



In the Name of God, the Compassionate, the Merciful

Open Source Real-Time OS (RTEMS) on SCI based Compute Clusters

Salman Taherian

Senior Sophister Final Year Project

Supervisor: Michael Manzke

B.A.I. Bachelor in Engineering

&

B.A. Bachelor in Arts

University of Dublin

Trinity College

School of Engineering

2003

University of Dublin

Abstract

Open Source Real-Time OS (RTEMS) on SCI based Compute Clusters

A new class of real-time application demanding high computation and parallel processing power has begin to emerge. It is believed that previously proven successful compute clusters could once again, offer an optimum and cost-effective solution to such demand. This project investigates the implementation of a specific real-time compute cluster (RTCC). A dedicated cluster interconnect, SCI (Scalable Coherent Interface) - *IEEE Approved Standard 1596-1992*, with hardware based functionality and deterministic performance was selected as the RTCC interconnect. Implementation focuses primarily on the hardware based distributed shared memory (DSM) offered by the SCI technology as the basis of inter-communication within the RTCC; while providing a solid platform for support of future SCI functionalities on the RTCC.

RTEMS, Real-Time Executive for Multiprocessor Systems - an “open source”, “licence free”, real-time dedicated operating system, was evaluated as the target real-time operating system performing on the RTCC. This project investigates the incorporation of SCI technology in RTEMS (supporting SCI functionalities and drivers), as well as implementation of a suitable real-time cluster computing library on RTEMS for support of DSM based RTCC. Finally, this project presents a mechanism of performing real-time computing by utilising the DSM based inter-communication. Synchronisation and mutual exclusion facilities are also supported through implementation of a suitable two stage lock mechanism.

ACKNOWLEDGMENTS

First and foremost I would like to thank my family, especially my parents for all their help and support throughout this year, not forgetting the past twenty. My supervisor, Michael Manzke, merits great appreciation for his advice, support and guidance on all aspects of this project.

I would also like to thank the RTEMS developers and the people on the RTEMS mailing list, particularly Joel Sherrill - Director of Research & Development of OAR Corporation, Ralf Corsepius for and Gregory Menke for their support and guidance throughout this project. Hugo Kohmann, from Dolphin Interconnect Solutions Inc., also merits special thanks for helping me to achieve an in-depth realisation, implementation and understanding of the SCI technology.

Finally, I would like to thank Sean McEvoy for his guidance on project, time and risk management in projects, which proved beneficial and Nichola Murphy for proof-reading the project report and presenting her useful comments.

TABLE OF CONTENTS

List of Figures	iv
List of Tables	v
Glossary	vi
Chapter 1: Introduction	1
Chapter 2: Compute Clusters & Cluster Interconnects	5
2.1 Clusters	5
2.2 Scalable Coherent Interface	7
Chapter 3: Real-Time Operating Systems	15
3.1 Hard Real-Time Systems	17
3.2 Soft Real-Time Systems	18
3.3 Real-Time Kernels	18
Chapter 4: SCI Drivers	21
4.1 Drivers	22
4.2 Drivers on Linux	22
4.3 Drivers on VxWorks	25
4.4 Structure of SCI Drivers	29
4.5 Compilation, Build & Configuration	31
4.6 IRM	32
4.7 SISI	39

Chapter 5: RTEMS	42
5.1 RTEMS Rate Monotonic Scheduling (RMS)	43
5.2 Comparison of RTEMS with others	44
5.3 RTEMS Structure	50
5.4 RTEMS Build	52
5.5 RTEMS Initialisation	57
5.6 RTEMS Device Drivers	59
5.7 Comparison of Device Drivers on RTEMS, VxWorks and Linux	61
Chapter 6: Implementation	63
6.1 SCI drivers on RTEMS	64
6.2 Building drivers under RTEMS	65
6.3 SCI Initialisation within BSP	67
6.4 SCI driver development on RTEMS	71
6.5 Application layer	79
6.6 Debugging	83
6.7 SISI layer	84
Chapter 7: Evaluation & Conclusion	87
7.1 Implementation	87
7.2 Project	88
7.3 Future work	90
Bibliography	92
Appendix A: RTCC Package API	96
A.1 SCI Initialisation Library (<code>sci_init.h</code>)	96
A.2 SCI Driver Interface (<code>sci_genif.h</code>)	97
A.3 Real-Time Cluster Computing Library (<code>librtcc.h</code>)	97

Appendix B: Discovered GCC bug	99
B.1 Symptom	99
B.2 Cause	99
B.3 Workaround	100
B.4 Resolution	100
B.5 Status	100
 Appendix C: Concepts & Tools	 101
C.1 Technical Concepts learnt	101
C.2 Tools and Software utilised	101

LIST OF FIGURES

2.1	Distributed Shared Memory diagram	10
2.2	PCI-SCI card overview	12
2.3	SCI Link Controller chip overview	13
4.1	Linux Module-Kernel Inter-communications Diagram	24
4.2	IRM Internal Structure Diagram	30
4.3	IRM Source Code Overview	33
4.4	SISCI Source Code Overview	40
5.1	RTEMS build diagram	53
5.2	RTEMS Makefiles	55
6.1	SCI Shared Memory Segment	80

LIST OF TABLES

5.1	RTOS's Features List	46
5.2	RTOS's Latency comparisons	47

GLOSSARY

API Application Programming Interface, hides low programming detail and provides a unique interface for higher level applications.

ATC Address Translation Cache, internal SCI cache for mapping PCI and SCI addresses.

ATT Address Translation Table, used on SCI card to translate PCI addresses to SCI addresses and vice versa.

BSP Board Support Package, all portions of RTEMS which are board specific.

BWCE Burst Write Combining Enable, an advanced PCI bus feature on HB_450NX_PXB host-bridges.

Context Switching Delay The time delay from the execution of the last instruction of a task, to execution of the first instruction of another task, which includes the time scheduler determines which task to run, time to save the context of first task and time to restore the context of the second task.

DSM Distributed Shared Memory, memory segment accessible by all cluster nodes at hardware level.

FAA Federal Aviation Administration.

HAC High Availability Computing

HPC High Performance Computing

HTC High Throughput Computing

Interrupt Latency The time elapsed from the moment of occurrence of an event (e.g. a hardware interrupt) until execution of the first instruction of the Interrupt Service Routine (ISR), which includes the overhead required by the executive at the beginning of each ISR plus the time required for the CPU to vector the interrupt.

IRM Interconnect Resource Manager, main section of SCI drivers which directly interacts with SCI hardware.

IRQ Interrupt ReQuest, the number of interrupt levels available within a system.

ISR Interrupt Service Routine, also known as Interrupt handler.

IT Information Technology.

LC Link Controller, interfaces the SCI card to the global SCI network.

LIBRTCC Real-time cluster computing library, supports real-time computing on a RTCC.

MMU Memory Management Unit.

OS Operating System, controls and provides a unique hardware independent interface for user applications to access hardware.

PIO Programmed I/O, capability of executing load and store instruction on remote cluster nodes.

PSB PCI-SCI Bridge, interfaces the PCI bus to the SCI card.

RMS Rate Monotonic Scheduling, special scheduling algorithm utilised in RTEMS to schedule periodic real-time tasks.

ROM Read-Only Memory.

RTC Real Time Clock, utilities in sensitive embedded applications to perform execution on high accurate basis.

RTCC Real-Time Compute Cluster.

RTEMS Real-Time Executive for Multiprocessor Systems, the RTOS used within this project.

RTOS Real-Time Operating System.

SCI Scalable Coherent Interface, IEEE standard interface technology, dedicated as cluster interconnects.

SMP Symmetric Multi-Processor, several processors on a single bus, forming a supercomputer.

VC Virtual Channel, forms the second layer of the SCI inter-communications protocol.

Chapter 1

INTRODUCTION

Not long ago, substantial computing power was only available within supercomputers. These high-performance systems have always been very expensive due to high design costs and the relatively small market for them. Today however, powerful computer clusters can be built for a fraction of the cost of traditional supercomputers by combining inexpensive, mass-produced PCs with dedicated cluster interconnects.

Clusters are known to have two significant characteristics: (i) They are a cost-effective alternative to large scale parallel systems and (ii) They are scalable. The most vital component in a cluster is the *interconnect*, which connects separate computer nodes to form a “unified computing resource” [1]. Communication latency, throughput and scalability are important parameters when building a cluster of interconnected computers. These parameters are all governed by the cluster interconnect, which is utilised in the formation of the cluster. The importance of interconnects in a cluster was recognised, hence Dolphin PCI-SCI (D310 model) interconnect cards were adapted for this project. This interconnect, referred to as SCI (Scalable Coherent Interface), is an implementation of the *IEEE, SCI: Scalable Coherent Interface, Approved Standard 1596-1992*. SCI is a dedicated cluster interconnect, implemented specifically for the task of constructing clusters from a series of standalone computers [29].

Parallel computing may be divided into two classes: Moderate Parallel systems based on shared memory (typical SMP machines) and Highly Parallel machines based on message passing. SCI offers low latency message passing through non-cached shared memory. Remote memory read, write, lock, interrupt and DMA operations are available. Hence, SCI cards for a standard PCI bus offer both low

latency message passing and distributed shared memory with a large number of processing nodes for a moderate price. Even though sources of performance loss within clusters are the the slow PCI bus and the SCI link latency, advancements within the SCI technology have shifted the focus of the performance loss purely on to the PCI bus.

Another class of systems, which are fast emerging into the IT¹ industry, are the *Real-Time* systems. Real-time systems are classified as systems which possess timing constraints on their execution. These constraints require the system to operate in a predictable and deterministic fashion. Real-time systems are widely used within embedded applications, where systems are task or operation specific and in most cases required to be fully reliable. Growth of computer system applications in all areas, together with the merging of old multi-media systems and modern computer systems, has increased the demand in real-time systems more than any other previous time.

Modern real-time systems increasingly demand higher computational power. This may be clearly seen within the developing network and communication technologies (3G, 4G, etc), which are required to support a tremendously large amount of network load on a real-time basis. While real-time based supercomputers may be utilised under such circumstances, this project investigates provision of a cost-effective solution, RTCC (Real-Time Compute Cluster). There have already been a number of attempts in implementation of the RTCC, but these have either been based on proprietary systems or they lack the high-performance required.

The RTOS (Real-Time Operating System) chosen for this project is an open source, real-time and embedded dedicated operating system named RTEMS. RTEMS, Real-Time Executive for Multiprocessor Systems, is a “licence free” operating system with an extensive set of features and a performance comparable with the most successful real-time operating systems in the industry (e.g. VxWorks). This project evaluates RTEMS on an SCI based compute cluster in order to achieve an inexpensive high-performance RTCC. The following points are considered noteworthy when

¹Information Technology

analysing the choice of elements within this project.

- RTEMS is a high performance RTOS, and SCI a dedicated cluster interconnect with a sufficiently high performance, they are hoped to contribute towards a high-performance RTCC.
- RTEMS is an “open source” and “licence free” system and thus not only significantly reduces the cost when compared to expensive proprietary systems, but it also offers future research and development possibilities in this area.
- Inexpensive D310 model PCI-SCI cards, which offer the required distributed shared memory capability were adapted for this project. They also represent a hardware based, deterministic behaviour, most suitable for real-time systems.

This project will initially aim at incorporating SCI compatibility into the RTEMS system. Analysis of SCI hardware and specifically SCI drivers, as well as RTOS's and specifically RTEMS system, were of extreme importance in this section. It was necessary to implement full functionality of the SCI hardware on RTEMS, without introducing any disadvantages or performance losses into the overall system. SCI technology has particularly close ties with the memory management of systems, which need extra attention. In addition, SCI drivers and operations must comply with requirement and standards of a real-time system. Efforts were made to implement SCI functionality on RTEMS most efficiently and in-line with other sections of the system. The second section of this project is dedicated to the implementation of the hardware based DSM (Distributed Shared Memory) within a cluster. Following this section a full RTCC is implemented with DSM as a basis for cluster inter-communications. In the final stage of this project, a two stage lock mechanism is developed to support inter-process synchronisation within a cluster, along with a simple demonstration program illustrating the result of project on a two node computer cluster.

This report is composed of seven chapters and three appendices. The following two chapters provide background information regarding clusters, SCI and RTOS

in general. Chapters four and five examine in detail, the SCI driver and RTEMS, respectively. Lack of documentation in many cases, particularly the SCI drivers, encourages us to present a detailed view of overall elements examined. It is further hoped that this project report will serve as an implementation guide and/or documentation for systems examined throughout this project. Chapter six details the implementation of the RTCC and analysis the implementation from various perspectives. Finally, chapter seven presents results of the implementation along with conclusions and an overall evaluation of this project.

Chapter 2

COMPUTE CLUSTERS & CLUSTER INTERCONNECTS

The wide spread use of digital technology and particularly computers within our daily lives has augmented the need for higher computation power within systems. In response to this demand hardware manufacturers are producing newer and faster hardware (and more specifically processors), on monthly basis. It is the high demand for computation power which contributes towards the rapid advancement of the computer technology. Unfortunately, there is a cost associated with this rapid advancement. As the technology becomes more advanced and complex, the cost of production and the related cost of purchase elevates.

Continuously maintaining supercomputers and mainframes to the highest level of technology is extremely costly and in most cases, uneconomical. The concept of a compute cluster offers construction of cheap supercomputers with the option of performance scaling, which is independent of technology development, and with minimum additional costs. Mentioned factors have resulted in rapid deployment of compute clusters within various industrial sectors, and in particular the replacement of high performance and large scale supercomputers with the more promising compute clusters. This chapter will provide a brief introduction to the significance of compute clusters, followed by a detailed examination of the interconnects employed within clusters. This project specifically employs SCI (Scalable Coherent Interface) interconnects manufactured by Dolphin. Hence the second section of this chapter will be focus entirely on this hardware and technology.

2.1 *Clusters*

A cluster is a collection of interconnected whole computers used as a single unified computer. Traditionally, parts of computer systems and mainframes had to be

replaced with new hardware if higher computation or processing power was desired. Due to compatibility issues, this process, in most cases, resulted in the replacement of multiple parts rather than the replacement of one desired section. The process was highly costly both in terms of money and time. Another disadvantage with this model was the poor residual value of the computer equipment. A system replacement often resulted in the invested capital being lost when the old system was replaced with a newer model.

In late 1993 Donald Becker and Thomas Sterling began sketching the outline of a commodity-based cluster system designed as a cost-effective alternative to large supercomputers [4]. The initial cluster computer consisted of 16 Intel 486 DX4 processors @ 100MHz, 16 MBytes RAM and 10 Mbit Ethernet Interconnects. The machine was an instant success. It was considered as an optimum solution without the many problems and disadvantages associated with the large mainframe and supercomputers.

Clusters can be built as result of interconnecting basic off-the-shelf computers, referred to as nodes. The main characteristics of a cluster are as follows:

- It consists of many of the same or similar type machines (heterogenous clusters are a subtype, still mostly experimental).
- It is tightly-coupled and uses dedicated interconnects.
- All machines share pre-specified resources such as a common memory segment.
- Initial software is required to setup the system for cluster computing.

The main advantages of clusters are that they are inexpensive and scalable. They are superior to supercomputers in terms of cost. Clusters need not be composed of the latest technology hardware available, and rarely need parts replacement. Their second advantageous factor, is evident scalability, when higher processing power is desired. Their performance may be improved by the addition of extra nodes rather than the replacement of parts. Scalability itself, however, is highly dependent on

the interconnect technology employed within the compute cluster [36]. The bandwidth and latency of the interconnect can determine the scalability of the hardware. Cluster architectures are physically more scalable than SMP¹ architectures. This is due to the increased aggregate interconnect bandwidth resulting from the addition of processors to clusters. In an SMP, the interconnect bandwidth remains constant.

The three main cluster application areas are as follows:

High Performance Computing (HPC) Executes programs with parallel algorithms.

High Throughput Computing (HTC) Used in parametric studies (same program executed many times with different parameters).

High Availability Computing (HAC) Provides fail-over redundancy.

Finally, cluster topology is a factor which users may use to their advantage. Clusters dependent on the employed interconnect can form various topologies, the most common of which is the *ring* formation. Although interconnects are considered one of the sources of performance loss within clusters, tuning of applications to a specific cluster topology may significantly minimise this loss of performance.

2.2 Scalable Coherent Interface

A key decision that will greatly affect the overall performance of a compute cluster is the method used to connect the nodes together. Performance and scalability of traditional systems were limited by the scalability of the processor bus. Scalable Coherent Interface (SCI) was introduced to enable the extension of systems beyond the scalability limit imposed by the processor bus [12]. The scalable coherent interface (SCI) provides computer-bus-like services [26]. Unlike a bus, however, it uses a collection of fast point-to-point unidirectional links to provide the far higher

¹Symmetric Multi-Processor

throughput necessary for high-performance multiprocessor systems. SCI is a dedicated cluster interconnect, implemented according to the *IEEE Scalable Coherent Interface (SCI) standard 1596-1992*.

The most significant service offered by the SCI is the provision of a single physical 64-bit address space across SCI nodes and the related transactions for reading, writing, and locking memory locations in this hardware based distributed shared memory (DSM). SCI supports distributed shared memory with optional cache coherence for tightly coupled systems and message-passing for loosely coupled systems[19]. The employment of the unidirectional point-to-point links eliminates the dependency of bus length and bus speed on the size and number of processors on SCI topology. Point-to-point unidirectionality of the network also ensures lack of congestion within a simple ring based SCI network [21]. Distributed shared memory provided at hardware level within a compute cluster results in a low latency, high performance and deterministic behaviour of the technology.

“The Scalable Coherent Interface (Local Area MultiProcessor) is effectively a combination computer backplane bus, processor memory bus, I/O bus, high performance switch, packet switch, ring, mesh, local area network, optical network, parallel bus, serial bus, information sharing and information communication system that provides distributed directory based cache coherency for a global shared memory model and uses electrical or fiber optic point-to-point unidirectional cables of various widths.” [5]

The scalable coherent interface as an “open” distributed bus also provides low-latency interconnections with full reliable communications for clusters [21]. High throughput, low latency and low CPU overhead are major factors which classify SCI as a high performance technology. Bandwidth and latency of SCI are sufficient in order *not* to impose any limitation on the performance and scalability of clusters. Meanwhile, the reliability of communication within SCI network, is also guaranteed through node-to-node request/response/acknowledge protocols, as well as the facilitation of split transactions (independency of the request and response signals) to prevent deadlocks. Address based communications, rather than stream based

methodology, result in the increased efficiency of SCI. Additionally, SCI protocols do not guarantee in-order delivery of transactions, hence the user must support this feature, through manual application, if desired.

One must appreciate that SCI is designed to serve both, message-based and shared-memory programming models [9]. Another feature of the SCI hardware, which will not be discussed in detail, is the ability to generate hardware interrupts on the remote nodes. This is performed by writing into the special status register address (mailbox) of the target node. Once the interrupt is triggered on the remote node, the SCI interrupt handler on the target node will be invoked and may perform the desired task or operation.

Exploiting SCI's flexibility in terms of its efficiency in supporting both parallel programming models - message passing and shared memory - leads to investigations targeting a distributed shared memory multiprocessor system similar to Stanford's FLASH [23], or MIT's Alewife [7] machines. The following subsection details how distributed shared memory (DSM) is achieved within a cluster through the use of SCI interconnects.

2.2.1 Shared Memory using SCI

SCI, with its origin as a distributed multiprocessor bus, provides possibilities to directly access remote memory by ordinary load and store operations [37]. Since remote accesses are going through the I/O bus, these remote load/stores are often referred to as programmed I/O (PIO).

Figure 2.1 illustrates a typical architecture of an SCI based cluster. SCI technology has also accounted for caching of remote memory and thus target higher performances. Hence the resultant memory hierarchy is: (1) CPU registers, (2) CPU cache, (3) local memory or SCI cache, (4) remote memory [35]. Unfortunately this is not possible on PCI bus based systems, since transactions on the processor bus of a system are not visible by the (slower) PCI bus. Nowadays, PCI-SCI cards are used as cluster interconnects. PCI technology, in contrast to the motherboard technology, offers a unique interface for SCI cards, accounting for its employment in

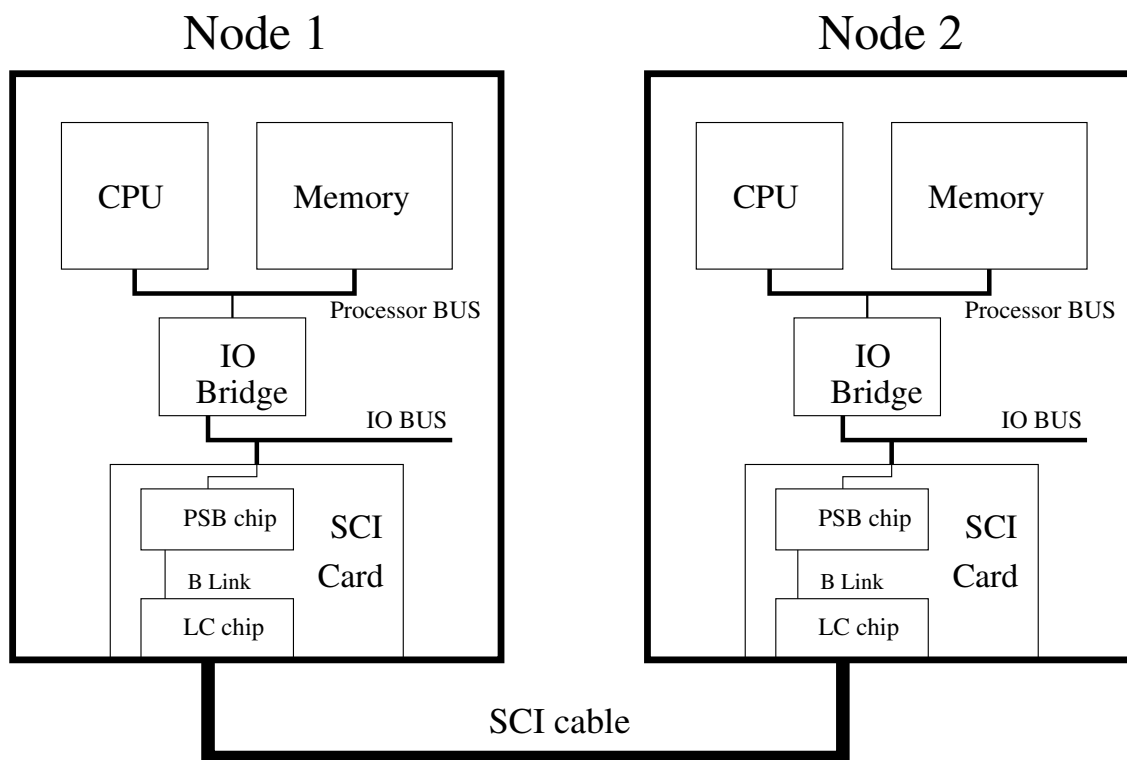


Figure 2.1: Distributed Shared Memory diagram

a much wider range of systems.

As noted earlier, SCI protocols do not guarantee in-order delivery of transactions. This is a result of the transaction buffering capability on SCI cards. The bandwidth of the SCI network is greater than the normal PCI busses, hence it is not uncommon for the SCI transactions to be delayed as result of the delay within the PCI bus. In order to eliminate this issue SCI makes use of two concepts. The first is the split transaction mechanism, which also eliminates the possibility of dead-locks within the SCI network, while the second is the transaction buffering on the SCI card. Multiple stores to adjacent locations in remote memory can be gathered (by hardware) into internal buffers on the local PSB chip [37]. When a buffer is full, the PSB will transmit the contents of the buffer as a single, fully loaded SCI packet. Thus the overhead of SCI packet generation is amortised over many store operations. This technique is referred to as *streaming*. A *read* or *store barrier* operation may be used to flush all pending operations on the SCI network.

Each node can create shared memory segments in its I/O address space and export them into the SCI network. Other nodes import these DSM (Distributed Shared Memory) segments into their I/O space. A process may further map DSM segments into its virtual address space and from that point on, use standard load and store instructions to access shared, potentially remote memory. A processor can also send a lock operation (atomic read-write operation) to another node. SCI transactions are atomic and guaranteed to be delivered to the destination node.

The memory segments are pinned down, meaning that they are non-swappable. They are assigned a unique id, which remote nodes use to import the segment. The remote node maps the exported memory segment (available through the SCI network) into its I/O address space, and creates an entry on its Address Translation Table (ATT). Additional options and configurations (known as segment flags and attributes) are also available to the user, which provide a series of services (such as security and access restrictions to the segment). Though potentially useful under some circumstances, they are undesirable in the context of this project, since they may introduce software interventions into the end system.

All remote operations are carried out by addressing part of the local address space, which is allocated for the SCI card on the PCI bus. Access to the local virtual memory is first mapped to a physical I/O address through the Memory Management Unit (MMU). The 32 bit PCI address is hence mapped into a 64 bit SCI address using the Address Translation Table (ATT) located on the board. The most significant 16 bits indicate the target node id, and hence also impose the restriction of maximum 64k nodes on any single SCI network. A portion of the PCI address is used as an offset to the remote memory segment and remains untouched. In most cases this is the 18 least significant bits. The remaining portion of the PCI address signifies the segment id. At the destination node, the least significant 32 bits of the SCI address are the physical I/O address on the target system.

Figure 2.2 shows the basic overview of a PCI-SCI card. The B-Link operates as a backbone link between the PSB (PCI-SCI Bridge) and the LC (Link Controller) chips. The PSB (PCI-SCI Bridge) chip translates I/O bus transactions into SCI

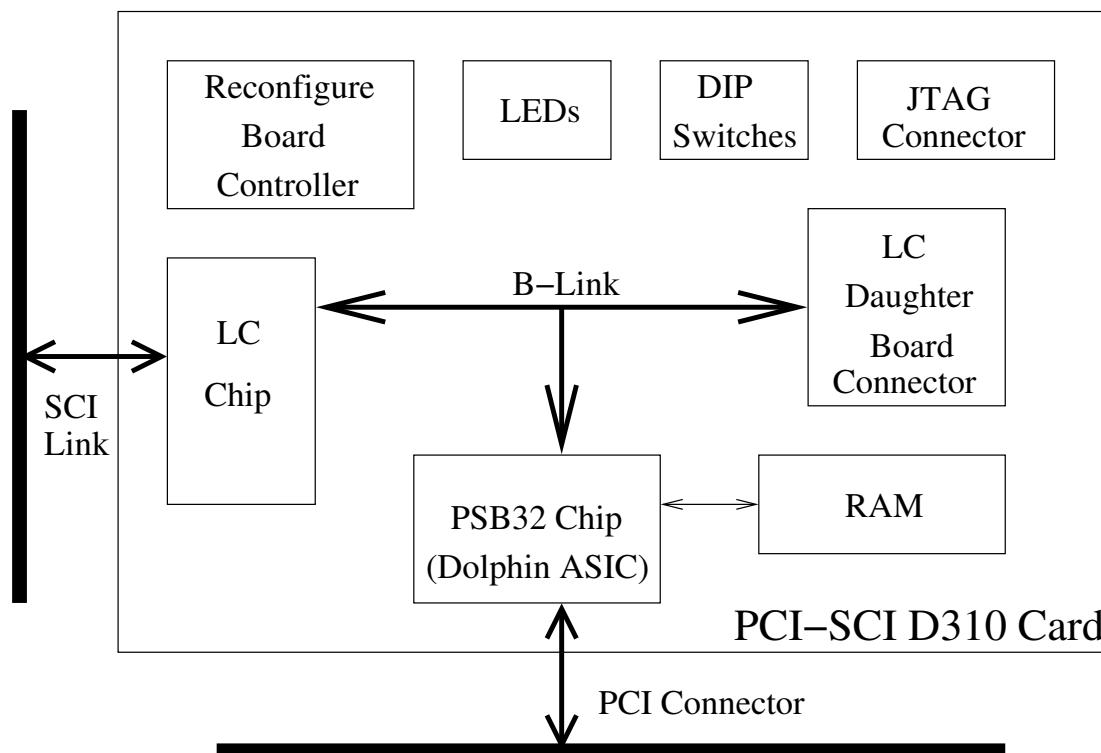


Figure 2.2: PCI-SCI card overview

transactions (and vice versa), and this is the SCI interface to the local machine (PCI bus). The on-board ATT (Address Translation Table) stored within the RAM chip, is used to convert local I/O addresses into SCI addresses (and vice versa). The LC chip is the interface of the card to the external nodes. It manages data transfer on the SCI physical layer, as well as performing routing and delivery guarantees.

The board controller controls reset and initialisation of the board, as well as board status reporting and LED management. DIP switches are used to set various parameters on the card, such as the SCI link frequency, B-Link frequency and SCI window size. The daughter board connector supports an additional card for high fault tolerance and routing ability.

The PSB chip is a combination of the following components [15].

PCI Interface Interfaces the PCI bus and contains a DMA controller for high performance memory-to-memory transfer.

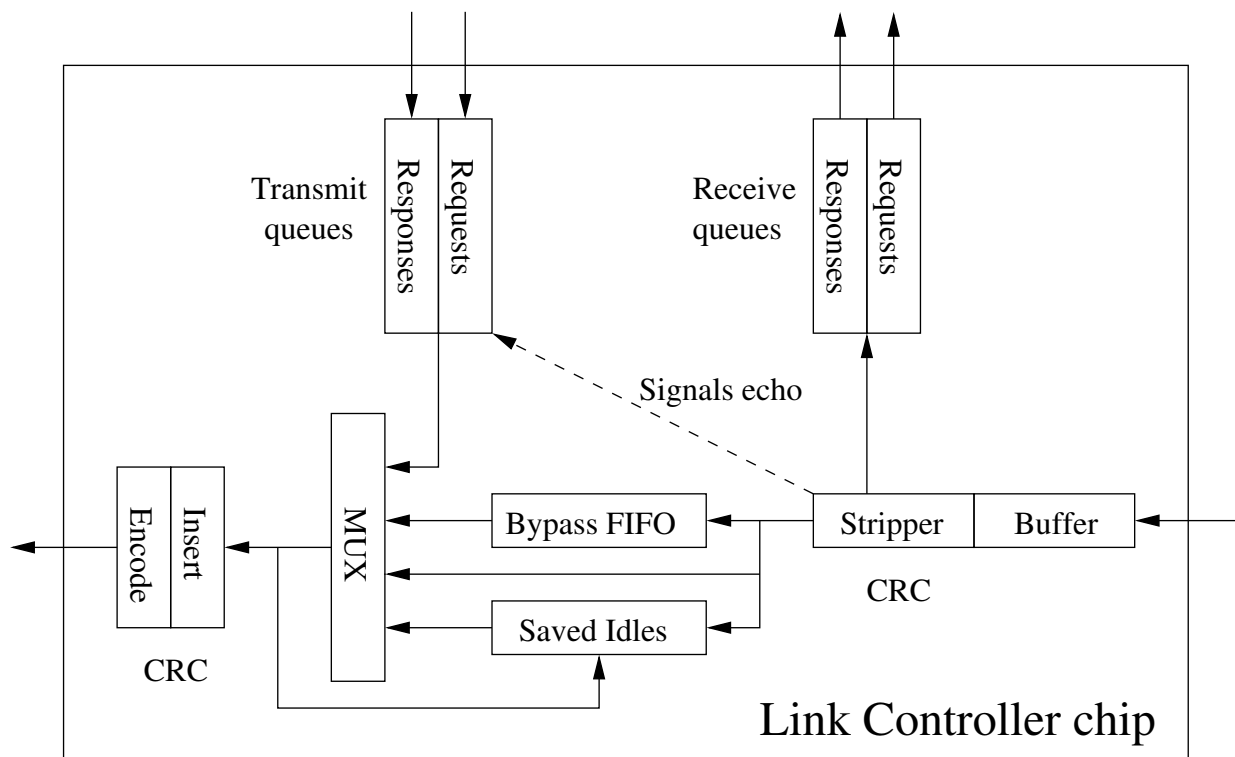


Figure 2.3: SCI Link Controller chip overview

Read and Write Buffers Read and write operation streams (16 streams with 128 bytes buffer each).

Address Translation Cache (ATC) Internal cache for mapping of 32 bit PCI addresses to 64 SCI addresses.

The PSB chip is formed by the top main components as well as a number of status and control registers for error logging, initialisation and status information.

The LC chip overview is illustrated in figure 2.3. Having already reviewed its role, the components present within the chip are briefly listed [15].

Receive buffer Input buffer composed of two queues, requests and responses. Packets are sent to B-Link from this buffer.

Transmit buffer Transmit buffer, again composed of two queues, requests and responses. Incoming packets from B-Link are placed within this buffer. In

some cases an “active buffer” is also present for retransmission in case of an error. Packets are discarded on receipt of a correct echo.

Bypass FIFO Stripped incoming packets, if not destined for this node, are placed within this queue for transmission to the next node.

Saved Idles These are SCI packets, inserted in between SCI packets for permanent synchronisation of the chip to the incoming data.

Chapter 3

REAL-TIME OPERATING SYSTEMS

Real-time operating systems are an important class of operating systems. An operating system is considered to be “real-time” if it possesses a strict set of policies and rules regarding its execution, the most important factor being the timing constraints [18]. Kernels designed for such operating systems are required to be extremely efficient, highly deterministic and to perform execution within a specified timing constraint.

If we model an operating system as a simple process, which takes a set of inputs and outputs a set of results after internal computations, then a real-time operating system is one which, is expected not only to produce a correct set of results, but also to *output results within a finite timing duration from the moment of presentation of inputs to the system.*

The strict timing issue of an operating system further implies that the operating system must perform tasks on a deterministic and highly efficient basis. The following is a fundamental description of a real-time operating system from the early 1980s.

“[An environment] characterised by processing activity triggered by randomly accepted external events. The processing activity for a particular event is accomplished by sequence of processing tasks, each of which must complete within rigid time constraints. ...Characteristically, the computer system is completely dedicated to the control application and has been configured to guarantee on-time responses even at peak loads. The environment is such that utilisation of equipment is less important than responsiveness to the environment.”[18]

Often a very strict hierarchy of commands is built-in to the real-time operating

systems. This is to allow for a certain level of flexibility for programmers while driving the fastest response possible from the real-time hardware. In recent years a set of rules and policies have been outlined which must be met by developers to achieve a real-time operating system. Some of these rules & policies are outlined below [22] [24]:

- A RTOS¹ should be a multitasking design in order to maximise the CPU's efficiency.
- The kernel should be driven in response to internal and external system events.
- The kernel should support a number of independent or interrelated tasks, each having its own priority associated with its scheduling importance.
- The kernel's performance should be highly deterministic.
- The kernel should be designed so as to impose minimal overhead to the application tasks and should have small RAM requirements.
- Common rules of task scheduling apply (higher priority task preempt lower priority tasks and the Null task always has the lowest priority).
- The kernel must be interruptible but not re-entrant.
- An interrupt service routine (ISR) must not issue kernel system calls except to signal another event or to terminate itself.

Real-time operating systems are rarely implemented fully in software. Hardware in most cases is custom designed with the intention of providing real-time capabilities. Hence, low level debugging tools such as simple background debuggers, logic analysers and emulators are used to address flaws within the system.

¹Real-Time Operating System

Applications of real-time systems are growing and will continue to do so. Many applications require real-time systems to deliver their objectives with timing constraints, from the simplest systems such as a watch to the most sophisticated systems like NASA space shuttles. They are classified into two types, based on their timing constraints. These types are called “hard real-time” systems and “soft real-time” systems. Each type will be addressed individually, and some general common issues and services involved within the kernels of such systems will also be discussed.

3.1 *Hard Real-Time Systems*

Hard real-time systems guarantee completion of a critical task within the specified amount of time. To achieve this, all delays within the system must be bounded and all operations must take a deterministic amount of time to complete. The scheduler plays an important role in hard real-time systems. Generally a process/task is submitted to the scheduler along with the time necessary to complete its operation. The scheduler will then either accept the process/task or reject it.

The scheduler will accept the task if it can guarantee the completion of the job within the specified amount of time. It will reject the task if it considers it as impossible to complete within the provided time limit. For the scheduler to be able to make such a judgement accurately, it must know the exact execution time length of each operation within the system.

Hard real-time systems are hence restricted as to what devices or operations they can support. Secondary storage devices and advanced operating system features are two major areas which are mostly absent in hard real-time systems due to the uncertainty involved in their response.

Examples of where hard real-time systems are used are NASA space shuttle systems, FAA (Federal Aviation Administration) systems, critical medical equipments, weapon & military systems and some automobile-engine fuel-injection systems. They all require objectives to be met within a specific time-line and failure to do so will be regarded as complete failure of the system.

3.2 *Soft Real-Time Systems*

Soft real-time systems are less restrictive than hard real-time systems. Tasks and objectives within such systems are allowed to be completed within a bounded period. The scheduler in such systems must be priority based. It must assign the highest possible priority to the real-time jobs, and ensure that their priority does not degrade over time. Non-real-time jobs are allocated variable priorities, and their priorities are allowed to vary throughout execution time.

The scheduler must at first level satisfy the real-time job requirements, and at second level handle non-real-time jobs. Although the scheduler is allowed to act more freely regarding non-real-time jobs, it should not allow long delays, starvation or unfair resource allocation among such jobs. This in most cases makes the design of the scheduler an extreme challenge, because the scheduler is expected to schedule jobs such that it completes all tasks with the minimum overall delay possible. In particular, dispatch latencies are minimised as much as possible (using efficient designs) so as to allow the execution of real-time as soon as they are executable.

Applications of such systems are extremely broad since unlike hard real-time systems, they can support various devices and operating system features. Thus they are used in many multi-media, graphics, virtual reality and advanced scientific systems and will continue to grow rapidly over the coming years.

3.3 *Real-Time Kernels*

Real-time operating system kernels as mentioned are very limited on the operations that they may perform. It is aimed that all supported operations within a kernel be deterministic and efficient. This has resulted in most RTOS's in today's market offering a similar set of services, which will be outlined here.

Kernels must manage system resources efficiently. The main system resources are the CPU, memory and time. The CPU is shared to increase efficiency and execute processes faster. Memory management is an issue because it is a finite resource. Time, as mentioned earlier, is the most important factor that every real-time system

should manage in order to deliver its objectives.

Considering that the use of real-time systems is mainly within embedded devices, which might impose size, power consumption, memory and other restrictions, having the kernel as small as possible is desirable [41]. The user may add further tasks if required, but these are at the expense of cost of a larger system, possibly slower performance and the variation of other parameters.

Schedulers, as discussed, are the most significant part of the kernel. There are three main types of schedulers which most real-time kernels use [22]:

- Round robin scheduling
- Time sliced scheduling
- Preemptive scheduling

Preemptive scheduling is the most popular, supporting the other two scheduling methods. It uses priorities along with event driven operation.

Services among kernels are more variable. In advanced kernels, the user must specifically request certain services at compile time if he/she wishes to use them within the application. Otherwise the kernel would not include such services, resulting in a reduced size and faster operation of the kernel. Some services, however, are critical and are present within all the systems. A short list of common kernel services are outlined below [22].

- Static and dynamic task services
- Queues and lists
- Semaphores
- Mailboxes
- Synchronous and asynchronous transmission
- Timers

- Memory management

Finally we present a list of properties which industrial users commonly use to compare and evaluate RTOS's for their applications [28].

- The interrupt latency (i.e. time from interrupt to task run).
- Maximum time period for execution of every single system call.
- The maximum time the OS² and drivers mask interrupts.
- System interrupt levels.
- Device driver IRQ³ levels.

The above parameters are fixed regardless of the application program. Users possess the knowledge that systems operate in a deterministic fashion and estimate whether their hard real-time jobs can complete on such systems or not. They choose the hardware together with the RTOS such that it meets their objectives with minimal cost.

²Operating System

³Interrupt ReQuest

Chapter 4

SCI DRIVERS

Device drivers are the most essential part of resource management within a system. In this context, resource refers to a hardware device which the user uses to obtain its objectives. SCI drivers play an important role in allowing systems to make correct, efficient and organised use of SCI hardware devices.

SCI provides high speed communication (an interconnection) between cluster nodes. It is “the” element, which allows independent, separated nodes to engage in cluster computing and processing of collaborative tasks. Hence examination of SCI drivers is extremely important in order to achieve a cluster with a performance comparable to those of supercomputers.

SCI drivers, as a portion of the real-time system, must obey all restrictions imposed on the system. Therefore, the real-time issues must also be examined and satisfied within the device drivers.

We shall start this chapter with a brief look at device drivers in general. Having identified the roles and objectives of a device driver, we will discuss the operation and implementation of device drivers on two of the most important operating systems (on which SCI drivers have already been implemented). The next section examines the general structure and arrangement of SCI source code drivers. Compilation, building, configuration and loading of SCI drivers are outlined in the following section. Finally, two major parts of the SCI drivers are discussed individually with a detailed analysis of their tasks, roles, initialisation sequences, resource requirements and inter-communications.

4.1 Drivers

A driver supplies a uniform, device-independent, logical interface which allows the user to interact with a device. They contain detailed knowledge of hardware devices and ensure correct and proper use of the device by the system and its users.

While they provide the mechanism of accessing hardware functionalities, they should not provide any policy regarding the use of such services. In other words, the drivers simplify the use of hardware functions, but may not impose any additional restrictions for doing so.

Device drivers are particularly important in providing compatibility between systems and a range of developed hardware devices. Along with the drift of technology, hardware becomes more advanced and offers new features and functionalities. Drivers must ensure that a *unified* interface is provided regardless of the version/model of a device. It is the drivers' responsibility to provide various implementations of their interface in order to support all models of a hardware device.

Device driver implementations vary among operating systems. Each operating system kernel has its own set of policies regarding loading, initialisation, configuration and access to resources within the system. This enforces various implementations of device drivers for different operating systems. We will examine policies and rules imposed by Linux and VxWorks operating systems, and later demonstrate how this is achieved within SCI source code drivers.

4.2 Drivers on Linux

Device drivers on Linux operating system are classified into two types based on their position in the system. The first class are the built-in device drivers. This set is compiled and linked into the kernel at the kernel build stage. Built-in drivers have the disadvantages of increasing kernel size and introducing additional overhead to the system. However, they provide better reliability, performance, security and speed due to being part of the system core (the kernel). Drivers are built-in to the kernel if (i) They are in constant use, (ii) A high operational speed is desired or (iii)

If security is a significant issue [8].

Built-in drivers initialise as part of the kernel initialisation process at system startup [13]. They do not generally terminate until the system shuts down, but privileged users may terminate some built-in drivers partially at run-time.

The second class of device drivers are known as *modules*. These are much more popular and more common within the system. Built-in drivers, as mentioned, are only reasonable if used on a very frequent basis. However, this is rarely the case and thus modules were introduced. Modules are, in a general sense, a set of kernel functionalities which are loaded into the system whenever an application or user desires their functionalities.

They have three distinct features which separate them from any other normal program. Once these three basic rules are met, any program can fall into the module category.

1. It must provide an initialisation routine (`init_module` function).
2. It must provide a cleanup routine (`cleanup_module` function).
3. It must either define, or be compiled with a relevant set of definition flags, indicating that it is a module, and it should be executed within the kernel space.

The `init_module`¹ function is the first function which is called when a module is loaded into the Linux kernel. It is responsible for allocating the necessary resources, tables and registration of its symbols into the kernel public symbol table. The kernel symbol table can be read in text form from the file `/proc/ksyms`. The table holds a list of functions which are supported by the kernel. In fact in most cases, the `init_module` makes use of such functions to allocate resources to the driver. The `init_module` calls the kernel `register_capability` function to register its capabilities. The kernel inserts the module capabilities within the system `capabilities[]` array, which indirectly references the start of function routines.

¹the user may specify an alternative initialisation routine using the `module_init()` function at the start of module code

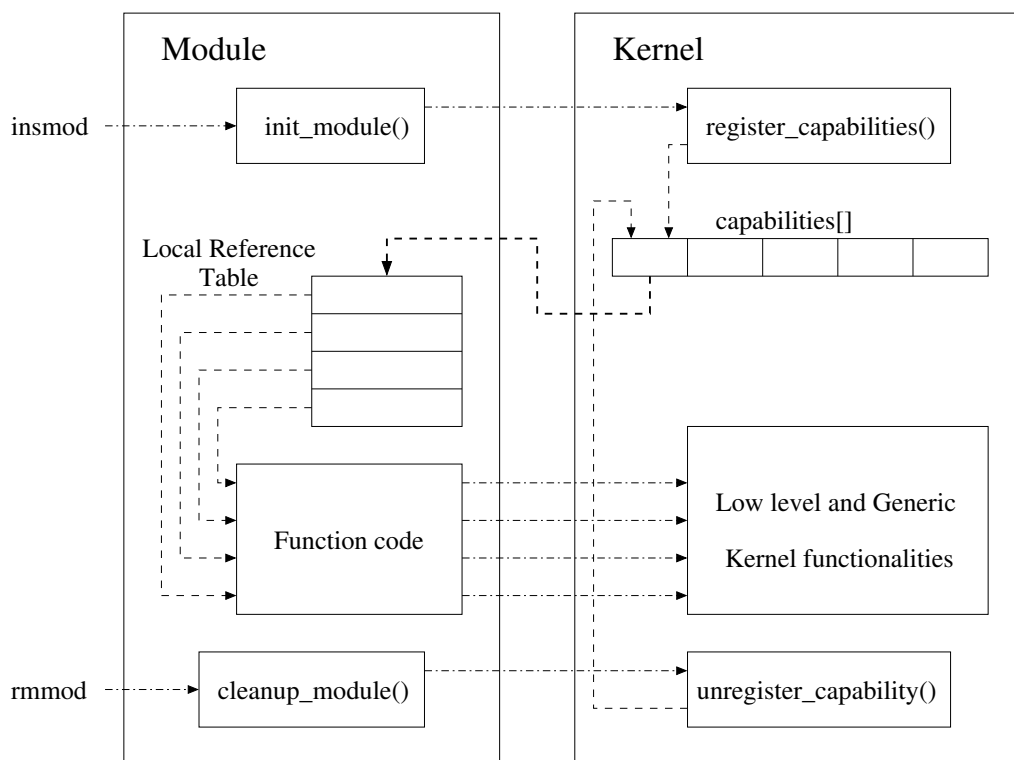


Figure 4.1: Linux Module-Kernel Inter-communications Diagram

Conversely the `cleanup_module`² function unregisters the capabilities from the kernel `capabilities[]` array and releases all held resources.

Figure 4.1 illustrates the basic relationship between kernel and module as explained. The `insmod` and `rmmod` functions may be used by privileged users to enforce module loading and unloading. A list of loaded modules can always be examined through `/proc/modules`. I/O region allocations can also be examined through `/proc/iomem` and `/proc/ioports` files.

Linux drivers are further classified into three types, and since they are mostly implemented as modules, they are termed *char modules*, *block modules* and *network modules*.

Character devices Similar to normal files, these devices are accessed as a stream of bytes. The driver implements basic `open`, `close`, `read` and `write` system

²the user may specify an alternative cleanup routine using the `module_exit()` function at the start of module code

calls. This data channel may only be accessed sequentially.

Block devices These devices are very similar to character devices. Data is usually accessed (read/write) in multiple blocks³. The only difference compared to character devices is the way in which data is handled internally by kernel.

Network interfaces These are hardware or software ⁴ devices which exchange data with other hosts. They transmit and receive data packets. The kernel also makes packet transmission based calls rather than reads and writes.

Every hardware device within the Linux system has a *major* and *minor* number associated with it. The major number indicates the class of the hardware, whereas the minor number is an index into the device ⁵. Drivers initially find their devices on the system using the major number. Having found the device on the system⁶, they can interact with the target device through kernel system calls. Often for optimisation and efficiency reasons, such system calls are defined as macros which reference lower level codings⁷.

4.3 Drivers on VxWorks

VxWorks is a commercial real-time operating system. It is widely used in industrial systems and has substantial support for a wide range of devices. However it is considered somewhat unsuitable for research and academic environments. In response to the “*What is a good RTOS?*” question, developers and users on the Comp.realtime newsgroup jointly agree on the following answer, which is presented in the FAQ section of the news group.

“A good RTOS is not only a good Kernel! A good RTOS should have a good documentation, should be delivered with good tools to develop and

³each block is usually one kilobyte

⁴such as loopback interface

⁵often zero, unless several devices of same type are present (e.g. multiple network cards)

⁶this generally implies knowing the bus and pci device numbers

⁷in most cases, Assembly language code

tune your application. So even if some figures like the Interrupt latency, Context switch time are important, there are a lot of other parameters that will make a good RTOS. For example a RTOS supporting many devices will have more advantages than a simple very good nano-kernel.”[6]

The lack of any proper documentation, information or source code of VxWorks *internals* are the main factors which prevent it being addressed in any research or academic program. Conversely, supporting a wide range of hardware devices, a significant amount of usage and interface documentation, along with extensive results on testing and evaluations have made it one of the most popular real-time operating systems in the industrial world.

It must be noted that substantial efforts were made in the study of VxWorks internals with regard to device driver implementations. The information provided here is comprehensive but by no means complete. While no reliable documents⁸ were found to verify the following concepts, we believe, to the best of our knowledge, that they are accurate and correct.

A short overview of the system will be presented, followed by an outline of the driver implementations of the two most important device types on VxWorks.

VxWorks provides a real-time kernel that interleaves the execution of multiple tasks by employing a scheduling algorithm. Thus the user sees multiple tasks executing simultaneously. VxWorks uses a single common address space for all tasks, thus avoiding virtual-to-physical memory mapping. Complete virtual memory support is, however, available with the optional vxMem library.

A task is an independent program with its own thread of execution and execution context. Every task contains a structure called the task control block which is responsible for managing the tasks’ context. A task has four states as outlined below:

Ready Task is ready to be scheduled for execution.

Delay Task is put to sleep.

⁸this section is mainly based on “VxWorks - Device drivers in a nut shell”, AyyalaSoft.

Suspend Task has been initialised, but not activated yet

or is debugged

or interrupted by an exception.

Pend Task is waiting for a resource.

The VxWorks scheduler runs a preemptive algorithm, which guarantees that the highest priority task preempts a lower priority task.

“Interrupt” is the mechanism by which a device seeks the attention of the CPU. The code which handles such interrupts is called interrupt service routine (ISR) or interrupt handler. Interrupt response time is the period starting from the occurrence of the interrupt to execution of the first ISR instruction. This period is composed of interrupt latency, delay for saving the task context and execution of kernel ISR entry function.

VxWorks provides a special context for interrupt service code to avoid task context switching, and thus renders faster responses. VxWorks supplies interrupt routines which connect to C functions and pass arguments to the functions to be executed at interrupt level [42].

All interactions with hardware devices in VxWorks are performed through the IOsub-system. VxWorks treats all devices as files. The two most important devices are *character devices* and *block devices*. There are two points which are common across both types of device drivers [42].

- Drivers in VxWorks can be dynamically loaded and unloaded.
- Drivers work in the context of the task invoked by an interface routine. Hence drivers are preemptable and should be designed as such.

There are four major issues involved in implementation of a Character device driver on VxWorks.

Driver installation The `iosDrvInstall()` and `intConnect()` directives are used

to provide basic driver functionality references⁹ and activate the hardware device related ISR, respectively.

Device descriptor registration The `iosDevAdd()` is used to add the device to the I/O system device list¹⁰.

Support for interface definitions Developer should provide code for all referenced driver functions¹¹.

Support for interrupt handler¹² Developer should provides device ISR, which is called when CPU receives an interrupt from target hardware.

Block devices interact with I/O systems through file-systems. Each block device is associated with a file-system. Block device drivers provide logical device structures. Logical device structures describe the device and contain routines to access the device in a general fashion via the underlying file system. Furthermore, it is the file system and not the block device driver which is installed as an entry into the I/O system driver table. The steps which define the development of block device drivers are as follows:

- Define and initialise all interfaces within the logical device structure;
- Associate and register the corresponding file system for the block device;
- Setup and connect the ISR.

Section 5.6 details device driver implementations on RTEMS, prior to a comparison of device drivers implementations on all three systems in Section 5.7.

⁹the user must provide function references for the *create*, *remove*, *open*, *close*, *read*, *write* and *ioctl* driver functions

¹⁰users can execute driver function calls through the I/O system device function table

¹¹the most important function is the `IOCTL` since users control the device through this function

4.4 Structure of SCI Drivers

The examination and study of SCI drivers have certainly been of crucial importance throughout this project. SCI drivers are by far one of the largest and most complex set of drivers seen for a PCI card. On Linux platform, the SCI set of drivers is at least twice as large as SCSI drivers¹³.

Throughout the introductory chapters, it was emphasised that all SCI protocols are carried out at hardware level, resulting in a low latency and high performance technology. One may wonder why SCI drivers (software) are of such importance? The reason is the tight connection of SCI and systems memory management in the nodes. Together with the requirement of connecting operating system kernels that have no knowledge of their counterparts on the other nodes, this makes SCI drivers rather complicated and probably the most complex of all drivers.

Unfortunately, lack of documentation, and the existence of few informative files within the driver source code forced us to thoroughly examine 37MB of source code! The information obtained is believed to be valuable and as stated within the Introduction Chapter, a simplified internal view of SCI drivers is presented in the hope that the information provided here will be useful for all future developers, who plan to examine, study, use, develop and extend the SCI technology and corresponding drivers.

The SCI drivers may be broken into two sections at the root. One section, labelled *adm*, allows users to compile, build drivers and conduct tests without the intervention of source codes and knowledge of SCI internals. Various scripts have been provided within the *bin* directory (located within *adm* directory) for each supported operating system to allow the user to compile, build and install drivers for different sets of cards.

The alternative to *adm* is *src*, which enters into the source code tree of SCI drivers. The SCI driver is split into two main parts.

¹³by many known to be the largest Linux device driver shipped with Linux distributions

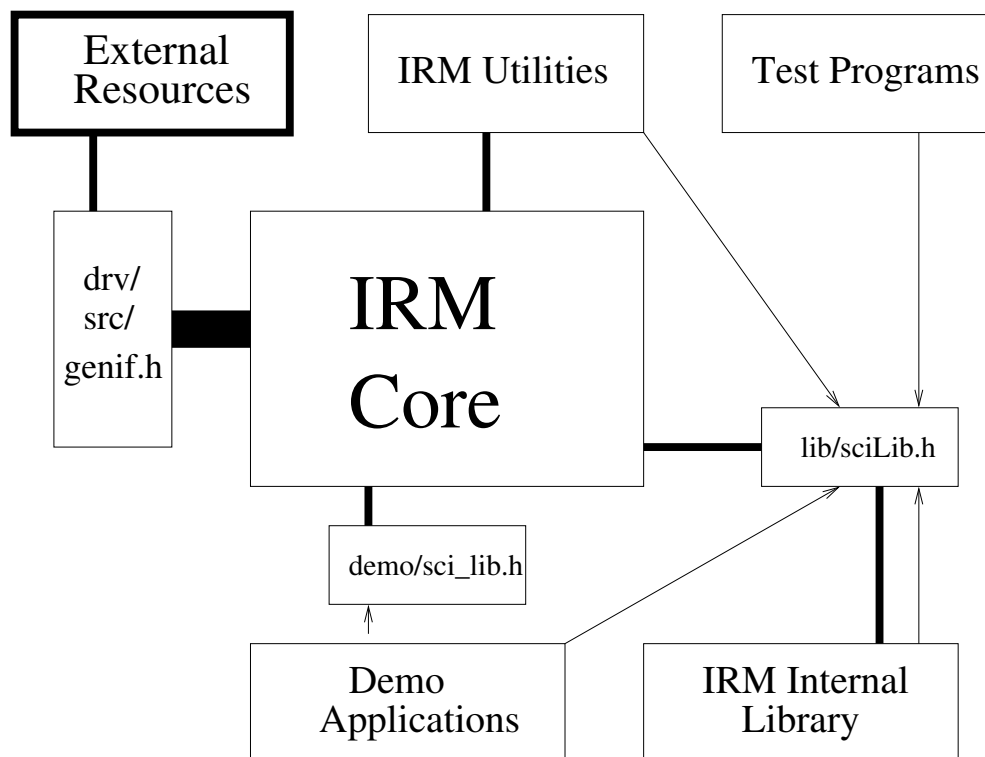


Figure 4.2: IRM Internal Structure

IRM IRM¹⁴ is a low level driver which interacts directly with the SCI hardware. It also provides higher level interfaces for low level programs, higher level drivers and modules such as SISCO, for reliable and simplified access to SCI resources.

SISCO SISCO is a higher level driver providing resource management and a simplified access interface for user level applications to the SCI adapter. While SISCO is a higher level driver relative to IRM, it is still considered as a low level driver and many additional layers are coded within the industry to sit on top of it for further simplified user interface.

Figure 4.2 illustrates the internals of the IRM and its interfaces. Three interfaces and one component directly interact with the *IRM Core*. The most significant interface is *genif.h* which provides full interface to all external resources. The *IRM Utilities* are a series of tools which allow users to configure the SCI adapter card. The

¹⁴Interconnect Resource Manager

test and demo programs are both a set of applications designed for testing and simple operations using IRM device drivers. Finally, while all external sections of IRM partially interact with the core directly, most communication is directed through an internal *sciLib.h* interface facilitated also by the *IRM Internal Library* section. All external resources and higher level drivers (such as SISI) **must** communicate with IRM through a general interface¹⁵.

4.5 *Compilation, Build & Configuration*

This section very briefly describes the procedure required to perform *custom* compilation, build and configuration of the SCI drivers. Of course, a set of scripts is already available within the root *adm* directory which simplifies all of this for the user.

There exists a build script within the IRM directory, which compiles and builds the IRM driver. The user must supply its operating system, architecture and adapter type in order for this script to compile the corresponding set of drivers. The result in most cases, is a binary object file, labelled *pcisci.o*. In the case of Linux operating system this would be the IRM module, which as indicated in previous sections, can be loaded into the system.

Similarly the SISI directory contains a build script which provides a second binary file, labelled *sisci.o*. This script also requires the operating system and adapter type to compile the relevant set of source codes.

The compilation of both sets of drivers is done using GNU Makefile utilities. SCI drivers use Makefiles in conjunction with some shell scripts to compile the relevant source codes. The makefiles relevant to IRM, are not only scattered through the source directories, but an exclusive set is also present within the *drv/src/MAKE* directory. SISI driver makefiles do not have a particular location and they are mostly scattered through the SCI driver structure.

SCI drivers seem to use run-time shell scripting as a hack to GNU Makefile procedures to achieve a finer control over the compilation of the drivers. Relevant

¹⁵*genif.h*

shell scripts are produced ¹⁶ and executed at the runtime stage of GNU Makefile utilities. This not only results in a high level of complexity in the compilation and build process, but it also increases the complexity of transferring such drivers to other platforms.

GNU Makefile utilities, nowadays, offer excellent flexibility and configurability[14]. We shall see how our target operating system (RTEMS) has used the latest of such tools to achieve a robust set of Makefiles within their source code structure.

The configuration of the adapter is considered when loading the driver into the system. The generated *pcisci.conf* specifies the configuration files to be used for configuring the *irm*, *psb*¹⁷ and *lc*¹⁸ sections. Only experts are recommended to specify their own configuration files, and in most cases it is recommended that *irmConfig.h* be used. This file contains default configuration for all sections of the board.

4.6 *IRM*

Interconnect Resource Manager is the lowest level of SCI driver code, which interacts directly with the SCI adapter. The source code supports a wide range of operating systems with all commercially available SCI cards. The present set of available SCI cards are:

- PCI1 (PSB32) adapter;
- PSB64 adapter;
- PSB66 adapter;
- SBUS2 adapter.

¹⁶mostly through *drv/src/MAKE/mklib.sh*

¹⁷pci-sci bridge

¹⁸link controller

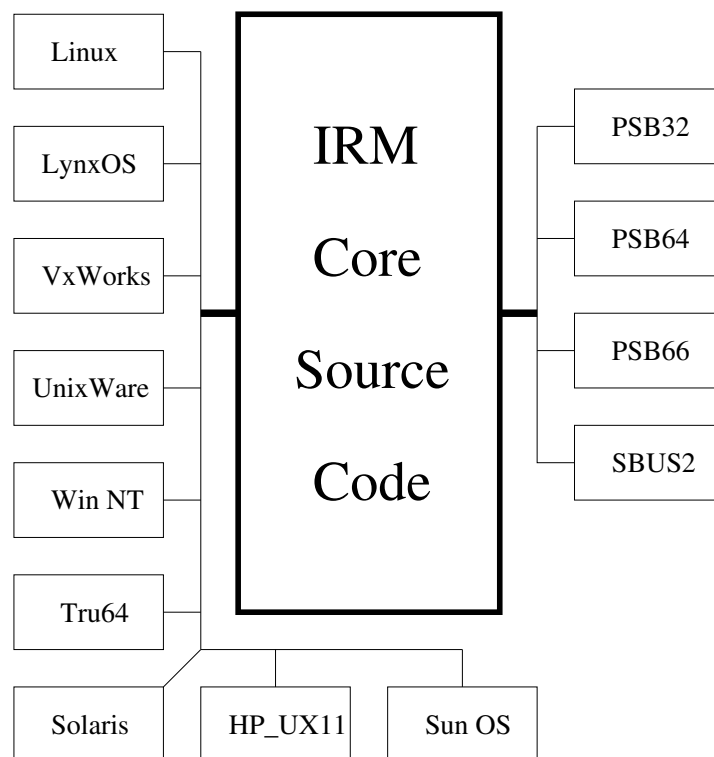


Figure 4.3: IRM source code structure

A general overview of the IRM source code is shown in Figure 4.3. On the left hand side and at the bottom of this figure, a series of operating systems which the SCI drivers support are shown. The ones of major importance are Linux and VxWorks as these will be examined in-depth in the Implementation Chapter. On the right hand side of the diagram, a list of all supported SCI card models is shown.

Depending on the set of options passed at compile time, the makefiles select the relevant sections and compile, build and merge them together for the final binary IRM driver object file.

Firstly, the interaction of the three sections ¹⁹ within the IRM source code will be examined.

The adapter dependent code itself splits into three sections. The first part is the *Link Controller* section. This controls the link controller chip on the SCI card. Several header files in this directory provide various structures which contain the

¹⁹IRM core source code, operating system dependent and adapter dependent code

location, type and significance of each register controlling the link controller chip. The second part relates to the *PCI-SCI Bridge* chip, known as the psb section. Similar to the link controller section, this section also provides informative structures regarding the registers on the PSB chip. The last section is the general adapter code which provides all the adapter specific functionalities listed in the *drv/src/adapter.h* file. This final section uses the structures provided from the previous two sections to read/write and configure the chips located on the SCI card. Hence, in an overview of this paragraph the adapter dependent section provides the adapter specific functionalities outlined in the *drv/src/adapter.h* file.

The operating system dependent code is the main interface through which the IRM driver interacts with the target operating system. This section is somewhat complicated due to the tight connection of the SCI drivers with the memory management of the operating systems. There are three main interfaces which the IRM driver requires to be supported by the operating system section. The first is comprised of various operating system related functionalities and is detailed within the *drv/src/osif.h* file. This section facilitates access to hardware, low level resource (data and register structures which reference on-chip registers) allocation and management, handling of events and interrupts, as well as locking, synchronisation and mutual exclusion which are required throughout the SCI drivers. The second is related to memory management and lists the desired functionalities within the *drv/src/memalloc.h* file. Tight connection of the SCI technology with memory management of operating systems requires support of a significant number of memory related functionalities, such as I/O buffer space allocation, memory locking (makes memory non-swappable), memory alignment (ensures the starting address of memory chunk is divisible by a certain alignment parameter), contiguous memory allocation, DMA transfer and many other functionalities. The last section is the SCI driver initialisation routine, which each operating system section must provide.

The section from the IRM core which relates to the initialisation routine is the bootstrap section. The file *drv/src/ldbootstrap.c* is used to indicate the approach taken for initialisation of IRM driver within the target operating system.

```

#if defined(Linux) || defined(OS_IS_VXWORKS)

extern ROUTINE init_module;

#ifdef OS_IS_VXWORKS
extern ROUTINE sci_get_local_csr;
#endif

static ROUTINE *syscall_required[] = {
    init_module,
    #ifdef OS_IS_VXWORKS
        sci_get_local_csr,
    #endif
};

#else

```

According to the above code²⁰, both Linux and VxWorks would like to initiate the driver by calling the `init_module` function, but VxWorks also proceeds with the `sci_get_local_csr` thereafter.

The IRM core section contains numerous interfaces and source files to perform various operations. The most important interface provided by the IRM core, as previously mentioned, is the *drv/src/genif.h* interface, which interfaces all allowable functionalities through the driver and should be used by all higher level code, driver and applications.

The IRM core consists of 2.1MB of source code, which supports all SCI technology related functionalities that are operating system independent, but not necessarily adapter independent. These functionalities include Session management, support for SCI communication protocol, Virtual Channel management, Mailbox and Interrupt management, Address Translation Table (ATT) configuration, adapter management, PCI and B-Link bus control, DMA engine support, Link Controller chip management, remote memory operations library, Switch, Timer, Topology support along with many other functionalities. The final IRM issue to be discussed is the initialisation routine, which must be implemented by the operating system dependent

²⁰taken from the *drv/src/ldbootstrap.c* file

section, and is thus investigated more thoroughly in following subsection.

4.6.1 *IRM Initialisation*

The SCI card initialisation routine is of substantial importance within the IRM drivers. Most functionalities of the card are performed at hardware level, which implies that the initialisation process holds greater responsibility in initialisation and configuration of the card. This is to allow for hardware based operation of the card at a later stage.

The initialisation routine is comprised of both operating system [OS] dependent and independent code. The OS dependent section mainly facilitates access to the card and registers within the card, as well as resource allocation and management in the system. The OS independent section configures and initiates the card and resources using the OS dependent layer and this section is primarily dependent on the card model.

Although we are only interested in the OS dependent section of the initialisation routine, one must stress that the OS independent section was also studied in-depth to ensure correct functionality and elimination of unnecessary code under the target operating system. Lack of documentation of this routine was another reason which supported the idea of representing the initialisation routine through a sequential process. Finally, every effort has been made to eliminate the complex nature and inter-communications of the routine, but where necessary sufficient detail has been provided to fully explain the objective.

PCI-SCI D310 Initialisation routine:

Find the PCI-SCI card Initialises the PCI BIOS interface and searches through the PCI bus for any device matching SCI card vendor and device tags.

Adapter Table creation Allocates memory to hold the adapter table information.

Adapter Table initialisation Places preliminary information obtained about the PCI device into the adapter table.

Setup PSB memory areas Sets up the memory descriptor for the PSB's csr, IO and prefetch space.

Store interrupt line Obtains the interrupt line number from the card memory.

Map PSB memory areas Maps the PSB's csr, IO and prefetch space into kernel memory.

Check PSB Verifies that the PSB is present and alive.

Lock creations Creates semaphore locks for Main_Driver, DMA, FlagIntr, ATTin-dex and timeQ.

Initiate the timeout job handler Initialises a separate thread which creates a message queue awaiting jobs to be dispatched from the timer thread to be executed ²¹.

Pre session and VC initialisations Preliminary configuration, resource allocation prior to attachment of any adapter.

Attach SCI interrupt Attaches SCI interrupt handler to the ISTAT register of the card ²².

PSB chip verification Reads and ensures that the PSB chip ID and revision match the ID and revision of the the current set of drivers ²³.

Obtain config info Obtains configuration info related to the card either from the adapter non-volatile memory or the config struct that has been previously filled in by a config operation.

²¹the `general_timeout_handler` is called which removes the job from jobs list and subsequently executes it

²²the handler reads and executes the jobs specified at the ISTAT register of the card

²³in our case the chip ID is 0x3D65806D signifying PSB32 and rev D

Insertion of config info Places the obtained config info into the SCI adapter structure.

Setup physical ID table Dependent on LC and PSB chip model, initialises the physical topology settings on the LC chip.

Configure LC chip Initialises and configures the LC chip related registers.

Initialise interrupt mask Stores the interrupt signal masks and their significance.

Set GX timeout Sets up the (possible) GX hostbridge to 30ms timeout on PCI.

Set NX In the case of the PCI host bridge type being HB_450NX_PXB, turn on the BWCE²⁴ and turn off the Assert SERR# on the inbound delayed read time-out.

Set ServerWorks Latency In the case of the PCI host bridge type being HB_SERVERWORKS_LE, set PCI Latency Timer to ensure enough time for PCI burst transfer.

Enable write posting Enables write posting on PCI host bridge.

Pre PSB initialisation Sets up BIU Control, SIDeadlkCnt, Misc Control registers, CSR access protection, interrupter, windows protection and Stream configuration in PSB and initialises PCI CSR.

Misc operations A mixture of random configuration reads and sets (related to both PSB and LC chips), plus the enabling of numerous board related functionalities (such as StoreBarrier, ATT initialise, nodeProbe, Client ErrCheck and Software packet buffer allocation).

VC initialisation Starts the VC²⁵ watchdog and module for the adapter.

²⁴Burst Write Combining Enable

²⁵related to receive and transmission functionalities of the card

DMA initialisation Starts the DMA engine²⁶ on the board.

Switch module configuration The Card predicts and sets the switch ID to which it belongs on the overall SCI topology.

Interrupt info initialisation Initialises the interrupt counters monitoring the jobs and tasks performed by order of the ISTAT register.

PSB initialisation Sets up final configuration of the PSB chip and starts the chip²⁷.

Final watchdog activations Starts the LC chip and session watchdog²⁸.

At the end of this routine the software intervention of the driver initialisation is over, but the hardware still performs necessary initialisations through a set of timer functions which it deems necessary. The `osif_timeout` is the function which is called with time related jobs. It creates watchdogs for each job which expire at the time chosen by the caller. Once a watchdog expires it submits the job into the jobs message queue, which is subsequently handled by the timeout job handler for execution. The majority of jobs are created through the hardware calls, ISTAT register and interrupt handler routine.

4.7 *SISCI*

SISCI is a higher level driver for enhanced resource management and simplified programming interface for user applications [16]. It maintains a table of resources held by user applications, and in the case of failure of the application task, it manages the resources and releases them appropriately. The SISCI is also responsible for bridging the user mode operations into kernel mode operations, allowing user applications, which are normally in the user space, to access IRM functionalities at kernel

²⁶enables for DMA transfer between nodes

²⁷resulting in full functionality of the chip on the PCI bus

²⁸bringing the card to a state where it can interact with external nodes as well

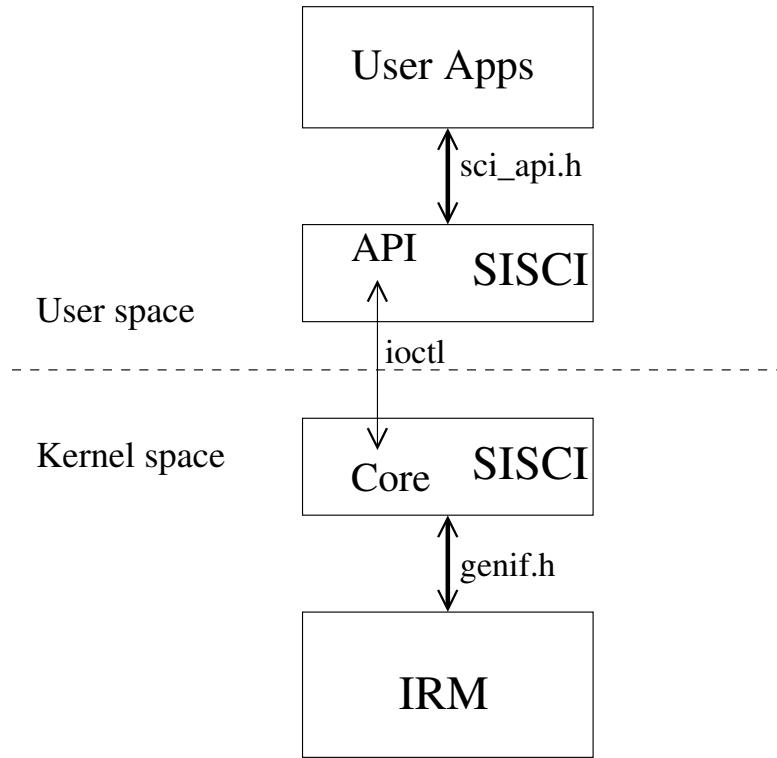


Figure 4.4: SISCi source code structure

or privileged user space [16]. Requests, addresses and data from the user space is copied and/or converted to the kernel execution space. Data in most cases, is simply copied from users space to the kernel space. Requests are converted/decoded into a series of IRM commands and functionalities, which are performed to deliver the objective. Addresses are also converted from the user mode to the kernel mode, referencing the copied data from the user space to the kernel space.

Figure 4.4 shows how IRM, SISCi and user applications interact with each other. As shown, the SISCi driver is mainly split into two sections, the *Core* and the *API*²⁹.

After compilation of the SISCi driver, the user is presented with a binary *sisci.o* file. In case of Linux, this is yet another module which has to be loaded into the kernel. This module contains the core section of the driver, which as shown in the diagram, interacts with IRM through the *genif.h* interface. The second section of

²⁹Application Programming Interface

the SISI driver³⁰ is compiled as a series of object files. User applications, when willing to use SCI functionalities, include header and interfaces of the API section to link their object files with the relevant SISI API object files. Hence, one can say the API section of the SISI driver is linked into user applications, allowing them to communicate with the Core section.

The main interface between the user applications and the SISI API is *sisci_api.h*. The API section transfers calls to *sisci_internals.h* through *kernel.h*, where `ioctl` calls are performed to invoke corresponding SISI core functions within the kernel space. Numerous core files are located within the core section which handle all types of calls. All files within the core section interface with the IRM through the *genif.h* interface.

In summary, the SISI layer, without providing any additional driver functionality, has three main objectives.

- To act as a bridge between the user and kernel execution mode;
- To simplify the programming interface, hiding a substantial amount of complexity behind the user interface;
- To handle resources at an internal level as much as possible.

These actions are implemented across all examined operating systems.

³⁰the API section

Chapter 5

RTEMS

RTEMS, Real-Time Executive for Multiprocessor Systems, is the real-time operating system which was selected for this project. RTEMS was developed by OAR Corp. which also co-ordinates developments and offers commercial support for RTEMS.

RTEMS was designed for real-time applications from the very beginning, targeting mainly embedded systems. The executive interface is presented to the user applications through a set of resource managers, each facilitating a certain functionality. The main functionalities are built-in to the executive core, while additional functionalities are wrapped externally to the core. Applications dependent on their needs would specify, compile and link the desired resources (and their corresponding managers) into the target system. The term “target” refers to the hardware on which the end result is expected to function and execute. Although tasks are internally synchronous, they execute independently resulting in an asynchronous processing stream.

Some RTEMS features are listed below.

- Multitasking capabilities
- Homogeneous and heterogeneous multiprocessor systems
- Event-driven, priority-based, preemptive scheduling
- Optional rate monotonic scheduling
- Intertask communication and synchronisation
- Priority inheritance

- Responsive interrupt management
- Dynamic memory allocation
- High level of user configurability

RTEMS also features POSIX (1003.1b), ITRON and a native API in C language. RTEMS's classic (native) API also facilitates Ada users by interfacing Ada language.

With portability in mind, RTEMS supports a range of architectures, such as A29k, ARM, H8300, I386, I960, M68k, MIPS, PPC, Sparc and Unix.

Throughout this chapter we will be examining RTEMS from various perspectives. Initially we examine how to achieve real-time systems¹ within RTEMS and satisfy hard deadlines. Next, RTEMS is evaluated against other RTOS's and its selection for this project is justified. The structure of RTEMS is of major importance and rather complex, we will be discussing those issues through a series of sections. Finally, system initialisation and device drivers on RTEMS are reviewed at the later sections.

5.1 RTEMS Rate Monotonic Scheduling (RMS)

One of many useful resource managers within RTEMS is the rate monotonic manager. It offer rate monotonic scheduling for periodic and hard real-time tasks. The rate monotonic scheduling algorithm is a hard real-time scheduling methodology. Using this scheduling algorithm a set of independent tasks are always guaranteed to meet their deadlines, even under transient overload conditions.

The algorithm examines the schedulability² of a task set under worst case conditions and models the system's behaviour predictably via schedulability analysis through a set of specific rules.

It has been proven that "RMS is an optimal static priority algorithm for scheduling independent, preemptible, periodic tasks on a single processor" [30]. This implies

¹one of the most important issues in any RTOS

²ability to schedule and meet dead-line of a task

that if a scheduler³ can schedule a task, then a rate monotonic scheduler is guaranteed to be able to schedule the task as well.

The algorithm assigns each task a priority based on its period⁴. RMS analyses the schedulability of tasks using task periods and execution times in conjunction with specific algorithms.

Processor Utilisation Rule can initially be used to examine the schedulability of the task set. If it fails one can still use the *First Deadline Rule* to further examine the schedulability of a task set. If either rules are satisfied, the task set is said to be schedulable by the Rate Monotonic Scheduler.

While it is important to note the scheduling methodology of real-time operating systems (and in our case RTEMS), we will not go into any further detail regarding the rules and algorithms. The following sources may be used to study Rate Monotonic Scheduling and its schedulability analysis in greater detail.

- C. L. Liu and J. W. Layland. “Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment.” **Journal of the Association of Computing Machinery**. January 1973. pp. 46-61.
- John Lehoczky, Lui Sha, and Ye Ding. “The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior.” **IEEE Real-Time Systems Symposium**. 1989. pp. 166-171.

5.2 Comparison of RTEMS with others

The number of real-time operating systems available within the industry is growing in number by the day. This section compares our choice of RTOS (RTEMS) with some other popular and successful real-time operating systems. The pros and cons of RTEMS will be laid out and the reasons for this choice of RTOS in this project will be outlined.

³based on a different static priority algorithm

⁴the interval between successive iterations of the task is referred to as its period

RTEMS shall be compared with other RTOS's from three perspectives [11]. The first of these is the availability of documentation and ease of development of the systems; the second is the features offered by each system and finally the comparison of Interrupt latency and Context switching parameters between the systems. RTEMS is examined against two RTOS's, one a successful commercial product, VxWorks, and the other a popular open source RTOS, RTLinux.

The documentation available in each case is subjective. While RTEMS and RTLinux both have extensive documentations available with regards to their operation, functionality, design and implementation, VxWorks has a considerable amount of documentation within the production, configuration, testing and application sectors. Industrial users tend to choose systems like VxWorks, which present clear statistics on their specifications, performance and applications of product. Conversely, research oriented industries, developers and academics are mainly interested in the systems of the first type for obvious reasons.

In relation to ease of developments RTLinux and RTEMS are both superior to VxWorks. RTLinux and RTEMS present a clear, free and full view of their internal core, system structure and parameters which a developer may use to his/her advantage in optimising the target system or exploiting the system to its highest potentials. Even though both are "open source" based, RTEMS is *licence free* which implies clients may freely modify, distribute, develop or even include it as part of a "proprietary work". VxWorks, on the other hand, is a proprietary system. Unfortunately the lack of detailed documentation of VxWorks internals results in developments which are related to VxWorks system being a serious challenge⁵.

Table 5.1 illustrates and compares the various features of the three real-time operating systems [20]. The table itself is detailed and requires little explanation. An important lesson that may be learnt from this table, however, is that RTEMS in most cases is superior to the other two systems in terms of availability of features and support of services. With regards to other services, such as watchdog timers, these were implemented on RTEMS if deemed necessary. In fact, some services such

⁵clearly experienced throughout this project!

	Multi-processor	Scheduling	Concurrency
RTLinux	Yes	FIFO	Pthreads
RTEMS 4.5+	Static allocation	FIFO,RR,Other	Pthreads
VxWorks 5.x	Optional	Fixed,RR	Lightweight processes

	Inter-process comm.	Dynamic Memory	Priority Inversion
RTLinux	Sem,Mutex,CondVar,FIFO	No	Ceiling
RTEMS 4.5+	Sem,Mutex,CondVar,PQ,Event	Yes	Inheritance,Ceiling
VxWorks 5.x	Sem,Mutex,Msg,RTsignal	Optional	Inheritance

	Filesystems	Timer res.	Timers
RTLinux	None	Hrd. dependent	None
RTEMS 4.5+	IMFS,DOSFS/FAT	Hrd. dependent	POSIX
VxWorks 5.x	FAT,NFS,TrueFFS	Configurable	Watchdog,POSIX

	API	Debug	Languages
RTLinux	POSIX 1003.1c	trace,GDB	C,C++
RTEMS 4.5+	RTEID,ITRON,POSIX 1003.1b	GDB,DDD,RDB	C,C++,ADA
VxWorks 5.x	VxWorks,POSIX 1003.1&1003.1b	GDB,RDB	C,C++

Table 5.1: Service comparison between RTLinux, RTEMS and VxWorks

as the watchdog timer itself were implemented on RTEMS as part of the SCI driver development ⁶.

Finally, we examine parameters which some people, mistakenly, consider as the most and only factors of importance in evaluation and consideration of an RTOS.

⁶detailed within the next chapter

	Interrupt Latency		Context Switching	
	max	avg $\pm\sigma$	max	avg $\pm\sigma$
	<i>Idle System</i>			
RTLinux	13.5	(1.7 \pm 0.2)	33.1	(8.7 \pm 0.5)
RTEMS	15.1	(1.3 \pm 0.1)	16.4	(2.2 \pm 0.1)
VxWorks	13.1	(2.0 \pm 0.2)	19.0	(3.1 \pm 0.3)
	<i>Loaded System</i>			
RTLinux	196.8	(2.1 \pm 3.3)	193.9	(11.2 \pm 4.5)
RTEMS	20.5	(2.9 \pm 1.8)	51.3	(3.7 \pm 2.0)
VxWorks	25.2	(2.9 \pm 1.5)	38.8	(9.5 \pm 3.2)

Table 5.2: Interrupt latency & Context switch delay comparison between RTEMS, VxWorks and RTLinux

They are the Interrupt latency ⁷ and the Context switching delay⁸ parameters.

At the *8th International Conference on Accelerator & Large Experimental Physics Control Systems, 2001, San Jose, California* table 5.2 was presented as results of performance measurement of RTEMS versus RTLinux and VxWorks. Measurements were carried out on a PowerPC 604 CPU (300MHZ), MVME2306 (PReP compatible) board manufactured by Motorola. This choice was made since all three systems supported this board, and also it featured a high resolution timer hardware ⁹. The software package consisted of initialisation, interrupt service routine and a simple “measurement” procedure.

“The initialisation code sets up the timer hardware, connects the ISR to the respective interrupt and spawns a task (MT) executing the measurement procedure

⁷time elapsed from the moment of occurrence of an event (e.g. a hardware interrupt) until execution of the first instruction of the Interrupt Service Routine (ISR), which includes the overhead required by the executive at the beginning of each ISR plus the time required for the CPU to vector the interrupt

⁸time delay from the execution of the last instruction of a task, to execution of the first instruction of another task, which includes the time scheduler determines which task to run, time to save the context of first task and time to restore the context of second task

⁹suitable for “on-the-fly” latency measurements

at the highest priority available on the system under test. The ISR determines the interrupt latency by reading the timer and notifies the MT by releasing a semaphore on which the MT blocks. This causes the system to schedule the MT (having become the highest priority runnable task), which, reading the running timer is able to determine the time that elapsed from the ISR releasing the semaphore until the MT actually getting hold of the CPU. After recording the delay, the MT again blocks on the semaphore.

This simple test was performed on a system heavily loaded with low priority tasks, networking and serial I/O traffic causing a large volume of interrupts (also at a priority lower than the timer hardware IRQ).” [38]

The smallest average and variance values related to the interrupt latency and the context switching delay, in most cases, is observed to be on RTEMS. VxWorks, though not the best in average latency and delay, holds the lowest maximum measured interrupt latency (in idle systems) and switching delay (in loaded systems). Apart from having the lowest average interrupt latency value on loaded systems, RTLinux has higher figures in comparison to RTEMS. Differential values between RTLinux and the other two systems are greater by far than the differential values between RTEMS and VxWorks. Casting a vote on the systems based on this table, RTEMS is first, VxWorks closely second and RTLinux is by far the third.

Almost all RTEMS directives execute in a fixed amount of time regardless of the number of objects present in the system. The primary exception occurs when a task blocks while acquiring a resource and specifies a non-zero timeout interval. Other exceptions are message queue broadcast, obtaining a variable length memory block, object name to ID translation, and deleting a resource upon which tasks are waiting. In addition, the time required to service a clock tick interrupt is based upon the number of timeouts and other ”events” which must be processed at that tick. This second group is composed primarily of capabilities which are inherently non-deterministic but are infrequently used in time critical situations. The major exception is that of servicing a clock tick. However, most applications have a very small number of timeouts which expire at exactly the same millisecond (usually

none, but occasionally two or three) [31] [10].

Summarising the examined factors, RTEMS as an “open source” real-time operating system together with RTLinux are more appropriate for developers and academics with regards to documentation and development. The fact that RTEMS is *licence free* would even place it ahead of RTLinux. In relation to the services and features available on each system, RTEMS and VxWorks are placed first and second respectively, with RTLinux placed last. Analysing systems from a performance point of view, RTEMS and VxWorks demonstrated high and tight performance competitiveness. Dependent on the test procedure or target hardware, results may vary, but RTEMS and VxWorks would be expected to produce results close to each other. RTLinux’s performance was considered poor and unsuitable for critical real-time applications. Overall one must agree that RTEMS is the most suitable RTOS for the purposes of this project.

RTEMS is available as an “open source” system and it is far superior to VxWorks. To illustrate this fact in today’s technology, a section of NASA’s report on the system which they use for flight applications is provided below.

“...At present, a commercial OS is commonly used (vxWorks) by Wind River Systems, which establishes a reasonable baseline for comparisons with respect to toolchains, runtime footprints and runtime performance. Due to the class of processors used in flight, operating systems are under some substantial constraints and have stringent requirements in areas like runtime size, reliability and task scheduling...

The principal requirement was the operating system be “open”- that is, the full source should be available. VxWorks is a closed, proprietary system- it is also very expensive to procure licenses, even more expensive to get source code, support is frequently poor- in general, its the normal litany one expects from a closed source vendor.”[3]

NASA considered RTEMS4.5 as a suitable replacement for the VxWorks systems!

5.3 RTEMS Structure

The structure of RTEMS has changed in recent months. The latest stable release version (RTEMS 4.5.0) is nearly two years old. Having initiated the project development on the latest stable release, a massive change in structure was observed on the migration of code from this version to the latest CVS version.

This section briefly touches on the structure of RTEMS based on the latest CVS release, rtems-ss-20030211 snapshot. Furthermore we will be focusing on the source tree structure related to the C language bindings only.

In order to relate our section directly to the source tree, we present a similar tree model, introducing each branch individually.

cpukit Executive RTEMS core.

ada Facilitates Ada API.

itron Facilitates ITRON API.

libblock Provides generic device and block controller functionalities, these are device and hardware independent IO functionalities.¹⁰

libcsupport Provides the required support for the newlib package (newlib glue). It also includes implementations of POSIX services as well as non-threading ANSI C library.

libfs Filesystem support, currently DOS FS and IM (In Memory) FS.

libmisc Miscellaneous libraries, including modules for RTEMS performance monitoring, high level OS functionalities and interface support for MicroWindows input devices.

libnetworking Supports FreeBSD TCP/IP network stacks.

librpc Facilitates the Sun RPC module (FreeBSD RPC/XDR).

posix Facilitates POSIX API (threading portion).

rtems Facilitates RTEMS API (resource managers).

¹⁰e.g. block devices, floppy, CD-ROM and RAM disk controllers

- sapi** This core provides API features which can't be addressed as part of any specific API and are beyond any RTOS standardisation. It is common code across all processors, and implements functionalities such as system initialisation, shutdown, I/O and error processing.
- score** This is the Super Core module, which provides the RTEMS internals. All APIs are implemented on this core (similar to wrappers around a core). Users should not use the functionalities offered by this core, but rather use the supported APIs.
- lib** Contains processor and board dependent libraries.
- libbsp** Contains processor and board dependent packages and routines, including shared routines among boards ¹¹, initialisation routines and board specific device drivers (processor and board dependent).
- libcpu** Specific processor functionalities and modules are present within this directory (processor dependent).
- libchip** Chip related library supporting hardware chip-sets available across range of boards (such as ethernet card, serial ports, IDE and rtc).
- libnetworking** RTEMS networking application layer (rtems_servers, rtems_telnetd, rtems_webserver and pppd).
- librdbg** RTEMS remote debugger support for M68k, PowerPC and i386
- librtems++** RTEMS C++ API.
- tests** Test packages.

The RTEMS source code structure design has been aimed at high portability and efficiency. Every section supports a specific set of functionalities or modules.

¹¹such as pci, io, irq and com modules for i386 hardware

Developers and advanced users must have a clear understanding of this structure, so as to be able to develop, optimise and debug their systems efficiently and correctly.

As a result of developments and enhancements, code sections are moved from one area to another. In most cases this is as result of generalising code among a range of target boards or processors. A developer not only has to code relative to the section he is examining, but must also consider the availability of other sections, modules and functionalities at the execution stage of his code. Such issues were carefully examined and analysed throughout this project and will be stated clearly within the next chapter.

5.4 *RTEMS Build*

Building RTEMS comes as a great surprise to many developers and users who have migrated from classical operating systems to RTOS's and specially RTEMS. The correct compilation, build, configuration and linking of RTEMS is one of the most important factors governing proper functionality of the end system. The RTEMS end result is a binary image file, called RTEMS image file, which entails all elements, the operating system, application and required modules. The RTEMS image file is placed on the target board or loaded into memory at the system start by a loader.

The RTEMS image file will take control of the system from the moment of initialisation of system, customly initiating the processor, board and related attached devices. The image file in most cases, is placed within a non-volatile memory (e.g. ROM) on the target board. On a i386 CPU, pc386 board, a grub loader can be used to load the RTEMS image file into RAM and transfer control to the RTEMS image file.

The RTEMS image file may be built on most platforms and is independent of the target environment ¹². The platform or system on which the RTEMS image file is developed on is referred to as the “host”.

In an aim to maximise the range of hardware platforms which may be used as *host* for development of RTEMS image file, RTEMS provides its own tools and required

¹²target processor and board jointly are referred to as *target environment*

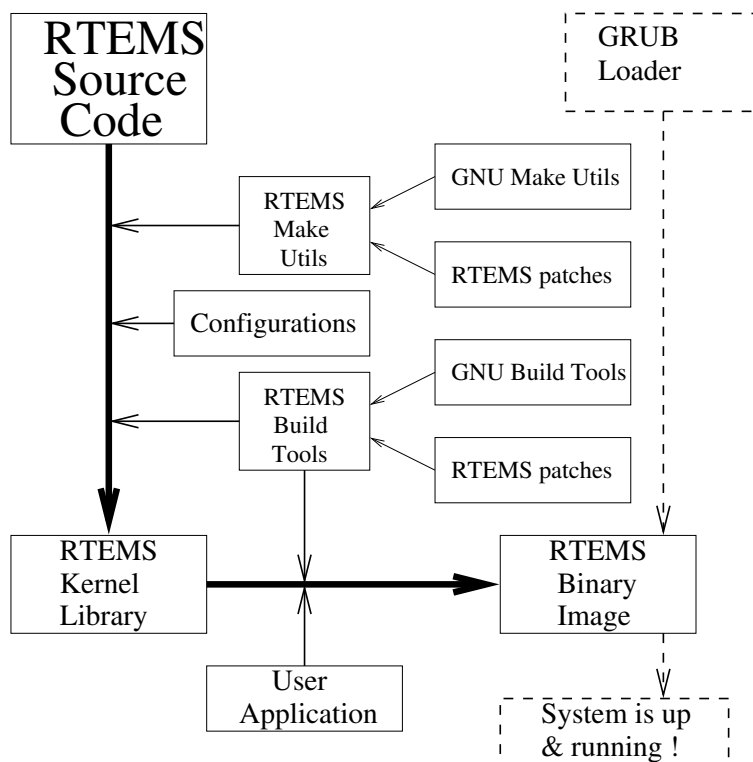


Figure 5.1: RTEMS Build Diagram

utilities for the development. Another factor which has resulted in RTEMS specific tools and utilities is the requirement of the *latest* tools and utilities for development of the RTEMS image file.

Figure 5.1 illustrates the steps and procedures required to develop the RTEMS image file from scratch. Following subsections will give details on each step and settings used throughout this project.

RTEMS as a rapidly developing platform encounters a significant number of bugs and malfunctionalities within the existent GNU tools and utilities. Most bugs and malfunctionalities are resolved within later versions of GNU tools and utilities, but RTEMS provides patches to resolve problems temporarily.

5.4.1 RTEMS Tools

RTEMS tools are composed of GNU tools and corresponding RTEMS patches. The packages referred to as tools are specifically the bin utilities, gcc, newlib¹³ and gdb.

All of the above gnu tools may be downloaded from the RTEMS snapshot site, or alternatively they are provided through the RTEMS CD. The source codes of all tools must be patched with their corresponding RTEMS patch files. Tools must be compiled and built in the following order (configurations used for each tool within this project are also provided below).

```
binutils configure --target=i386-rtems --prefix=/opt/rtems
```

```
gcc configure --target=i386-rtems --with-gnu-as --with-gnu-ld
      --with-newlib --verbose --enable-threads --prefix=/opt/rtems
```

```
gdb configure --target=i386-rtems --prefix=/opt/rtems
```

The `i386-rtems` option is an indication of the target processor, the `/opt/rtems` signifies the installation directory and `--with-newlib` is specified to link the gcc package with the newlib package.

5.4.2 RTEMS Makefiles

The fast development and complexity of RTEMS structure demands the use of the latest Makefile utilities to ease and generalise RTEMS developments. The latest GNU Makefile utilities not only simplify the making process significantly, but also result in a less error-prone and highly unified structural model when multiple developers work on a single project [14] [17]. All this, however, comes at a cost of realisation and understanding of the latest utilities, while maintaining make utilities up-to date.

The RTEMS development team has recently begin providing the latest GNU Makefile utilities with corresponding RTEMS patches. Even though users with the

¹³portable ANSI C library implementation intended specifically for embedded systems

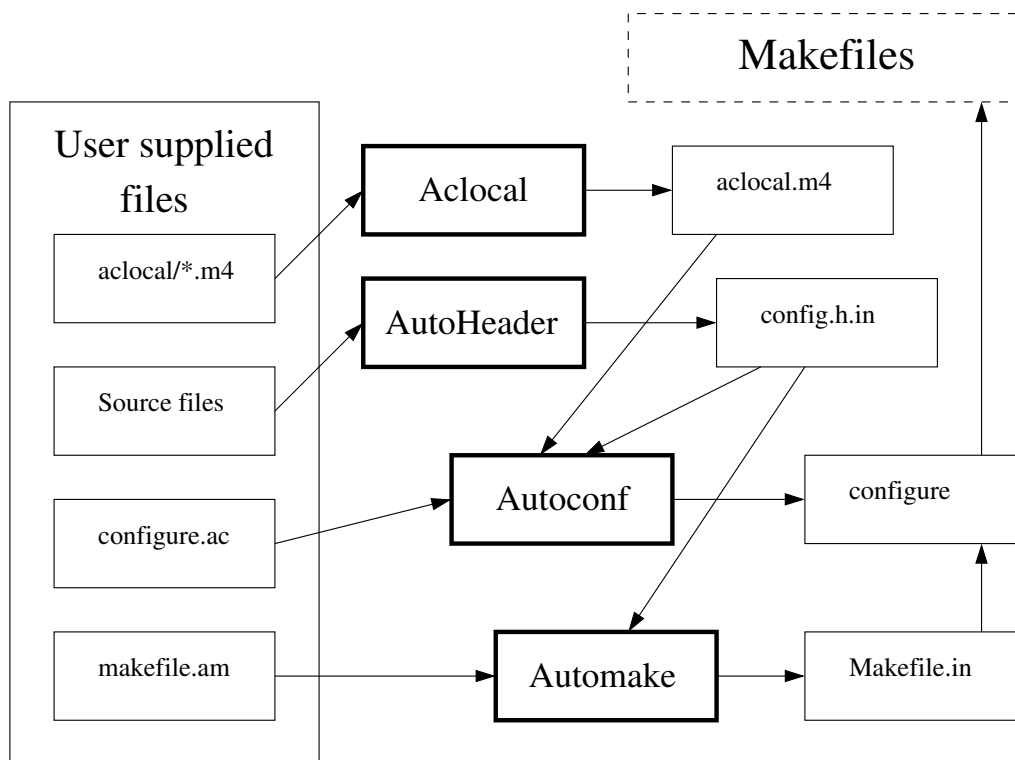


Figure 5.2: RTEMS Makefiles

latest GNU Makefile utilities may still make RTEMS makefiles without installation of RTEMS patches, in the near future the use of RTEMS patches will become necessary to maintain a more structured and unified approach towards further enhancement of RTEMS.

Makefile utilities required for RTEMS are autoheader, aclocal, autoconf, automake and make. Figure 5.2 illustrates the process for creation of Makefiles. While most developers directly code makefiles for their projects, some use the configure and makefile.in approach (increasing portability and configurability, together with minimisation of error) and very few (such as RTEMS developers) use GNU make utilities to their fullest [17].

In the final step, `configure` script is used to achieve configured makefiles from the `Makefile.in` files. Next section details the use of `configure` script to configure the RTEMS build process.

5.4.3 RTEMS Configuration

RTEMS configuration is the process where the user indicates the specifications of the target hardware and his/her desired modules for the application system. This allows users to compile and build only the necessary and relevant sections of the RTEMS source code. Space and time are the main considerations here, the object and library files related to a single target environment may occupy as much as 800MB of space!

For the purpose of this project the following configuration was supplied to the `configure` script.

```
configure --target=i386-rtems --enable-rtemsbsp=pc386
--enable-posix --disable-itron --disable-cxx --disable-tcpip
--disable-networking --disable-multiprocessing --disable-rdbg
--disable-tests --disable-docs --prefix=/opt/rtems
```

The above configuration, while minimal, provides the necessary functionalities for the purposes of this project. The RTEMS kernel library may be built using the ‘‘`make all install`’’ command.

5.4.4 RTEMS Image File

The RTEMS image file is the final resultant system, which is both target hardware and user application dependent. The user compiles and builds his/her application using the RTEMS kernel library. The user may need to provide a suitable `Makefile` to perform this process.

The application must state all resource managers and assets it requires. Hence only the required parts and segments of the RTEMS kernel library will be built into the resultant binary image file. Furthermore, according to the specifications of the application (which the user states within his/her source code) the RTEMS linker allocates memory segments for different sections of the execution environment (i.e. size and position of code, heap, data, etc). Users may, by aid of linker scripts, customise the memory allocation process. It is through the application source code and the linker script which the user can fully customise the resultant image file [32].

While the user may develop applications which drift the system towards a dynamic behaviour, the RTEMS itself is static. By the term *static* we imply that the operating system has fixed behaviour with regards to functionality, services and resources as predefined by the user application and the RTEMS configuration. Hence, one can not introduce new functionalities, resources or services at run-time level (e.g. no installation of new OS functionalities *unless* the user application facilitates the issue).

5.5 RTEMS Initialisation

The initialisation routine of a system is the first piece of code which is executed on the system processor after a reboot or restart. This section presents a general overview of the RTEMS initialisation routine. While the system is “open source” and open to manual customisation for specific target environments, RTEMS developers have designed an initialisation routine where users can customise and configure the routine to great extent through their application source code !

The initialisation routine starts by execution of a processor specific assembly language code, located within the `libbsp` directory. This code initialises the processor and board at the lowest level, performing basic tasks such as stack initialisation and disabling external interrupts. It is the minimum necessary code to allow continuation of the routine through C code.

The invocation of the shared `boot_card()` method signifies the end of assembly language code and the initiation of execution through C language. This routine introduces a series of configuration tables (e.g. CPU, RTEMS, BSP configuration tables) and initiates the board specific initiation via the `bsp_start()` directive. Following the `bsp_start()` function, the `rtems_initialize_executive_early` and the `c_rtems_main` directives are invoked sequentially. Finally the `bsp_cleanup()` directive ends the system.

Note that the user application is executed within the `c_rtems_main` function and once finished the system shuts down by calling the `bsp_cleanup()` function. The following subsections describe each of the four main stages briefly.

5.5.1 *bsp_start()*

In the i386 processor package (with which we are mainly concerned) this directive is weakly aliased to `bsp_start_default`, hence `bsp_start()` does *not* exist! This allows the user to override the default `bsp_start()` function with his/her own function if he/she desires to do so. This capability is in-line with the earlier noted flexibility in initialisation routine.

This routine performs three main tasks:

- It inserts the processor and board specific information into the earlier created configuration tables
- It initialises the RTEMS interrupt manager
- It initialises the RTEMS exception manager

5.5.2 *rtems_initialize_executive_early*

This routine is called when the target environment is ready for initiation of the RTEMS system environment. Hence this is considered as the start of the RTEMS itself.

The directive initiates the RTEMS using the configuration tables obtained through the previous routines and the user supplied configuration tables. The following are the main tasks performed by this routine.

- Initiating a series of managers and handlers (such as user extensions manager, IO manager, object handler and interrupt handler) starting with the Debugging manager to enable debugging at all stages.
- Initialising the RTEMS API and other supported APIs (upon request)
- Creating the “idle thread” which is considered as the starting point of thread initialisations as well. Before this point no thread should be created since `_Thread_Executing` and `_Thread_Heir` are not set yet.

- Initialising all device drivers. It should be noted that the `_API_extensions_Run_predriver` and the `_API_extensions_Run_postdriver` functions are provisions for the user to perform desired tasks before or after the device driver initialisations (ignored in most cases).

5.5.3 *c_rtems_main*

This routine after setting the `rtems_progname` (program's name) variable, simply calls the `rtems_initialize_executive_late` directive.

The `rtems_initialize_executive_late` directive initiates multitasking within the system, starts the main application and simply goes away. Execution is resumed within this thread once the application calls `rtems_shutdown_executive`. `rtems_shutdown_executive` routine stops multitasking and resumes execution of the `c_rtems_main` directive.

5.5.4 *bsp_cleanup()*

Even though the `bsp_cleanup()` directive is executed as part of the shut down sequence, we briefly describe actions of this directive for the completeness. This directive simply calls the `rtemsReboot` routine, which performs board specific reboot operation.

Within the i386 environment, the `rtemsReboot` routine performs reboot using the keyboard controller.

```
outport_byte(0x64, 0xFE);
```

5.6 *RTEMS Device Drivers*

This section examines how device drivers are implemented and how they are structured within RTEMS. Devices within RTEMS are classified according to the RTEMS source code structure.

Some drivers are generic and hardware independent. An example of this would be the hard-disk controller which is hard-disk independent but necessary as part of hard-disk device drivers. These drivers are mostly processor and board independent.

They are located within the `libblock` section. In some cases, they may even be considered as an intermediate interface between RTEMS and device drivers.

The next class of drivers are ones which are generic among most boards and some processor types. These are at a lower level than the ones described above (located within the `libchip` directory).

At present, the section is composed of `ide`, `network`, `rtc` (real-time clock) and serial device support. For example, the network chip (ethernet) is commonly used within the i386 and PowerPC architectures, hence both architectures share a section of device drivers (which is chip dependent but not board dependent) and this portion is located within the `libchip` section. In the future, PCI chip code would also be moved into this section to be shared across supporting architectures.

Finally processor and board dependent device drivers are located within the `libbsp` section. Most device drivers are initially placed in this section. After generalisation across a set of boards and processors a significant portion of them are relocated to the `libchip` section. Sections of the device drivers which are heavily board dependent remain in `libbsp` section.

Network device drivers are an example of device drivers which are spread across most of the areas mentioned above. The interface section of network device drivers, since common, is placed within the `libchip` section. Low level read, write and configure operations are board and card specific, and hence are placed within numerous directories in the `libbsp` section.

Drivers dependent on their placement (within the source tree structure) would be implemented differently and are required to support a different set of functionalities. Since our project focuses on a specific processor (i386) and board (pc386), our SCI device driver implementation is expected to be located within the `libbsp` section. We will examine the requirements of the drivers in this section in detail in the next chapter.

5.7 Comparison of Device Drivers on RTEMS, VxWorks and Linux

Having discussed device driver implementations on each of the systems (RTEMS, VxWorks and Linux), this section presents an overview of the differences between device driver implementations on above systems.

Classification of the drivers on Linux and VxWorks systems are quite similar and heavily based on the *type* of target devices. Developers on such systems are presented with an IO Systems Model which allows them to integrate their device capabilities into the operating system. This IO System model was described in sections 4.2 and 4.3.

Classification of devices, based on their type, arises as a result of operating system, and more specifically the IO System, requiring the knowledge of communication with the hardware device and its device drivers. For example, a device is classified as a Block device if the IO System and user applications are required to communicate with the device on block basis (send and receive chunks of data at a time). In VxWorks and Linux systems, user applications communicate with the device and its drivers through this IO System.

The IO System simplifies a number of issues on the system, these including access and resource management as well as provision of device interfaces for user applications. The `/dev` directory on each system presents a list of devices available and/or registered on the system. Presence of the IO System Model in both, VxWorks and Linux, places certain rules and policies which drivers and hence their developers must obey and account for in the coding and delivering of their objectives through the device drivers.

RTEMS classification of device drivers are different to that of Linux and VxWorks's. Device drivers on RTEMS, as mentioned in the last section, are classified on the basis of *portability*, *dependency* and hence their *location* on the RTEMS tree structure. RTEMS does not have an IO System Model similar to that of Linux and VxWorks's. RTEMS's IO Manager provides basic support for an IO System model. However, it is mainly designed to manage the initialisation of device drivers at the system (BSP) initialisation stage. Our implementation, as described in the

next chapter, is dependent on the RTEMS IO Manager **if** user desires to perform the SCI driver initialisation at the BSP initialisation stage. RTEMS IO Manager dependency is *eliminated* if one decides to perform the SCI driver initialisation at the application level, similar to that of network drivers.

Absence of a strict IO Systems Model in RTEMS is beneficial to programmers as it removes a significant overhead from the system. This, however, implies that the developer of the device driver (on RTEMS) has to account for provision of some means of communication between the system and its drivers. The simplest approach is through exportation of libraries and header files. Users may import related libraries to communicate directly with the corresponding device drivers. This has numerous advantages over the IO System model used in Linux and VxWorks systems, outlined below.

- The device drivers have direct access to all system resources (higher customisation, flexibility and efficiency).
- The developer is flexible in providing relevant services and functionalities to the user applications (less overhead).
- The user has full direct access to the device driver functionalities (higher access speed).

Conversely, it challenges the developer in providing a suitable set of interfaces to the system, as well as performing efficient resource handling, management and routine execution on the system platform.

Chapter 6

IMPLEMENTATION

The implementation chapter is where all the acquired knowledge converges to develop an end result. Necessary background information was covered in Chapters 2 and 3. Detailed study and analysis of available resources was performed throughout Chapters 4 and 5. This chapter demonstrates how available resources (in our case SCI and RTEMS) can be combined and developed to achieve a RTCC (Real-Time Computer Cluster).

The first and major section of this chapter will focus on the implementation of the SCI drivers on RTEMS. Initially the position, role and status of the SCI drivers on RTEMS will be discussed. This will be followed by a description of the methodology used to insert SCI drivers in to RTEMS as a native section. Section 6.3 will analyse how the SCI can be initialised as part of the system initialisation process; it will also deal with the challenges, obstacles and issues involved within this process. The “SCI driver development on RTEMS” section details the steps and procedures involved in full implementation of the IRM section, of the SCI drivers, on RTEMS.

The second section of the implementation involves developing an application layer specific to the objectives of the project. The application layer uses the SCI drivers to provide “hardware based distributed shared memory” segments among cluster nodes, hence resulting in a real-time compute cluster. The next chapter details a simple application which employs the SCI driver and the application layer implemented to perform simple real-time cluster computing between two nodes.

The last two sections within this chapter discuss the debugging of the implementation and the SISC layer respectively. The debugging section outlines the tools used to perform debugging on the resultant implementation.

Throughout this chapter the SCI drivers mainly refer to the IRM section of the

SCI drivers as oppose to the whole SCI drivers (IRM and SISCi, both). This is due to the fact that IRM is the main and essential section of the SCI drivers, which is capable of providing full functionality regardless of the presence of the SISCi section. The last section of this chapter reasons and justifies the exclusion of the SISCi layer from the implementation. It is worth noting that, until recently, the SISCi layer did not exist as part of the SCI drivers, and only very recently has it been introduced to simplify some issues, in particularly the resource handling and management which users had to pay incredible attention to at the IRM drivers level.

6.1 *SCI drivers on RTEMS*

Support for hardware devices and high level functionalities comes with time in any development system. RTEMS developers continuously expand systems support for hardware devices based on the priority of a device within real-time embedded systems. In the case of the i386 processor and the pc386 board, support for console, hardware clock and timers were among the first to be addressed on RTEMS. RTEMS as an open source system also benefits from its external developers and users. Users based on their needs may also code drivers and provide hardware support for RTEMS, which upon contribution to the group results in an expansion of RTEMS's hardware device support. In fact, this project partially falls into the above category, where in order to achieve a Real-Time Compute Cluster one must first provide support for SCI interconnects on RTEMS.

Support for PCI cards on the i386/pc386 target environment, on RTEMS, had been limited to network (ethernet) cards until the contribution of this project. Network cards of class 3c509, ne2000 and wd8003 chipsets are currently supported on RTEMS. Network device drivers, however, due to their integration within other boards and processor architectures (particularly PowerPC), have been partially relocated to the `libchip` section. Furthermore, due to the use of FreeBSD network stacks and facilitation of GDB debugging over ethernet in RTEMS, network drivers are further integrated within RTEMS. In fact, network card initialisation within RTEMS does not take place at BSP initialisation stage, but is carried out as part

of the `rtems_bsdnet_initialize_network` directive.

Following observation of the above factors, it was decided to implement a stand alone set of drivers for the PCI-SCI cards. The device driver was designated to be placed within the `libbsp` section, with a minimal amount of RTEMS resource dependencies. The device driver functionalities are accessible by the RTEMS directives and user applications through a set of exported libraries and interfaces placed within the `libbsp` include section. Device driver initialisation may optionally take place at the BSP initialisation stage or at the application runtime stage by users request. While the latter approach simply implies that the user calls the initialisation routine, the first approach requires attention, which is detailed in Section 6.3.

In future, given that PCI-SCI cards are also compatible on PowerPC architecture, sections of this device driver may be relocated to the `libchip` section. New sections such as `librtcc`¹ may also be initiated to support high level cluster computing functionalities on RTEMS.

6.2 *Building drivers under RTEMS*

The makefile mechanism used within SCI drivers is far more different than ones used within the RTEMS source code. The compilation and making of SCI drivers were detailed in Section 4.5, similarly the making mechanism used in RTEMS was described in Section 5.4.2. Clearly the makefile mechanism used within our implementation of device drivers had to be compatible with the RTEMS makefile mechanism. This conclusion came as a result of realising that the SCI driver functionalities and directives had to be part of the RTEMS kernel library in order to be available for use by RTEMS directives and user applications. Hence, they must be present in the first stage of the compilation and build process (refer to figure 5.1).

Relevant source code was extracted from the SCI drivers source code and a sample set of stand alone device drivers was implemented. A high level GNU makefile mechanism (similar to the RTEMS makefile mechanism) was implemented and tested on the sample device driver files. Dependencies and functionalities were satisfied

¹real-time cluster computing library

by providing “fake” declarations and functionalities ². The sample device driver directory was labelled `sciauto`, to signify the SCI drivers and the use of GNU auto tools (high level makefile mechanism) within its make and build process.

Macro and compiler definitions (and flags) were other elements that required careful examination. SCI drivers, in order to maximise portability and efficiency, heavily use macro definitions throughout the source code. Necessary macro definitions were identified by thoroughly examining both the SCI driver makefiles and the source code. They were analysed first so as to not cause in any conflict with RTEMS, and hence they were included within the implemented source files and makefile mechanism.

Macro definitions used within our makefile mechanism are as follows:

```
DEFINES += -DEXPORT_SYMTAB -DOS_IS_RTEMS -DADAPTER_IS_PCI1
-DLITTLE_ENDIAN -DCPU_ARCH_IS_X86 -DMODULE -D__KERNEL__ -D_KERNEL
-D_X86_ -DUNIX -DIRM_INTERNAL -DDBG -DRTEMS_KERNEL_LIBRARY_SCI
-DRTEMS_BSP_SCI
```

Macro definitions are maintained throughout the source code as much as possible to allow for future expansion and portability of the drivers.

Finally, the sample device driver was integrated into the RTEMS `libbsp` section. Driver source code was placed in the `$RTEMS_ROOT/c/src/lib/libbsp/i386/pc386/sci` directory. The relevant makefiles from the sample device drivers were also migrated to this location with required modifications. Target BSPs’ `configure.ac` and `Makefile.am` files were accordingly modified to include compilation and building of the SCI drivers.

IRM’s main interface, `genif.h`, was also placed within the BSP’s library files, to redirect any references to SCI directives to the SCI drivers directory. Further enhancements include integration of the `--enable-sci` option into the RTEMS’s building mechanism. This involved coding `aclocal` files as well as making modifications within the main RTEMS scripts. It allows users to specify whether to include

²required functions were declared and implemented to return *true* at all times without performing any action

SCI drivers into the RTEMS kernel library or not at the compile time ³. Other relevant files (such as the target BSP's wrapper files) throughout the RTEMS were also modified to account for insertion of the new module into the i386/pc386 BSP as well as the new `--enable-sci` option.

6.3 *SCI Initialisation within BSP*

Initialisation of device drivers within the BSP initialisation routine is, in most cases, the simplest and most preferable method of initialising devices from the user perspective. Unfortunately in most cases, simplicity for the user implies extra attention and development from the developer's point of view. The RTEMS initialisation routine was detailed in Section 5.5. Device driver initialisations are carried out within the `rtems_initialize_executive_early` directive. Device driver initialisation routines are called using dynamic reference links from the pre-configured `IO_Driver_address_table`.

The `IO_Driver_address_table` is extracted from a higher level `BSP_Configuration` table. The `BSP_Configuration` table is the default configuration table provided by RTEMS, defined within the `$RTEMS_ROOT/cpukit/sapi/include/confdefs.h` file.

The SCI drivers library is linked into the RTEMS image file, subject to the user defining the `CONFIGURE_APPLICATION_NEEDS_SCI_DRIVER` macro.

```
#ifdef CONFIGURE_APPLICATION_NEEDS_SCI_DRIVER
#include <scidrv.h>
#endif
```

The following code is a *simplified* version of how the SCI and other device drivers are included in the `Device_drivers` table if requested by the user.

```
\#ifndef CONFIGURE_HAS_OWN_DEVICE_DRIVER_TABLE
rtems_driver_address_table Device_drivers[] = {
\#ifdef CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
```

³by default SCI drivers are not compiled into the RTEMS kernel library

```

        CONSOLE_DRIVER_TABLE_ENTRY,
    \#endif
    \#ifdef CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
        CLOCK_DRIVER_TABLE_ENTRY,
    \#endif
    \#ifdef CONFIGURE_APPLICATION_NEEDS_RTC_DRIVER
        RTC_DRIVER_TABLE_ENTRY,
    \#endif
    \#ifdef CONFIGURE_APPLICATION_NEEDS_SCI_DRIVER
        SCI_DRIVER_TABLE_ENTRY,
    \#endif
    \#ifdef CONFIGURE_APPLICATION_NEEDS_STUB_DRIVER
        DEVNULL_DRIVER_TABLE_ENTRY,
    \#endif
};
\#endif /* CONFIGURE_HAS_OWN_DEVICE_DRIVER_TABLE */

```

The `CONFIGURE_HAS_OWN_DEVICE_DRIVER_TABLE` macro is used to indicate that the user is supplying a customised table. The `SCI_DRIVER_TABLE_ENTRY` is defined as follows within the `scidrv.h` file.

```

#define SCI_DRIVER_TABLE_ENTRY \
{ sci_initialize, sci_open, sci_close, \
  sci_read, sci_write, sci_control }

```

sci_initialize Initialisation procedure invoked as part of the BSP initialisation routine ⁴

sci_open Open request procedure, invoked by the open directive of the IO manager, `rtem_io_open`. Not required for SCI, but can be used to open a SCI session with a target node.

⁴called within the `rtems_initialize_executive_early` directive

sci_close Close request procedure, invoked by the close directive of the IO manager, `rtem_io_close`. Not required for SCI, but can be used to close an open SCI session with a target node.

sci_read Read request procedure, invoked by the read directive of the IO manager, `rtem_io_read`. Not required for SCI, but can be used to receive data from a target SCI node.

sci_write Write request procedure, invoked by the write directive of the IO manager, `rtem_io_write`. Not required for SCI, but can be used to send data to a target SCI node.

sci_control Special functions procedure invoked by the control directive of the IO manager, `rtem_io_control`. Not required for SCI, but can be used as an SCI IOCTL directive.

All of the above routines require major, minor and argument variables. The major variable is a unique number that identifies the target hardware (e.g. SCI) among all other hardware present within the system. The minor variable is an index variable reflecting the device number *if* multiple devices of the same hardware type exist within the system (e.g. SCI switch nodes with multiple SCI cards), otherwise it is set to zero. The argument variable is an object reference pointer on whose nature and type the developer decides, depending on the data required for the implementation of this directive.

Initialising drivers as part of the BSP initialisation routine has some constraints that limit full initialisation of the SCI cards at the BSP initialisation stage. The first and most important factor is the concept of the “mono task execution” system. As noted earlier, RTEMS does not allow for multi-tasking until invocation of the `rtems_initialize_executive_late` directive. Hence, even if the SCI “timeout job handler” task is started through the SCI driver initialisation, it would not be scheduled for execution until initiation of multi-tasking within the system. Similarly watchdog (VC & LC watchdogs) routines cannot be initiated until activation

of multi-tasking within the system.

Board interrupts are also disabled at the start of the `rtems_initialize_executive_early` directive. While the SCI interrupt handler may be attached through the SCI initialisation process, it would be inactive until board interrupts are re-enabled. Finally, the SCI drivers should note the unavailability of certain RTEMS resources prior to completion of the system initialisation process. Since the SCI drivers cannot assume any order in initialisation of drivers, they may not have any dependencies on the routines supported by other drivers or ones which are unavailable at runtime stage. The above issue forced us to code the drivers with the minimal amount of RTEMS Managers required.

The implemented SCI driver dependencies are limited to the RTEMS IO, Task, Timer, Interrupt Managers, as well as POSIX Message and Semaphore Managers; all initialised and available prior to SCI initialisation. For debugging purposes the console driver was also included within the dependencies list, but in return initialisation of console drivers prior to SCI routines was guaranteed through custom configuration of the RTEMS initialisation routine.

Debugging at BSP initialisation level is extremely tedious since it involves re-compilation, making and building of the entire RTEMS kernel library and image file per debugging session. Restrictions and constraints present within the BSP initialisation stage clearly block the full initialisation of SCI cards. The initialisation section may be divided into two sections: one is the configuration of the card at the BSP initialisation stage; the other, activation of the card at the user application stage.

The last element blocking any further developments in this area was the presence of a discovered gcc related bug within the RTEMS tools set, detailed in Appendix B. Having realised that the initialisation of the SCI cards as part of the BSP initialisation routine was almost impractical, the main focus was steered towards the alternative option of initialising SCI drivers through user applications. While providing flexibility for the user applications⁵ and considerably easing the debugging

⁵i.e. user may initialise SCI drivers whenever he/she desires so

process (now only involving make and building of the RTEMS image file), this approach resulted in the SCI card initialisation process being somewhat similar to the network card initialisations, which also take place at the user application level.

6.4 *SCI driver development on RTEMS*

The development of SCI device drivers on RTEMS accounts for a significant portion of this project. Having realised that the IRM section of the SCI drivers was the basis and most important element of drivers, the focus of attention was steered towards this element in the first section of the project. We shall discuss the SISI section later in this chapter. The result of implementing the IRM section on RTEMS was roughly 3.7MB of source code (approx 110,000 lines). However, we do not intend to discuss the 110,000 lines of code, but rather present an overview of the approach taken to achieve this objective.

Prior to reaching this objective, it was *correctly* predicted that the task would be of massive scale and extreme complexity. Hence, utilising project and time management skills, the task was divided into a series of stages; each stage targeting a local objective, while all together collaboratively approach a global objective.

This approach not only eased the development process significantly, but also ensured the correctness and fineness of the approach towards achievement of the objective. We shall be detailing each stage, assessing inputs and outputs of each. The first three stages (the analytical sections) provide guidelines and a basis for the latter two sections, which are purely coding and development oriented stages.

6.4.1 *Foundation Analysis*

Foundation analysis was the first stage of the development process. The local objective of this stage was to provide a set of code that could be considered as a solid bases for SCI driver development on RTEMS. We shall refer to the outcome of this stage as the “foundation core”. SCI drivers were the only resource required throughout this stage.

SCI source code was analysed, and relevant sections were extracted according to the needs. The main elements extracted from the SCI source code are as follows:

- IRM functionality core (shared code among all boards)
- PSB chip related sections (source codes related to the PSB32 chip specifically)
- LC chip related sections (source codes related to the LC1 chip specifically)
- Operating system related codes (Linux, LynxOS and VxWorks source codes and interfaces)

The extracted sections were further scanned for any undesirable dependencies. Irrelevant code and functionalities were eliminated to reduce the scale of the code for processing in all following stages. While all mentioned sections were reduced, most were eliminated from the IRM functionality core, due to providing extra functionalities for PSB64 and PSB66 cards. Conversely, in some cases extra code had to be included to satisfy dependency requirements within the foundation core.

With regard to the operating system section, source code from Linux, LynxOS and VxWorks was used. Linux was chosen for its support and its “open source” feature. VxWorks was included since it was the closest match for the RTEMS within the supported operating systems. LynxOS was chosen as an alternative operating system, which is both real-time based and Linux compatible. Due to the lack of SCI driver documentation, VxWorks and Linux code were analysed throughout this project, and if they were in strong *disagreement*, the LynxOS code was analysed to reach a clearer understanding of the issue.

6.4.2 Functionality Analysis

Functionality analysis is a process that is carried out on the outcome of the last stage (Foundation analysis). The local objective is to identify functions, their role, dependencies and inter-relationships within the foundation core. Functions were divided into groups, referred to as “segments”, based on their location within the

foundation core. Segments, however, were analysed with the aid of examining the wider set, this being the full IRM section, rather than just the foundation core. Furthermore, a minor objective of this stage was to further eliminate any unnecessary function or source code.

The output of this stage was important in the development of the SCI drivers on RTEMS. Through this stage segments were classified under three categories:

Solid segments Segments with functions holding unique coding across the IRM section, regardless of the configuration ⁶ set used. They should not be modified under any circumstances and generally provide internal core functionalities of the driver, independent of any factors.

Soft segments Segments with functions that are present (with same functionality) throughout the IRM section for almost all configurations, but their implementation varies depending on the factors chosen within the configuration set.

Loose segments Segments present to generally satisfy dependencies and provide minor functionalities. Their existence is dependent on the configuration used and in most cases the implementation is specific to a certain factor used within the corresponding configuration set.

Segments were classified by inspection and by the use of a set of logical rules. The presence of dynamic reference links and calls made this process harder than anticipated.

A summary of outcomes and actions in each case is presented below.

Solid segments Some portions of IRM internals, PSB and LC related sections were classified under this category. They are to be included without any modifications in the foundation core.

Soft segments Most of the operating system dependent section plus some of the IRM internals, PSB and LC related sections were categorised in this branch.

⁶a specific set of operating system type, architecture, PSB and LC chip model is referred to as configuration

Functionality of all functions is to be understood. Functions must be re-implemented to be made RTEMS compatible, while maintaining their functionality and interface with other functions.

Loose segments Some of the operating system dependent sections plus a small portion of PSB and LC related sections fell into this category. They are to be eliminated as much as possible, and replaced with new loose segments which are RTEMS specific *if* required.

Conclusions made from this stage are important in the achievement of our global objective. While solid segments should not be altered, both the soft and loose segments require modification and developments. One must realise the functionality of the soft segments and develop the corresponding functions on the target operating system (RTEMS). Loose segments require little or no attention, and, while techniques in achievement of certain objectives may be learnt from them, they may be almost entirely eliminated.

It must be noted that realisation of the functionality of routines was by far the hardest challenge within this stage, reaching its peak when attempting to digest the functionality of some VxWorks implemented functions which were not only of significant complexity, but also utilised VxWorks system calls whose source code and function was beyond our reach.

6.4.3 *Implementation Analysis*

Functions within the segments identified as soft or loose should be re-implemented to perform the desired functionalities on RTEMS. This stage analyses the resources and facilities available to a developer within RTEMS to perform such a task.

Following study of RTEMS and examination of Linux and VxWorks implementation of target routines, a set of goals and aims was developed. These were targets to be met throughout the development process. The list is outlined below.

- Given that portions of the SCI drivers may be desired to be initiated at the BSP initialisation level, dependencies of the developing routines must rely on

available resources at the BSP initialisation stage ⁷. Note that this may not be possible for all routines (such as initiation of the timeout job handler, which introduces an unpermitted multi-tasking). Hence, only a portion of SCI drivers can be initiated at the BSP initialisation stage at any time.

- Maximise load handling, by developing the relevant routines such that they could be executed in parallel, independently. Furthermore, minimise the critical sections within the “favourable routines” ⁸.
- Utilise multi-tasking to allow for a more flexible system. Variable priorities may also be used to give preferential treatment to critical section of routines.
- Given that interrupts preempt any executive task (even ones of highest priority), polling should be considered as a suitable solution for lower priority events.
- As a rule of thumb within real-time systems, avoid blocking ISRs and minimise processing within ISRs as much as possible.
- Implementations are encouraged to employ the RTEMS API in achievement of their objectives. RTEMS API as an all-time available library, not only ensures the operation of routines under all circumstances, but in most cases this also results in minimum system overhead in execution of the routines.
- Implementations should perform their functionalities (properly) with a minimal amount of executive operations. In real-time systems the user takes responsibility of resource management, resulting in a much more optimised and deterministic system.

⁷available resources at the BSP initialisation stage were discussed earlier in this chapter

⁸routines which are executed on frequent basis either by users request or occurrence of frequent event, signal and/or interrupts

- Implementations should minimise assumptions made regarding the status of the system at various executive stages. For example, the SCI driver initialisation routine should account for the possibility of the PCI bus not having been initialised yet, if invoked as part of the BSP initialisation routine !
- Considering the absence of the virtual memory, user/kernel mode space and swap space concepts within RTEMS, routines associated with such concepts should preferably be eliminated.
- Since VxWorks itself is an RTOS, its implementations may be used as a guideline for RTEMS development of routines.

Portability of applications across different operating system platforms is of significant importance. This is especially important for applications designed for longevity, where the hardware and software infrastructure may change during the application's life cycle. Dolphin SCI drivers have also been designed with portability in mind. In real-time systems, however, where predictability and low overhead are important, portability is often sacrificed [25]. While every effort was made to maintain the portability of the SCI drivers throughout the implementation process, the highlighted goals were of higher priority and were permitted to influence the portability of the end result, if necessary⁹.

Throughout the next two development stages, we will briefly touch on techniques and approaches used to achieve the various goals listed above.

6.4.4 *First stage development*

The SCI initialisation routine is a significantly large routine. Implementation of this routine itself was considered as the first stage of development.

The `sci_init` routine was chosen as the start of the initialisation process. The routine (detailed in Section 4.6.1) was studied in conjunction with the results obtained from the functionality analysis stage. Relevant¹⁰ functions, which were clas-

⁹yet another difference between normal and real-time systems

¹⁰in this context meaning functions involved in the SCI initialisation process

sified as part of the soft segments, were re-implemented on RTEMS. Throughout the implementation process, the goals and aims set at previous stages were constantly targeted.

In order to allow for the use of major sections of the code within the BSP initialisation stage, the implementation was made dependent on only three RTEMS resource managers: IO, Task, and Timer Managers, and two POSIX managers: Message and Semaphore Managers. All other functionalities were implemented at RTEMS C standard library level, minimising dependencies and in most cases maximising execution speed of implemented routines.

The SCI initialisation routine is a one time routine, so the issue of “load handling” is irrelevant in this section. Multi-tasking was utilised wherever appropriate. The timeout job handler is a perfect example of this, where functions submit a section of their job to the timeout job handler and facilitate the next requests. The SCI interrupt does not block other ISRs and was developed to minimise processing within its ISR. The SCI interrupt handler (ISR) processes the ISTAT register, and submits the necessary jobs to other routines, minimising the processing within the ISR itself. While VxWorks and Linux implementations had certain influences over the implementation, developments were primarily based on functionality analysis and relevancy of routines within the RTEMS.

The outcome of this stage was a complete and correct SCI driver initialisation routine for RTEMS.

6.4.5 Second stage development

The second stage of development was a continuation of the first stage. Second stage development targeted the remaining functionalities of the SCI drivers.

The local objective of this stage was implementation of all of the necessary (remaining) functionalities for the ported foundation core on RTEMS. This stage resulted in implementation of various sections of the foundation core plus a major section of the operating system dependent interface of the SCI driver.

Since the routines addressed within this stage were guaranteed to be addressed

at the user application level, multi-tasking, optimisation and efficiency were heavily exploited. A large number of simple routines were implemented. Parallel execution and multi-tasking were considered wherever appropriate.

Simplicity of implementations ensured deterministic behaviour of functions - one of the most desirable factors within any RTOS. Use of simple, low level system routines in implementation of functions, in most cases, ensured optimisation. A routine was considered optimised *if* the number of its run-time instruction executions on the processor, to deliver its corresponding functionality, was reduced.

One of the major areas where the above issues were utilised was the memory management section of the SCI drivers. The use of RTEMS Partition and Region Managers and POSIX Memory Managers was avoided. Instead, optimised and simple functions were implemented using the RTEMS C standard library to address the specific functionalities necessary.

The outcome of this stage was the full SCI drivers implemented on RTEMS. Some statistics on the overall result are provided below.

Source code size 3.7MB, approx 110,000 lines of code

C source files 46 files

C header/library files 88 files

Re-Implemented C source files 16 files

Re-Implemented C header files 9 files

RTEMS libraries used assert.h asm/system.h stdarg.h stdlib.h stdio.h
 asm/io.h bsp.h fcntl.h errno.h sys/types.h sys/conf.h sys/types.h
 pcibios.h irq.h mqueue.h time.h semaphore.h unistd.h time.h
 rtems/error.h rtems/score/coremsg.inl confdefs.h string.h assert.h

6.5 Application layer

The development of SCI drivers on RTEMS was certainly the major portion of the project and a great achievement. However, without an application layer, SCI drivers by themselves would not result in a real-time compute cluster !

The SCI driver development had a global objective of full driver development on RTEMS. This section, however, focuses entirely on the objectives of this project. We had discussed in earlier chapters how SCI provides various useful functionalities on clusters. The functionality that this project is interested in achieving is the “hardware based distributed shared memory” among the cluster nodes.

Specific nodes on a cluster topology will share segments of their memory space with other SCI nodes in the topology. We shall call these type of nodes “servers” since they offer their memory resource to the SCI network. Nodes that map the offered memory segments as part of their memory address space are referred to as “clients”. All nodes access the shared memory by referring to a specific address in their local memory address space.

Figure 6.1 illustrates a simplified process of memory sharing among SCI nodes. Following this procedure, the server and client nodes may access the shared memory segment by accessing a certain memory address on their *local* address space.

The server nodes need to perform the following steps to share their memory segments within the SCI address space.

sci_create_segment Creates a unique segment on the local node

sci_export_segment Makes the segment available for remote access (the local node can access the segment after this procedure)

sci_set_local_segment_available Permits nodes within the SCI topology to access the exported segment

sci_local_kernel_virtual_address Returns the local virtual address¹¹ of the locally created segment

¹¹in RTEMS this would same as the allocated I/O address

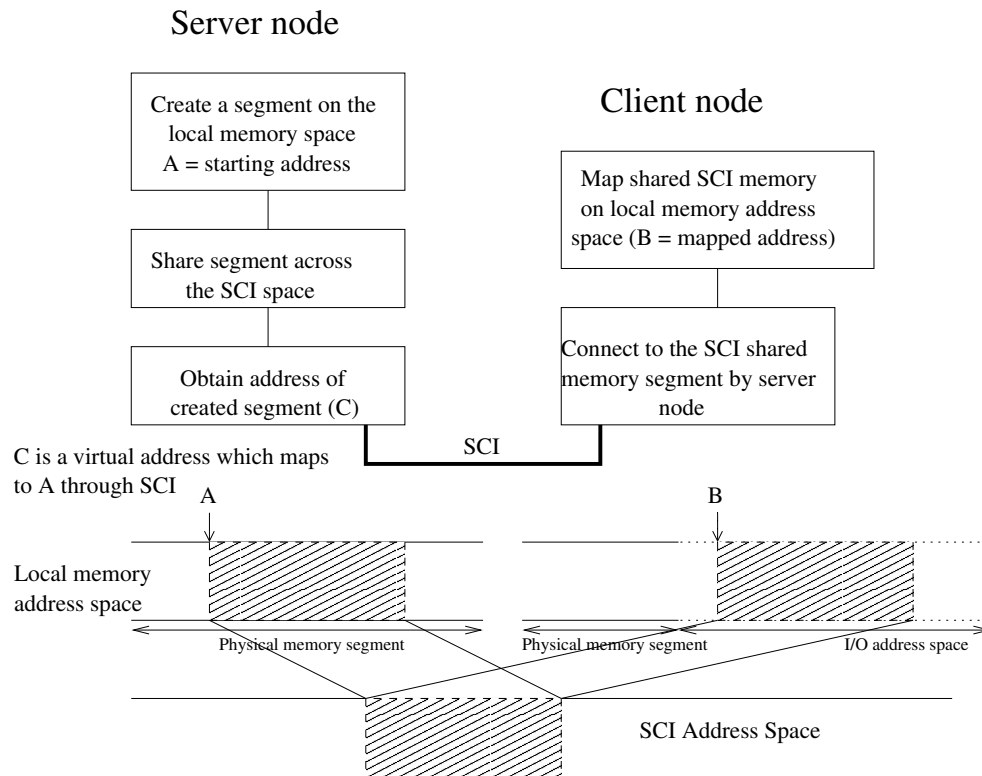


Figure 6.1: SCI Shared Memory Segment

At segment creation stage, the user provides a handle to a callback function that handles incoming connect requests from remote nodes. The handle may be used to carry out client verification and other appropriate operations.

The attributes provided by the user at the segment creation stage determine the behaviour of SCI regarding local segments. Special features or functionalities (briefly discussed in Chapter 2) may be enabled on the local segments. The use of such factors may introduce driver (software) interventions, which are undesirable within a real-time system. Hardware processing in most cases results in a more deterministic system.

For the purposes of this project, however, attributes were analysed and set so as to minimise any processing by the SCI card. In fact, the appropriate set of attributes were adopted to fully eliminate the intervention of software (SCI drivers) after setup of the RTCC.

The following attributes were provided for the segment creation function to en-

sure complete hardware processing of the packets.

```
sci_create_segment(NULL,
                  0xBD,           // Module id
                  0x10000001,     // Segment id
                  0,              // Attributes
                  size,           // Segment size
                  local_segment_msg_callback,
                  (void *)0xbd1,  // Callback arguments
                  &local_segment_handle);
```

The memory segment is a physically locked memory segment within the main memory. Physically locked implies that this memory segment is non-swappable and has a fixed location for the duration of its existence.

The physical address of the created segment, A , is not usually provided to the user. Instead the I/O address, C , of the SCI card is provided first. If no special configurations have been setup on the SCI card regarding the segments, the card simply forwards the packets to the main memory, otherwise it performs the necessary operations and then forwards the packets to the main memory.

The procedure is carried out fully through the SCI card hardware. Limited segment information (VC¹² information) and the ATT (Address Translation Table) on the SCI card allow hardware level handling of the packets.

The client nodes perform the following steps to obtain access to the shared memory segment on the SCI address space.

sci_connect_segment Connects to a segment on the SCI address space.

sci_map_segment Allocates and maps I/O space for the connected segments on the local machine.

sci_kernel_virtual_address_of_mapping Returns the local I/O address of the mapped segment.

¹²Virtual Channel

Similar to the creation of a segment, the connect routine has its own complexities. It initiates the connection from the local side and transmits a request for connection to the remote node. As an argument, this routine must also be provided with a callback reference, which is invoked upon response of the remote node. The connection is not established until the callback function indicates so. Note that the connect routine returns following transmission of the request packet, hence return of the function by no means signifies the completion of the objective.

The following arguments were used from the client side to minimise SCI interference.

```
sci_connect_segment(NULL,
                    target_node_id,
                    adapter_id,
                    0xBD,           // Module id
                    0x10000001,    // Segment id
                    0x00000001,    // INCLUDE_SHORT_CONNECT_MESSAGE
                    remote_segment_callback,
                    (void *) ptr_array, // Callback arguments
                    &remote_segment_handle);

sci_map_segment(remote_segment_handle,
                0,      // Flags
                0,      // Offset
                sci_remote_segment_size(remote_segment_handle), // Size
                &remmap);
```

After execution of the application layer, all nodes are provided with a local pointer that points to the globally shared SCI space. Users may treat this address pointer similarly to a local address pointer. While users perform normal read and write operations on the memory space addressed by the pointer, the SCI processing is hidden at hardware level.

It must be noted, that the above functions perform a certain task and return

following the execution of that. The functions return irrespective of the remote nodes' reply to the issued request. They do not *wait* for the reply, rather they provide a *callback* routine which is called once reply is received. This approach results in high determinacy and predictability of the routine executions. System may execute any of the above functions, knowing that they will return within a fixed time duration. It may schedule other processes, following the request, until reply is received. The callback routines are also short and predictable. Overall, to perform a connect operation, the user and the system know it will take a *fixed* amount of processing time, this accounting for the execution of the function and the callback, this demonstrates determinacy and predictability of execution of the above routines in RTEMS.

This application layer provides a significant resource for real-time cluster computing. It has been specifically coded to provide “hardware based distributed shared memory” segments among cluster nodes. Furthermore, a useful 2-stage lock mechanism was developed for synchronisation and mutual exclusion services in the cluster. The lock is 2-stage based to eliminate the need of atomic-execution on the remote nodes, hence normal `load/store` operations are used to perform the locking. FSP (Finite State Process) model and LTSA (Labeled Transition System Analyzer) were used to ensure the implementation of a fair, fail-safe and suitable lock for the purposes of the project.

6.6 Debugging

Testing and debugging of implementations are always an important section of projects and usually the most time consuming element of project work.

Debugging in this project has been more difficult than any experienced before. While debugging device drivers requires significant amounts of information for troubleshooting purposes, it is quite hard to extract this information from the real-time operating systems (in particular the RTEMS).

It was decided to initially use a simple console debugging mechanism and if further needed, GDB over ethernet or serial to be approached. It was also decided

to perform SCI driver initialisation at application level for the following reasons.

Only choice It was clear that the SCI initialisation routine could not take place at the BSP initialisation stage, hence it was decided to be performed at the application level. This placed the SCI driver at a class similar to the network card drivers in the RTEMS, as well as offering a range of useful features.

Time saving The RTEMS kernel library did not necessitate rebuilding for each test session, rather only the RTEMS image file had to be re-built as a result of modifications made in the application code.

Console debugging tool available Given that the driver initialisation routine is fully performed at application level, the console device is fully in service for use in the debugging process.

Flexible debugging Using the console debugger one has a more flexible choice in debugging, execution and flow of the program. (i.e. can print relevant parameters anywhere deemed necessary and can hold program execution, using the `getchar` routine, at any stage desired)

Fortunately, console debugging was sufficient for this project. Proper management and implementation seemed to have saved a significant amount of time within the debugging process.

The discovered gcc bug, however, seemed to be the only element that required extra attention and debugging. Essential debugging was carried out through the execution and source code to investigate the source of the problem. When it was discovered that the source of problem was a gcc related issue within RTEMS build tools the issue was considered out the scope of the project and reported to the RTEMS maintainers. Once again, Appendix B details this issue.

6.7 *SISCI layer*

In Chapter 4 we highlighted two sections of the SCI drivers, IRM and SISCI. The connection between the two was examined as well as examining each individually.

It was noted that the IRM was the main driver section with direct interaction with hardware, and SISCO was a higher level driver using IRM to provide resource management, simplified programming interface and connection between user and kernel mode executions.

This chapter, however, had focused purely on the IRM section. SCI driver implementation was only considered in the context of the IRM section as well. This section of this chapter details our reasons behind the decision not to include SISCO as part of our implementation. Each objective of the SISCO will be addressed and its attainment will be analysed.

The first and most important role of SISCO was to handle user mode calls within kernel execution mode. This involved copying/converting user data, request and addresses. This factor, however, is completely useless in RTEMS. User applications and RTEMS both execute in kernel mode, hence there is no need for a layer to connect the application mode execution to the system mode execution. Application of the SISCO layer, here, is considered as an overhead. IRM routines, if supplied with the address, can access user data at any time. The user address space, is the same as the RTEMS address space, both being the physical address space of the system, due to the flat-memory scheme employed in RTEMS. Furthermore, upon the application program including the necessary libraries at the link time, it may execute any instruction in much the same way as the RTEMS operating system. Linux, however, as an operating system which is open to a wide range of users, includes and applies user mode protection. VxWorks system facilitates user mode protection, which if desired may be applied to the system. Though application of user protection mode in VxWorks this is not recommended, SCI drivers have covered the general case of accounting for this through SISCO layer.

The second role of SISCO was to simplify the programming interface, hiding substantial amounts of coding from user applications. This, also, is an undesirable element. The extra code and overhead involved in the SISCO layer targets fail safety, error checking and resource management mechanisms. While useful, a real-time system developer in most cases prefers to overtake such tasks personally, since he/she

can optimise a system based on known relevant factors rather than general issues. A real-time system developer would, however, appreciate the existence of a real-time cluster computing library that eases relevant issues, such as sharing segments of main memory to the SCI network, mapping segments from SCI address space, implementation of locks on shared memory segments, implementation of shared objects or message queues among cluster nodes. Hence, the SISCO layer implementation was replaced with the implementation of a real-time compute clustering library (LIBRTCC). Within the implemented library, DSM as well as locking mechanisms are supported to facilitate the synchronisation and the mutual exclusion services within a cluster.

The last task handled by the SISCO layer is the resource management, which, as described in the last paragraph, is not a demanding element. The SISCO layer overall, on a real-time system, is considered as an overhead layer, which was not at any stage justified throughout this project to be implemented on RTEMS. In summary, the IRM section was implemented on RTEMS, which provides *full* support for all capabilities of the PCI-SCI card. However, the SISCO layer was not implemented, and as a result users obtain more control over the performance of the system. Instead, a much more useful library for real-time systems was developed, to perform useful clustering functionalities with minimum overhead, optimised and once again flexible to users objectives.

Chapter 7

EVALUATION & CONCLUSION

In this final chapter, the end result of the project work is evaluated. Suggestions for further work in-line with this project are also presented.

7.1 *Implementation*

SCI drivers were fully implemented on the RTEMS. Additionally, a new library (`librtcc`) was coded to enable DSM based real-time cluster computing on the resultant RTCC. A 2-stage lock mechanism was also implemented and included in the `librtcc`, in the hope that it can be used and possibly generalised for future purposes. The API of the full implemented package is provided in Appendix A.

A sample real-time cluster computing application was developed to examine the projects implementation. The program first invokes the SCI device driver initialisation routine. The existence of a bug prevented full execution of this stage. However the cause of the bug was discovered and a “work around” solution was developed to allow testing and full execution of the implementation until the bug in the RTEMS is completely removed.

The application uses the second section of the project (the RTCC application layer) to obtain a hardware based shared memory segment in between the cluster nodes. Following this step, the system was ready for real-time cluster computing and hence a simple cluster data processing application was deployed on the cluster.

The application employed two cluster nodes, one transferring (writing) data to the shared memory segment and the other retrieving (reading) data and printing on the console screen. The application also employed the implemented 2-stage lock mechanism to ensure mutual exclusion for access to the shared memory segment. In order to further complicate the scenario and slow the process (for the user to be

able to observe the results on the console), sleeping mechanisms were used to place the task executions out-of-phase with each other.

Print functions were extensively used throughout the program to fully monitor and control the execution of the application. While they indicated how successful the implementation was, step by step, the absence of the debugging print info suggested the lack of software intervention within the last section of the application. Hence, achievement of a complete hardware based distributed shared memory, among the RTCC nodes.

This project's implementation is hoped to start a new area of research development and activities. Applications of RTCC systems are foreseen to be wide and extensive in the near future. This is particularly in the area of networking where voice and modern data systems are under close examination to be merged under 3G or 4G standards. New telecommunication systems are required to have real-time capabilities (to satisfy multimedia features) as well as high performance computing power. At present, the availability of the already implemented OpenH323 protocol on RTEMS allows for replacement of highly loaded H323 Gateways with the RTCC H323 Gateways. Further applications of the RTCC systems would include support of high level OS functionalities on hard real-time basis; possible on RTCC due to the high level of distributed computation power available.

7.2 Project

Success of a project is, in most cases, determined by the success of the end result. An engineer should examine his/her approach to the achievement of the end result, as well as comparing his/her work against the finest models present in industry.

The closest industrial implementation available to the achieved result is the real-time compute cluster comprised of VxWorks as the real-time operating system and SCI as the cluster interconnect. The end result of this project is considered comparable, if not superior, to the equivalent VxWorks implementation for the following reasons:

- RTEMS is a "licence free", "open source" based system, freely available world-

wide. VxWorks, however, is an expensive proprietary system, with little info on its internals.

- RTEMS is a fast developing system, with frequent updates, keeping up with the latest tools and highest technology standards.
- The performance of RTEMS is comparable and in some cases better than VxWorks (as analysed in Section 5.2).
- SCI drivers implemented on the RTEMS are finely tuned for RTEMS ¹.
- SCI drivers on the RTEMS are optimised and are in direct interaction with core sections of the operating system, unlike the Linux and VxWorks implementations which are confined to an IO Systems Model and structure.
- Some differences between RTEMS and VxWorks (such as unconditional lack of user/kernel mode executions and also unconditional full access to the flat memory scheme within RTEMS) were exploited to achieve a higher optimisation and lower SCI drivers overhead on RTEMS than the VxWorks implementation ².
- Provisions have been made to allow for both, partial initialisation of the SCI drivers at the BSP initialisation stage and dynamic load/unloading of the SCI drivers at the application level stage. This flexibility is not offered by the SCI driver implementation on VxWorks.
- Unlike the VxWorks implementation, the SISCO layer was completely eliminated and replaced with a much more useful library (real-time cluster computing library) on RTEMS.

¹having the adverse effect of breaking compatibility with the Dolphin SCI drivers, as happens regularly in real-time system developments

²VxWorks has support for both user mode execution and virtual memory space. They may be disabled through configuration of the system, but the SCI drivers in order to function on either case, support these issues to a variable extent. This is mainly seen throughout the SISCO layer

The following factors are worthy of consideration when one evaluates the approach, management and work employed throughout this project.

- Non-working SCI drivers on Linux ³ and unexaminable drivers on VxWorks ⁴ were studied to achieve a *correct* and *working* set of SCI drivers on RTEMS.
- The developed SCI driver on RTEMS is currently the largest set of drivers on RTEMS (i386/pc386 BSP).
- The project objective was achieved without any loss in functionality or presentation of any disadvantages to the system at software level.
- Though the implementation was first targeted at the latest RTEMS Stable Release (RTEMS4.5.0), implementation was modified and steered towards the latest snapshot of the RTEMS to develop a package compatible with the latest system with all its improvements and new services.
- Unexpected threat elements (such as the gcc bug and the unexpected structural change of RTEMS from the latest stable version to the latest snapshot version) were identified and worked around within the projects time frame, without any influence on the end result.

7.3 *Future work*

This project is believed to provide solid foundations for a new range of applications, a new research area and numerous activities within the near future. A number of issues related to this project, which could be considered as future work are presented below.

- This project had focused on the PCI-SCI D310 model cards. Support for other SCI card models can be incorporated into the implementation model and this

³in fact, non-functionality of the SCI drivers (the October 2002 distribution) on Linux resulted in a student changing the focus of his project away from SCI this year !

⁴Unexaminable, since no VxWorks systems were available for analysis, testing and drivers could not be built on a non-VxWorks system

involves extraction of the related source code from the original Dolphin SCI driver source codes and its insertion into the implementation. Sections of the current implementation would also need modification since they have been made adapter card specific through the implementation process.

- The testbed for this project consisted of two nodes. Even though the implementation can support any number of nodes within a cluster, load analysis must be carried out with a higher number of cluster nodes. The implementation provides hardware based communication between cluster nodes, but the performance is lower than a single supercomputer system due to the existence of the PCI bridge and SCI links bottlenecks ⁵. The effects of traffic, congestion and an increase in the number of nodes must be analysed on the cluster. These issues are highly critical within a real-time system.
- Implemented SCI drivers on RTEMS support full functionality of the hardware. Series of such functionalities were used to achieve the required objective of the project (within the application layer). Other functionalities may be used for different task or objectives. Hence extension of the new `librtcc` (real-time cluster computing library) is recommended as further future work. Some useful functionalities which could be developed are the extension of the IO bus from one node to another or utilisation of the DMA facilities within the RTCC.

⁵source of performance loss is mainly the low speed and bandwidth of the PCI bus

BIBLIOGRAPHY

- [1] http://cmp.ameslab.gov/cmp/cluster_computers
- [2] http://microcontroller.com/wp/DeviceDrivers/device_drivers.htm
- [3] <http://that.gsfc.nasa.gov/osgroup/benchmarks.html>
- [4] <http://www.beowulf.org>
- [5] <http://www.cs.tcd.ie/Michael.Manzke/research.html>
- [6] *Comp.realtime*: FAQ.
- [7] A. Agarwal, D. Chaiken, G. D'Souza, K. Johnson and D. Kranz. *The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor*. In Proceedings of the Workshop on Scalable Shared-Memory Multiprocessors. Seattle, USA, June 1990. Kluwer Academic Publishers.
- [8] Alessandro Rubini. *Linux Device Drivers*. February 1998
- [9] Alexander Reinefeld and Jens Simon. *A High Performance Compute Cluster with SCI*.
- [10] Antoine Colin, Isabelle Puaut. *Worst-Case Execution Time Analysis of the RTEMS Real-Time Operating System*. IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France.
- [11] Avionic Systems Standardisation Committee. *Evaluation of Real Time Operating Systems - The Role of Standards*. March 1997.
- [12] Bjarne G. Herland. *SISCI - Implementing a Standard Software Infrastructure on an SCI Cluster*. Parallab, University of Bergen.

- [13] Bryan Henderson. *Linux Loadable Kernel Module HOWTO*. 21 May 2002.
- [14] David MacKenzie. *GNU Auto-tools*. 2001.
- [15] Dolphin Interconnect Solutions. *PCI-SCI Adapter Card D320/D321 Functional Overview*. version 1.01, November 30 1999, part no.: D1950-10299.
- [16] Dolphin Interconnect Solutions. *SISCI API User Guide*.
- [17] Eleftherios Gkioulekas. *Developing software with GNU*. Department of Applied Mathematics, University of Washington.
- [18] Harold Lorin and Harvey M. Deitel. *Operating Systems*. (Reading, Massachusetts: Addison-Wesley Publishing Company, Inc.), p.65. 1981.
- [19] *IEEE: IEEE Standard for Scalable Coherent Interface (SCI)*. IEEE standard 1596-1992, New York, 1993.
- [20] Ismael Ripoll. *RTOS State of the Art Analysis*. DISCA, Universidad Politecnica de Valencia.
- [21] Jason C. Fan. *Assessment of Scalable Coherent Interface (SCI)*. IEEE 802 Plenary - La Jolla, CA: RPRSG.
- [22] John P. Kraus. *Real Time Operating Systems CS384 Design of Operating Systems*. 1998.
- [23] K.J. and D. Ofelt. *The Stanford FLASH Multiprocessor*. In Proceedings of the 21st International Symposium on Computer Architecture. volume 22, pages 302-313, Chicago, IL, 1994. ACM.
- [24] Kang G. Shin. *Real-Time Operating Systems: Principles and a Case Study*. Real-Time Computing Laboratory, EECS Department, University of Michigan.

- [25] Kevin M. Obenland. *The use of POSIX in Real-Time Systems, Assessing its Effectiveness and Performance*. The MITRE Corporation, 1820 Dolley Madison Blvd. McLean, VA 22102.
- [26] Knut Omang, and Bodo Parady. *Scalability of SCI Workstation Clusters, a Preliminary Study*. Department of Informatics, University of Oslo, Norway.
- [27] LynxOS Release 4.0. *Writing Device Drivers for LynxOS*.
- [28] Matt Verber. *Real-Time Operating Systems*. 1998.
- [29] Maximilian Ibel, Klaus E. Schauser, Chris J. Scheiman, and Manfred Weis. *High-Performance Cluster Computing Using SCI*. Department of Computer Science, University of California, Santa Barbara.
- [30] On-Line Applications Research Corporation (OAR). *RTEMS C User's Guide*. 2001.
- [31] On-Line Applications Research Corporation (OAR). *RTEMS Intel i386 Applications Supplement*. 2001.
- [32] Red Hat Inc.. *Using ld*. Edited by Jeffrey Osier.
- [33] Richard M. Stallman and Roland H. Pesch. *Debugging with GDB*. Seventh Edition, for GDB version 4.18, February 1999.
- [34] Robert W. Todd, Matthew C. Childester and Alan D. George. *A Direct Flow Control for Real-Time SCI*. HCS Research Laboratory, 2000..
- [35] Roger Butenuth, Hans-Ulrich Heiss. *Shared Memory Programming on PC-based SCI Clusters*.
- [36] S. Millich, A. George, and S. Oral. *A Comparative Throughput Analysis of Scalable Coherent Interface and Myrinet*. HCS Research Lab, ECE Dept., University of Florida, Gainesville, FL 32611.

- [37] Stein J. Ryan, Stein Gjessing, Marius Liaaen. *Cluster communication using a PCI to SCI interface*.
- [38] T. Straumann. *Open Source Real Time Operating Systems Overview*. 8th International Conference on Accelerator & Large Experimental Physics Control Systems, 2001, San Jose, California. 2001.
- [39] The Linux Document Project Organisation. *Linux PCI-HOWTO*.
<http://www.tldp.org>
- [40] The Linux Document Project Organisation. *The Linux Kernel API*.
<http://www.tldp.org>
- [41] Yanbing Li, Miodrag Potkonjak and Wayne Wolf. *Real-Time Operating Systems for Embedded Computing*. Department of Electrical Engineering, Princeton University.
- [42] WindRiver. *VxWorks Programmer's Guide 5.3.1*. Edition 1.

Appendix A

RTCC PACKAGE API

This appendix outlines the API of the implemented packages on RTEMS. The three main libraries, which provide implementation interfaces for application programs are detailed below.

A.1 SCI Initialisation Library (sci_init.h)

This library contains routines involved in the SCI driver initialisation process. The full SCI driver initialisation process is carried out by calling the `sci_init(0)` directive.

Some individual initialisation routines are listed below (please refer to the Section 4.6.1 for description on any of the following).

- `static void getDevices(char *drvname)`
- `static int createAdapterTable(u_int count)`
- `static int initAdaptorTable(char *drvname)`
- `static int createAdapter(Sci_p up)`
- `signed32 gen_init_pre(osif_init_args_t args)`
- `static int openAdapter(Sci_p up)`
- `scibool gen_adapter_init(Sci_p sci_p, osif_instance_t instance, scibool attaching)`

A.1.1 Application Level Initialisation

User may simply import the above library, and call the `sci_init(0)` directive. After return of this directive, the SCI card is not fully functional yet (still a set of timer jobs need to be completed at the background), but user may initiate his/her program and make SCI calls and the card will catchup with the application.

A.1.2 BSP Level Initialisation

Developers keen to perform a portion of the SCI driver initialisation within the BSP initialisation stage, may still do so by placing their desired routines in the `$RTEMS_ROOT/c/src/lib/libbsp/i386/pc386/sci/sci.c:sci_initialize` routine. The above routine shall be called at the BSP initialisation stage.

NOTE: Full SCI driver initialisation can not take place at the BSP initialisation stage, hence user must execute the complementary initialisation routines at the start of his/her application.

A.2 SCI Driver Interface (`sci_genif.h`)

This is an extensive library, covering all supported SCI driver functionalities. The library contains useful documentation on each routine, which user may refer to for further guidance.

The number of routines available in this library run into hundreds and are well documented, therefore we make no further comments on this library except to say if you imports the above library he/she has full access to all SCI functionalities.

A.3 Real-Time Cluster Computing Library (`librtcc.h`)

This library contains the application layer developed within the second section of the project. It uses the general SCI driver interface (described above) to deliver the following functionality. It hides substantial amount of coding and resource management from user, and specifically targets DSM based real-time cluster computing.

- **volatile unsigned32 *Export_SCI_segment_RTCC(
int segment_size,
int source_node_id,
int target_node_id)**

Creates and exports a local segment into the SCI address space. The returned address is the local address of the segment on the local machine. If an error occurs (such as uninitialised SCI drivers), a NULL pointer is returned.

- **volatile unsigned32 *Connect_SCI_segment_RTCC(
int *segment_size,
int source_node_id,
int target_node_id)**

Connects to a shared memory segment of a specific node on the SCI network. It returns the size of the segment as well as the local address on the local machine used to access the remote segment.

- **void lock(volatile unsigned32 *address, int node_id)**

Provided the local address of a shared memory segment, it will lock the segment for the use of local node only.

- **void unlock(volatile unsigned32 *address, int node_id)**

Provided the local address of a shared memory segment, it will unlock the lock.

NOTE: The lock implementation is provided to facilitate synchronisation and mutual exclusion with minimum overhead, it does not provide safety or security against abuse of the shared memory segment.

Appendix B

DISCOVERED GCC BUG

This appendix details a bug which was discovered throughout this project. It was sufficiently analysed, and without any side effects on the project, a work around solution was adopted until further attention on the issue. Following the realisation that bug was GCC and RTEMS build tools related, RTEMS developers were informed and efforts were made to diagnose and resolve the problem. Below is a short description of this bug, which if not handled correctly, could have placed a halt on the project.

B.1 Symptom

The bug was encountered when a new task was created and initiated for execution. Various threads and tasks were initialised through SCI drivers, hence it was observed that whenever a new thread was initiated the system would halt with a Faulty Thread message.

B.2 Cause

Crash occurs in the middle of context switch between the running task and the newly initiated task. When a new thread is created, three minimal contexts are created with `_Thread_Handler` as the entry point. When performing task delete the first context is used, and performs correctly. But when calling the `_Context_Switch`, unfortunately EIP is corrupted, hence the new thread is not initiated properly, resulting in the crash.

B.3 Workaround

As a workaround solution for this problem, a “dummy victim thread” was created with special attributes which would be killed in place of the main execution thread.

B.4 Resolution

The following line in the new lib package was identified as the cause of this bug.

```
*ptr = (struct _reent) _REENT_INIT((*ptr));
```

Replacement of the above line with the following, would result in correct functionality without corruption of the EIP.

```
_REENT_INIT_PTR((ptr));
```

B.5 Status

Problem was identified on the new lib package of GCC 3.2.1 and 3.2.2. Above resolution would resolve the problem. However, at the time of writing of this report, RTEMS is already swifiting towards GCC 3.2.3 (pre-release GCC version), which is yet to be analysed with regards to this bug.

If GCC 3.2.3 does not address the issue, the RTEMS patches will certainly be responsible to do so.

Appendix C

CONCEPTS & TOOLS

C.1 Technical Concepts learnt

- High-performance and parallel computing, with regards to compute clusters.
- SCI (Scalable Coherent Interface), role and importance of cluster interconnects.
- Embedded and Real-Time systems.
- Operating system concepts, with special regards to hardware initialisation, task and object handling, process scheduling and device drivers.
- In depth knowledge of RTEMS, an “open source”, high-performance dedicated real-time operating system.

C.2 Tools and Software utilised

- Ctags - analyse flow of system calls within a large project
- CVS - for extensive version and source code management
- GDB (GNU debugger) - for monitoring step by step execution of processes.
- Latex - for writing this project report.
- SourceNavigator - for studying/analysing large amount of project source codes.
- VIM - as an extremely powerful text editor for programming and writing documents, both.