_Mode;
  case Operational_Mode is
   when   Nominal=> --take one path;
      null;
   when Mode_1 =>-- take another path
      null;
   -- do other paths if needed
   end case;
   --requisite delay statement
 end loop;
end App_Task_x;
```

Figure 18: Switching Modes

The first instruction that a task executes within the loop is a call to the Check_Mode Function to read the MODE protected variable present in the SWITCH . This MODE variable determines the control flow of the application task. The different paths through the program converge before the task delays itself or loops. The Change_Mode Procedure of the SWITCH Protected Object is the only way to change the value of MODE, and is accessed by the Monitoring Chip Interrupt Handling Task (MCIHT) to issue a mode change instruction.

6.4.3 The Monitoring Chip Interrupt Handling Task

As first discussed in the *Monitoring Chip As A Component Of RavenHaRT* section of Chapter 5, subsection 5.1.5, the MCIHT will be the task preempting the erroneous task as a consequence of the Monitoring Chip's intervention. The deployment of the fault tolerant resources preventing the erroneous task to damage the system is therefore undertaken by the MCIHT. The behavior of the MCIHT is illustrated in Figure 19.

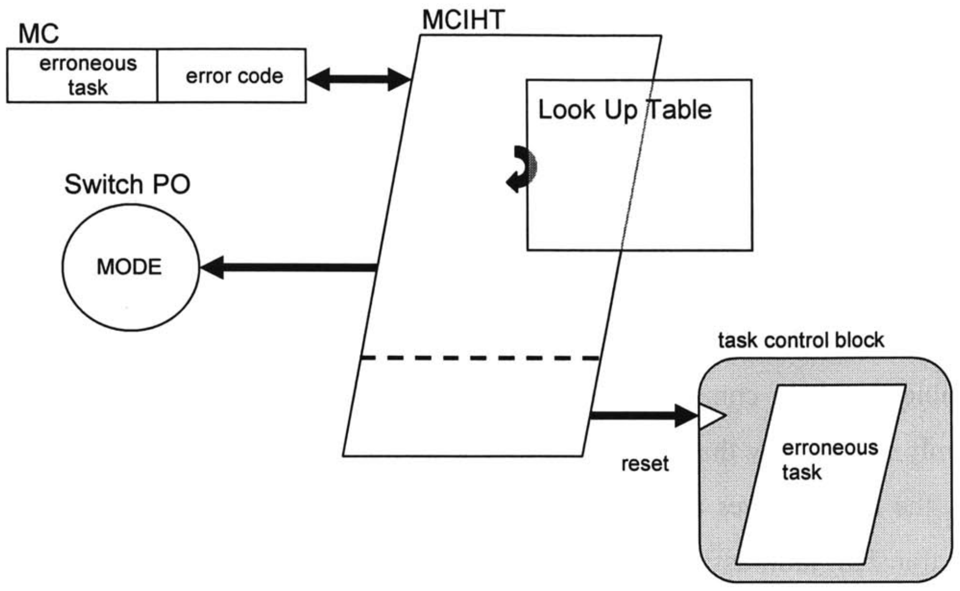


Figure 19: MCIHT Activity

First, the MCIHT is responsible for polling the register of the Monitoring Chip that contains the errtask and errcode information. By doing so, the fault tolerant strategy can adapt to the particular error that occurred. The action of reading from that register acknowledges the interrupt (the errclr signal is set) so that no other explicit interrupt acknowledgment is needed to be accomplished by the software.

Second, the MCIHT task accesses the fault mode Lookup Table and retrieves the next mode of operation given the current mode of operation, the identity of the erroneous task, and the task error code.

Third, the MCIHT calls the Change_Mode Procedure of the Switch PO, as illustrated in Figure 18. For safety purposes, the MCIHT is the only task possessing the privilege to modify the MODE variable.

Fourth and when appropriate, the MCIHT initiates backward error recovery of the erroneous task. This is done in software by calling a special function which restores the state of the erroneous task's last checkpoint. To keep software and hardware synchronized, the appropriate hardware PO circuits of RavenHaRT are reset by the task itself.

Finally, the MCIHT loops around, and suspends on the Entry of the corresponding interrupt PO, thus respecting the system interrupt model presented in Chapter 4.

6.4.4 Isolating the Erroneous Task

Upon MCIHT suspension, the erroneous task resumes execution. Since the task was rolled back, it will first check the mode it should operate in and then start along the path of execution corresponding to the new mode. The behavior of the erroneous task in this new mode, i.e. the fault mode it triggered, can vary. The fault tolerant strategy can implement the concept of stand-by sparing presented in [Ran75] by making use of *alternates*, which are computations that attempt to deliver a gracefully degraded version of the service required by the task when executing in its Nominal mode. Another strategy can consider that the fault sustained is transient and thus enable the same computations of the Nominal mode to be executed in the new fault mode. The only reason why this second strategy would be implemented is to bet that the second time around, the task behaves as expected. This is the strategy chosen for our implementation, based on the transient fault assumption. This time around however, measures need to be taken to prevent the damage of the previous or possible new error to spread to the rest of the system. We want to isolate, or quarantine the erroneous task. These measures consist in part of flagging each transaction that the erroneous task undertakes with its correspondents. In Ravenscar, each transaction is accomplished through the used of a PO and so this measure is easy to implement.

Under the principles of Model-Integrated Computing, the new operating mode applies for all tasks, and consequently the mode should change the behavior of the system so that the fault is tolerated. Therefore, once the tasks that interact with the faulty tasks have changed mode, they will adapt their behavior so as to ignore the faulty task. This is the other part of the measures required to quarantine the erroneous task. The reason that flagging of interactions is necessary is that flagging acts as a synchronization mechanism. It is necessary to warn the interacting tasks that they should change fault modes before proceeding. The flagging is necessary to enforce a consistent mode of operation for all tasks that depend on the rolled back erroneous task. A task processing PO data that has been flagged knows it should rollback and check the MODE variable of the switch PO. In Figure 20, the need for both a flagging mechanism and fault mode change to isolate the erroneous task is made explicit.

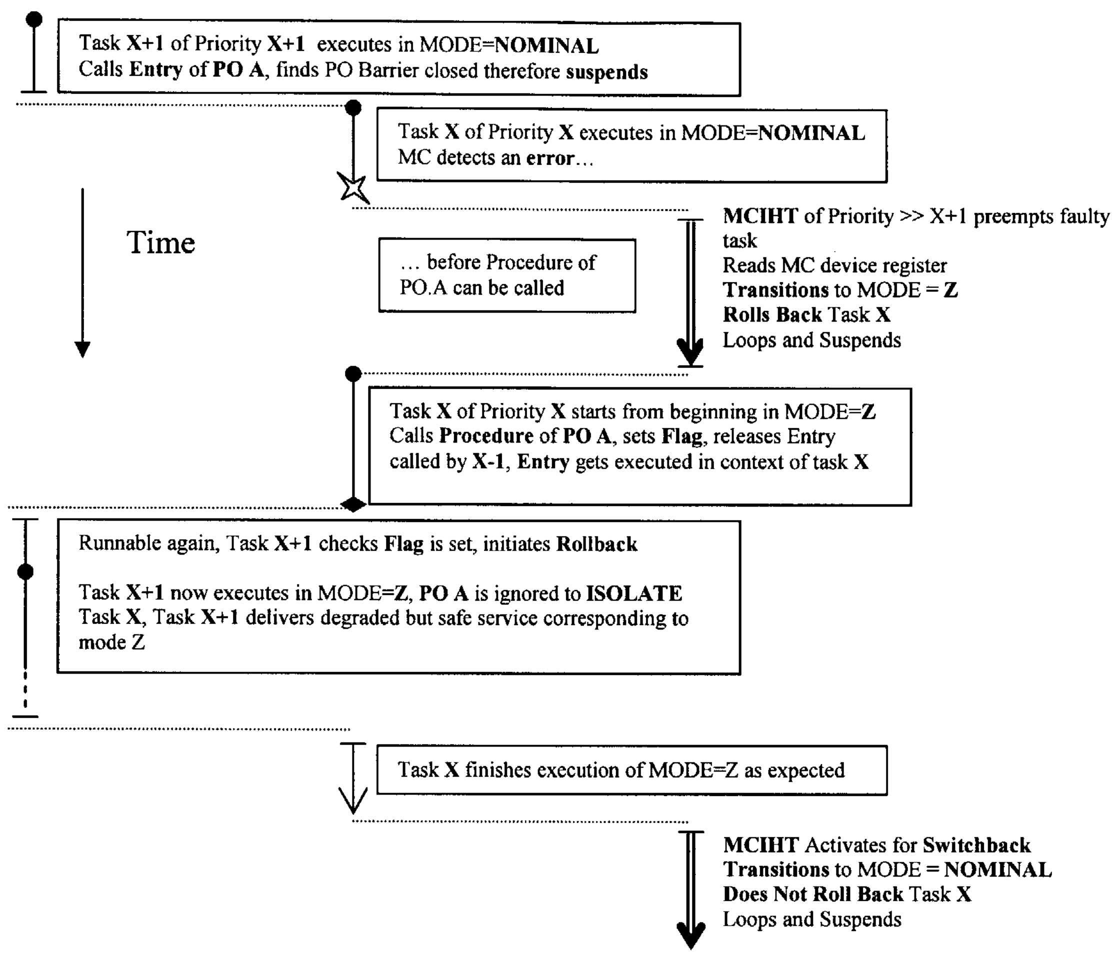

Task X+1 of Priority X+1 executes in MODE=NOMINAL
Calls Entry of PO A, finds PO Barrier closed therefore suspends
Task X of Priority X executes in MODE=NOMINAL
MC detects an error…
Time
… before Procedure of PO.A can be called
MCIHT of Priority >> X+1 preempts faulty task
Reads MC device register
Transitions to MODE = Z
Rolls Back Task X
Loops and Suspends
Task X of Priority X starts from beginning in MODE=Z
Calls Procedure of PO A, sets Flag, releases Entry called by X-1, Entry gets executed in context of task X
Runnable again, Task X+1 checks Flag is set, initiates Rollback
Task X+1 now executes in MODE=Z, PO A is ignored to ISOLATE Task X, Task X+1 delivers degraded but safe service corresponding to mode Z
Task X finishes execution of MODE=Z as expected
MCIHT Activates for Switchback
Transitions to MODE = NOMINAL
Does Not Roll Back Task X
Loops and Suspends

Figure 20: Fault Mode Transition and Switchback

The task restarting in the new mode ignores communication with the erroneous task. In an effort to feature graceful degradation, it outputs safe results to the other tasks it interacts with. In the new mode, Flags from the erroneous task are ignored because communication with the erroneous task is a non-issue. The erroneous task becomes effectively quarantined. The Domino Effect discussed before is avoided simply because the dependent tasks are restarted in a fault mode designed to make them the shield preventing the error from propagating to the rest of the system.

Tasks not dependent on the erroneous task will eventually restart in the new mode. Their independence will make them unaffected by the error so the new mode for these tasks can be the same as the Nominal mode.

6.4.5 Monitoring Chip Task Graph Fault Mode Synchronization

As stated in Chapter 5, the Task Graphs of the MC are models of the application tasks of the system. Therefore, proper Task Graph coding in the MC ROM tables should include a starting call to the Check_Mode Function to read the MODE protected variable present in the SWITCH Protected Object. If different MODE paths diverge, the MC will be able to track each path. Therefore synchrony between what is executing on the processor and what is monitored is established without the need for an explicit synchronization mechanism.

Two cases are interesting to note.

- The erroneous task can fail at any point in the Task Graph. To account for this possibility, we saw in section 5.3 that the tgr_addr signal, first described in subsection 5.2.2.6's description of the sc process for each task, is set to error_handler_address upon error detection. This allows the MC to branch to any part of the Task Graph if an error is detected. In our case, we specify the error_handler_address to be the start address of the Task Graph.
- The tasks depending on the erroneous task can self-trigger their own rollbacks we saw above. They can only do so after having checked the Flag following communication with a PO. An appropriately coded Task Graph will therefore account for the possibility that task may then initiate a rollback by including a branch to the start address of the Task Graph.

6.4.6 Switchback

We will now look at the strategy used to resume Nominal mode operation despite the isolation of the erroneous task under fault mode behavior.

Because of the self-imposed requirement stating that only the MCIHT shall modify the MODE variable of the Switch PO, switchback to Nominal mode must involve execution of the MCIHT and intervention of the Monitoring Chip.

Under the transient fault assumption, we decided earlier to have the behavior of the erroneous task in the new fault mode be the same as the nominal task behavior. While keeping the erroneous task quarantined, this scheme presents the advantage that monitoring for errors can continue as before. The Task Graph of the Monitoring Chip for this task in this fault mode can be adapted to the capability desired from the Monitoring Chip. Consider that the Task Graph of the fault mode is the same than the Task Graph of the Nominal mode. Two outcomes are then possible, depending on if an error is detected.

- An error is detected; the MCIHT runs and decides if a new MODE switch is necessary. More likely than not, a MODE switch will be unnecessary since the system should stay in the fault mode in which it already is.
- No errors are detected

The second case basically says that this time around, the fault that triggered the error that lead to the present fault mode did not reoccur.

Other switchback strategies might be considered, depending on the design-time fault assumptions. If the fault is unlikely to occur two times in a row, the MCIHT and the MC ROM Table might be coded so that switchback occurs automatically after one loop of the erroneous task under the fault mode. Conversely, if a fault is likely to occur several times in a row, the MC ROM Table might be coded not to react to it, thereby saving processor execution time otherwise wasted on the purposeless execution of the MCIHT. A possible implementation would see the task's fault mode worst-case execution times be changed to higher values. The problem in this case is to find a way for the MC to identify that conditions for switchback are met. This could be done by setting the task's fault mode best-case execution time to the values of the task's Nominal mode worst-case execution time. The MC would then detect if the timing bounds are broken from below, indicating that the nominal upper timing bounds are not broken.

The case of similar Task Graphs for Nominal and fault modes is the one we decide to choose for our implementation. In our implementation, the second outcome described above (no errors detected) is a sufficient condition for switchback initiation. To signal the MCIHT in cases of no errors, the Monitoring Chip must be altered. The modification involves the use of an extra bit, the *Switchback Bit*, in the ROM table and the use of a new error code. The details are given in the implementation section below.

6.5 Discussion

The fault tolerant strategy presented fits with the constituent phases of fault tolerance outlined in section 6.1 and the mode change requirements identified in [RC04].

6.5.1 Damage Confinement

Damage confinement is ensured in part by the present Ravenscar structure, by the MC error detection mechanism, and the Fault Mode paradigm.

Let us recall the decoupling between faults and errors. Errors are what are detected and faults are what cause errors. Our error detection mechanism tracks only certain kinds of errors: execution time bound errors and task flow errors. Thus three cases can be distinguished:

1. A Fault occurs that leads to an error detectable by the MC.
2. A Fault occurs that leads to an error not detectable by the MC.
3. A Fault occurs that leads to an error that is detectable by the MC and another that is not.

Since the types of faults of case 2 are non-detectable, we cannot protect the system against them. In case 3, the fault that caused a detectable error might also have caused an undetectable error; for example, corrupted data. We want to protect the system against these errors also. Note that the only way that erroneous task can communicate corrupted data to the external world is by making use of POs. Invoking a PO involves making transactions with RavenHaRT that are picked-up by the MC, which by then will have detected the detectable error caused by the same fault. The MC signals errors it detects immediately. In our single processor system, before any other task can run, the MCIHT will take action to handle the error. Thus, if the fault causing the detectable error also caused data corruption, the erroneous data will not get propagated. After the MCIHT changed the system's mode of operation, the erroneous task will announce to the task that are depending on it that it is in quarantine, using the Flag on the PO communication. Once all interacting tasks are in the same fault mode of operation, the erroneous task is prevented from providing data to the rest of the system.

In case 1, the damage to the rest of the system caused by the error might simply be considered to be the extra execution time taken. In this case the fault mode change will make the system adapt to the problem and the damage will be contained.

6.5.2 Damage Assessment

The damage assessment is a design time issue. Making no assumptions on the fault, but making assumptions on the error, we predict the extent of the damage to be limited to the damage caused by the simple elimination of the erroneous task (the consequence of its isolation while in quarantine). By preparing for the worse, the other tasks can make guarantees on their level of service.

6.5.3 Error Recovery

At the task level, the error is recovered with backward error recovery using the rollback mechanism. At the system level, the error is masked with forward error recovery. By functionally taking the erroneous task out of service, the behavior of the system displays graceful degradation. The erroneous task is ready to be switched back into operation any time.

6.5.4 Fault Treatment and Continued Service

Our fault tolerant strategy does not attempt to identify and eradicate the faults causing the errors that are detected. The transient fault assumption allows us to do so.

Resumption of normal operation is implemented by the switchback mechanism which is achieved by the coherent action of a composition of elements: the MC Fault Mode Task Graph coding, the erroneous task fault mode behavior and the MCIHT behavior.

6.5.5 Mode Change Transition

In [RC04], Real and Crespo survey the issues related to the temporal overload produced during Mode Change Transitions for real-time systems. They identify four aims to be reached during the Mode Change Transition to ensure system feasibility. Each objective is addressed in the context of our fault tolerant strategy [LSG05]:

1. Schedulability – During the Mode Change Transition, the control flow of the erroneous task and its dependents change due to the rollback mechanism. The Mode Change Transition scheduling verification is made prior to system implementation for each task using the smallest of the Nominal or fault mode deadlines and using a WCET for the task consisting of the sum of the WCET of the task in both Nominal and fault mode of operation.

2. Periodicity – The behavior of RavenHaRT is not altered during the transition and the kernel will continue to ensure activation of periodic tasks.
3. Promptness – The promptness requirement is necessary to answer to Mode Change Requests that come from outside the real-time system. In our system, the Mode Change Request comes from within the real-time system to handle the error detected. Thus the promptness requirement simply falls back on the requirement that tasks in the fault mode shall meet their deadlines.
4. Consistency – The use of Protected Objects for inter-task communication, combined to the fixed priority requirement of Ravenscar ensures that shared resources are used consistently.

Our fault tolerant strategy gives us the possibility to meet the Mode Change Transition objectives.

6.5.6 Limitations

The fault tolerant strategy presented has a number of limitations in terms of mode change transients, space requirements, design efforts and verification complexity.

The switchback policy enforced is a tradeoff between lack of promptness to return to nominal operation and inefficient usage of computing cycles while processing the MCIHT. Each execution of the MCIHT causes transients in the scheduling of the system tasks which must be designed to handle these.

Physically, the required logic space to implement the elements is constrained by the space available on the FPGA chips used. For RavenHaRT, space required is proportional to the number of task. For the MC itself, the space required is proportional to the same number. The fault tolerant strategy proposes the elaboration of fault modes for different classes of failures, as in Figure 17. If more than one erroneous task is to be tolerated at the same time, the number of fault modes grows at an exponential rate. Each task must have a defined behavior for each fault mode. Not counting the design effort necessary to ensure coherent and predictable system-wide behavior of tasks for each fault mode, each new fault mode must be accounted for in the MC's ROM Tables. FPGA gate space becomes a limiting factor. For the same reasons, design and verification efforts grow significantly with increasing number of fault modes. The complexity of the fault tolerant strategy can then become sufficiently large that the error proneness of the fault tolerant system overweighs the error proneness of the original fault intolerant system.

Ultimately, the focused design of a restricted number of fault modes used to tolerate targeted errors in the context of a specific application will provide fault tolerant services satisfyingly increasing the dependability of the embedded system.

6.6 Implementation Issues

We now examine the specific changes brought to the Gurkh Project to implement the fault tolerant strategy proposed.

6.6.1 Checkpointing and Rollback

New low-level software capability must be added to the system to implement checkpointing and rollback. The checkpointing is done at the start of every loop of the system's task. The rollback mechanism is either called by the MCIHT, when the rolledback task is the erroneous task, or by a task itself, when it is a dependent of an erroneous task. The rollback mechanism can therefore accommodate both.

The development version of our system uses C-code in a Ada Ravenscar structure. For reference, the checkpointing function is called checkpoint(). A chkpoint_TCB_LIST construct is created for each task to provide the storage space necessary to save a task's context. The assembly function store_registers in tsw.S is the function used to copy the current task context in the chkpoint_TCB_LIST construct.

The function restore_tcb(int taskid) called from another task than the task specified by taskid is used to copy the content of the checkpoint structure back in the appropriate memory locations, thus implementing rollback. The function restore_running_tcb is used to copy the content of the checkpoint structure back in the appropriate registers when the calling task needs to self-rollback.

6.6.2 Switchback

The MCIHT's behavior depends on the current value of the MODE variable and the information provided by the MC. In non-switchback situations, the MCIHT must rollback the erroneous task, whereas in switchback situations, the MCIHT does not need to do so. For the MCIHT to distinguish the two situations, we need to add the capability for the MC to provide the information. This capability is thus implemented in the MC hardware circuit. An extra bit, the Switchback Bit is added to the ROM Tables of the tasks. The MC following a fault mode path in the Task Graph will check if this bit is set on completion of the Task Graph. Completion of the Task Graph is detected by a command having a branch address corresponding to the start address of the ROM Table.

System developers will therefore have to code into the MC's Task Graph if successful completion of a Mode of operation should trigger switchback or not. If so, the Switchback Bit of the correct entry of the ROM Table should be set to one. On completion of the Task Graph, the MC finding the Switchback Bit set communicates the error_code signal 1111. Receiving this code, the MCIHT recognizes a switchback situation. This mechanism requires new logic to be added to the VHDL description of the sc process of the sc_x circuits.

6.6.3 Mode Change Coordination

POs provide a practical interface for task inter-communication. The Flagging mechanism described above is necessary only for task suspending on Entries, and task calling PO Procedures to activate blocked Entries. Each PO in the system contains an extra boolean, the Flag. All PO Procedures opening PO Barriers must have an extra input boolean containing the value of the Flag. If the task calling the Procedure is the erroneous task of the current fault mode, it will set the Flag. In parallel, all tasks suspending on PO Entries must check the value of the PO's Flag on returning from the Entry call to verify if they should initiate self-rollback or not. These changes are straightforward to implement in the Ada code, as we can see in the example below.

```
-- Event PO declaration ---------------------
protected type Any_Object(Ceiling : System.Priority) is
    entry       Wait(D : out message_datatype);
    procedure Signal(D : in message_datatype;
                  Flag : in Boolean);
     Function    Should_Rollback ( Flag : out Boolean);
  private
    pragma Priority(Ceiling);
    Current : message_datatype;
     Mode_Transition : Boolean := False; -- the Flag
     Signalled : Boolean := False;
  end Event_Object;

  protected body Any_Object is
```

```
    entry Wait(D: out message_datatype) when Signalled is
    begin
       D := Current;
       Signalled := False;
    end Wait;
    procedure Signal(D: in message_datatype;
                    Flag : in Boolean) is
    begin
       Current := D;
       Signalled := True;
        Mode_Transition := Flag;
    end Signal;
    function Should_Rollback ( Flag : out Boolean) is
    begin
          Flag := Mode_Transition;
     end
end Any_Object;
```

6.6.4 Building the MCIHT Lookup Table

Our implementation currently uses if-else software constructs to implement the Lookup Table required for the MCIHT to determine its actions. Since it is the only entity with writing privileges to the MODE variable, the MCIHT keeps an internal copy of the current MODE. This software solution becomes problematic and inefficient quickly as the number of fault mode in the system increases. For optimal efficiency the Lookup Table should be implemented in hardware and integrated in the MC. The MCIHT would then simply have to retrieve the next mode to switch to. Table 5 shows what the construct looks like for the GGNS application example.

Current Mode	errtask	errcode	Next Mode
⋮			
Nominal	1	0001	A
⋮			
Nominal	2	0000	Failure
Nominal	2	0001	B
Nominal	2	0010	B
Nominal	2	0100	B
Nominal	2	0110	B
Nominal	2	1111	Nominal
⋮			
B	2	1111	Nominal
⋮			

Table 5: MCIHT Lookup Table

6.6.5 Coordination with RavenHaRT PO Circuits

In our tightly coupled software / hardware design, software rollback should be conducted carefully. In particular, a task forced to rollback during PO transactions will leave the corresponding RavenHaRT PO circuits hanging. To avoid problems of unsynchronized hardware we allow the erroneous task in its fault mode to reset the hardware circuit about to be used. To enable this capability, two new commands were added to the vocabulary used to communicate with RavenHaRT. The RST_E command resets the entry component of the PO circuit specified by writing in the Lower Parameter Register. Similarly, the RST_PF command resets the proc_func component of the PO circuit passed as a parameter. These modifications required editing the myvariables.vhd, interface_opb.vhd and kernel_internal.vhd hardware description files. Thus, the erroneous task executing in its fault mode behavior will call RST_E before calling Es (Entry Start) and RST_FP before calling FPs (Function / Procedure Start).

The tasks initiating self-rollback do not need to make use of these commands, because they cannot be interrupted and rolled back in the middle of a PO interaction, and consequently their PO circuits will already be reset.

Chapter 7
Conclusion and Future Work

The work presented in this thesis detailed a proof-of-concept implementation of a non-intrusive fault tolerant framework for mission critical systems. This framework fits in the larger context of the Gurkh Project, which provides a tool-supported environment for the design, development, formal verification and validation, and maintenance of mission critical real-time systems. The Gurkh Project process benefits from the Ada Ravenscar Tasking software model, hardware based kernel scheduling, and non-intrusive error detection. The Gurkh Project process aims at producing mission critical real-time systems that are faster and cheaper to develop, and that prove to be as dependable as their counterpart designed using non fault tolerant and cyclic executive approaches.

7.1 Major Contributions

In this thesis, major components of the Gurkh Project environment were for the first time integrated and functionally described. The RavenHaRT kernel was augmented with system level hardware interrupt capability and the Monitoring Chip was fully integrated in hardware with RavenHaRT. Description of both elements was detailed to allow future system designers to modify or use the functionality these elements provide at their full extent.

To provide higher dependability to the systems developed using the Gurkh Project process, a fault tolerant framework was provided that exploits the benefits of the Ravenscar profile, the RavenHaRT kernel scheduling and the Monitoring Chip.

The fault tolerant approach developed provides system level handling of errors using the non-intrusive task overrun and flow error detection services of the Monitoring Chip. Multiple system operation modes are used to isolate the erroneous task and reconfigure the rest of the system, thus providing a degraded but predictable level of service. Implementation issues are explored at the conceptual and at the practical level. These include the necessary fault mode design, the provision of damage confinement, the coordination between hardware and software during task rollback and mode transition, and the provision of efficient switchback capability for continued nominal service.

7.2 Directions for Future Work

Future work should be undertaken to complete the objectives of the Gurkh Project.

We classify the directions of future work into different classes depending on their immediate relevance with the fault tolerant framework presented. In order, the more relevant directions are presented first.

7.2.1 A Full-scale Application

Currently, many implementation details of the fault tolerant strategy suggested are still to be refined against a full-scale application example. The best driver for the further development of the fault tolerant framework would be transition from workbench example (such as GGNS) to a full-scale application. Real-world requirements would drive the development of the framework towards a more mature implementation. The performance of the implementation of these solutions could subsequently be compared to those of traditional non-fault tolerant cyclic executive approaches, and confidence would be gained in our approach.

7.2.2 Refinement of Modeling Techniques

The model-checking tools used for system verification are sensitive to the complexity of the models used. In the face of complex systems, modeling heuristics are required to allow efficient modeling of the different components of the system. Such heuristics would allow the total system complexity to be broken down into hierarchies of models with increasing levels of detail. Certain properties would be verified on models providing only the required level of detail. The modeling techniques would need to be established by considering the safety and bounded liveness properties to be checked. A better understanding of the formal analysis process would therefore help the modeling task.

7.2.3 Automated Monitoring Chip Hardware Description Generation

The Monitoring Chip performs using a model of the application. The model consists of a Task Graph for each task in the system. Currently, the Task Graphs are hand-coded in the ROM Table in the hardware description file of the Monitoring Chip. As the number of tasks and the number of fault modes in a design increases, the hand coding of the Task Graphs becomes tedious and time-consuming. A tool that would automatically analyze the verified state machines of the system application and create Task Graphs designed to be used in the MC would be very useful to the Gurkh Project process. The Task Graphs produced would be guaranteed to never yield the Task Graph coding error mentioned in Chapter 5. The tool could be extended to input Ada Ravenscar Gurkh software directly, and produce the MC VHDL file directly. In the VHDL hardware description of the MC, only the number of task and each task's ROM Table changes depending on the application. Automatic MC Hardware description generation would enable a faster, systematic and less error prone Task Graph coding technique than the current manual technique.

7.2.4 Transfer of Fault Tolerant Software Functionality to Hardware

In the current fault tolerance approach, the MC Interrupt Handler is responsible for deciding which mode of operation to switch to. To do so, a lookup table is used in software. As the number of tasks and fault modes increases, the size of the Lookup Table will increase as well. The time required for the MC Interrupt Handler to search the Lookup Table will increase as well, making the mode switching mechanism possibly too time-consuming to be acceptable as a solution. For faster mode switching, a simple solution would be to transfer the Lookup Table to hardware, keeping the execution time of the MC Interrupt Handler as small as possible, since the latter would only have to retrieve which mode to switch to from a register.

Pushing the same idea one step further, the functionality and execution time of the MC Interrupt Handler could be further minimized by moving the functionality of the Switch PO and MODE variable to hardware. Each task would start its execution by verifying the hardware register holding the MODE variable and would be instructed to follow its corresponding task path.

For large systems implementing our fault tolerant strategy, transfer of fault tolerant software functionality to hardware would be very advantageous in terms of system responsiveness to errors.

7.2.5 RavenHaRT Kernel Upgraded for System Interrupts

The system interrupt solution described in Chapter 4 was designed to be integrated to the finished RavenHaRT design. As mentioned in Chapter 4, a RavenHaRT kernel upgrade could benefit from a more direct system interrupt solution that would have direct access to the kernel's Ready Queue. Such a design would be easier to prove correct.

Other avenues for RavenHaRT design optimization are described in [Sil04].

7.2.6 Reversing the role of the Monitoring Chip

The MC will signal an interrupt based on a timing model of the executing task. Thus, given pre-estimated execution times, the MC can inform the system of an irregular timing behavior. With little modifications, the reverse functionality could also be provided. The MC could be used as a passive device non-intrusively counting the execution times of the system application tasks. Using the very same hardware architecture and supporting logic, a task's timer circuit could be enabled to count up at a predetermined frequency as its task starts executing. Reading from the same Task Graph ROM Table mechanism described in section 5.3, the hardware entity could store the counter's elapsed time after each new task software call to the kernel. A special comparator circuit would only keep the longest elapsed time for the WCET value and only the smallest elapsed time for the BCET value. Those two quantities would be stored for, and associated to, each Task Graph node sequence. At run-time, the "Reverse Monitoring Chip" configuration would effectively record the *extrema* execution time bounds monitored for each node sequence of each task.

This "Reverse Monitoring Chip" could be used during system development to assist the work on execution time analysis of the application tasks. Since one of the outcomes of the execution time analysis in the Gurkh Project process is to provide useful BCET and WCET values for the MC to use, the Reverse Monitoring Chip could help the calibration of the bounds to be used for the system's operational service. Extensive testing would be required to provide extrema bounds that are useful: rarely accessed but perfectly normal system states that yield execution-time extremum values should be sought for. This is why the technique of the Reverse Monitoring Chip measurement would be used as a complement, not a substitute, to traditional execution time analysis techniques.

While yielding no guarantees, the information provided by using the Monitoring Chip in a reverse configuration would be useful to confirm the accuracy and the usefulness of the execution-time bound values ultimately chosen to be used in the Monitoring Chip for the system's operational service.

References

[AL03] Asplund, L. and Lundqvist, K., "The Gurkh Project: A Framework for Verification and Execution of Mission Critical Applications", 22nd Digital Avionics Systems Conference, 2003.

[AL03b] Asplund, L. and Lundqvist, K., "Ravenscar-Compliant Run-time Kernel for Safety-Critical Systems Real-Time Systems", 24, 29-54, 2003, Kluwer Academic Publishers, The Netherlands, 2003.

[ALL+04] Avizienis, A., Laprie, J.-C., Landwehr, C., Randell, B., "Basic Concepts and Taxonomy of Dependable and Secure Computing", IEEE Trans. on Dependable and Secure Computing, 1(1):11-13, 2004

[BAJL99] Burns, A, Asplund, L., Johnson, B, Lundqvist, K, "Session Summary: The Ravenscar Profile and Implementation Issues", Ada Letters, June 1999, Vol. XIX, No. 2.

[BDR98] Burns, A., Asplund, L., Romanski, G., "The Ravensccar Tasking Profile for High Integrity Real-Time Programs", Ada-Europe 98, LNCS 1411, 1998, pp 263-275.

[BDV03] Burns, A., Dobbing, B., Vardanega, T., "Guide for the Use of the Ada Ravenscar Profile in High Integrity Systems", University of York Technical Report YCS-2003-348, 2003.

[Bur03] Burns, A., "How to Verify a Safe Real-Time System: The Application of Model Checking and Timed Automata to the Production Cell Case Study", Real-Time Systems, 24, 135-151, 2003, Kluwer Academic Publishers, The Netherlands, 2003

[BW01] Burns, A., Wellings, A., "Real-Time Systems and Programming Languages", Third Edition, Addison-Wesley, 2001

[BW+93] Burns, A., Wellings, A., Bailey, C.M., Fyfe, E., "The Olympus Attitude and Orbital Control System, A Case Study in Hard Real-time System Design and Implementation", European Space Agency ESTEC Contract 9198/90/NL/SF

[DY00] David, A., Yi, W., "Modelling and Analysis of a Commercial Field Bus Protocol", Proceedings of the 12th Euromicro Conference on Real-Time Systems pages 165-172, 2000

[Gor04] Gorelov, S., "Development and Testing of the hardware interrupt PO-entry release mechanism for the RavenHaRT kernel", Embedded Systems Lab Working Paper, Aeronautics and Astronautics, Massachusetts Institute of Technology, September 2004

[HA03] Harbour, M.G., Aldea Rivas, M., "Managing Multiple Execution-Time Timers from a Single Task", Ada Letters Vol.XXIII, No.4, December 2003

[HR+98] Harbour, M.G., M.A. Rivas, et.al., "Implementing and Using Execution Time Clocks in Ada Hard Real-Time Applications", Proceedings of the 1998 Ada-Europe International Conference on Reliable Software Technologies, pp 90-101, June 08-12, 1998

[ISO99] "Programming Languages – Guide for the Use of the Ada Programming Language in High Integrity Systems", ISO/IEC DTR 15942, July 1st, 1999.

[KK95] Kantz, H., Koza, C., "The ELEKTRA Railway Signalling-System: Field Experience with an Actively Replicated System with Diversity", Proc. of FTCS. 1995

[LA90] Lee, P.A., Anderson, T., "Fault Tolerance: Principles and Practice" (Second Revised Edition), Dependable Computing and Fault-Tolerant Systems Vol. 3, Springer-Verlag Wien New York, 1990

[LBM00] Ledeczi, A., Bakay, A., Maroti, M., "Model-Integrated Embedded Systems", IWSAS 2000, LNCS 1936, pp. 99-115, 2000, Springer-Verlag Berlin Heidelberg 2000

[Liu00] Liu, Jane W.S., "Real-Time Systems", Prentice-Hall, Inc. 2000

[LSG05] Lundqvist, K., Srinivasan, J., Gorelov, S., "Non-Intrusive System Level Fault-Tolerance", 10th International Conference on Reliable Software Technologies - Ada-Europe 2005 York, United Kingdom, June 20-24, 2005

[MM88] A. Mahmood, E. J. McCluskey, "Concurrent Error Detection Using Watchdog Processors - A Survey", IEEE Transactions on Computers, vol 37, no .2, pages 160-174, Feb. 1988

[MM01] Ram Murthy, C.S., Manimaran, G., "Resource Management in Real-Time Systems and Networks", The MIT Press, Cambridge, Massachusetts, 2001

[Mos00] Mosensoson, G., "Practical Approaches to SOC Verification", Technical Paper, Verisity Design, Inc. 2000. http://www.verisity.com/resources/whitepaper/technical_paper.html

[PL00] Pettersson, P., and Larsen, K.G., "Uppaal2k", Bulletin of the European Association for Theoretical Computer Science, volume 70, pages 40-44, 2000

[PPC] "PowerPC Processor Reference Guide", Xilinx Embedded Development Kit 6.1 September 2, 2003

[Pun97] Punnekkat, S, "Schedulability Analysis for Fault Tolerant Real-Time Systems", PhD. Thesis, Uniiversity of York, Department of Computer Science, June1997

[PZ03] de la Puente, J.A., and Zamorano, J., "Execution-Time Clocks and Ravenscar Kernels", Ada Letters Vol.XXIII, No.4, December 2003

[Ran75] Randell, B, "System Structure for Software Fault Tolerance", IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, June 1975.

[RC04] Real J., Crespo, A., "Mode Change Protocols for Real-Time Systems: A Survey and a New Proposal", Real-Time Systems, 26, 161-197, 2004.

[See04] Seeumpornroj, P., "pGNAT: The Ravenscar Cross Compiler for the Gurkh Project," Massachusetts Institute of Technology, 2004

[Shaw01] Shaw, A., "Real-Time Systems and Software", John Wiley and Sons, Inc, New York, 2001.

[Sil04] Silbovitz, A., "The Ravenscar-Compliant Hardware Run-Time Kernel," Massachusetts Institute of Technology, 2004

[SK97] Sztipanovits, J., Karsai, G., "Model-Integrated Computing", IEEE Computer, April 1997.

[TP00] Torres-Pomales W., "Software Fault-Tolerance: A Tutorial", NASA Technical Report, NASA-2000-tm210616, 2000.

[VAR99] Vardanega, T., "Development of on-board embedded real-time systems: An engineering approach", Technical Report ESA STR-260, European Space Agency, 1999. ISBN 90-9092-334-2.

[WA02] Ward, M., Audsley, N.C., "Hardware Implementation of the Ravenscar Ada Tasking Profile", CASES 2002, October 8-11, 2002, Grenoble, France, 2002 ACM 1-58113-575-0/02/0010

[Xil04] "Xilinx Virtex-II Pro Embedded Development Platform Documentation", ML310 User Guide, November 2, 2004.

[XRR00] Xu, J., Romanovsky, A. and Randell, B., "Concurrent Exception Handling and
Resolution in Distributed Object
Systems", IEEE Trans. on Parallel and Distributed Systems Vol. 11, No. 10, 1019-1032,
October 2000

APPENDIX
FPGA Space Synthesis Requirements For Full-Scale Applications

This section looks at the FPGA space synthesis required to implement the RavenHaRT and the Monitoring Chip on full-scale applications.

The Olympus Attitude and Orbital Control System

We base our analysis on the case study of the Olympus Attitude and Orbital Control System presented in [BW+93]. When it was launched in 1989, the Olympus satellite was the largest civil three-axis-stabilized communication satellite. The Olympus Attitude and Orbital Control Sub-System (AOCS) is a prime example of a mission critical real-time system.

In this case study, the authors demonstrate the feasibility of re-implementing the AOCS using Ada and its concurrent tasking facilities. The AOCS is required to operate in 6 modes of operation. The case study concerns only the re-implementation of the Normal Mode of operation, which is the most complex and the one used for the greatest percentage of the satellite's lifetime [BW+93]. Note that the AOCS modes of operation are not related to the Gurkh Project fault tolerant modes of operation discussed in Chapter 6.

Assumptions

We chose this example application because we believe that its size requirements are reflective of the type of full-scale applications that the Gurkh Project targets. The software architecture used, even though developed in Ada 83 using the HRT-HOOD design method, is believed to have similar structure requirements than that of an Ada 95 Ravenscar implementation.

Specifically, the HRT-HOOD Ada 83 implementation of the AOCS contains 9 cyclic objects, 4 sporadic objects, 14 protected objects and 16 passive objects for a total of 3300 lines of code.

The following assumptions are made concerning the transition from Ada 83 to Ada 95 Ravenscar.

- All cyclic and sporadic objects are considered to be tasks.
- All protected and passive objects are considered to be Protected Objects.

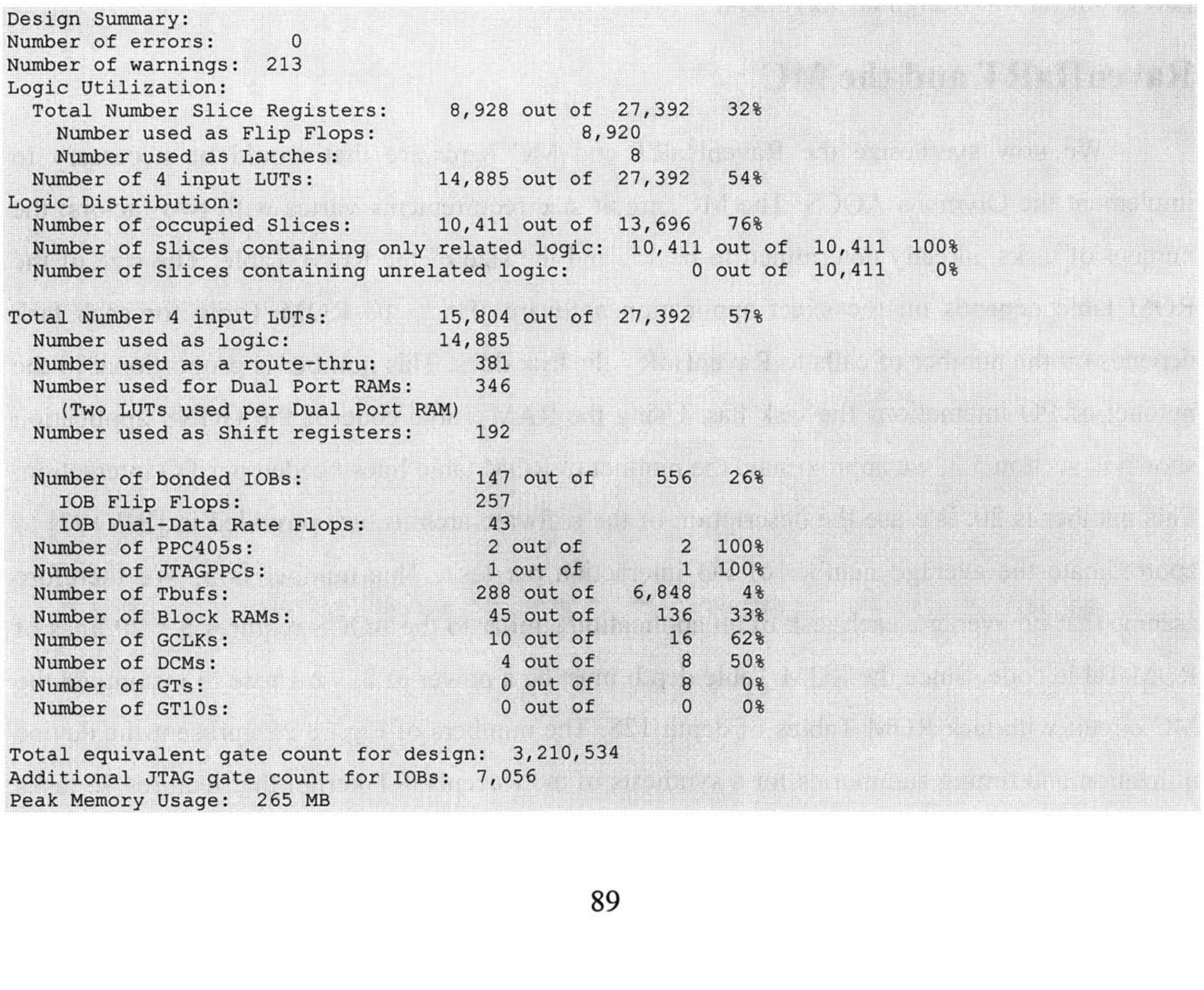
Synthesis Platform

The numbers of Figure 21and Figure 22 were obtained using the Xilinx XST tool in the development environment described in section 2.2.2. More specifically:

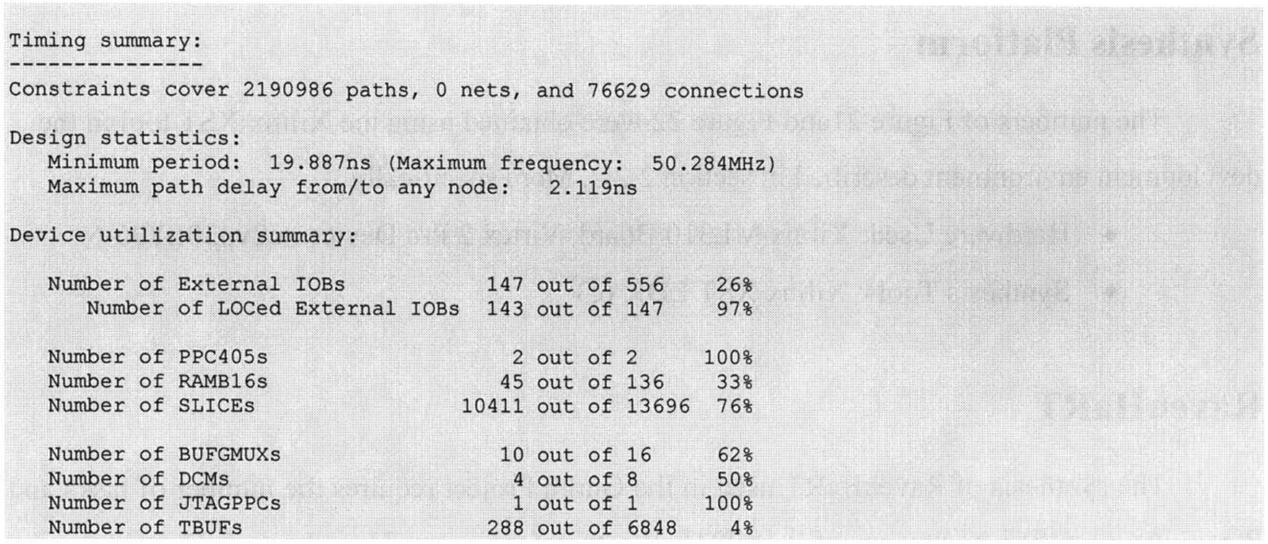
- Hardware Used: Xilinx ML310 Board, Virtex 2 Pro Device xc2vp30ff896-6
- Synthesis Tools: Xilinx XST EDK 6.3

RavenHaRT

The synthesis of RavenHaRT used in the Gurkh Project requires the number of tasks and POs to be specified as powers of 2 [Sil04]. We therefore count 16 tasks and 32 POs for the synthesis of a RavenHaRT capable to support the Olympus AOCS. The numbers of Figure Figure 21 represent the device utilization and timing summaries for a synthesis of the RavenHaRT kernel that features 16 tasks and 32 POs. Note that one task and one PO must be reserved for the null task and the null PO. Such a synthesis would therefore accommodate up to 15 tasks and 31 POs.



```
Design Summary:
Number of errors:      0
Number of warnings:  213
Logic Utilization:
  Total Number Slice Registers:     8,928 out of  27,392   32%
    Number used as Flip Flops:                8,920
    Number used as Latches:                       8
  Number of 4 input LUTs:         14,885 out of  27,392   54%
Logic Distribution:
  Number of occupied Slices:      10,411 out of  13,696   76%
  Number of Slices containing only related logic:  10,411 out of  10,411  100%
  Number of Slices containing unrelated logic:          0 out of  10,411    0%

Total Number 4 input LUTs:        15,804 out of  27,392   57%
  Number used as logic:           14,885
  Number used as a route-thru:       381
  Number used for Dual Port RAMs:    346
    (Two LUTs used per Dual Port RAM)
  Number used as Shift registers:    192

  Number of bonded IOBs:             147 out of     556   26%
    IOB Flip Flops:                  257
    IOB Dual-Data Rate Flops:         43
  Number of PPC405s:                   2 out of       2  100%
  Number of JTAGPPCs:                  1 out of       1  100%
  Number of Tbufs:                   288 out of   6,848    4%
  Number of Block RAMs:               45 out of     136   33%
  Number of GCLKs:                    10 out of      16   62%
  Number of DCMs:                      4 out of       8   50%
  Number of GTs:                       0 out of       8    0%
  Number of GT10s:                     0 out of       0    0%

Total equivalent gate count for design:  3,210,534
Additional JTAG gate count for IOBs:  7,056
Peak Memory Usage:  265 MB
```



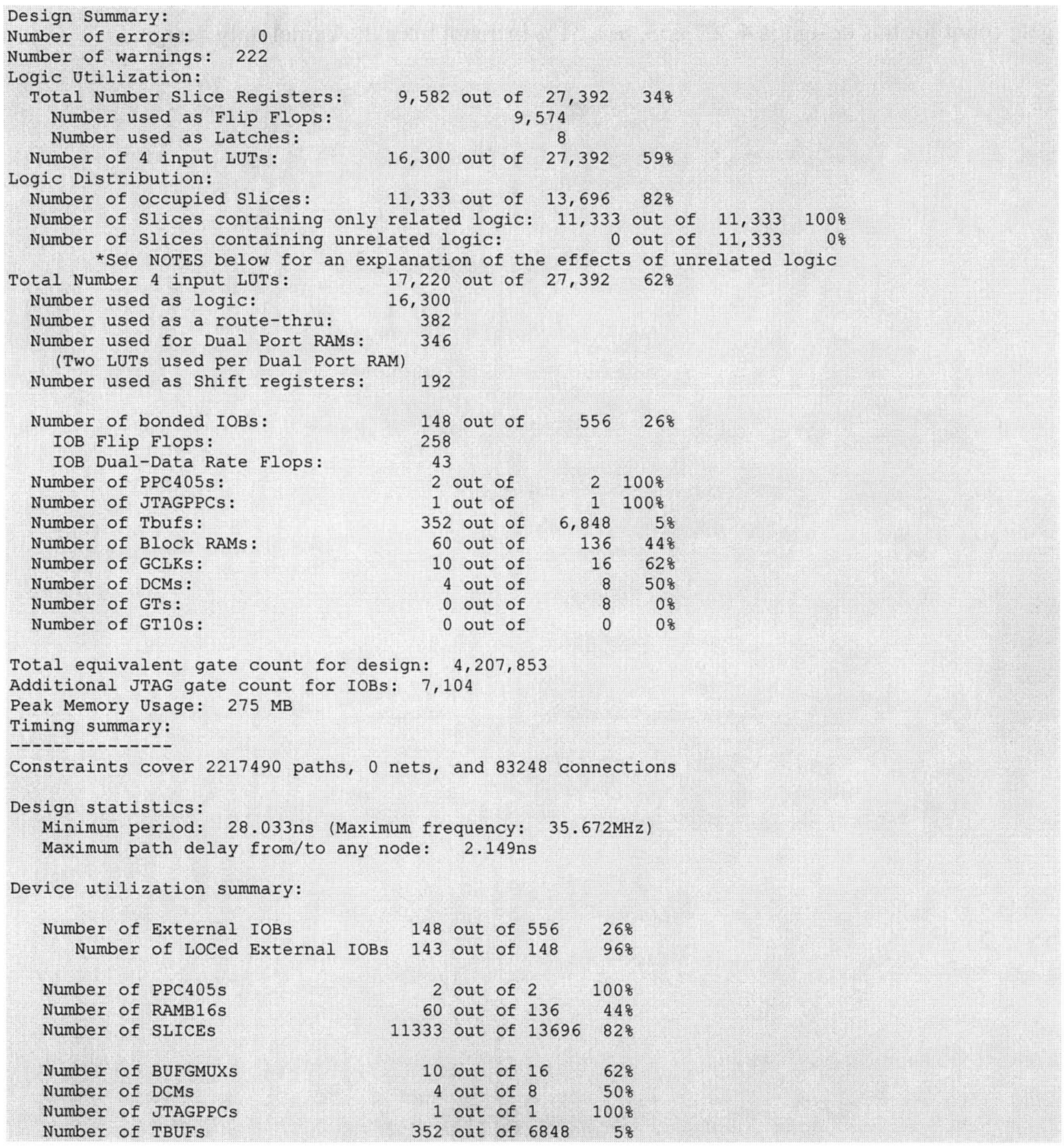
```
Timing summary:
---------------
Constraints cover 2190986 paths, 0 nets, and 76629 connections

Design statistics:
   Minimum period:  19.887ns (Maximum frequency:  50.284MHz)
   Maximum path delay from/to any node:   2.119ns

Device utilization summary:

   Number of External IOBs             147 out of 556    26%
      Number of LOCed External IOBs    143 out of 147    97%

   Number of PPC405s                     2 out of 2     100%
   Number of RAMB16s                    45 out of 136    33%
   Number of SLICEs                  10411 out of 13696  76%

   Number of BUFGMUXs                   10 out of 16     62%
   Number of DCMs                        4 out of 8      50%
   Number of JTAGPPCs                    1 out of 1     100%
   Number of TBUFs                     288 out of 6848    4%
```

Figure 21: Xilinx 2vp30 Chip Utilization and Timing Summary for RavenHaRT synthesis of an application requiring up to 15 tasks and 31 POs

As can be seen in Figure 21, this configuration uses about 10 400 Virtex Slices, the unit of logic used in Virtex's Configurable Logic Blocks (CLB) (Virtex uses two slices per CLB). This represents about three quarters of the total capacity of the 2vp30 Chip. The total equivalent gate count for this design is 3,210,534.

RavenHaRT and the MC

We now synthesize the RavenHaRT and MC hardware that would be necessary to implement the Olympus AOCS. The MC circuit size requirements varies with two factors: the number of tasks, already determined to be 15, and the size of the ROM Table. The size of the ROM table depends on the exact application software. Then, the ROM Table for each task depends on the number of calls to RavenHaRT the task does. This number is proportional to the number of PO interactions the task has. Using the RAM Table code of the GGNS application shown in section 5.3, we approximate the number of ROM table lines needed per PO interaction. This number is 20. We use the description of the software architecture provided in [BW+93] to approximate the average number of PO interaction per task. This number is 5. We therefore assume that on average, each task of an application similar to the AOCS requires 5 x 20 lines of ROM Table code. Since the ROM Table depth must be a power of 2, we chose to instantiate the MC circuit with task ROM Tables of depth 128. The numbers of Figure 22represent the device utilization and timing summaries for a synthesis of the RavenHaRT kernel that features 16 tasks and 32 POs and a Monitoring Chip that features 15 task timing circuits and 128 lines of ROM Table coding per task circuit.



```
Design Summary:
Number of errors:      0
Number of warnings:  222
Logic Utilization:
  Total Number Slice Registers:      9,582 out of  27,392   34%
    Number used as Flip Flops:                 9,574
    Number used as Latches:                        8
  Number of 4 input LUTs:           16,300 out of  27,392   59%
Logic Distribution:
  Number of occupied Slices:        11,333 out of  13,696   82%
  Number of Slices containing only related logic:  11,333 out of  11,333  100%
  Number of Slices containing unrelated logic:          0 out of  11,333    0%
        *See NOTES below for an explanation of the effects of unrelated logic
Total Number 4 input LUTs:          17,220 out of  27,392   62%
  Number used as logic:             16,300
  Number used as a route-thru:         382
  Number used for Dual Port RAMs:      346
    (Two LUTs used per Dual Port RAM)
  Number used as Shift registers:      192

  Number of bonded IOBs:               148 out of     556   26%
    IOB Flip Flops:                    258
    IOB Dual-Data Rate Flops:           43
  Number of PPC405s:                     2 out of       2  100%
  Number of JTAGPPCs:                    1 out of       1  100%
  Number of Tbufs:                     352 out of   6,848    5%
  Number of Block RAMs:                 60 out of     136   44%
  Number of GCLKs:                      10 out of      16   62%
  Number of DCMs:                        4 out of       8   50%
  Number of GTs:                         0 out of       8    0%
  Number of GT10s:                       0 out of       0    0%

Total equivalent gate count for design:  4,207,853
Additional JTAG gate count for IOBs:  7,104
Peak Memory Usage:  275 MB
Timing summary:
---------------
Constraints cover 2217490 paths, 0 nets, and 83248 connections

Design statistics:
   Minimum period:  28.033ns (Maximum frequency:  35.672MHz)
   Maximum path delay from/to any node:   2.149ns

Device utilization summary:

   Number of External IOBs           148 out of 556    26%
      Number of LOCed External IOBs  143 out of 148    96%

   Number of PPC405s                   2 out of 2     100%
   Number of RAMB16s                  60 out of 136    44%
   Number of SLICEs                11333 out of 13696  82%

   Number of BUFGMUXs                 10 out of 16     62%
   Number of DCMs                      4 out of 8      50%
   Number of JTAGPPCs                  1 out of 1     100%
   Number of TBUFs                   352 out of 6848    5%
```

Figure 22: Xilinx 2vp30 Chip Utilization and Timing Summary for RavenHaRT and MC synthesis of an application requiring up to 15 tasks and 31 POs

As can be seen in Figure 22, this configuration uses about 11 300 Virtex Slices, the unit of logic used in Virtex's Configurable Logic Blocks (CLB) (Virtex uses two slices per CLB). This represents about three quarters of the total capacity of the 2vp30 Chip. The total equivalent gate count for this design is 4,207,853, or a 31% increase from the kernel only design.

6075-48