Enhancing a System-Level Design Flow by RTOS Integration

A Thesis Presented
by

Rohan Kangralkar

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Master of Science

in

Electrical Engineering

Northeastern University
Boston, Massachusetts

July 2012
**Thesis Title:** Enhancing a System-Level Design Flow by RTOS Integration.

**Author:** Rohan Kangralkar.

**Department:** Electrical and Computer Engineering.
I lovingly dedicate this thesis to my parents and wife, for their unconditional love and support.
## Contents

List of Figures .......................................................... vi
List of Tables ........................................................... vii
List of Acronyms ......................................................... viii
Acknowledgments ........................................................ x
Abstract of the Thesis .................................................. xi

1 Introduction ................................................................ 1
   1.1 Challenges .......................................................... 1
   1.2 System Level Design .............................................. 2
   1.3 Background ........................................................ 3
      1.3.1 System on Chip Development Environment ............ 3
      1.3.2 Real-Time Operating Systems ............................. 6
   1.4 Problem Definition ............................................... 7
   1.5 Related Work ....................................................... 9
      1.5.1 System Level Design Languages ......................... 9
      1.5.2 Electronic System Level Design ......................... 9
      1.5.3 Code Generation ........................................... 9
      1.5.4 RTOS Synthesis ........................................... 10

2 RTOS Selection ......................................................... 11
   2.1 Criteria ............................................................. 12
      2.1.1 Processor ..................................................... 12
      2.1.2 Programming Language .................................. 12
      2.1.3 Debugging Features ....................................... 13
      2.1.4 Scheduling Algorithms .................................. 13
      2.1.5 Inter-task Communication ............................... 14
      2.1.6 Documentation ............................................ 14
      2.1.7 Portability ................................................... 15
      2.1.8 Configurability ............................................ 15
      2.1.9 Source Code ................................................ 16
# List of Figures

1.1 Productivity gap. .................................................. 2  
1.2 SCE design methodology. ......................................... 4  
1.3 Share of operating systems in embedded market. .......... 7  

3.1 Architectural view of ADSP-BF527 core. ..................... 32  
3.2 Program sequencing and interrupt processing. ............ 33  
3.3 ADSP-BF527 core nested interrupts handling. ........... 34  
3.4 Layered architectural view of RTEMS. ....................... 35  
3.5 RTEMS code organization. ...................................... 36  
3.6 RTEMS context switch on yield ................................ 39  
3.7 RTEMS context switch on interrupt. ......................... 41  
3.8 RTEMS priority structure. ....................................... 43  
3.9 RTEMS task ready queue. ....................................... 44  
3.10 RTEMS priority table. .......................................... 45  

4.1 Software synthesis flow. ......................................... 47  
4.2 ADSP-BF527 block diagram. .................................... 51  
4.3 RTOS software stack. ............................................ 57  
4.4 OVP semihosting interception. .................................. 59  

5.1 Multi-dimensional RTOS exploration space. .................. 60  
5.2 Measurement of response. ...................................... 61  
5.3 Experimental setup specification model. ..................... 62  
5.4 Experimental setup mapping. .................................... 63  
5.5 Experimental setup architecture. .............................. 64  
5.6 RTOS integration specification. ................................. 64  
5.7 Complexity in terms of LOC. ................................... 65  
5.8 RTEMS ARM926EJS response time. ............................ 67  
5.9 μC/OS-II ARM926EJS response time. .......................... 68  
5.10 Static footprint size. ............................................ 69  
5.11 Execution on Virtual platform and Hardware platform. .... 70  
5.12 RTEMS ADSP-BF52X response time on TLL6527M. .......... 71  
5.13 RTEMS ADSP-BF52X response time. .......................... 71  
5.14 Selection of RTOS. .............................................. 72
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Selection Criteria Variables, Prescribed Value and Importance</td>
<td>20</td>
</tr>
<tr>
<td>2.2</td>
<td>µC/OS-III Score</td>
<td>21</td>
</tr>
<tr>
<td>2.3</td>
<td>FreeRTOS Score</td>
<td>22</td>
</tr>
<tr>
<td>2.4</td>
<td>RTEMS Score</td>
<td>23</td>
</tr>
<tr>
<td>2.5</td>
<td>eCos Score</td>
<td>25</td>
</tr>
<tr>
<td>2.6</td>
<td>UClinux Score</td>
<td>26</td>
</tr>
<tr>
<td>2.7</td>
<td>QNX Score</td>
<td>27</td>
</tr>
<tr>
<td>2.8</td>
<td>VxWorks Score</td>
<td>29</td>
</tr>
<tr>
<td>2.9</td>
<td>RTOS ranking</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>SW database components and dependencies</td>
<td>49</td>
</tr>
<tr>
<td>5.1</td>
<td>Measured interrupt response latency on ARM926EJS</td>
<td>68</td>
</tr>
<tr>
<td>5.2</td>
<td>RTEMS interrupt response latency</td>
<td>71</td>
</tr>
</tbody>
</table>
List of Acronyms

SCE  System-on-Chip Environment. SCE represents a new technology that allows designers to capture system specification as a composition of C-functions. These are automatically refined into different models required on each step of the design process.

SoC  System on Chip. Collection of analog and digital computing components on a single silicon chip.

SLD  System Level Design. A design methodology.

SLDL  System Level Design Languages. Modeling language that can represent hardware and software aspect of an electronic system.

RTOS  Real Time Operating System. Specialized operating system with predictable execution.


ISS  Instruction Set Simulator. Simulator for target processor

MMR  Memory Mapped Registers.

ESL  Electronic System Level. Designing at system level using tools as aids for exploration or synthesis.

BCAM  Bus Cycle Accurate Model.

CCAM  Computation Cycle Accurate Model.

TLM  Transaction Level Model.

RTEMS  Real-Time Executive for Multiprocessor Systems. It is a free real-time operating system designed for deeply embedded systems

RAL  RTOS Abstraction Layer.

TRON  The Real-time Operating system Nucleus.

ITRON  Industrial TRON.

TCB  Task Control Block.

ISA  Instruction Set Architecture.
ISR  Interrupt Service Routine.

MMACS  Million Multiply Accumulate Cycles per Second.

MSA  Micro Signal Architecture.

GPIO  General Purpose Input Output.

HAL  Hardware Abstraction Layer.

LOC  physical Lines Of Code.

VP  Virtual Platform.
Acknowledgments

The research work is the outcome of help and support of many. I want to thank everybody who supported me during the process of the thesis work. At the outset, I would like to express my sincere gratitude to my advisor, Prof. Dr Gunar Schirner. Without his perseverance, criticisms, guidance and support this work would not have been possible. The theses would not have been so beautiful without his organizational talent, technical skills, automation and quest for perfection. I appreciate his understanding and helpfulness in shaping my future life. I am highly indebted to him.

It is an honor for me to have Prof. Dr David Kaeli and Prof Dr. Kaushik Chowdhury as thesis committee members.

I wish to record my gratefulness to northeastern’s ESL group for their cooperation and help during my research work. This thesis work was dependent on SCE and influenced by groups discussion, meetings and entertainment. I would specially like to thank Akash Agarwal for integrating Blackfin simulator model which was instrumental in achieving the results. I owe my deepest gratitude to Prof Dr Mihir Ravel for his ideas in selecting an operating systems and providing ESL group with a robust learning platform (TLL6527M). I am indebted to all my colleagues who supported me in any respect during completion of my thesis.
Abstract of the Thesis

Enhancing a System-Level Design Flow by RTOS Integration

by

Rohan Kangralkar

Master of Science in Electrical and Computer Engineering
Northeastern University, July 2012
Prof. Dr. Gunar Schirner, Adviser

The complexity of designing modern embedded system is increasing at exponential rate. Electronic System Level (ESL) design methodology and System Level Design Languages (SLDL) have been introduced to address the complexity in design. ESL focuses on higher design abstraction level, and SLDL aid in capturing the specification.

In modern embedded systems, the design cost of software content is high compared to hardware. Very often embedded system software consists of a Real Time Operating System (RTOS) with multi-threaded applications executing on heterogeneous multi-processor systems. Dynamic scheduling behavior, task distribution, priority assignment and latencies of an application executing on an RTOS are important non-functional considerations during design space exploration. ESL aids in generation and synthesis of hardware and software models. The synthesized hardware models offer functional simulation of physical hardware. The functional simulation model of physical hardware forms virtual platform. The virtual platform can be used for the purpose of testing the generated software. Execution of software on virtual platform aids for better profiling, analyzing and debugging of systems facilitate design space exploration.

The work in this thesis introduces support for a commonly used RTOS Real-Time Executive for Multiprocessor Systems (RTEMS) for both ARM and Blackfin processor into an ESL flow tool System-on-Chip Environment (SCE) [25]. We use the Portable Operating System Interface (POSIX) standard for interfacing the generated software to RTEMS. Using POSIX standard allows future integration of POSIX compliant operating systems into ESL.

To demonstrate the flexibility offered by our approach, we synthesize a JPEG decoder along with RTEMS on ARM and Blackfin processor. The synthesized JPEG encoder is used as load while measuring the cumulative probability of response time for each RTOS. The response time provides system designers with accurate system latencies of the RTOS. In this thesis we explore the
design space by synthesizing a JPEG decoder with $\mu$C/OS-II, RTEMS on ARM and ADSP-BF527 processor. The experimental results show that the maximum response latency of $\mu$C/OS-II is about 41\% lower than RTEMS on ARM processors. Thus using our work the system designer can automatically synthesize an RTOS and explore the RTOS characteristics for an embedded application.
Chapter 1

Introduction

An embedded system is a reactive computing system integrated into a larger physical system in order to performing specific function. Modern day life has constant interaction with embedded devices. They span multiple domains ranging from but not limited to telecommunication systems (e.g. mobile phones, routers, bridges), consumer electronics (E.g. camera, video players, television, microwave oven), transportation and industrial systems (e.g. GPS, motor control, automobiles) and medical equipment (e.g. EEG, ECG MRI, CT). Current market forecasting and research expects embedded market to increase each year with compound annual growth rate of 7% every year reaching $152.4 billion by year 2015 [49].

The application and operational environment of an embedded system are already known at design time. Designing embedded systems is complex because it is done under tight functional and non-functional environmental constraints. Since embedded system performs specific functions therefore their functional requirements can be concretely defined at design time by the system designer. The non-functional restrictions are often diverse including cost, footprint, power consumption or real-time constrains. Due to multi-dimensional functional and non-functional constrains, customizing and optimizing an embedded system platform is time consuming and complex task.

1.1 Challenges

Complexity of an embedded system is increasing in both software and hardware. This is shown in more detail by Figure [1.1]. Processing capacity of the chip doubles every 18 months according to Moore’s law [33]. But the complex hardware utilizing transistor technology advances is estimated to improve at 1.6x over 18 months. There is a gap between capability of the chip tech-
CHAPTER 1. INTRODUCTION

Figure 1.1: Hardware, Software and System Design Gap(Source:[59]).

Technology and HW design and shown as HW Design Gap in Figure 1.1. Similarly at present software design is reported to increase by about 2x every 24 months. But current need for software design growth is estimated at 2x over 10 months. Thus there is software design productivity gap and hardware design productivity gap. Combining both productivity gaps, as shown in Figure 1.1 results in a large system design productivity gap. These conflicting demands lead to a significant productivity gap. Therefore, new approaches are needed to dramatically increase design productivity and to close the productivity gap. One such approach is utilizing hierarchy and designing at a higher level of abstraction, which enables constructing larger and more complex systems.

1.2 System Level Design

Design complexity of embedded systems is exploding due to the market demand for more complex features and tight coupling between hardware and software. Due to rapid advances in technology there is high pressure of short time-to-market for many consumer electronics embedded products. However the traditional way of designing hardware system components followed by software development separately, does not optimally parallelize design task hence consumes time causing industries to miss marketing deadlines. Also achieving the set functional and non-functional goals, involves analyzing a multi-dimensional design space which is difficult to achieve in traditional methodology due to increased cost and time. System Level Design addresses these challenges by using a holistic approach. Instead of designing the hardware and software components separately, a complete embedded system is designed together enabling tight coupling between hardware and software. An essential aspect of System Level Design (SLD) is the hardware and software co-
CHAPTER 1. INTRODUCTION

design, where both aspects of the system are concurrently designed aiding in faster time to market and tight coupling between hardware and software.

Most modern day embedded systems consists of heterogeneous processors, interconnecting bus and custom hardware components. Selection of components depends on functional and non-functional objectives of the embedded system. A system may contain a mix of software running on one or more generic processors and specialized hardware to meet the set objectives. With a system-level view, the embedded system design starts early with a specific algorithmic system description independent of a particular hardware-software implementation. System Level Design Languages (SLDL) is able to capture both hardware and software features at a higher abstraction level. Jointly designing both aspects has the potential for exploiting the acceleration achieved by hardware and flexibility provided by software for more efficient designs. For exploring the design space quickly SLDL are used along with Electronic System Level (ESL) Design tools. The ESL process facilitates automatic synthesis, rapid prototyping and facilitates larger design space exploration.

1.3 Background

1.3.1 System on Chip Development Environment

System-on-Chip Environment (SCE) [25] is an ESL tool. This tool allows system designers to capture system level specification as a composition of behaviors. Behaviors are then transformed into models by the specify-explore-refine engine. The generated models are fully functional and are validated for functionality and performance with varying accuracy and simulation speed. Since refinement is done automatically, the system designer can focus efforts on system design itself. Consequently, automated generation and validation allows designers to perform architectural exploration in a larger design space.

SCE not only performs vertical synthesis of models with increasing accuracy but also allows horizontal exploration of PE, mapping of tasks and connectivity. The Figure 1.2 gives a brief overview of SCE design methodology. The embedded system designer captures the specification of the application algorithm via SLDL (SPECC [34]) models as composition of behaviors communicating via abstract communication channels. The specification model is purely functional model, free of any architectural implementation details. The model focuses on capturing the algorithmic behavior and allows a functional validation of the application. Since the specification model
CHAPTER 1. INTRODUCTION

does not contain any architectural information it is untimed and allows for causal ordering. This model forms an input to the Specify-Explore-Refine engine for automatic refinement, exploration and synthesis.

The exploration engine is composed of computation, scheduling and communication refinement stages followed by synthesis stage. The refinement tool contains a database of hardware/software components. The component database aid the refinement tool by providing hardware/software models with timing statistics for processing elements. As the first step the embedded system designer explores the computational architectural space. The computational architectural decision information is added to the specification model during computation refinement. The processing elements are inserted into the system and the previously defined functional behaviors are mapped to them. A processing element can be predefined standard component such as generic processor core or custom hardware component. Based on internal statistics of processing elements, early estimations about the run time performance are made. This gives the designer, initial feedback about computational component selection. After computation refinement the architecture model that captures the decisions is generated. Since statistical information is utilized for identifying the computation latency therefore this model is the approximately timed model. It takes computing time into account.
CHAPTER 1. INTRODUCTION

but all communication between the processing elements are still un-timed.

The next step in the refinement is the scheduling refinement. This refinement allows the designer to select suitable scheduling policy for the tasks executing on the processing elements. The scheduling capabilities range from static scheduling, which allows most predictability without using an OS, to a priority based dynamic scheduling based on Real Time Operating System (RTOS). Software task on a processing element can be scheduled statically or dynamically. If the embedded system designer selects static scheduling then an RTOS is not selected instead the task are serialized to execute as a state machine. Most modern embedded systems have task that are created dynamically and interact with the environment dynamically. Static scheduling does not support the dynamic characteristic of tasks. In SCE an abstract RTOS model is used to emulate the dynamic scheduling behavior of tasks. The priority based scheduling allows the system designer to assign priority to tasks executing on processors. A high level abstract RTOS model [19] provides the key features that define a dynamic scheduling behavior independent of any specific RTOS implementation. The RTOS model aids in exploring the priority and also may model RTOS timing for switching between tasks.

The communication refinement allows the system designer to explore bus interconnect between the processing elements. Earlier defined abstract communication channels are replaced by busses models. Pin accurate model or Transaction accurate model of the bus is inserted between the processing elements. The pin accurate model contains cycle accurate timing information about the selected protocol whereas Transaction accurate models only the transactions. The model resulting from the communication refinement includes communication stack for the selected bus implementation. The pin accurate bus contains the arbitration and the access logic for a bus master or bus slave. Thus the model now has high timing accuracy on the bus transaction.

To summarize, the Specify-Explore-Refine starts at specification model and passes through computation, scheduling and communication refinement stages. Each refinement stage captures the designers decision into models. The tools include information collected from previous stages of refinement into the models. These models called Transaction Level Model (TLM) contain architectural and functional behavior information necessary for the synthesis engine to perform software/hardware synthesis.

The synthesis step concludes the design flow. The hardware synthesis generates the RTL model of hardware component. The hardware synthesis uses a HW database of RTL components for synthesizing a cycle accurate description of each custom hardware processing element. Similarly the software synthesis uses a SW database. From the SW database, specific code for the selected
CHAPTER 1. INTRODUCTION

RTOS and drivers for communication are compiled along with the generated application code to generate target binary code. The SW database also contains Instruction Set Simulator (ISS) for the generic processing elements. The software synthesis generates ISS-based system model for executing and validating the generated target binary code. The resulting combination gives a cycle accurate execution of computation and communication of the embedded system. As a result of these theses, additional RTOS is added to software database. The section 1.3.2 gives the brief overview of the importance of an RTOS in modern embedded systems.

1.3.2 Real-Time Operating Systems

To manage the hardware resources operating systems are used as an additional abstraction layer. Operating systems are special computer programs to manage computer resources. Modern operating systems perform task management, memory management, task synchronization and communication. They manage resources and provide an interface for the application to use computing resources.

In process of managing the computing resources operating systems add an overhead to the overall application execution. But the overhead amortizes over the productivity gain that can be achieved by using the resource management features provided by the operating system. As a result Operating Systems are being used for increased productivity. Embedded systems have tight cost constrains and computation power is expensive. But now-a-days with advances in semiconductor technology the cost of computing resources is decreasing. Embedded systems are now designed with slightly higher processing power budget to accommodate operating system overhead. This has enabled operating systems to penetrate into domain of embedded systems where computation.

An embedded system is a specialized computer system designed to perform some specific task. These tasks often have real-time interaction with the environment in which the systems execute. Embedded system usually has timing and resource constrains. Traditional OS lack predictability because they were designed for non-critical desktop systems. As a result specialized operating systems with addition of timing constrain called Real Time Operating System (RTOS) were developed. RTOS are playing important role in time critical embedded systems. Real-time embedded systems are widely used in the domain of military, telecommunication, aviation, automotive, medical and industrial automation. RTOS provides all standard OS features like task management, memory management, task synchronization, communication. But in addition to standard OS features RTOS guarantee to execute the OS features deterministically. Designers can reuse
code that is already validated and verified to work on similar hardware. Using RTOSes also reduces the software development time. This is evident by a survey report [59]. The survey as shown in Figure 1.3 reports that 81% of all embedded products now use some RTOS. This shows that RTOS is becoming an important part of embedded systems development.

Each RTOS provide set of API’s for user to develop their application. They guarantee deterministic execution of APIs. However meeting the system deadline is not only dependent on the RTOS but also depends on how the overall system is designed. The overall system must be designed to meet deadline. In this thesis, the system designer is provided with additional flexibility in exploring an RTOS based application by accurately capturing execution on ISS-based system model.

1.4 Problem Definition

Depending on system requirements the choice of an RTOS may change. Significant effort is needed to port an embedded system application to RTOS. The system designer needs to spend significant time to manually port an application to new RTOS. Without automatic synthesis, it is infeasible to explore multiple RTOS for the embedded system. Automatic synthesis allows the system designer to explore multiple RTOS, identify latencies and mapping. Hence to facilitate RTOS exploration and identify system latencies, multiple RTOS have to be integrated into Electronic System Level design flow.

There are plethoric RTOS each with unique characteristic available in the market. Depending on functional and non-functional requirements the choice of an RTOS for an embedded system can change. It is difficult to choose the right RTOS for the embedded system. Hence key RTOS characteristic are identified and weighed. Later multiple RTOS are surveyed and ranked to
CHAPTER 1. INTRODUCTION

identify the \textbf{RTOS} that fits best into our requirements.

As described earlier in section 1.3.2, \textbf{RTOS} is used in most modern embedded systems. \textbf{RTOSes} support dynamic scheduling of software tasks on processing elements. The scheduling policy is usually based on priorities of the software task. The assignment of priorities to the task defines the overall system behavior. Change in the assigned priorities changes the system behavior. In the worst case wrong mapping of task priorities can lead to deadlocks and prevent tasks from execution. It may ultimately cause real-time system to miss critical deadline. Similarly, the response time of an \textbf{RTOS} is the time an \textbf{RTOS} takes to respond to an external event. Response latency is unique to an \textbf{RTOS} it depends on the length of critical section. Ideally the system designer needs to capture the response time during the system design. Using the response time measurement the system designer can choose or design the system appropriately to meet applications real-time deadlines. The affects of missing deadline can be catastrophic for a real-time system. Hence it is important to identify and accommodate \textbf{RTOS} latencies while designing the embedded systems. \textbf{SCE} \cite{25} facilitates generation of C code targeted to a RTOS Abstraction Layer. The generated code can then be executed on an Instruction Set Simulator based Virtual platform. The virtual platform can be use to find the response time of the system even before the actual hardware platform is built.

ISS-based system model simulation of the application requires implementation of \textbf{RTOS} port consisting of; interrupt management for synchronization and communication with external custom hardware, context switch for executing multitasking application and drivers for synchronization and communication. Effort for detailed implementation of all the above described software components is time consuming and hence limits design space exploration. Therefore if RTOS code is automatically integrated into the application then embedded system designer can focus on functional correctness rather than porting an RTOS for the platform. This encourages functional and design space exploration.

As a result of this thesis new \textbf{RTOS} es were added into \textbf{SCE} SW DB. Chapter 2 identifies features those are important for selecting an \textbf{RTOS}. The selected \textbf{RTOS} is ported as describes in chapter 3 to ADSP-BF527 processor. Chapter 4 describes the integration of Real-Time Executive for Multiprocessor Systems (RTEMS) into \textbf{SCE} for ARM926EJS \cite{27} and ADSP-BF527 \cite{24} processor. In the experimental result chapter 5 response time of RTEMS on ADSP-BF527 and ARM926EJS and compare it to UCOS-II.
1.5 Related Work

1.5.1 System Level Design Languages

In order to specify a design, System Level Design Languages that can contain both the hardware and software contents are needed. System Verilog [17] is an example of a System Level Design Language (SLDL) that is based on Verilog which is a hardware description language. Verilog has been extended for the description of software aspects to create system verilog. There are many system languages are based on imperative programming languages (e.g. C, C++, and Java). SpecC [34], SystemC [20] are system languages based on examples imperative programming languages. The languages provide a means of capturing system specification abstractly.

1.5.2 Electronic System Level Design

The abstract models captured via SLDL are use by ESL implementation for automatic synthesis of models. It allows co-design of hardware and software without special attention to the final hardware software split. There are many commercial and academic tools implementation of ESL. The Cadence Virtual System Platform [6] simplifies the creation and support of virtual prototypes with automated modeling and faster hardware/software debugging. Synopsys Platform Architect [57] provides architects and system designer’s with SystemC TLM tools and methods for the efficient design, performance analysis and optimization of multicore SoC architectures. The tools from cadence and Synopsys focus on ISS based co-simulation. Significant research effort has been invested into development of an academic ESL tool, SCE [25] framework for system level design and software synthesis. It is based on the SpecC language. In addition to ISS based co-simulation SCE also includes abstract processor models and software generation. In this thesis we augment SCE development environment and enhance the design space exploration of real-time operating system.

1.5.3 Code Generation

The paper [65] presents method of automatically generating embedded software from system specification written in SLDL. The authors demonstrate the effectiveness of the proposed method implementing sc2c tool which can generate efficient ANSI C code from system models written in SLDL. The sc2c is part of the SCE tool. In our thesis, the sc2c generated C code is inte-
CHAPTER 1. INTRODUCTION

grate along with RTOS. This enables automatic integration of an RTOS along with the application and enhances the exploration capabilities of SCE.

1.5.4 RTOS Synthesis

SpecC language does not support modeling of dynamic real-time behavior often found in embedded systems. However, an approach has been presented that does not require any specific language extensions [19]. In the approach, an RTOS model with timing annotation for the selected RTOS interacts with the event-based simulator and the application. This approach has been integrated into ESL tool [25]. In this thesis, we have implemented the RAL APIs using POSIX [1] interface call to the RTEMS operating system.

The paper [21] presents an RTOS-centric hardware/software co-simulator. Co-simulator has a complete simulation model of an RTOS. All the software service calls are natively executed on a host computer. Their techniques improve simulation speed but do not show execution time of the software on target processor. Since performance of the application cannot be obtained on the target processor, therefore the performance matrix obtained cannot be used for RTOS design space exploration.

Similar to [19], the work [35] presents an abstract RTOS model with a POSIX API on top of SystemC. SystemC specification using POSIX functions is converted automatically into a timed simulation estimating the execution time of the application SW running on the POSIX platform. The resulting software is then executed on SystemC event-based simulator and hence is not an accurate measure of the latencies. In our approach, we target POSIX [1] interface of RTEMS for execution on actual processor. So we get fairly accurate results at the cost of increased simulation time.
Chapter 2

RTOS Selection

Hardware and software of embedded systems are increasing in complexity. New efficient and complex hardware resources are being introduced continuously. To manage the hardware resources operating systems are being used. Operating systems are special computer programs to manage computer resources. Modern operating systems perform task management, memory management, task synchronization and communication. They manage resources and provide an interface for the application to use computing resources.

Recently there have been plethoric of free and commercially available general RTOS developed by communities, academia and industries. With time these general RTOS are continuously improving in performance and the variety of services that they provide. Its becoming extremely difficult to identify and select a RTOS matching our requirements. To overcome the difficulty, first the selection criteria are identified in section 2.1. Then, to select an RTOS for our requirements, values are prescribed to selection criteria. System designer assigns weight ranging from 0 to 100 based on importance associated for the prescribed values. This forms propitious RTOS feature set and ideally the selected RTOS needs to match all the prescribed values. Next, multiple RTOS are surveyed and features are assessed. Depending on availability of prescribed value in the RTOS prorated weight is assigned to individual assessed criteria. The assessed prorated weight is then scaled on basis of importance associated with the prescribed values. The values for individual RTOS are aggregated and then RTOS es are ranked. System designer then selects the best RTOS from plethora of surveyed RTOS.

In this chapter, a RTOS is selected for ADSP-BF527 processor with focus on synthesizability. ISS is available as processing element in SCE development environment. Comparison aspects are weighted with goal of software synthesis onto ADSP-BF527 processor. Each RTOS
CHAPTER 2. RTOS SELECTION

selection criteria is weighted in a range of 0 to 100% (0 - not required at all and 100% is absolutely required). After weighing the selection variables, multiple RTOSes are surveyed for supported features. Aggregate values from different RTOSes under consideration are calculated and RTOS receiving the highest weight matches closest to requirements of synthesizability and portability. Section 2.1 identifies, describes the selection criteria and then prescribes values and importance to individual selection criteria.

2.1 Criteria

2.1.1 Processor

RTOS interacts closely with the processor to provide multi-tasking facility. Adding new processor support to an RTOS requires significant work and development time. Embedded systems, like consumer electronics, have tight cost and time-to-market constrains. For such embedded systems, the processor support for platform is heavily weighted.

Implementing new processor support for an operating system is complex and time consuming. In our academic environment, a RTOS that supports any of the Blackfin family processors would aid porting effort. If an RTOS already has support for one of the Blackfin family of processors, then only the code specific to ADSP-BF527 processor has to be written and validated. This criterions value is prescribed as Blackfin family processor support. Importance of 100 is assigned to the criterion, since it is absolutely necessary for Blackfin processor family support to be available.

2.1.2 Programming Language

Compiler is a set of programs that convert source code written in human understandable language to binary form to match the Instruction Set Architecture of the target processor. The binary can then be decoded and executed by the target processor. The survey done in 2006 shows that 51% of embedded application development is done using C and around 30% using C++ [58]. Most of the free, commercial operating systems are written in C. There are several language features which are desirable like strong typing, exception handling and modularity.

Object oriented languages provide many features that encourage a high degree of modularity and information hiding. Some information hiding is achieved by binding while execution with time penalty. A study shows code written in objective C was 43% slower than the same code written in conventional C [26]. Embedded systems have predefined set of tasks to be performed efficiently.
CHAPTER 2. RTOS SELECTION

They are constrained by the resources available. To achieve this task the compiler produce code that is highly optimized in size and speed for the target processor. Hence most RTOS are implemented using C. Since RTOS es surveyed are implemented in C this criterion is not explored in the survey.

2.1.3 Debugging Features

Debugger is tool that is used to debug and test target programs. A survey done in 2006 shows that 25% of the project time is spent on testing and debugging, 32% engineers felt an improvements in debugging tool is needed. And 53% need good debug features and debugger integration into graphical interface. An integrated debugger is rated higher than the support for a processor. Most debuggers provide basic functions like single step execution, setting breakpoints, viewing and modifying variables, support for stack analysis and performance analysis. Multi-threaded application has multiple task kernel objects and synchronization kernel objects. Information about the kernel objects are maintained in the RTOS kernel. To efficiently debug an application on RTOS it desirable for the debugger to be aware of kernel object structure. This allows the application designer to get insight into the state of tasks and synchronization objects. Since an RTOS contains multi-threaded reactive application, it would be beneficial to view real-time system events to better understand the real-time context. For example, a graphical trace of the occurrence of system events like interrupts and context switches aids developers to resolve programming problems by identifying unexpected behavior. Since we are developing application on top of RTOS the criterion is given maximum weight.

2.1.4 Scheduling Algorithms

Task scheduling and dispatching is one of the basic services provided by an operating system. In Real-time systems all tasks are assigned a level of priority. Scheduler performs the activity of identifying the next task to execute. Wide varieties of scheduling algorithms have been developed and most RTOS have support for multiple configurable scheduling algorithms. The developer chooses a scheduling algorithm depending on application and response requirements. It is desirable for and RTOS to support different configurable scheduling algorithms so that system designer can choose most appropriate scheduling scheme for the application. There are different strategies employed in designing a real-time operating system like polled loop systems, phase/state-driven code, cooperative multi-tasking and interrupt driven systems.
CHAPTER 2. RTOS SELECTION

The strategy to be used depends on application of the real-time system. If precise time of an event is known then phase/state driven scheduling strategy can be used. For more complex application general RTOS es provide interrupt driven or cooperative multi-tasking. Interrupt driven systems usually support various scheduling policies like the round-robin, Rate-monotonic. There are hybrid systems that support interrupt occurring at both fixed rates and sporadically.

SCE environment currently uses preemptive priority based scheduling algorithm for scheduling tasks. SCE generates code for preemptive priority based scheduling therefore the criterion is prescribed preemptive priority based scheduling. Since the scheduling requirement is well defined the criterion receives absolute weight.

2.1.5 Inter-task Communication

Real-time application has multiple communicating tasks executing in parallel. The tasks often communicate with each other to synchronize or to transfer data. RTOS provides service primitives like file, signal, semaphores, mutex, condition variables, message queues, files and shared memory for synchronization and data transfer. The synchronization primitives are sometimes implemented with special features such as priority inheritance. The inter-task primitives help in communication and synchronization between the tasks.

SPECC has synchronization primitives to communicate with behaviors. The synchronization primitives can be emulated using minimum of semaphore and mutex IPC mechanisms. Since IPC requirements are well defined the criterion receives absolute points. If RTOS under consideration has support for semaphore and mutex then the RTOS receives maximum points.

2.1.6 Documentation

There are ever increasing complex processors being introduced in the embedded market. To port an RTOS to a platform requires substantial effort in understanding the source code and its organization. Without proper documentation and comments understanding the mechanism an RTOS uses to accomplish task can be daunting. Well documented code can help developers in understanding source code, but still requires significant time. Architecture and porting guides help the developer to understand the RTOS architecture, thereby reduces the time to understand. This aids the developer to adapt the RTOS quickly to new processors. The variable is of high importance to consumer industries where time to market is tight.
We are performing RTOS porting activity for the first time and would need porting guide. Porting guide containing procedure to add new processor to the RTOS is necessary to aid faster porting. Since the criterion is prescribed with well defined value therefore the criterion receives 60% points.

### 2.1.7 Portability

Significant time is consumed when the embedded system application is to be moved to a different board or processor. If the RTOS code is modular then the RTOS can be easily ported. RTOS code is split into processor dependent assembly code and processor independent C code. It is desirable to have minimal assembly code since it is non-portable. If majority of the implementation is in high level language like C with portability, alignment, endianness, then the RTOS can be easily ported to the new hardware. Porting is also easy when the code size is smaller. The number of line of code can give some idea for the value of this variable.

Portability gains higher importance if the RTOS does not have support for the desired processor. Since we are porting RTOS to a new processor, this criterion is assigned weight of 80%. The percentage of RTOS written in a high level language and a layered architecture determines the points given to a RTOS.

### 2.1.8 Configurability

Modern RTOS have rich feature set. Many of these features are not utilized by the application. The application developer chooses the features that are required by the application. To aid the developer to choose the required features, reduce memory footprint RTOS provide facilities to enable-disable features. This helps in customizing RTOS features and reduces the memory footprint by removing unwanted features. Fine grained configuration will enable in proper selection of components and makes development of components easier, faster and reduces the memory footprint.

The software synthesis selects only the required components from RTOS. With view of software synthesis, high configurability is desired. Hence, configurability criterion is given 90% weight. Fine grain configurability, nested selection helps in lowering the memory footprint and selection the dependent components.
CHAPTER 2. RTOS SELECTION

2.1.9 Source Code

Most commercial vendors do not provide source code. Any bugs found in the RTOS are often reported to the commercial vendor for debugging. Some vendors take substantially more time to rectify the bug and this may impact the marketing deadline. If the source code is available, then it is easier and faster to debug errors by the developers. Having the source code of the kernel can help in faster debugging and reduces the dependency on other organization to fix bugs. The source code also provides opportunity for the engineer to tweak the code to gain maximum performance from the hardware.

SCE includes part of source code into the software database. It also annotates the source code to gather profiling data. Software database contains source, it can also used obtain profiling results. Therefore source code is absolutely required. This is highly desired feature and hence receives 100% weight.

2.1.10 Application Programming Interface

Explosion of embedded systems has created a need for industries to work together and create a standard for open-ended architecture for RTOS like POSIX real-time extension, real-time specification for Java, OSEK. Standardization will help in faster development of product and easy portability of application. Standard API interface aids in porting application to different RTOS’es. Sometimes there is a performance penalty associated with abstraction layer. However, the benefits of portability out-weigh the performance benefits.

Adhering to standards of compatibility aids in better understanding of the application and reduces the learning time. It also helps in portability of the application to other complaint operating systems with little or no modification. To allow portable of the synthesized code to different operating systems compliance to standards is necessary. Portable Operating System Interface (POSIX) standard is widely used and supported by major RTOS’es. The criterion is prescribed, Standard interface POSIX, TRON. These standard API’s can be used for interfacing to synthesized application and hence the criterion receives 100% weight. If the RTOS under consideration contains some of the standard interfaces then it receives weight relative to ratio of supported interfaces.

2.1.11 Memory Footprint

Memory is a scarce resource in an embedded system. Around 14% of the project reject the use of operating system because of memory constrains. Embedded systems usually execute
CHAPTER 2. RTOS SELECTION

with tight memory constraints on memory. An RTOS with low memory footprint is desirable, so that
large amount of space is provided to application. The memory footprint depends on configurations
of the RTOS, stack, memory layout etc. RTOS to application memory footprint in the range of
512K to 8M is usually acceptable overhead. Our platform has around 64MB of main memory hence
RTOS overhead of around 1Mb is acceptable. Since the platform has huge memory, this criterion
receives 60% points.

2.1.12 Predictability/Performance/Latency

The basic characteristic of an OS that qualifies it as an RTOS is predictability. If the ser-
vice provided by an operating system are deterministic then the operating system qualifies to be a
real time operating system. The responsiveness to events is more important than the execution speed
of the event. The interrupt latency and context switch time provides a measure of responsiveness of
the system.

2.1.12.1 Interrupt Latency

Interrupt latency is the time between assertion of the interrupt pin to the first instruction
is executed. The factors influencing the interrupt latency are

- **Processor response time**: The processor needs to finish or kill the instruction in flight inside
  the pipeline before responding to the interrupt. The time to respond is usually a function of
  the pipeline depth and instruction in flight. This is usually few cycles and is neglected since.
  Having a longer pipeline often affects the deterministic nature of the interrupt. The interrupt
  must be identified and this is usually done by dedicated hardware like the vectored interrupt
  controller.

- **Context save time**: After an interrupt arrives the RTOS saves the processor registers into
  memory. While saving the registers interrupts are disabled. The duration of interrupt disable
  depends on the number of registers that needs to be saved. Processors having large set of
  registers disable interrupts for longer time.

- **Interrupt disable time**: While executing the application RTOS performs multiple context
  switches between threads and multiple lock, unlocks during synchronization. As the RTOS
  performs these activities, the RTOS accesses shared kernel data structures (ready queue, pri-
  ority map etc). The shared data structures have to be accessing atomically to avoid data
corruption by multiple threads. Hence the RTOS disable interrupts at the beginning of atomically accessed code and re-enable interrupts at the end. While the interrupts are disabled the processor does not receive any new interrupts. This interrupt disable time is the largest contributor to interrupt latency.

2.1.12.2 Task Switch

A real-time application has multiple tasks active. State of a task include a value of the program counter (future continuation point), contents of the processor registers (storing the state of the processor) and stack pointer which points to memory containing temporary data (function calls variables, return address etc). To switch to a new task RTOS saves the state of the task into memory and restores the processor state with state from the new task. While performing this activity the kernel modifies shared data structure and hence interrupts are disabled. There are two components for task switch time

- **Scheduling**: This time depends heavily upon the type of scheduling algorithm and the efficiency of the algorithm. This algorithm selects the eligible task from pool of ready task for execution. The data structure for maintaining the pool influences the selection time. Ideally the time to select an eligible task needs to be constant and as low as possible.

- **Dispatch**: After the scheduler identifies the next eligible task for execution the dispatcher performs the activity of saving, restoring task state and modifying the task structure parameters. The time for dispatch depends on the number variables that need to be saved and restored. Since most multitasking application spend significant time in context switch during which interrupts are disabled therefore it’s desirable to have low task switch time.

Interrupt latency and context switch time are dependent on the processor hardware. Experiments have to be performed to achieve accurate measurements. Major RTOS advertise rough latency measurements. Since accurate latency measurements are not published by RTOS vendors, the criterion is assigned relatively low points (40%) in our survey.

2.1.13 Device and Power Management

Device management aids in identifying the devices that are being actively being used. Power to un-used device can be turned off. Thus device management aids in managing the power to different parts of the hardware platform. Since some embedded applications have power constrains
CHAPTER 2. RTOS SELECTION

therefore RTOS have device management feature to aid power management. This criterion is not considered while surveying the RTOSes.

2.1.14 Licensing

In market there are various RTOS available and have different licenses and costs. Depending on the business requirements, they can be categorized into Free Source RTOS, Open Source RTOS, Royalty charged RTOS and Proprietary RTOS.

Since an academic environment has tight budget constrains therefore free RTOS is needed for our purpose. This section is very important as money cannot be invested in buying commercial RTOS. Since, constrain is well defined it receives absolute weight.

2.1.15 Community

There are different discussion forums and communities for developers and users. These communities are a pool of resource for understanding and debugging. The communities can sometimes help greatly in development of the RTOS features by providing new ideas. The communities can also be divided into academic, non-profit and industry based communities. Academic and non-profit based communities provide help for free but is usually not time bound and depends on the expertise of the person answering. Usually the academic communities are easily accessible and provide quick support.

The traffic, number of users and developers can be a great source of information for finding the current development activity and to understand how much help can be received from the community. The hit rate in popular search engines can also be taken as a matrix for finding the popularity and use of the operating systems.

Community is important for collaboration, understanding, development and sustainability of our application. We need community to help, direct and sustain the project we are undertaking of porting an RTOS. We look at the popularity and the activity rate of the investigated RTOSes for comparison. This criterion receives a relatively low percentage weight of 40%. Popularity of the RTOS is obtained by using the hit rate in Google. Other methodologies like commit rate or number of active developers can also be use as matrix for prescribing value to the criterion.
CHAPTER 2. RTOS SELECTION

2.2 RTOS Survey

In section 2.1 RTOS selection criterion are identified and weighed based on system requirements. In this section several RTOS’es are surveyed. Then selection methodology is applied to the surveyed RTOS’es. Table 2.1 summarizes the variable, value of the parameters required and the weight assigned to the variable. For example the weight 100 is equally distributed between Blackfin and ADSP-BF527. If an RTOS port supports Blackfin processor but not ADSP-BF527 then Processor variable will be assigned 50. Based on the available features RTOS under consideration are assigned points relative to supported parameters. In the next subsequent sections μC/OS-III, FreeRTOS, RTEMS, QNX, eCos, UClinux and VxWorks operating systems are surveyed to select an RTOS.

2.2.1 μC/OS-III

μC was originally written by Jean J. Labrosse. μC/OS-III is a third generation realtime OS by Micrium [32]. Table 2.2 shows the scaled weight for features available in μC. The RTOS works well with 8, 16 and 32 bit processors. However, support for Blackfin is still not available on μC/OS-III. Since μC/OS-III does not support any of the Blackfin processors the Processor variable receives zero points. The evaluation of the μC/OS-III source code is FREE for 45 days then license
## CHAPTER 2. RTOS SELECTION

Table 2.2: μC/OS-III Score

<table>
<thead>
<tr>
<th>#</th>
<th>Criteria</th>
<th>Features</th>
<th>Prorated</th>
<th>Scaled</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Processor</td>
<td>None</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Debug</td>
<td>Available</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>Scheduling Algorithm</td>
<td>RR preemptive</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>Inter-task Communication</td>
<td>Semaphore, event flags, Message Queue</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>Documentation</td>
<td>Book describing the kernel internals available</td>
<td>100</td>
<td>60</td>
</tr>
<tr>
<td>6</td>
<td>Portability</td>
<td>Easy to port</td>
<td>100</td>
<td>80</td>
</tr>
<tr>
<td>7</td>
<td>configurability</td>
<td>Highly configurable via configuration file</td>
<td>100</td>
<td>90</td>
</tr>
<tr>
<td>8</td>
<td>Source Code</td>
<td>Available</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>9</td>
<td>Application Interface</td>
<td>POSIX</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>10</td>
<td>Interrupt Latency</td>
<td>NA</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>11</td>
<td>Context switch Latency</td>
<td>NA</td>
<td>50</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>Memory Footprint</td>
<td>6K-24K</td>
<td>100</td>
<td>60</td>
</tr>
<tr>
<td>13</td>
<td>Licensing</td>
<td>Licensed</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>14</td>
<td>Community</td>
<td>1,470,000 hits in Google</td>
<td>100</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td><strong>Total Scaled Weight</strong></td>
<td></td>
<td><strong>920</strong></td>
<td></td>
</tr>
</tbody>
</table>

is required for commercially use. If μC/OS-III is used for commercial purpose then license is not required. Since the RTOS is not completely free, so 50 points is assigned to License variable.

μC/OS-III has support for interrupt based preemptive, priority based kernel. μC/OS-III added support for Round-Robin scheduling. The RR enables having multiple tasks at the same priority. The RR scheduling can be enabled or disabled during runtime. All of the scheduling algorithms and IPC mechanisms needed as described in Table 2.1 are supported by μC/OS-III.

The author of μC/OS-III has written books describing and documenting the internal working of the RTOS. There are many papers available on the Internet describing the μC/OS-III. It has very good documentation and hence receives maximum points for Documentation variable.

μC/OS-III is written in few thousand line of C code. Porting μC/OS-III to new processor is relatively easy to processors that have and ANSI C compiler support. Since μC/OS-III is highly portable therefore Portability variable receives maximum weight.

μC/OS-III is highly configurable and can produce kernel with low footprint. The RTOS is also popular and has a huge development/support community and hence can be assigned full points for Community and Footprint variable. The latency measurements are not available, as its a popular safety critical real-time operating system we assume that the RTOS has low latency. After
aggregating the scaled weights $\mu$C/OS-III receives a total score of 920 points. These points will be compared against other RTOS for selection.

### 2.2.2 FreeRTOS

#### Table 2.3: FreeRTOS Score

<table>
<thead>
<tr>
<th>#</th>
<th>Criteria</th>
<th>Features</th>
<th>Prorated</th>
<th>Scaled</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Processor</td>
<td>ADSP-BF533 [41]</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>2</td>
<td>Debug</td>
<td>Available</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>Scheduling Algorithm</td>
<td>Priority based preemptive [42]</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>Inter-task Communication</td>
<td>Semaphore, event flags, Message Queue [43]</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>Documentation</td>
<td>Fair</td>
<td>60</td>
<td>36</td>
</tr>
<tr>
<td>6</td>
<td>Portability</td>
<td>Easy to port [40]</td>
<td>100</td>
<td>80</td>
</tr>
<tr>
<td>7</td>
<td>configurability</td>
<td>Not configurable</td>
<td>20</td>
<td>18</td>
</tr>
<tr>
<td>8</td>
<td>Source Code</td>
<td>Available</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>9</td>
<td>Application Interface</td>
<td>POSIX</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>10</td>
<td>Interrupt Latency</td>
<td>NA</td>
<td>50</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>Context switch Latency</td>
<td>NA</td>
<td>50</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>Memory Footprint</td>
<td>around 4K [38]</td>
<td>100</td>
<td>60</td>
</tr>
<tr>
<td>13</td>
<td>Licensing</td>
<td>Free [39]</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>14</td>
<td>Community</td>
<td>398,000 hits in Google</td>
<td>100</td>
<td>40</td>
</tr>
</tbody>
</table>

**Total Scaled Weight** 844

FreeRTOS [37] is an open source real-time microkernel OS developed by Richard Barry & FreeRTOS team. FreeRTOS is distributed under modified GPL license. The kernel has been ported to many microcontrollers. The FreeRTOS development community has contributed Blackfin 533 port [41]. As shown in Table 2.3, Processor variable receives 60 points because at least one processor from the Blackfin family is supported.

FreeRTOS has fully prioritized preemptive scheduler [42]. But it does not have support for round robin scheduling. Since we require prioritized preemptive scheduler only, therefore Scheduling Algorithm variable receives maximum points. It has support for queues, binary semaphores, mutexes and counting semaphores but it lacks signals and events. Hence it receives prorated score of 80. The RTOS is relatively simple since it has 3 core files that implement the bulk of the kernel. Hence porting is easy but it does not have good documentation and hence the learning curve will be
CHAPTER 2. RTOS SELECTION

long and slow. It is not easy to configure the kernel compared to the other kernels seen and hence it scores low on kernel configuration.

The kernel code is free under modified GPL with an exception that allows the application code can be proprietary and hence the application code need not be free. This protects the interest of the company also. There are commercial port to FreeRTOS like openRTOS and SafeRTOS. The RTOS has very small memory footprint as shown in Table 2.3 and hence score good in memory footprint. The latency measurements are not available; as its a popular safety critical real-time operating system we assume that the RTOS has low latency. FreeRTOS scores an aggregate of 804 points.

2.2.3 RTEMS

<table>
<thead>
<tr>
<th>#</th>
<th>Criteria</th>
<th>Features</th>
<th>Prorated</th>
<th>Scaled</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Processor</td>
<td>ADSP-BF533 [13]</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>2</td>
<td>Debug</td>
<td>Partly Available as scripts</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>3</td>
<td>Scheduling Algorithm</td>
<td>Rate-Monotonic, RR, FIFO(preemptive) [10]</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>Inter-task Communication</td>
<td>Semaphore, Mutex, Conditional variable, Mailbox, signals [10]</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>Documentation</td>
<td>Structured coding and manuals. [15]</td>
<td>100</td>
<td>60</td>
</tr>
<tr>
<td>6</td>
<td>Portability</td>
<td>Easy to port [12]</td>
<td>100</td>
<td>80</td>
</tr>
<tr>
<td>7</td>
<td>Configurability</td>
<td>Configurable</td>
<td>100</td>
<td>90</td>
</tr>
<tr>
<td>8</td>
<td>Source Code</td>
<td>Available [14]</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>9</td>
<td>Application Interface</td>
<td>POSIX 1003.1b, µITRON 3.0 [10]</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>10</td>
<td>Interrupt Latency</td>
<td>8.17uS [53]</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>11</td>
<td>Context switch Latency</td>
<td>17.48uS [53]</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>12</td>
<td>Memory Footprint</td>
<td>64K-128K</td>
<td>80</td>
<td>48</td>
</tr>
<tr>
<td>13</td>
<td>Licensing</td>
<td>Free [11]</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>14</td>
<td>Community</td>
<td>315,000 in Google search engine</td>
<td>80</td>
<td>32</td>
</tr>
</tbody>
</table>

The OAR Corporation originally developed Real-Time Executive for Multiprocessor Systems [9] in 1994 as a research project. Real-Time Executive for Multiprocessor Systems supports a large number of CPU’s and more are being in process of being added. The Table 2.4 shows the scores assigned to variables. The RTOS has support for Analog devices Blackfin 533 processor
CHAPTER 2. RTOS SELECTION

which was added in CVS version 4.8. Since it has support for one member of Blackfin family the processor variable is assigned 60 points. The debugger provided for the kernel uses scripts to identify the kernel objects and provide debug information.

RTEMS has rich set of scheduling algorithms ranging from traditional priority based preemptive scheduling with support for multiple tasks at same priority level to Rate-Monotonic. It supports Semaphore, Mutex, Conditional variable, Mailbox, signals for inter process communication. The executive comes along with full set of documents describing the kernel internals. There are occasional on-line moodle based classes for training. Since it supports all the scheduling, IPC features and documentation is excellent the variables receive maximum weights.

The RTOS is mostly written in modular C, the processor family code is separated from processor specific code. Adapting to new processors is easily due to provided abstraction layers. The modularity also allows the executive to be highly configurable and can be customized to smaller memory footprint. The RTOS has also been ported to many processors and hence scores high in porting as shown in Table 2.4. It has low interrupt and context switch latency of 17.48uS and hence scores high in this category. The executive is freely provided under GNU GPL license v2.0, compliant to POSIX 1003.1b, µITRON 3.0 and hence scores high in this category. RTEMS receives an aggregate score of 990 points.

2.2.4 eCos

Embedded Configurable System (eCos) [44] was developed by Cygnus Solutions which was later bought by Red Hat. Redhat has stopped the development of eCos but former developers of eCos now contribute to the project. The software is under GPL license as per our requirements.

Support for Blackfin family of processors is paramount for our development, but eCos lacks support for Blackfin range of processors [47]. Hence the RTOS scores low for Processor variable.

eCos scheduler uses either multilevel queue or bitmap to implement preemptive, time-sliced scheduling. In the multilevel queue two processes can have the same priority where as in bitmap each process has a unique priority. The scheduler satisfies our basic requirement of prioritized, preemptive scheduler. The author Anthony Massa has written books that covers architecture, installation configuration of eCos. However, eCos has relatively little documentation for its latest releases and thus receives lower score.
CHAPTER 2. RTOS SELECTION

Table 2.5: eCos Score

<table>
<thead>
<tr>
<th>#</th>
<th>Criteria</th>
<th>Features</th>
<th>Prorated</th>
<th>Scaled</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Processor</td>
<td>None</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Debug</td>
<td>Partly Available as scripts</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>3</td>
<td>Scheduling Algorithm</td>
<td>Preemptive with time slicing</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>Inter-task Communication</td>
<td>Semaphore, Mutex, Cond Variable, Message box</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>Documentation</td>
<td>structured coding but relatively less document-</td>
<td>80</td>
<td>48</td>
</tr>
<tr>
<td>6</td>
<td>Portability</td>
<td>Layered architecture</td>
<td>100</td>
<td>80</td>
</tr>
<tr>
<td>7</td>
<td>configurability</td>
<td>Highly Configurable</td>
<td>100</td>
<td>90</td>
</tr>
<tr>
<td>8</td>
<td>Source Code</td>
<td>Available</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>9</td>
<td>Application Interface</td>
<td>POSIX 1003.1, µTRON(R,S)</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>10</td>
<td>Interrupt Latency</td>
<td>10.42µS</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>11</td>
<td>Context switch Latency</td>
<td>NA</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>12</td>
<td>Memory Footprint</td>
<td>300K</td>
<td>80</td>
<td>48</td>
</tr>
<tr>
<td>13</td>
<td>Licensing</td>
<td>Free</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>14</td>
<td>Community</td>
<td>4,980,000 hits in Google</td>
<td>70</td>
<td>28</td>
</tr>
</tbody>
</table>

**Total Scaled Weight** 914

The RTOS is covered under modified GNU GPL license, code is freely available, system is highly configurable and scores high as shown in the Table 2.5.

It has compatibility layers allowing developers easily adapt the kernel to new processors. eCos supports the standards required by us like POSIX and µTRION standards. eCos is a realtime OS and has low interrupt and context switch latency well within our acceptable limit of 20µsec as shown in Table 2.5 and hence has high score.

Since it is highly configurable the memory footprint is low. The kernel requires about 300K which matches our memory footprint requirement. Hence eCos scores high in this section. eCos score an aggregate of 914 based on our selection criteria.

2.2.5 UClinux

UClinux [2] project was founded by D. Jeff Dionne and Kenneth Albanowski. It is MMU less derivative of GNU/Linux kernel version 2.0 intended for microcontrollers. The first major criterion for our selection is the processor support. UClinux supports Blackfin range of processors and hence scores high on processor selection criteria. The operating system is not an RTOS, but is
Table 2.6: UClinux Score

<table>
<thead>
<tr>
<th>#</th>
<th>Criteria</th>
<th>Features</th>
<th>Prorated</th>
<th>Scaled</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Processor</td>
<td>Blackfin [3]</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>Debug</td>
<td>Available</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>Scheduling Algorithm</td>
<td>Priority based RR, FIFO [4]</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>Inter-task Communication</td>
<td>Semaphore, msg queue, signals, Transparent IPC [5]</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>Documentation</td>
<td>Excellent documentation</td>
<td>100</td>
<td>60</td>
</tr>
<tr>
<td>6</td>
<td>Portability</td>
<td>Fairly easy to port</td>
<td>90</td>
<td>72</td>
</tr>
<tr>
<td>7</td>
<td>configurability</td>
<td>Highly Configurable</td>
<td>100</td>
<td>90</td>
</tr>
<tr>
<td>8</td>
<td>Source Code</td>
<td>Available</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>9</td>
<td>Application Interface</td>
<td>POSIX 1003.1</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>10</td>
<td>Interrupt Latency</td>
<td>14.8uS</td>
<td>50</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>Context switch Latency</td>
<td>20uS</td>
<td>50</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>Memory Footprint</td>
<td>1MB</td>
<td>100</td>
<td>60</td>
</tr>
<tr>
<td>13</td>
<td>Licensing</td>
<td>Free</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>14</td>
<td>Community</td>
<td>341 commits in June 2010; 4,980,000 hits in Google search engine</td>
<td>50</td>
<td>20</td>
</tr>
</tbody>
</table>

Total Scaled Weight 972

very well supported by Analog Devices hence we added the OS to the survey.

GNU/Linux supports rich set of IPC and scheduling algorithms. UClinux is a derivative of Linux. Like GNU/Linux UClinux supports wide range of scheduling algorithms and rich set of IPC and synchronization mechanisms as mentioned in Table 2.6. It satisfies the scheduling algorithm and IPC requirement of our application as mentioned in Table 2.1.

UClinux like GNU/Linux has well documented structured code. Like Linux, UClinux is mostly written in C and has layer architecture the kernel can easily be ported to different architectures. In addition ADI supports Blackfin processors and has a dedicated web-site for Blackfin developers. UClinux has an interface similar to Linux to configure and select the components required. The kernel is highly configurable and hence UClinux scores high in Configurability. We are looking for free kernel for our development and the UClinux kernel is under GNU GPL license and the code is freely available for download and use.

The UClinux kernel implements POSIX standard interface. But it does not conform to any of the realtime OS standards like µTRION. So it does not fully satisfy our compatibility requirement. UClinux is a derivative of general purpose OS GNU/Linux. The kernel has been modified to support
CHAPTER 2. RTOS SELECTION

embedded file system and microcontrollers. So UClinux has a huge memory footprint of around 1M and hence scores low on memory footprint. UClinux reports low interrupt latency of 14.8uS and 20uS, however it is not a RTOS as the worst case latency distribution is not released. Still UClinux is assigned a high score on latency.

The UClinux development and user community is large and spread around the world. The UClinux kernel development takes place in both academic and industries continuously and has a high activity rate. ADI has added support of their processor to UClinux. They also provide technical support to the Blackfin UClinux developers and users. UClinux is being continuously being upgraded by contribution from many students and universities. The kernel has been in the market for a long time and has been tested and deployed by companies developing network systems, consumer products like cameras and evaluation boards. On the downside since UClinux does not claim the operating system to be realtime and UClinux also has huge memory footprint as shown in Table 2.6. Due to strong support from Analog devices it scores a high score of 972 points.

2.2.6 QNX Neutrino

<table>
<thead>
<tr>
<th>#</th>
<th>Criteria</th>
<th>Features</th>
<th>Prorated</th>
<th>Scaled</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Processor Support</td>
<td>Not Supported [51]</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Debug Support</td>
<td>Available</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>Scheduling Algorithm</td>
<td>Sporadic, FIFO, RR [56]</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>Inter-task Communi-</td>
<td>Message Passing, Signal, Shared Memory, FIFO</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>cation</td>
<td>[56]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Documentation</td>
<td>Excellent documentation</td>
<td>100</td>
<td>60</td>
</tr>
<tr>
<td>6</td>
<td>Portability</td>
<td>Not available</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>configurability</td>
<td>Highly Configurable</td>
<td>100</td>
<td>90</td>
</tr>
<tr>
<td>8</td>
<td>Source Code</td>
<td>Not available</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>Application Interface</td>
<td>POSIX 1003.1 [56]</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>10</td>
<td>Interrupt Latency</td>
<td>5.4µS [50]</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>11</td>
<td>Context switch Latency</td>
<td>11.1µS [50]</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>12</td>
<td>Memory Footprint</td>
<td>Not available</td>
<td>50</td>
<td>30</td>
</tr>
<tr>
<td>13</td>
<td>Licensing</td>
<td>Proprietary</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>14</td>
<td>Community</td>
<td>6,520,000 hits in Google</td>
<td>100</td>
<td>40</td>
</tr>
</tbody>
</table>

| Total Scaled Weight | 660 |

Table 2.7: QNX Score
CHAPTER 2. RTOS SELECTION

QNX Neutrino [52] is a microkernel RTOS developed by QNX software systems. For our development we require a free RTOS. However QNX is not a free operating systems and hence we cannot choose the OS for our development purpose. We require the kernel to support Blackfin processors but QNX does not have a port for any of the Blackfin processors [51]. So it scores low.

QNX supports preemptive, priority based scheduling. It supports our basic scheduling algorithm requirement. The RTOS also supports all our IPC and synchronization requirements as shown in Table 2.7. So it scores high in scheduling, IPC and synchronization. QNX has layered architecture and has excellent documentation. The kernel is also compliant to POSIX standards so it is easy to port application from other POSIX compliant OS. QNX satisfies our requirement of good documentation and portability.

It is a realtime operating system it has a low interrupt and context switch latency. The latencies are well within our requirements and hence the RTOS scores well. The RTOS has a good support forum but support is paid. It has a good hit rate in Google as shown in the Table 2.7. QNX has excellent support, latencies and features but the source code is not available therefore it receives low aggregate score of 782 points.

2.2.7 VxWorks

VxWorks Realtime Operating system [61] is developed by Wind River technologies. It has been in the market for over 25 years. VxWorks is a proven RTOS that is being used in many mission critical applications. VxWorks is a commercial RTOS it is not free, and there is no port available for Blackfin processors. So VxWorks does not satisfy our basic requirement and hence scores low as shown by Table 2.8.

VxWorks supports priority, preemptive scheduling. It supports FIFO and Round Robin scheduling policies. So it satisfies our requirements of scheduling algorithm. VxWorks has a layered architecture, POSIX compliant and good documentation and satisfies our requirements of this section. VxWorks has low interrupt and context switch latency. The latency is within our requirements and hence scores high in this section. It also has small memory footprint of around 150K and hence scores high as shown in the Table 2.8. Similar to QNX it has a good hit rate in Google as shown in the Table 2.7. QNX has excellent support, latencies and features but the source code is not available therefore it receives low aggregate score of 660 points.
CHAPTER 2. RTOS SELECTION

Table 2.8: VxWorks Score

<table>
<thead>
<tr>
<th>#</th>
<th>Criteria</th>
<th>Features</th>
<th>Prorated</th>
<th>Scaled</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Processor Support</td>
<td>Not Supported</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Debug Support</td>
<td>Available [63]</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>Scheduling Algorithm</td>
<td>preemptive FIFO, RR [62]</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>Inter-task Communication</td>
<td>Semaphore, Mutex, Cond Variable, Signal,</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transparent IPC [62]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Documentation</td>
<td>Excellent</td>
<td>100</td>
<td>60</td>
</tr>
<tr>
<td>6</td>
<td>Portability</td>
<td>Not available</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>configurability</td>
<td>Highly Configurable</td>
<td>100</td>
<td>90</td>
</tr>
<tr>
<td>8</td>
<td>Source Code</td>
<td>Not available</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>Application Interface</td>
<td>POSIX 1003.1b [62]</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>10</td>
<td>Interrupt Latency</td>
<td>13.90uS [53]</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>11</td>
<td>Context switch Latency</td>
<td>20.80uS [53]</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>12</td>
<td>Memory Footprint</td>
<td>150K</td>
<td>50</td>
<td>30</td>
</tr>
<tr>
<td>13</td>
<td>Licensing</td>
<td>Proprietary</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>14</td>
<td>Community</td>
<td>2,420,000 hits in Google</td>
<td>100</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>Total Scaled Weight</td>
<td></td>
<td></td>
<td>660</td>
</tr>
</tbody>
</table>

2.3 Summary and Selection

Section 2.2 surveyed 7 popular OS for our embedded system. Weights are assigned to each criterion for individual RTOS and scaled relative to the importance and relevance to our embedded system feature requirement.

Table 2.9: RTOS ranking

<table>
<thead>
<tr>
<th>RTOS</th>
<th>RTEMS</th>
<th>UClinux</th>
<th>(\mu)C/OS-III</th>
<th>eCos</th>
<th>FreeRTOS</th>
<th>QNX</th>
<th>VxWorks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score</td>
<td>990</td>
<td>972</td>
<td>920</td>
<td>914</td>
<td>844</td>
<td>660</td>
<td>660</td>
</tr>
</tbody>
</table>

Table 2.9 summarizes and ranks the scores obtained for 7 surveyed RTOS es. Clearly RTEMS receives maximum points; it is a popular free RTOS. It has support for Blackfin ADSP-BF533 processors, highly configurable, decent development community, small footprint, and the OS is compliant to the compatibility standards. The RTOS best fits our requirements and hence is the best choice for synthesizability and porting to ADSP-BF527 processor.

At second position is UClinux, it is a free OS. Since it is officially supported by Analog...
CHAPTER 2. RTOS SELECTION

Devices Inc, it already been ported to Blackfin processors. It is very popular, highly configurable and has a large development community. However, UClinux scored low due to large memory footprint. Though the latencies are low, the OS does not claim to be an RTOS and hence scored low on latency. The OS is not compliant to uITRON compatibility standards. Hence the OS receives the second position in our ranking.

μC/OS-III finishes third. It is stable popular, feature rich, well documented RTOS with low memory footprint and latency. However it lacked processor support and is not totally free, hence it dropped to third position. eCOS finishes very close to μC/OS-III. It has features comparable to μC/OS-III. But lacks good documentation and hence moved to fourth position. FreeRTOS is a small, free RTOS, it has support for Blackfin processors. However, it loses points because of poor documentation and no declared compliance to standard APIs. Hence the RTOS drops to fifth position.

Finally, VxWorks and QNX together obtain sixth position. They are the most popular commercial RTOS available in the market. However, their source code is not available and hence inhibits synthesizability. It lacks processor support, and is expensive to request the RTOS vendors to add support. Hence both the RTOS score low of 660 points.

In this chapter, 14 RTOS features are identified that can influence selection of an RTOS for our embedded system. Section 2.2 surveyed 7 popular RTOS’es, weighted and scaled the feature requirements for synthesizability and execution on Blackfin-527. RTOS’es were ranked and RTEMS is selected for porting to TLL6527M platform because it is a popular free RTOS, has support for Blackfin range of processors, highly configurable, decent development community, small footprint, and OS is compliant to most of the compatibility standard.
Chapter 3

RTOS Porting

In chapter 2 we surveyed uClinux, RTEMS, FreeRTOS, eCos, uC, QNX and VxWorks as potential candidates for porting to TLL6527M platform. After weighting, RTEMS is selected as target RTOS, because it is a popular free RTOS, has support for Blackfin range of processors, highly configurable, decent development community, small footprint, and OS is compliant to most of the compatibility standard. As RTEMS best fits our requirements and hence it is the best choice for synthesis and our target board. TLL6527M platform contains ADSP-BF527 DSP processor, set of sensors and Xilinx Spartan-3E FPGA device is mapped to General Purpose Input Output signals and memory region of the processor.

ADSP-BF527 processor is both a DSP and a micro-controller. It can perform 1200 Million Multiply Accumulate Cycles per Second (MMACS) and is powerful enough for execution of application along with OS. FPGA on the platform provide capability of implementing customized hardware for acceleration. Due to generic processor and FPGA, TLL6527M is suitable for HW/SW partitioning, allowing software to execute on the ADSP-BF527 and hardware components to be synthesized to FPGA.

3.1 Introduction to ADSP-BF527

Understanding the ISA of ADSP-BF527 is paramount for porting RTEMS to the ISA. ADSP-BF527 belongs to Blackfin family of processors from Analog Devices. The Blackfin Micro Signal Architecture core was developed jointly by Analog Devices, Inc. and Intel Corporation. Figure 3.1 describes the architectural view of ADSP-BF527 core. “ADSP-BF527 combines two MAC signal processing engines, an orthogonal RISC-like microprocessor instruction set, flex-
CHAPTER 3. RTOS PORTING

ible Single Instruction, Multiple Data (SIMD) capabilities, and multimedia features into a single instruction set architecture. The Blackfin processor core contains two 16-bit multipliers, two 40-bit accumulators, two 40-bit arithmetic logic units (ALUs), four 8-bit video ALUs, and a 40-bit shifter. They process 8-, 16-, or 32-bit data from the register file" [22]. Since the ALU can process 8 bit data the smallest C data type char is of width 8 bits.

As shown in Figure 3.2 and Figure 3.1 ADSP-BF527 has 44 core registers exposed to the programmer in Instruction Set Architecture (ISA) and they define the current state of the processor. Figure 3.1 shows the 34 registers of core. They are divided into 8 general purpose registers (R0 to R7), 8 pointer registers (P0-P5, SP, FP), 16 data address registers (I0-I3, L0-L3, B0-B3 and M0-M3) and two accumulate registers (A0, A1). Program sequencing blocks shown in Figure 3.2 contains additional 6 loop control registers (LC0/1, LT0/1, LB0/1), 5 return registers (RETI, RETS, RETX, RETN, RETE) and one status register ASTAT. State of the processor is defined by values stored in the 44 core register. To save the state of the processor the 44 core registers have to be saved to memory block.

To communicate and synchronize external devices use interrupt signal lines connected to the processor. Understanding interrupt handling behavior of ADSP-BF527, is important for porting

Figure 3.1: Architectural view of ADSP-BF527 core. [22]
activity. Figure 3.2 describes the Blackfin interrupt architecture. ADSP-BF527 provides two-level interrupt processing; the first level consists of the System Interrupt Controller (SIC), it groups, prioritizes interrupt requests signaled by on-chip or off-chip peripherals and forwards them to the second level consisting of Core Event Controller (CEC) which interacts closely with program sequencer and manages the Event Vector Table (EVT) as shown in Figure 3.2. EVT contains the vector address for handling all the interrupts. The core events EMULATION, RESET, NMI, EXCEPTIONS, HARDWARE ERRORS and CORE TIMER are connected directly to the CEC. Core timer interrupt is used to provide system ticks to RTEMS. External interrupts from custom hardware are routed via the General Purpose Input Output interrupt line connected to the SIC. The mapping of the custom hardware to the processor allows the processor to synchronize and communicate with the custom hardware via interrupt or polling.

In addition to interrupt architecture, it is important for interrupt handling code to know the behavior of the core pipeline hardware after an interrupt occurs so that the software can modify critical software structures and keep global interrupt disable time to bare minimum. Figure 3.3 describes the behavior of the core after an interrupt is delivered to the core by CEC. ADSP-BF527 has a ten stage fully inter-locked pipeline. There are no visible pipeline effects when executing
Interrupt Processing has completed. The first instruction in an interrupt service routine that supports nesting must save the return address currently held in RETI by pushing it onto the Supervisor stack (\(--SP\) = RETI). This clears the global interrupt disable bit IPEND[4], enabling interrupts. Next, all registers that are modified by the interrupt service routine are saved onto the Supervisor stack. Processor state is stored in the Supervisor stack, not in the User stack. Hence, the instructions to push RETI (\(--SP\) = RETI) and pop RETI (RETI = [SP++]) use the Supervisor stack.

Figure 4-9 illustrates that by pushing RETI onto the stack, interrupts can be re-enabled during an interrupt service routine, resulting in a short duration where interrupts are globally disabled.

<table>
<thead>
<tr>
<th>PIPELINE STAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CYCLE: 1</td>
</tr>
<tr>
<td>IF 1 A9</td>
</tr>
<tr>
<td>IF 2 A8</td>
</tr>
<tr>
<td>IF 3 A7</td>
</tr>
<tr>
<td>DEC A6</td>
</tr>
<tr>
<td>AC A5</td>
</tr>
<tr>
<td>DF1 A4</td>
</tr>
<tr>
<td>DF2 A3</td>
</tr>
<tr>
<td>EX1 A2</td>
</tr>
<tr>
<td>EX2 A1</td>
</tr>
<tr>
<td>WB A0</td>
</tr>
</tbody>
</table>

Figure 3.3: ADSP-BF527 core nested interrupt handling.

Instructions with data dependencies. The minimum latency from the interrupt trigger on the custom hardware till it reaches the CEC is about 5 core clock cycles due to the core and system clock ratio. The core starts to react to the latched interrupt on the third cycle. Interrupts are disabled from the 3rd cycle after interrupt latch till the core reads RETI register and the read reaches DF2 STAGE as shown in Figure 3.3 cycle 10. Interrupts are also disabled during write to RTI register shown by m+1 to m+4 cycles. Hence the read of RTI instruction re-enables interrupt on the core. After an interrupt, the core jumps to the beginning of the interrupt handling code contained in Event Vector Table (EVT). An interrupt is an asynchronous event during processor execution. To return to the interrupted code the processor states has to be saved. Since read of RTI re-enables interrupts on the core and new interrupt can arrive potentially corrupting kernel data structures. Hence important registers are saved by the RTOS before re-enabling interrupts in the core. Section 3.2 gives an introduction to RTEMS and the modification made to port RTEMS to TLL6527M.

### 3.2 Introduction to RTEMS

In chapter RTEMS is selected as the target operating system for porting. Real-Time Executive for Multiprocessor Systems is a scalable, preemptive free open source Real Time Operating System that supports a variety of open API and interface standards [10] and is designed for
CHAPTER 3. RTOS PORTING

Figure 3.4: Layered architectural view of RTEMS (Source: [8]).

deeply embedded systems. Figure 3.4 describes the architecture of RTEMS executive. RTEMS contains a real-time kernel, higher-level services such as network protocol stack, Graphical User Interface (GUI), file management and other components. RTEMS supports the BSD TCP/IP stack for inter networking. The network stack [18] is free and supports standard TCP/IP protocol. Most RTOSes also have graphical front end and RTEMS supports many of the free open source graphical user interfaces like MicroWindows, OpenGUI, picoTK. In addition RTEMS also supports number of file systems like FAT, TARFS and IMFS. As an interface to the operating system RTEMS supports variety of standards. Original RTEMS API is referred as the Classic API. It was based upon a VMEbus Industry Trade Association [60]. Other OS APIs supported by RTEMS are Industrial TRON (ITRON), POSIX. RTEMS is modular, has a rich feature set ranging from networking to graphical user interface which can be utilized after porting the operating system to our platform. Section describes the code organization and is followed by description on modules for porting RTEMS to TLL6527M

3.3 RTEMS Code Organization

Source code of RTEMS is organized to maximize core-reuse. It encourage development of modular components by isolating processor and target-dependent code, while allowing as much common source code as possible to be shared across multiple processors and target boards. It allows
CHAPTER 3. RTOS PORTING

Figure 3.5: RTEMS code organization.

multiple RTEMS users to perform simultaneous compilation of RTEMS. Figure 3.5 shows basic directory structure. Processor and target-dependent source files are isolated from generic files. The code is broadly divided into processor-independent code to manage OS features, processor-specific source code to handle context switch, interrupts and portable application library function to aid application developers. Processor-specific source for processors is located at $c$/src/lib. The $libbsp$ location contains code specific to the platform and constitutes the board support package. The $⟨$platform$⟩$ directory contains core startup code that is common for the processor family and is shared between multiple platforms. Core processor startup code is written in assembly and is processor-specific and cross-compiler dependent. Platform-specific compiler-independent board initialization code is also placed under $⟨$platform$⟩$ directory. Code for the processor peripherals is placed under $libcpu$. The $⟨$processor$⟩$ folder contains to access MMR of peripherals specific to the processor. Code under $libcpu$ is processor-dependent but xcompiler independent code. Most of the processor-independent code is placed under cpukit. It contains generic application code, OS API code and some cross-compiler, processor dependent code. Generic code containing networking protocol stack, GUI and
file system management code is placed under cpukit. OS specific generic C API code is placed under sapi, rtems, posix, score. Processor and cross compiler specific code for performing context switch, interrupt handling and processor configuration is stored under score/cpu/(processor). Details of processor-dependent implementation under score/cpu is described in sub-section below. Apart from separation of code into processor-dependent and processor-independent section, RTEMS also separates the compilation into two separate components. At end of compilation, library containing BSP(libtemsbsp.a), CPU(libtemscpu.a) and RTEMS APIs(libtemsbsp.a) are created for linking with the application. This allows separate compilation of RTEMS and the application. To add ADSP-BF527 processor support to RTEMS the folders (platform), (processors) and cpu/(processors) are modified which are described in subsequent sections.

3.4 RTEMS Porting

An operating system consists of hardware-dependent and hardware-independent code. Hardware-dependent code interfaces with processor core registers and peripherals, the code is unique for a processor type. Processor startup, context-switch, interrupt and code accessing MMR constitute hardware-dependent code. Startup code initializes processor core registers and prepares an environment for HLL code execution; Context switch code allows switch between two tasks; interrupt code allows RTOS to react to external events and MMR code handles processor peripherals. Porting is the endeavor of adapting an operating system for execution on a new processor. The activity adapts hardware-dependent code to new processor for enabling execution RTOS on new processor. Next sections describe the hardware-dependent code for adapting RTEMS to ADSP-BF527 processor.

3.4.1 Startup Code

De-asserting ADSP-BF527 RESET pin completes reset state of ADSP-BF527. Depending on boot-mode pin configuration, the processor begins execution in supervisor mode. While booting from memory it begins execution from address 0. ADSP-BF527 remains in this mode until an emulation event or return instruction occurs to change the execution mode. Listing 3.1 shows ported RTEMS processor startup code, it is placed under c/src/lib/libbsp/libbfin/shared/start/start.S. Start and length of sections that constitute RTEMS are obtained from linker description file. Startup code is minimalistic and sets the C run time environment. First, code on line 1 sets the stack point-
CHAPTER 3. RTOS PORTING

ers, after this C function call are possible. Next on line 5 startup code obtains location for Block Starting with Symbols (BSS) section. Then on line 9 the startup code clears BSS to reset global symbols to zero. After this point the basic C environment setup is completed. The code can now transition to use C execution environment. Function boot_card on line 12 performs initialization of RTOS data structures, heap, drivers and begins multitasking. Initialization code at c/src/lib/libbsp-sp/bfin/TLL6527M/startup/bspstart.c boosts the ADSP-BF527 core clock to 500 Mhz and system clock to 100Mhz. After this point the processor is ready for fast execution of code.

Listing 3.1: Processor startup code

```c
sp.l = __end - 12;
sp.h = __end - 12;
usp = sp;
...
p0 = _bss_start;
p1 = _bss_length;
r0 = 0;
setup(loop,loop) lc0 = p1;
loop: [p0++] = r0;
...
p0 = _boot_card;
call (p0);
```

3.4.2 Context Switch

A real-time application can have multiple tasks. OS maintains TCB kernel data structure for each task. Program counter (future continuation point), contents of the processor registers and stack pointer together constitute context of a task. An instance of Task Control Block data structure contains context of a task. Context switch code saves the core processor registers into current task’s Task Control Block (TCB) context instance and restores context of new task’s TCB into the processor registers. Thus a context switch provides capability to multi tasking systems to switch between two tasks. The code sequence of performing context switch access core processor registers in defined sequence. But C abstracts register access. Since context switch code accesses registers in defined sequence therefore it is coded in assembly. Assembly code maintains proper sequence between storing and restoring of processor registers during context switch. The context switch code accesses processor registers and hence is highly processor-specific. The code is in assembly to maintain proper register read write order and hence is compiler specific.
Figure 3.6: RTEMS context switch on yield

RTEMS scheduler described in section 3.4.4 identifies, currently executing TCB instance and new task (heir task) TCB instance, and provides the pointers to the dispatcher. The dispatcher performs context switch between two tasks. Figure 3.6 shows procedure for context switch, following description describes the flow of events during context switch.

1. Currently executing task is selected and pointer to context memory, located inside TCB instance is assigned to register r0.
CHAPTER 3. RTOS PORTING

2. Heir task is selected by the scheduler and pointer to context memory located inside TCB is assigned to register r1.

3. In listing 3.2 line 1 assigns currently executing tasks context memory pointer to p0. Current values of the processor registers are saved into current task TCB context area. Line 2-13 shows context store code into tasks processor context memory area.

4. After storing current task context, the context switch code then reads registers values from heir TCB and stores them to processor register. Listing 3.3 shows context restore code, line 1 assigns heir context area to p0, then line 2-13 reads heir context memory location and restores processor registers. Finally, line 14 restores program counter from rtes register. At this point RTEMS completes context switch.

5. RTEMS changes statistic information of the task. It marks the old TCB as Ready or Blocked. and new heir TCB as executing.

At this time the context switch is complete. Heir task will be in executing state and previous current task enters the waiting state. Sequences of restoring and storing registers are mirror images. In section 3.4.3 we look at the interrupt handling functionality

3.4.3 Interrupt Handling

An interrupt is an asynchronous event to the processor. After receiving an interrupt, RTOS saves processor context of the currently executing task for later execution. It then uses processor registers to execute Interrupt Service Routine. Context switch due to an interrupt is similar to task context switch, however the context is saved onto a separate interrupt stack. ADSP-BF527 has two levels of interrupt handling, CEC and SIC. Similarly, RTEMS also has two levels of interrupt handling code. Core interrupt handler identifies the CEC interrupt source. Code for core interrupt handling is place at cpukit/score/cpu/bfin/cpu_asm.S. Figure 3.7 and code listing 3.4 describes core interrupt handling sequence.

1. An interrupt occurs and signals the processor of an external event. To handle the external event the processor finishes current activities as described in section 3.1. Processor, then branches to RTEMS core interrupt handler routine, address stored in EVT.
2. In listing 3.4 line 15 core interrupt handler saves scratch registers r0, r1, p0, p1. It uses these registers as temporary locations for all computation performed in the core interrupt handler. The core interrupt code then identifies the interrupt stack location.

3. If required core interrupt handler code changes stack pointer to interrupt stack pointer. Line 9 updates the stack pointer. Then at line 9 task stack is stored onto interrupt stack.

4. Next at line 10 core interrupt handler code reads and save the RETI to interrupt stack. Reading RETI re-enables interrupts on the processor as detailed in section 3.1.

5. Core interrupt handler then identifies CEC interrupt source (not shown in listing for simplicity).

6. All the processor state registers to the interrupt stack as shown on line 12-16.

7. Finally, core interrupt handler pushes 3 arguments as needed by gcc blackfin ABI onto stack and branches to first level of C Interrupt Service Routine (ISR) on line 18 for identifying and servicing SIC interrupt.
At end of handling an Interrupt Service Routine (ISR), RTEMS restores context of the task into the core registers. As shown on line 1-9, RTEMS core interrupt handler code reads values from interrupt stack and stores them to processor registers. Next, if there is a new task to be dispatched then the code raises lowest priority software interrupt (line 12). Then remaining registers are restored as shown on line 14-18. Thus, processor can now respond to interrupts and also resume task at the end of the interrupt. After receiving interrupt and performing a context save, the C user interrupt service routine identifies the line responsible for the interrupt.

The core interrupt handler call C ISR for identifying SIC interrupt source. Code for the SIC interrupt handler is placed at `c/src/lib/libcpu/bfin/bf52x/interrupt/interrupt.c`. Previous implementation of SIC interrupt handler utilized linked list containing function pointers. An list holds CEC mask for each SIC register and an additional pointer for linked list, it takes $9 \times 3 \times 4 = 108$ Bytes and additional 8 byte mask for complete set of 56 lines. A secondary data structure holds a list of 56 entries containing 7 word structure taking space $56 \times 7 \times 4 = 1568$ Bytes. In worst case all the lines can be on one CEC line to 56 entries have to be traversed to find the right user ISR. Thus, the previous implementation has variable time in identifying SIC source and consumes $108 + 1568 + 8 = 1684$ Bytes.
CHAPTER 3. RTOS PORTING

However, our modified implementation contains static array implementation. Space for data structure and time to identify interrupt source is constant. Our modified implementation contains an array of function pointers. Array of function pointers are indexed by SIC interrupt line. Space taken up by SIC interrupt handler is as follows

1. ADSP-BF527 system interrupt controller has 4 byte SIC.IMASK0 and SIC.IMASK1 for external interrupts on each of the 9 CEC line. The masks are used to identify the external interrupt and they take a total space of \(9 \cdot (4 + 4) = 72\) Bytes.

2. There 56 external interrupts sources on ADSP-BF527. The central dispatcher maintains user function pointers (4 bytes), user function argument (4 bytes), priority (4 bytes) and source (4 bytes) for each of the 56 interrupt source taking a total space of \(56 \cdot (4 + 4 + 4 + 4) = 896\) Bytes.

Total space consumed by modified SIC interrupt handler is \(72 + 896 = 968\) Bytes. Thus in worst case the modified SIC interrupt handler consumes 41% less memory than original implementation. In modified implementation jump tables are used to identify interrupt source hence time to identify an interrupt source is constant.

On TLL6527M, custom hardware sends interrupt signals to the processor on port F which is mapped to secondary SIC pin 13. The secondary SIC pin is mapped by default to CEC interrupt line 13. Central interrupt handler identifies the interrupt arriving on CEC pin 13 and executes SIC ISR. SIC ISR then identifies SIC line interrupt source and execute GPIO interrupt handling routine. GPIO interrupt handling routine identifies GPIO line source and calls C user function registered for individual General Purpose Input Output (GPIO) line for servicing interrupt source.

3.4.4 RTEMS Priority Structure

![Figure 3.8: RTEMS priority structure.](image)

This section, describes scheduling mechanism and minor optimization used for identifying the highest priority task. Priorities of tasks in RTEMS are maintained inside TCB instance. The
RTEMS port for ADSP-BF527 has 8 bit in the priority field. The 8 bit priority field is split into “major” and “minor” components as shown in Figure 3.8. The most significant 4 bits is the major component, while the least significant is the minor component. The tasks that are ready to execute are placed in ready queue. Figure 3.9 shows the ready queue. TCB is indexed in the ready queue based on 8 bit priority field inside each TCB.

![RTEMS task ready queue](image)

Figure 3.9: RTEMS task ready queue.

The number of priorities supported defines the size of the array. Each slot in the array contains pointers to the tasks control block. Tasks are arranged in the array according to their priority. Task at the same priority level are linked together into one slot as shown by entry at index 1. All ready tasks are placed on the ready queue. For example Figure 3.9 shows task with priority 1, 5 and 135 added to the ready queue.

RTEMS scheduler identifies the highest priority ready task for execution. Searching through the ready queue list is expensive and inefficient. The time to identify a ready task depends on the interrupt level and hence is not constant. To overcome these issues, RTEMS uses priority table to calculate index of highest priority task that is ready to execute. Before adding ready TCB to ready queue, RTEMS manipulates RTEMS priority table as shown in Figure 3.10. RTEMS priority table is defined in `cpukit/score/include/rtems/score/priority.h` and consists of “.Priority_Bit_map” and “.Priority_Major_bit_map”. “bitmap_table” is a 16x16=256 bit field table, each bit-field in the
Task Priority 0 to 256

Priority Minor
7 4

Priority Major

Highest Priority
Lowest Priority

Ready Queue

TCB priority
1

bitmap_table
1
0
7
8
15

1
0
0
0
0
0
0
1
0
0
0
0
0
0

Figure 3.10: RTEMS priority table.

table refers to a priority level. “_Priority_Major_bit_map” is 32 bit field value. RTEMS uses first 16 bits of “_Priority_Major_bit_map” as a row index into “bitmap_table”. Before adding ready TCB to ready queue, RTEMS performs two activities on priority bit map data structure. First RTEMS uses “major” value of task priority as an index to set one bit in “_Priority_Major_bit_map” and also selects the row in “_Priority_Bit_map”. In second step RTEMS uses “minor” value of tasks priority as in index to set one bit of the selected row. For example as shown in Figure 3.10 a task with priority 135 (0b10000111) is ready to be added to the ready queue. Higher nibble (0b1000) forms an index and sets the eighth field in “_Priority_Major_bit_map” and selects the eighth row of “_Priority_Bit_map[8]”. Then the lower nibble (0b0111) is used to set seventh bit in the row “_Priority_Bit_map[8][7]” as shown by 1 place in the seventh column of the eighth row”. With these two activities RTEMS logs the new ready task. The table is maintained so that the search can be computationally efficient.

Search for a new task is performed by the scheduler at scheduling points as described in section 3.4.2. The scheduler performs frequently identifies the first set bit to calculate priority of ready task. RTEMS provides generic C implementation for calculating first set bit as shown in listing 3.6. The original implementation code consists of 16 entry lookup table to identify the bit

45
set. And using blackfin compiler the code consumes around 72 bytes of code space.

However, while porting bit array search is optimized by using ones population count as shown in listing 3.7. As shown in listing 3.7, the code converts LSB bit value to number of ones using \( (n \land (\neg n)) + 1 \) implemented in assembly from line 1-5. Using value obtained, next instruction uses ADSP-BF527 “ONES” instruction to perform population count of one as shown at line 6. Using “ONES” instruction, the scheduler identifies least significant bit set. Time in identifying LSB set is constant, deterministic and computationally efficient. Modified optimized code occupies 26 bytes of code space. As shown in listing 3.7, the optimized code has no branches. Thus optimized searching method uses less memory and code requirements compared to generic RTEMS implementation.

<table>
<thead>
<tr>
<th>Listing 3.6: RTEMS original find LSB set</th>
<th>Listing 3.7: Modified LSB set</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. _number = 0</td>
<td>1. \texttt{R2 = %1;}</td>
</tr>
<tr>
<td>2. if _value &gt; 0x00ff</td>
<td>2. \texttt{R4 = \neg R2;}</td>
</tr>
<tr>
<td>3. _value &gt;&gt;=8</td>
<td>3. \texttt{R4 = \neg R4;}</td>
</tr>
<tr>
<td>4. _number = 8;</td>
<td>4. \texttt{R3 = R2 ^ R4;}</td>
</tr>
<tr>
<td>5. if _value &gt; 0x0000f</td>
<td>5. \texttt{R4 = 0;}</td>
</tr>
<tr>
<td>6. _value &gt;&gt;=8</td>
<td>6. \texttt{R4.L = ONES R3;}</td>
</tr>
<tr>
<td>7. _number += 4</td>
<td>7. \texttt{R4 += -0x1;}</td>
</tr>
<tr>
<td>8.</td>
<td></td>
</tr>
<tr>
<td>9. _number += \texttt{bit_set_table[}_value</td>
<td></td>
</tr>
</tbody>
</table>

Thus after selecting RTEMS as target operating system, it is ported to TLL6527M by modifying the startup, interrupt handling. As an optimization, context switch, interrupt identification are implemented in Blackfin assembly and uart console is implemented via DMA. At the end of porting the port is tested with RTEMS test suite. The port is contributed back and made available to RTEMS users [7]. Chapter 4 uses compiled RTEMS libraries for linking against synthesizing user application.
Chapter 4

RTOS Integration

4.1 Software Synthesis

Chapter 1 section 1.3.1 introduced SCE development environment. Figure 1.2 described various refinement and synthesis stages of SCE tool. In this thesis, SCE is augmented by adding Real-Time operating system ports into the software database. SCE accepts the specification model and finally generates target binary for execution on ISS-based virtual platform. As shown in Figure 1.2, the software synthesis stage is the final stage that generates target binaries for execution on the ISS-based system models (Virtual platform). Figure 4.1 provides details of software synthesis stage. TLM output from specify-explore-refine engine contains architectural and functional

![Software synthesis flow](image-url)

Figure 4.1: Software synthesis flow. [54]
CHAPTER 4. RTOS INTEGRATION

behavior information, captured during previous refinement stages. This forms an input to the software synthesis stage. Software synthesis consists of two parts, code generation and hardware dependent software synthesis. Code generation stage generates C code targeting RTOS Abstraction Layer (RAL) APIs. RAL provides an abstraction layer for operating system, task, synchronization and time management RTOS services. C code generation transforms behaviors mapped to generic processors during computational refinement to tasks. It utilizes hierarchy, concurrency and communication constructs to generate equivalent C code targeted towards RAL APIs. Scheduling refinement decisions encoded inside input TLM aid in priority assignment. Utilizing the hierarchy, concurrency and communication information the C code generation transforms SPECC primitives into RAL APIs. The result of C code generation is transformation of logic from TLM into equivalent ANSI C code implementation. The C code generation is described in the paper [65]. The binary executing on virtual platform often interacts with virtual platform hardware. The hardware dependent software synthesis provides hardware dependent code necessary for interaction with virtual platform. It includes code for interrupt handler, drivers for communication and configuration to select the software database components. The software database contains components necessary for assembling and building the target application. The hardware dependent software synthesis is outline in [54]. Generated C code along with hardware dependent software and software database components are cross-compiled and linked together to generate target binary. The target binary is executed for validation on the hardware platform or ISS-based system model (Virtual platform). Automatic generation of target binary, allows embedded system designer to focus on functionality rather than the implementation details. In this thesis we have added RTEMS which was ported earlier as described in chapter 3 into the software database. This enables automatic mapping of applications to RTEMS for ARM926EJS and ADSP-BF527 processor. Hence, it allows the platform developer to identify performance and cost without performing the actual port.

4.2 Integration of RTOS into SCE Software Database

Software database consists of components necessary to build the target binary. It contains RTEMS port and the hardware dependent software code. Software database is extended with RTEMS for ADSP-BF527, ARM926EJS processors. It also contains hardware dependent software for executing ADSP-BF527 application without an operating system. Table 4.1 shows the database files and directories structure in the first column. The second column gives a brief description followed by the dependency of modules. It shows the dependency of modules on the software database. To
modify a module, the marked software components have to be changed. For example adding a new cross-compiler to the software database requires changes in `xcompiler` directory.

The software database splits code into different components to achieve maximum flexibility and to minimize code duplication. Software database contain software components required for cross-compiling and linking to target code. All the components cannot be stored within the database. So the software database contains link to external components. Cross-compilers compile the generated code along with the RTOS to target binary. The cross-compiler, itself is external to SCE software database. The base path references to external packages are captured in top level `Makefile.macros.common`. The software packages external to database are installed relative to the path defined by `EXTPATH_BASE` variable. The external package tools in the software database are referenced relative to the path defined in `Makefile.macros.common`. Placing the base path of external packages at single configuration file allows flexibility in adding external packages to the software database.

### Table 4.1: SW database components and dependencies

<table>
<thead>
<tr>
<th>Database</th>
<th>Description</th>
<th>Processor Core</th>
<th>Cross-Compiler</th>
<th>Processor Peripheral</th>
<th>Platform Peripherals</th>
<th>Operating System</th>
<th>Root of the software Directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Makefile.macros.common</td>
<td>Paths to the externally installed packages for cross-compiler and RTOS libraries.</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>compiler</td>
<td>Contains set of crosscompilers.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;tool&gt;_&lt;isa&gt;_&lt;format&gt;</code></td>
<td>Cross-compiler and output format specific directory.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Makefile.macros</td>
<td>Contains path to cross-compiler tools like compiler, linker etc.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hal</td>
<td>Hardware Abstraction Layer contains directories for individual processors.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;Processor&gt;</code></td>
<td>Each processor contains processor peripheral initialization, interrupt dispatcher and driver code.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>bsp.c</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>stdPic.c</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>ambaAhb_mac.c, pack.c</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;platform&gt;</td>
<td>Code specific to platform peripherals for example initialization of FPGA on the board.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>os_wrapper.c</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Contains implementation of the RAI APIs.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;Processor&gt;</code></td>
<td>Contains directory unique to OS, processor core and cross compiler specific code.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;tool&gt;_&lt;isa&gt;_&lt;format&gt;</code></td>
<td>Contains OS, processor core and cross-compiler specific code.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>alinkcmds</td>
<td>Provide the memory map to the startup and cross-compiler to generate output file.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>start.S</td>
<td>Initiates the processor core.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bsp_specs</td>
<td>Cross-compiler specific settings to select the startup code and linker settings</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Makefile.macros</td>
<td>The most specific configuration file. The configuration file contains links to the cross-compiler tools and hal. The generated code will reference this file to aggregate all the components to form the final target binary.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The RTEMS port uses a customized cross-compiler tool-chain for compiling the RTOS kernel.
CHAPTER 4. RTOS INTEGRATION

and application. The cross-compiler and processor independent RTEMS libraries described in 3.3 together forms the RTEMS installation. RTEMS installation is external software and hence installed at ⟨EXTPATH_BASE⟩/rtems-4.10. A new variable EXTPATH_RRTOS_RTEMS defines the base path location to RTEMS installation. All reference to RTEMS libraries and the cross-compiler is relative to EXTPATH_RRTOS_RTEMS.

Software database contains RTOS ports, hardware dependent software implementation and configuration information to link to external software packages. The software database is designed to minimize code duplication between RTOS ports by splitting code into three major directories.

1. **xcompiler**: has reference to cross compiler installed outside of the software database.

2. **hal**: contains processor specific hardware abstraction layer.

3. **os**: contains OS port and sources.

The next sub-sections describe them individually.

### 4.2.1 Cross Compiler

Cross-compilers are used for compiling RTOS and the application code target processor ISA. Output format of generated binary allows the loader to load sections of program onto the target platform for execution. Cross-compilers are usually obtained from third party (GCC) and are installed outside the database. Reference to the cross-compiler and tools has to be stored, so that RTOS ports can select cross-compilers. Directory xcompiler contains folders specific to cross-compiler and output format as shown in Table 4.1. A new cross-compiler is added under a new directory ⟨tool⟩⟨processor⟩⟨format⟩. The configuration file Makefile.macros under new directory defines variables (CC defines compiler, AR defines archiving tool, LD defines linker ect) with complete path reference to the cross compiler tools. This allows RTOS with flexibility to select between different cross-compiler tools.

Since RTEMS installation provides customized cross-compilers, therefore directories containing cross-compiler tools are added to xcompiler. The file xcompiler/gcc-bfinrtems/Makefile.macros contains variable definition pointing to rtems-4.10 compiler tools for the Blackfin ISA. Similarly, the file xcompiler/gcc-arm-rtems/Makefile.macros contains definition of variables pointing to rtems-4.10 tools for the ARMv5 ISA. The two cross-compilers provide tools
CHAPTER 4. RTOS INTEGRATION

necessary to compile, debug and query the binaries compiled for on ARM926EJS and ADSP-BF527 processors.

4.2.2 Hardware Abstraction Layer

![Figure 4.2: ADSP-BF527 block diagram](image)

Most processors are composed of core and peripherals. The core performs execution of the instructions and peripherals are devices external to the core. Modern processors contain rich set of peripherals like interrupt controller, timer and bus controller. Within a family of processors, the core remains identical but the peripherals are different. To reduce code duplication, software database separates core from the peripherals. Code for the processor peripherals is placed under `hal/processors` directory. `hal/processors` directory contains `BspInit()` function for initialization of the processor peripherals; `UserIrqRegister()` registers user ISR with the interrupt controller, `UserIrqClearAll()` function clears all the interrupts; pack.c and ambaAhb_mac.c contain MAC layer implementation of the communication controller. Code contained in these functions access MMR and hence is specific to the processor. SW database separates peripheral from the core, there by allowing separation of code. As a result peripheral initialization can be shared between multiple operating systems thereby allowing reuse of peripheral driver code.
CHAPTER 4. RTOS INTEGRATION

Listing 4.1: Processor initialization

```c
void BspInit(void) {
    gpio_init(PORT_MASK, OUTPUT_MASK);
    CEC_EVT13 = &_ISR_Disp;
    CEC_IMASK |= 0X1 << 13;
    SIC_IMASK1 |= 0X1 << 13;
    boardInit();
}
```

Figure 4.2 shows the block diagram of ADSP-BF527 processor. It contains rich set of peripherals like the interrupt controller, DMA controller, timer, uart etc. Hardware Abstraction Layer for bf52x implements basic functionality to use and operate the interrupt controller. Listing 4.1 shows implementation code for BspInit() for ADSP-BF527. On line 2, BspInit() initialize General Purpose Input Output port-F [23] as input with interrupts on rising edge. Port F has sixteen pins and thus the port can support 16 external interrupt lines. These 16 lines are multiplexed into secondary system interrupt controller register on line 13. Eventually secondary SIC source 13 is mapped to CEC 13. To support interrupts BspInit() registers the core interrupt handler for source 13 on line 3. Next on line 4 and 5 the CEC and secondary SIC is allowed to accept interrupt only on source 13. The initialization finally completes by calling board specific function to configure the FPGA device as shown on line 6. The board specific code is placed under hal/bf52x/tll6527m and it configures the FPGA device on TLL6527M. The function UserIrqRegister() implement user ISR registration with centralized GPIO ISR dispatch. The function isrDisp_isr() implements centralized GPIO dispatcher that identifies source of interrupt and calls user registered ISR for handling interrupt. ambaAhb_mac.c file implements MAC layer communication drivers. ARM926EJS port for UCOS-II and RTEMS uses the same hal code. Identical HAL code can be used across different RTOS ports. Therefore it prevents code duplication for the processor port.

4.2.3 Operating System Ports

As described in section 3.3 code for RTOS is composed of processor dependent, cross-compiler independent code for peripherals; processor independent OS services; processor and cross-compiler dependent code for initialization, task management and optimization. Software database identifies these differences and separates code into different directories. Top level os directory contains operating system ports. Each operating system is placed in separate folder (os_name) under os directory. Processor-dependent, cross-compiler independent code is placed under hal as
described in section 4.2.2. Processor-startup and context switch code is written in assembly to overcome register abstraction provided by HLL. Hence, code is processor-dependent and cross-compiler dependent, it is placed under processor/(tool_processor_format) folder. The folder contains processor specific startup code, linker script, compiler configuration parameters and core configuration file. The startup code initializes the processor code, sets up the C run-time environment with aid of the linker description file. The startup code finally function calls main() for further initialization. The file processor/(tool_processor_format)/Makefile.macros includes variables from external packages and selects xcompiler and hal components. The configuration file Makefile.macros aggregates files from software database and libraries from external database to generate target binary. Thus os/(os_name)/processor/(tool_processor_format)/Makefile.macros is the centralized configuration file that defines components for inclusion while building the final binary.

ADSP-BF527 specific RTEMS configuration and code is placed under os/rtems/bf52x/gcc-bfin-rtms. Operating system libraries are maintained in the external database. The startup file os/rtems/bf52x/gcc-bfin-rtms/start.S contains code for initialization of the processor. As onset, the code sets up stack pointer, at this point the C function call can be used. The startup also code clears the Block Starting with Symbols (BSS) section so that the global symbols are reset to zero. The start and length of the various sections that make up a C program are obtained from the linker description file os/rtems/bf52x/gcc-bfin-rtms/linkcmds. The linker file and configuration file os/rtems/bf52x/gcc-bfin-rtms/bsp_specs describe cross-compiler configuration settings for building the final binary. After this point, basic C environment setup is completed. The code now transitions to use C calling convention. main() is the first function called after processor initialization. os/rtems/bf52x/gcc-bfin-rtms/Makefile.macros contains definition to aggregate code from sw/hal/bf52x, RTEMS libraries from external database and selects cross-compiler tools from sw/xcompiler/gcc-bfin-rtms.

Similarly to ADSP-BF527 RTEMS port, the startup code for ARM926EJS RTEMS port is placed under os/rtems/arm926ejs/gcc-arm-rtms. The startup file os/rtems/arm926ejs/gcc-arm-rtms/start.S contains code for initialization of ARM926EJS processor. As the first step the code sets up the processor interrupt stacks for IRQ, FIQ, UNDEF, SWINT and SVC modes. It then remains in the SVC mode and initializes the Block Starting with Symbols (BSS) section so that the global symbols are reset to zero. Start and length of various sections that make up a C program are obtained from the linker description file os/rtems/arm926ejs/gcc-arm-rtms/linkcmds. The linker file and configuration file os/rtems/arm926ejs/gcc-arm-rtms/bsp_specs describe the cross-compiler configuration settings for building the final binary. After this point the basic C environment setup
CHAPTER 4. RTOS INTEGRATION

is completed. The code now transitions to use C calling convention. `main()` is the first function called after processor initialization. `os/rtems/arm926ejs/gcc-arm-rtems/Makefile.macros` contains definition to aggregate code from `hal/arm926ejs` RTEMs libraries from external database and select cross-compiler from `sw/xcompiler/gcc-arm-rtems`.

The operating system’s processor-independent services are common for all supported processors. Processor-independent operating system code is placed under `<os_name>` or is compiled as library and stored as external package. Generated C code described in section 4.1 interfaces with the operating system code via RAL APIs. The RAL API provide service in four categories

1. **Operating System Management**: handles initialization of the RTOS during system start-up.

2. **Task Management**: Manages creation, deletion and execution of tasks

3. **Synchronization**: Provides API primitives to emulate the SLDL synchronization primitives.

4. **Time modeling**: Provides function to delay execution of task.

The operating system management API’s provide interface for initialization of the operating system. The initialization stages are separated into OS data structure initialization and initialization before peripheral initialization after processor peripheral initialization. The function `OSWrapperInit()` is the first function called from `main()` after processor core initialization. It performs first stage of operating system initialization by initializing the RTEMs data-structures. The second stage of initialization is performed by `OSWrapperInit2()` function. It performs initialization after the processor peripherals are initialized and interrupts are registered. `OSWrapperEnd()` shuts down the operating system and hence is the last call in the model.

RAL’s `OSWrapperInit()` function for RTEMs port initializes RTEMs data structure. The function `OSWrapperInit()` first identifies the workspace for RTEMs, it configures the workspace and setup the heap area. All objects during initialization of RTEMs are allocated from the workspace. `OSWrapperInit()` initializes RTEMs data structures and C library framework. Multi-Tasking is not yet enabled. `OSWrapperInit2()` is a stub function and does not perform any function for RTEMs. `OSWrapperEnd()` starts multi-tasking of the operating system by executing `rtems_initialize_start_multitasking()`. The function `rtems_initialize_start_multitasking()` starts the first task and remains in multitasking mode till completion of all tasks. Finally after all the tasks have exited `rtems_shutdown_executive()` function is executed to stop RTEMs The initialization and shutdown is operating system specific and is not captured by any standard and hence we use RTEMs specific functions to initialize the operating environment.
CHAPTER 4. RTOS INTEGRATION

Tasks are dynamically created and deleted during the lifetime of the application. The RAL task management API provides interfaces for creation, deletion and execution of tasks. A task is an independent thread of execution. Each thread has its own stack space, priority and execution space. RAL API TaskCreate() accepts the entry point for execution, priority and stack space required for creation of tasks. It returns thread handle for identification of the created thread. TaskDelete accepts the thread handle and stops execution of the thread. It deletes the thread from RTOS domain. Threads some time need to wait for completion of other threads before they continue execution, TaskJoin() waits for a thread to complete execution. Thus these basic primitives are used to manage task execution in an RTOS.

```
Listing 4.2: TaskCreate

os_task_handle TaskCreate(
    os_task_fct pFunc, void* arg,
    int stkSize, int prio) {

    thread_id = (pthread_t *)malloc(1, sizeof(pthread_t));
    pthread_attr_init( &attr );
    pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setdetachstate( &attr, PTHREAD_CREATE_JOINABLE );
    pthread_attr_setstacksize( &attr, stkSize );
    pthread_attr_setschedpolicy( &attr, SCHED_OTHER );
    pthread_attr_setschedparam(&attr, &sch_param);
    pthread_create(thread_id, &attr, pFunc, arg);
    return (os_task_handle)thread_id;
}
```

Tasks management interfaces are standardized and RTEMS is POSIX compliant operating system. RAL task management API’s are targeted to RTEMS POSIX interface. Before multitasking starts TaskCreate() collects task parameters. The tasks are created after RTEMS starts multitasking. The listing 4.2 shows the TaskCreate() function for RTEMS port. It allocates memory for the thread (line 5), then pthread_attr_init() sets attributes for explicit scheduling (line 7), detachment (line 8), stack size (line 9), scheduling policy (line 10) and priority. Finally pthread_create() creates the thread and adds to the ready queue for execution (line 12). TaskDelete() uses pthread_detach() function to delete tasks. TaskJoin() joins threads using POSIX call pthread_join(). Thus POSIX interface is implemented for task management interfaces of RAL allowing facile porting to other POSIX compliant operating systems.

The task executing in RTOS communicate and synchronize with each other to perform ac-
activities like transferring data, sending signals etc. The event management feature of RAL provides event and semaphore APIs to synchronize between tasks. An OS kernel object must be allocated and initialized before using the events. The `EventCreate()` function provides the interface to allocate and initialize an event. When the event kernel object is no longer needed it is given back to the memory pool using `EventDelete()`. The SLDL constructs to send and receive events are implemented by `EventNotify()` and `EventWait()` RAL APIs. Similar to events RAL API `SemCreate()` allows allocation and initialization of semaphore, `SemDelete` destroys and de-allocates memory of the semaphore object, `SemAquire` is a blocking call, it acquires a semaphore object if the object is already available else it blocks till the object is available. `SemAttempt` is non-blocking call, it check the semaphore object if available it return success else returns failure. Thus using the semaphore and event constructs RAL implements synchronization and event handling features for SLDL.

POSIX provides task synchronization primitives. Since RTEMS is a POSIX compliant operating system therefore the event and semaphore RAL APIs are implemented using POSIX interface calls. Conditional variable provide facility to broadcast messages to all pending objects. Hence `EventCreate()` RAL API is implemented using conditional variables as shown in listing 4.3. The API allocates an object of type `posix_event_t` (line 2, 3). Then `posix_event_t` uses a conditional variable of type `pthread_cond_t`, mutex of type `pthread_mutex_t` and buffer variable of type int to implement events.

```c
void *EventCreate(void){
    posix_event_t *event =
        (posix_event_t *)calloc(1, sizeof(posix_event_t));
    pthread_cond_init(&(event->cond), NULL);
    pthread_mutex_init( &(event->Mutex_id), NULL );
    pthread_mutex_lock( &(event->Mutex_id) );
    event->buffered_event = 0;
    status = pthread_mutex_unlock( &(event->Mutex_id) );
    return (void *)event;
}
```

The conditional variable is initialized using `pthread_cond_init()` (line 4) and mutex is initialized using `pthread_mutex_init()` (line 5). Since SPECC events are buffered `EventCreate()` function also clears the buffer variable on line 8. Now event structure is ready for use by `EventNotify()` and `EventWait()`.
CHAPTER 4. RTOS INTEGRATION

EventNotify() is a non blocking call, it notifies corresponding EventWait() of an event. If objects are not waiting on event then the event is buffered. EventNotify() uses pthread_cond_broadcast() to notify all objects waiting on the event. EventWait() is a blocking call and the task waits for an event, pthread_cond_wait() blocks till it receives the event. Thus we use POSIX interface calls to implement event functionality. Similar to event, semaphore are implemented using POSIX interfaces. RAL APIs SemCreate(), SemDelete(), SemRelease(), SemAquire() and SemAttempt() are implemented using POSIX APIs sem_init(), sem_destroy(), sem_post(), sem_wait() and sem_getvalue() respectively. Thus entire synchronization primitives are implemented using POSIX interfaces allowing effortless porting to POSIX compliant operating systems.

Thus POSIX interface to RTEMS is use for implementing most of the RAL APIs. The task management and synchronization management uses POSIX 1003.1b implementation. The os management is specific to specific to operating systems and hence does not have POSIX interface. To add a new POSIX compliant operating system the operating system management APIs needs to be re-implemented. Since other APIs have POSIX interface they can be effortlessly used. Thus using POSIX implementation for RAL allows facile addition of POSIX compliant operating system as linux.

4.3 Target Binary Generation

![Figure 4.3: RTOS software stack.](image)

Section 4.1 described components generated by software synthesis stage. The result of C code generation is transformation of logic inside TLM into equivalent ANSI C code implementation. ANSI C code is targeted to RTOS via RAL described in section 4.1. Execution on ISS-based
system models needs platform dependent glue logic for communication drivers, startup code glue between reset to start of C code and the configuration file for linking the software database. The hardware dependent software synthesis generates the interrupt handlers and communication drivers. Hardware dependent software uses decisions made during refinement stages for generating Makefile used for compiling generated C code. Makefile file select os, processor and cross compiler specific Makefile.macros. This Makefile.macros aggregates components from the database and the generated C code for compilation. Thus by including the os, processor and cross compiler specific Makefile.macros, the application is compiled and a target binary is generated for execution on ISS-based platform (Virtual platform). Figure 4.3 shows generated software stack. The C code generation generated SW Application code targeted towards RAL, the hardware dependent software generates the communication drivers, interrupt handlers and RTOS with RAL interfaces. Thus the generated application code can execute on an RTOS via RAL interfaces.

4.4 ISS Based Execution

Generated target binary is executed on ISS-based virtual platform for binary validation. The application can be simulated on ISS much before the actual hardware platform is available. It also allows greater flexibility in debugging and identifying the platform bugs. This reduces time to market compared to traditional way of hardware based validation because validation is performed even before the hardware is available. Since virtual platform execute on the host operating systems therefore they allows additional flexibility of intercepting address or memory access. SCE instantiates Imperas OVP ARM9EJS simulator in the virtual platform generated for ARM926EJS port of RTEMS. Imperas OVP simulator provides techniques to intercept instruction/addresses/function and allow execution of the host function. The benefits are twofold, first the target binary simulate faster because the execution is performed on the host instead of the simulator and second code accessing Memory Mapped Registers can be trapped and host function are executed. Figure 4.4 describes Imperas OVP intercept technique. Imperas simulator allows developers to define intercept libraries, which are loadable shared objects. The intercept library can augment or specify alternate behavior for a particular instruction, when execution reaches a particular address or a symbol. The Imperas simulator utilizes the DWARF information from the generated ELF file to translate a symbol to an address. The intercept library contains a table of target address and corresponding host functions to execute. Imperas OVP simulator breaks execution on the ISS after encountering a matching instruction or address and instead execute a host function registered in the mapping table.
Thus interception allows behavior that would normally occur on a simulated system, to be imple-
mented using features of the host system instead [28]. The standard Imperas OVP provided intercept
functionality for libc functions. But RTEMS implemented the standard IO on RS232 with differ-
ent symbol name. These symbols are not capture in the standard newlib implementation. Hence the
standard Imperas OVP newlib intercept table was augmented to accommodate standard IO functions
provided by RTEMS. Thus the standard IO function of RTEMS are executed on the host thereby
improving execution speed and provide the standard IO messages on the host console.
Chapter 5

Experimental Results

Exploration of RTOS characteristic is a daunting task due to multi-dimensional exploration space as shown in Figure 5.1. Each RTOS shown on Operating System Axis has numerous unique features for comparison as shown by criteria axis. To select a RTOS fulfilling system space, designer needs to explore features for multiple RTOS executing on different ISA. Manual porting and exploration of multiple RTOS'es to identify these features is infeasible. To demonstrate our contribution, an application is automatically synthesize and integrated with a RTOS. Experiments are performed to exploration response-latency and static memory footprint as RTOS characteristics.

To demonstrate automatic integration, AC3 decoder and JPEG encoder are synthesized to execute along with μC/OS-II for ARM926EJS; RTEMS on ARM926EJS and ADSP-BF527. In next experiment, characteristic like footprint and latencies of real-time JPEG encoder application are explored with goal to choose an RTOS. Automatic integration of RTOS allows the system designer to focus on application functionality and performance rather than on porting an application to RTOS. It also allows system designer to explore RTOS selection features. Next section describes experimental setup used for performing experiments.

Figure 5.1: Multi-dimensional RTOS exploration space.
CHAPTER 5. EXPERIMENTAL RESULTS

5.1 Setup

During the execution of an application, RTOS performs multiple context switches between threads and multiple lock, unlocks during synchronization. As RTOS performs these activities, it accesses shared kernel data structures (ready queue, priority map etc). Shared data structures can be corrupted if multiple threads modify the code simultaneously. To avoid data corruption the shared data structure are accessed atomically. To access the code atomically, RTOS disable interrupts at the beginning of atomically accessed code and re-enable interrupts at the end. In multi-threaded application there are multiple atomic block of code called critical section. During execution of critical section the RTOS does not respond to IRQ. Hence critical sections have to be as short as possible.

Accurate measurement of interrupt latency can be obtained by instrumenting the RTOS code. However interrupt latency does not exhibit time taken to respond to the external event. Time to respond to IRQ is the response latency. It is the time elapsed from assertion of HW IRQ to execution of user task. Length of critical sections, memory architecture, pipeline architecture influence response latency. Critical sections are by far the largest contributors to response latency. Hence in experiment setup, the length of response latencies is measured.

![Figure 5.2: Measurement of response.](image)

Figure 5.2 shows behavior of the test setup on arrival of IRQ from HW behavior. A high priority task $T_H$ responds with MMR write to an IRQ. Embedded application Load to be evaluated.
is placed at lower priority than $T_H$. The IRQ received during execution of non-critical code are serviced immediately by the RTOS. As shown in Figure 5.2, RTOS is executing non-critical code at $T_0$, hence it immediately switches to interrupt context, then services the HW IRQ. At end of IRQ a scheduling decision wakes up task $T_H$, which in turn responds with MMR write. HW behavior measures the latency between IRQ and MMR response write. Since the task immediately responds to the IRQ therefore the measured response latency is least.

However, IRQ's received during execution of critical section, wait until completion of critical section. They are serviced at end of critical section. As shown in Figure 5.2, RTOS receives an IRQ at $T_3$ while executing critical code. Since interrupts are disabled, IRQ is not serviced immediately. Servicing of interrupts are delayed till completion of critical section. After critical section completes at $T_4$, RTOS begins processing of IRQ. The response to IRQ is delayed by unfinished length of critical section. Latency to respond to the interrupt depends on where the interrupt hits the RTOS code. Thus using this experiment instead of instrumenting the RTOS, the system designer can measure the latency to respond to an external event on HW. The response latency includes latencies due to critical sections, memory architecture, processor pipeline architecture, interrupt peripherals and bus access.

Application enters RTOS critical section for executing atomic code. Time and length of executing critical section is unique to an application. Response latency depends length of critical section and hence response latency is load specific. JPEG encode is selected as RTOS load in the test setup. JPEG is compute intensive DSP application; it has multiple pipelined and parallel execution stages with data flow between each stage. Pipeline and parallel execution stages in the application are suitable for execution on the RTOS as threads. Figure 5.3 shows specification model of the experiment. HW sends 4096 signals via a channel CH to behavior PE at random
interval. It then waits for change in Register variable, if there is a change then the variable is reset. Time elapsed between assertions of signal to detection of Register change is measured in the behavior. Behavior PE is composed of Response Task and JPEG encoder. Response Task receives signal from channel CH and responds with write to Register variable. JPEG encoder is load on RTOS and it encodes JPEG images. JPEG encode consists of about 10 sub-behaviors for stages receive, quantization, discrete cosine transform, Huffman encoding and output. This specification model is provided as an input to SCE tool. Figure 5.4 shows architecture behavior mapping. Behavior HW is mapped to a custom hardware unit and the behavior PE is mapped to processor. Communication between the custom hardware and processors is through interrupt lines and AMBA bus. Behaviors mapped to the processor are assigned priorities for execution on an RTOS. Response task behavior is assigned the highest priority and behaviors within JPEG encoder are assigned lower priority. This architecture mapping and scheduling information is fed into SCE. SCE encodes architectural and schedule decisions information into TLM models and generates virtual platform. TLM model containing architecture and scheduling information encoding is used by sc2c tool for software synthesis and integration of ISS into VP. The sc2c tool generates C code and hardware dependent software generates the platform dependent code as described in section 4.1. The generated C code is compiled along with RTOS.

The virtual platform and synthesized RTOS, executing the application is shown in Figure 5.5. Ultimately the platform consists of a processor executing an application along with RTOS and a custom hardware. The JPEG encoder contains multiple sub-behaviors and performs multiple context switches during execution. For reason leading to context switch and during context switches, RTOS enters multiple critical sections. In this experiment the response latency is measured without instrumenting RTOS. The custom HW generates interrupt randomly to hit RTOS critical sections.
CHAPTER 5. EXPERIMENTAL RESULTS

Figure 5.5: Experimental setup architecture.

at different points. Using response latency timings, cumulative probability of interrupt latency is calculated and plotted.

5.2 Automatic RTOS Integration

Embedded system designer captures applications behavior in SLDL. Specification model and system designer’s choices are provided as an input to SCE tool. SCE tool refines the specification model to TLM. The resultant TLM contains architectural, scheduling information. Using information from TLM, C code generation tool generates application C code and configuration file to links to RTOS in SW database. Application code along with RTOS is built automatically and hence saves porting and debugging effort for the system designer. To demonstrate automatic RTOS integration, complexity of synthesized real-time application is capture in terms of physical Lines Of Code (LOC) for two DSP application, JPEG encoder and AC3 decoder. The application is split into threads of well defined blocks of execution. Application threads synchronize and communicate with each other and hence use most of RTOS features. Specification model, captured in SPECC as composition of communicating behaviors is shown in Figure 5.6. Design for the experiment consists of a stimulus, an application (either JPEG encoder or AC3 decoder) and a monitor. Stimulus transmits data stream to application via channel CH1, the processed stream is then sent to monitor.
CHAPTER 5. EXPERIMENTAL RESULTS

for validation via CH2. The application is mapped as a real-time software behavior to processor. Software synthesis stage of SCE automatically generates ISS-based system models and C code. The C code is cross-compiled and linked to selected RTOS for execution on ISS-based system model. Number of lines of C code though not accurate can still show the complexity of code. Using target binary’s object dump the number of lines of C code are extracted. The lines of C code are separated into application code and RTOS code. For each of the application the Figure 5.7 shows physical lines of C code. The x-axis shows two sets of bar graphs the first set shows AC3 decoder and the second set shows JPEG decoder. The Y-axis shows physical LOC. System designer captures the specification model hence the system designer writes 1698 LOC for AC3 decoder and 757 LOC for JPEG encoder.

SCE refines the specification model and C code generation tool generates 1883 LOC for AC3 and 1907 LOC for JPEG application. RTOS C code is automatically integrated into the application. The final target binary for AC3 decoder is composed of about 16441 (RTEMS on ARM916EJ-S), 17542 (UCOS on ARM926EJ-S) and 28893 (RTEMS on ADSP-BF527) LOC automatically integrated RTOS C code. Similarly the final target binary for JPEG encoder is composed of about 12594 (RTEMS on ARM916EJ-S), 8673 (UCOS on ARM926EJ-S) and 18130 (RTEMS on ADSP-BF527) LOC automatically generated RTOS C code. JPEG and AC3 decoder are simple application

![Figure 5.7: Complexity in terms of LOC.](image-url)
and use most of the RTOS services and hence the RTOS consume a large share in the target binary. However, as application increases in complexity the share of RTOS in the overall application reduces. Thus using our approach most of the RTOS code can be automatically generated. This allows the system designer to focus on functionality and performance rather than porting an RTOS to the processor. Procedure to refine and synthesize from specification model to target binary is automatic and only needs architectural information from system designer. It takes 50 sec to obtain an application executing with an RTOS on ISS-based system platform. This shows that the RTOS code can be automatically integrated and system designer can focus on functionality and performance rather than porting an RTOS to the processor. In next section automatic RTOS integration feature is utilized to explore RTOS characteristic for JPEG encoder.

5.3 Exploration on ARM926EJS

As a result of our work the system designer can explore between different RTOS'es for the embedded application. TLM containing system designer’s decision information is provided as input to the software synthesis tool. At this stage the system designer has the flexibility in choosing either RTEMS or μC/OS-II as an operating system for executing the application. The effort for system designer is iota of work to select between two RTOS in the graphical SCE environment. Synthesis of application code and RTOS requires minuscule effort. Automatic synthesize of application and targeting of the application to different RTOS'es allows the system designer to explore RTOS characteristics. In next section response latency and footprint are measured between RTEMS and μC/OS-II on virtual platform consisting of imperas OVP ARM926EJS ISS model and a custom hardware.

5.3.1 Response Time

5.3.1.1 RTEMS

Synthesized application along with RTEMS executes on ARM926EJS ISS model and response latency is measured on custom hardware. Some of the interrupts arrive at RTEMS while executing critical section, leading to higher response latency. Figure 5.8 shows cumulative probability of response latency for RTEMS executing on an ARM926EJS platform. The x-axis denotes the response latency in cycles. The y-axis denotes the cumulative probability of completing the interrupt. RTEMS on ARM926EJS needs a minimum of 1226 cycles to process an interrupt and
CHAPTER 5. EXPERIMENTAL RESULTS

Figure 5.8: RTEMS ARM926EJS Response Time.

respond with an MMR write. Response time includes it includes interrupt context switch, ISR, task context switch and a write to MMR. Probability of processing interrupts in minimum time is 0.4, that shows none of RTEMS critical section with probability of 0.4. Critical sections are of varying length. An interrupt that hits shorter pending critical section has shorter response time. Shorter critical section have steeper slope. Graph has slope starting at probability of 0.4, slope of the curve relates RTEMS guarantees that interrupts will be processed within a maximum of 1686 cycles on ARM926EJS. This measurement includes the length of longest critical section. It shows that RTEMS has longest critical section of length 460 cycles (1686 – 1226). Thus using this graph, system designer can calculate probability of RTEMS to respond to external stimulus.

5.3.1.2 μC/OS-II

In second exploration, μC/OS-II is synthesized for execution on ARM926EJS OVP model base virtual platform. The model executes with a quantum of single instruction. Figure 5.9 shows response time for μC/OS-II executing on an ARM926EJS platform. The x-axis denotes the response time in cycles. The y-axis denotes the cumulative probability. Figure 5.9 shows that μC/OS-II on ARM926EJS needs a minimum of 580 cycles to respond to interrupt. The probability of not hitting any critical section and responding in 580 cycles is 0.6. μC/OS-II guarantees that, interrupts will be processed within a maximum of 994 cycles. This measurement includes length of longest critical section interrupt context switch, ISR, task context switch and a write to MMR. The length of longest critical section hit is of length 414 cycles (994 – 580). Thus using this graph, system designer can
calculate probability of \( \mu \)C/OS-II to respond to external stimulus.

Table 5.1: Measured interrupt response latency on ARM926EJS

<table>
<thead>
<tr>
<th>RTOS PORT</th>
<th>Min</th>
<th>Avg</th>
<th>50% tile</th>
<th>90% tile</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTEMS</td>
<td>1226</td>
<td>1303</td>
<td>1258</td>
<td>1434</td>
<td>1686</td>
</tr>
<tr>
<td>( \mu )C/OS-II</td>
<td>580</td>
<td>599</td>
<td>582</td>
<td>650</td>
<td>994</td>
</tr>
</tbody>
</table>

Table 5.1 summarizes the response latency measurements. When executing RTEMS on ARM926EJS, 50% of the interrupts will be responded by 1258 cycles, 90% of the interrupts will be delayed by 1434 cycles. While executing \( \mu \)C/OS-II on ARM926EJS 50% of interrupts are responded within 582 cycles and 90% of the interrupts are delayed for 650 cycles. Variance in the response latency on virtual platform is due to critical section. Memory latency does not influence since the VP models memory as cache and hence there are no stalls while fetching data from memory. Table 5.1 show that \( \mu \)C/OS-II has lower response time due to shorter critical sections compared to RTEMS. Clearly \( \mu \)C/OS-II is superior compared to RTEMS on ARM926EJS platform. Thus using response time measurement, system designer can choose or design the system appropriately to meet applications real-time deadlines.

5.3.2 Static Memory Footprint

Apart from response time there are large numbers of other RTOS parameters that a system designer can consider during exploration. Combination of these characteristics fulfilling system de-
CHAPTER 5. EXPERIMENTAL RESULTS

Sign parameters, identify the optimal RTOS for embedded application. Target binary is composed of read-only code, data and read-write data. The target binary and if required a loading program is stored on non-volatile memory of the embedded platform. At boot-time the complete or part of target binary is copied to RAM for faster execution. At execution time the application may dynamically allocate and free memory for heap and stack objects. However, static memory footprint does not capture dynamic allocation but captures the minimum memory required. Since most embedded systems have tight memory constrains therefore static memory footprint of the target binaries is measured in second parameter exploration.

Figure 5.10 shows static memory footprint, x-axis shows the RTOS ports and y-axis shows size in bytes. RTEMS port of the application needs at least total of 197480 bytes of memory, consisting of 59800 bytes of read-only memory and 38580 bytes of read-write memory for execution. Application executing on µC/OS-II for ARM926EJS needs total of 169819 bytes of memory, consisting of 79799 read-only memory and 90020 bytes of read-write memory. The comparisons show that µC/OS-II has a smaller memory footprint for the experimental application. Thus using memory footprint information system designer can explore different compiler, optimization flags and select the operating system that fulfills memory constrains of the system.
CHAPTER 5. EXPERIMENTAL RESULTS

5.4 Exploration on ADSP-BF527

5.4.1 Static Memory Footprint

In traditional design methodology HW is built and then software is developed on top of the physical hardware. However, SCE allows validation of target binary on Virtual Platform (VP) even before the hardware platform is available. After exploration of design space as described in Chapter 3, an optimal platform is selected. Actual hardware similar to the designed virtual platform can then be built. Application example described in section 5.1 is used to demonstrate execution on actual hardware and ADSP-BF527 based virtual platform. Figure 5.11 shows the same target binary can execute on VP and hardware platform. The virtual platform that is generated contains ADSP-BF527 ISS and is similar to TLL6527M platform described in Chapter 3. Target application encodes raw image of size 116x96 to compressed JPEG image. The generated target binary needs a minimum of 256576 bytes consisting of 147164 of read-only memory and 109412 bytes of read-write memory. It is executed on ISS-based system model consisting of instruction accurate simulator. To encode 116x96 pixels of raw image to JPEG the application consumes average 81428921 cycles (160ms). Identical target binary is loaded onto physical hardware platform’s SDRAM for execution. The execution takes about 163929567 cycles (330ms). This shows that identical code can be executed on VP and HW.
CHAPTER 5. EXPERIMENTAL RESULTS

5.4.2 Response Time

The custom hardware is replaced by general purpose timer, to generate 4096 interrupts periodically at 1ms interval on TLL6527M. Figure 5.12 shows the response time for RTEMS executing on an TLL6527M platform. The x-axis denotes the response time in cycles. The y-axis denotes cumulative probability. Figure 5.12 shows that RTEMS on ADSP-BF527 needs a minimum of 7359 cycles to process an interrupt. RTEMS on ADSP-BF527 guarantees that the interrupt will be processed within a maximum of 14519 cycles.

Similarly, identical target binary is executed on ADSP-BF527 instruction accurate simulator with an execution quantum of 500 instructions. Figure 5.13 shows the response time for RTEMS executing on TLL6527M platform. The x-axis denotes the response time in cycles. The y-axis denotes cumulative probability. Figure 5.12 shows that RTEMS on ADSP-BF527 needs a minimum of 2540 cycles to process an interrupt on the virtual platform. RTEMS on ADSP-BF527 guarantees that the interrupt will be processed within a maximum of 4500 cycles.

<table>
<thead>
<tr>
<th>RTEMS interrupt response latency</th>
<th>RTOS PORT</th>
<th>Min</th>
<th>Avg</th>
<th>50% tile</th>
<th>90% tile</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware Platform</td>
<td>7359</td>
<td>9625</td>
<td>7839</td>
<td>8154</td>
<td>14519</td>
<td></td>
</tr>
<tr>
<td>Virtual Platform</td>
<td>2450</td>
<td>3210</td>
<td>3000</td>
<td>3500</td>
<td>4500</td>
<td></td>
</tr>
</tbody>
</table>

![Figure 5.12: RTEMS ADSP-BF52X Response Time on TLL6527M.](fig/rtems_bf52x_iss.pdf)

![Figure 5.13: RTEMS ADSP-BF52X Response Time on VP.](fig/rtems_bf52x_iss.pdf)
CHAPTER 5. EXPERIMENTAL RESULTS

Table 5.2 summarizes response latency of RTEMS on ADSP-BF527 VP and HW. There is a gap by a factor of more than 2 between measurements on VP and HW. The difference can be attributed to multiple factors like memory access latency, memory bank access latency, target binary layout on memory, accuracy of ISS model, execution quantum and bus model. However, the experiments show that an application can build and validate on ISS-based virtual platform. And after the hardware is available the target binary with almost no modification to the source code can be executed on the hardware.

![Graph comparing latency vs footprint](image)

Figure 5.14: Selection of RTOS.

Thus, system designer can measure RTOS characteristics on accurate platform, explore response latency and memory footprint for multiple RTOS. Figure 5.14 shows graph comparing latency vs footprint that were explored during experiments. Area in green is defined acceptable system requirements. RTEMS on ADSP-BF527 fulfills size requirements but fails on latency requirements of the system. RTEMS and μC/OS-II both fulfill system requirements. However, μC/OS-II clearly has better parameters than RTEMS and hence best suits our requirements. Additional exploration such as cost, royalty, community and other factors as described in chapter 2 can also be explored for RTOS selection.
Chapter 6

Conclusion

RTOS are used in most modern embedded systems. RTOSes support dynamic scheduling of software tasks on processing elements. The scheduling policy is usually based on priorities of the software task. Assignment of priorities to tasks defines the overall system behavior. Change in the assigned priorities, modifies the system behavior. In the worst case wrong mapping of task priorities can lead to deadlocks and prevent tasks from execution. It may ultimately cause real-time system to miss critical deadline. The affects of missing deadline can be catastrophic for a real-time system. SCE generates application C code and our work has enabled the system designer to automatically integrate RTOS code into the generated C application code. The generated code can then be executed on an Instruction Set Simulator based Virtual platform. This facilitates validation and exploration of application task priority assignment early. The procedure to get from specification model to refinement is automatic and only needs architectural and scheduling information from system designer. Automatic generation allows system designer to focus on functional correctness rather than porting an RTOS onto the platform. This encourages functional and design space exploration.

As a result of our work system designer can choose between RTOSes. TLM containing architectural information is provided as in input to the software synthesis tool. At this stage system designer has the flexibility in choosing either RTEMS or uCos-II as an operating system for executing the application. The effort is iota of work to select between two RTOSes in the graphical SCE environment. It is easy to generate application code targeted to different RTOSes with minuscule effort. Automatic synthesize of application and targeting of the application to different RTOSes allows system designer to explore RTOS characteristics. The system designer can measure RTOS characteristics, weigh the results and then select an RTOS for the embedded application.

Our work introduces support for commonly used RTOSs into ESL for both ARM
CHAPTER 6. CONCLUSION

and Blackfin processor. Use of POSIX standard for interfacing the generated software to RTEMS allows future integration of POSIX compliant operating systems into SCE.

Thus in this theses RTEMS was ported to ADSP-BF527 processor, SCE software database is augmented with RTEMS for ARM926EJS and ADSP-BF527 processor. This work will enable SCE users to explore application with different RTOS on ARM926EJS and ADSP-BF527 processors.
Chapter 7

Future Work

In this thesis, RTEMS is added to software database to allow system designer to explore multiple RTOS'es. POSIX interface is used to implement RTEMS interface with SW database interface. There are plethoric RTOS available in the market, which are not yet been added into the SW database. New RTOS'es compliant to POSIX can use interface identical to RTEMS for implementation of task management, synchronization management and time management. However minor changes in RTOS specific initialization code is needed for RTOS to be integrated into the SW database. The effort is minimal and can be taken up in the future to add variety of RTOS into SW database.

System designer can synthesized RTOS code and execute on ISS-based system platform. RTOS introduces latencies for services it provides for application execution. System designer has to synthesize application, generate target binary and execute on ISS-based execution for identifying latencies introduced by RTOS. Also, execution on ISS-based system platforms is slow. However, RTOS latencies can be identified once and then back-annotated during scheduling refinement phase in SCE. System designer can then explore RTOS'es at scheduling refinement stage, without executing on ISS-based system model.

Generated C code is target towards an abstract RTOS added into the SW database, implements the interface for abstract RTOS. There is minimal overhead caused due to the interface of abstract RTOS. In future, the C code generation tool can be re-implemented to generate code targeted towards selected. Thereby eliminating the need for an abstract RTOS. Since application C code is generated, the C code generation tool can identify synchronization objects, task objects and other RTOS configuration. This would aid in configuring RTOS automatically to system designers requirements.
Bibliography


BIBLIOGRAPHY


[23] Analog Devices Inc. ADSP-BF52x Blackfin Processor Hardware Reference, (0.31), 5 2008.

BIBLIOGRAPHY


BIBLIOGRAPHY


