NTNU
Det skapende universitet

En Time Management Unit (TMU) for sanntidssystemer

Bjørn Forsman

Master i teknisk kybernetikk
Oppgaven levert: Juni 2008
Hovedveileder: Amund Skavhaug, ITK

Norges teknisk-naturvitenskapelige universitet
Institutt for teknisk kybernetikk

Oppgavetekst

I et sanntidssystem må tidsfristene nås, ellers degraderes eller feiler systemet. For å kunne garantere at tidsfristene blir nådd må programmenes verste kjøretid (WCET) være kjent, maskinvaren til systemet må være predikterbar og hyppigheten av eksterne avbrudd må være begrenset. Å oppnå alle disse faktorene er vanskelig og derfor er hensikten med denne oppgaven å utvikle en Time Management Unit (TMU) som kan måle og begrense kjøretiden prosessoren bruker på vanlige prosesser og avbruddsrutiner under kjøring.

Prosjektet kan deles inn i mindre oppgaver:
*Utfør et litteratursøk omkring temaet
*Lag en spesifikasjon for TMUen
*Evaluer mulige plattformer for implementering og diskuter fordeler og ulemper
*Utfør en implementering på en passende platform
*Test implementasjonen for å verifisere at designet fungerer etter spesifikasjonene

Oppgaven gitt: 07. januar 2008
Hovedveileder: Amund Skavhaug, ITK

Problem description

A Time Management Unit (TMU) for Real-Time Systems

In real-time systems the time deadlines must be met or the system degrades or even fails. To guarantee that the deadlines are met, the worst-case execution time (WCET) of the programs must be known, the underlying hardware must be deterministic and the occurrences of interrupts must be bounded. All these factors may be hard to achieve, therefore this project aims to provide a Time Management Unit (TMU) that is capable of measuring and controlling the execution time spent by tasks and interrupt service routines (ISRs).

The assignment can be split into several smaller tasks:

1. Perform a background research on the subject and related work
2. Propose a design for the TMU
3. Evaulate possible implementations of the proposed TMU and discuss its pros and cons
4. Implement the TMU on a suitable platform according to its proposed design
5. Perform tests on the implementation to verify that the design is operating according to its specifications

Assignment given:	2008-01-07
To be handed in by:	2008-06-23
Supervisors:	Amund Skavhaug (ITK)
	Bjørn B. Larsen (IET)

Preface

This master thesis is written as a compulsory part of the study for the degree of Master of Science in Engineering Cybernetics with the specialisation Dedicated Computer Systems at the Norwegian University of Science and Technology (NTNU).

Although my native language is Norwegian, I have chosen to write this report in English because most literature on the subject at hand is written in English, and also because I would like to improve my English skills.

The thesis is written in the LaTeX document preparation system and the figures are created with the diagram creation program Dia.

I would like to thank my supervisor Amund Skavhaug for his guidance and inspiration throughout this project and both Mikael K. Eriksen and Christopher T. P. de Lange for their help on the report. Also, I would like to thank Jan Anders Mathisen for being kind enough to lend me his FPGA development board.

Trondheim, 2008-06-23

Bjørn Forsman

Abstract

In real-time systems the time deadlines must be met or the system degrades or even fails. To support timely behaviour, the real-time system must be predictable at the hardware and software level. Estimating the Worst Case Execution Time (WCET) of any non-trivial code is difficult and when taking into account the underlying hardware with features like deep execution pipelines, caches and DMA, the overall effort will be unmanageable [20]. Further, real-time systems are often subjected to unbounded external stimuli, known as asynchronous events or interrupts, which may drive the system into overload. Because of these two key problems, there is a need for a dynamic approach to measuring and controlling the execution times of tasks and interrupt service routines (ISRs).

In this project, a study of scheduling in real-time systems have been carried out. Based upon this study, a hardware unit called Time Management Unit (TMU) is designed aiding real-time systems in keeping deadlines by measuring and controlling the execution times of tasks and asynchronous events. A test system was implemented, comprising an Field Programmable Gate Array (FPGA) with a LEON3 soft-core processor running the eCos embedded operating system.

The result is that with the use of the proposed TMU, execution times for tasks and ISRs can be bounded. Thus one can achieve system predictability and guarantee that the system will not be overloaded, if the unit is used appropriately. The TMU introduces very little processor time overhead and utilise little to moderate amounts of FPGA logic resources.

The principal conclusion is that the TMU brings predictability to real-time systems operating in unpredictable environments at a relatively low cost; both concerning processor overhead and the increased use of FPGA logic resources.

Contents

List of Abbreviations

API .... Application Programming Interface
ATC .... Asynchronous Transfer of Control
CPLD .... Complex Programmable Logic Device
CPU .... Central Processing Unit
DMA .... Direct Memory Access
DS .... Deferrable Server
DSR .... Deferred Service Routine
DUT .... Device Under Test
EDF .... Earliest Deadline First
FPGA .... Field Programmable Gate Array
GNU .... GNU's Not Unix
GPL .... General Public License
GUI .... Graphical User Interface
HDL .... Hardware Description Language
ISA .... Instruction Set Architecture
ISR .... Interrupt Service Routine
KIB .... kibibit = 2^{10} bits = 1,024 bits
LGPL .... Lesser General Public License
LUT .... Look-Up Table
MIB .... mebibyte = 2^{20} bytes = 1,024 kibibytes
OS .... Operating System
PS .... Polling Server
RTOS .... Real-Time Operating System
SOC .... System-on-Chip

SS . . . . Sporadic Server

TMU . . . . Time Management Unit

VSR . . . . Vector Service Routine

WCET . . . . Worst Case Execution Time

XSR . . . . Exception Service Routine

Chapter 1

Introduction

Real-time computing systems play a vital role in our society, and they cover a spectrum from the very simple to the very complex. Examples of real-time systems include multimedia players, microwave ovens, control systems for cars, airplanes, military weapon systems and factory control systems.

In real-time computing the correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced. Typically, a real-time system consists of a controlling system and a controlled system. For example, in an automated factory the controlled system is the factory floor with its robots, assembling stations and the assembled parts. The controlling system is the computer and human interfaces that manage and coordinate the activities on the factory floor. Thus, the controlled system can be viewed as the environment with which the computer interacts.

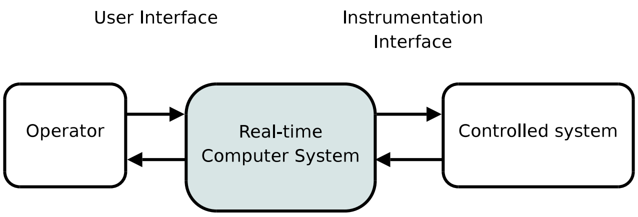


Figure 1.1: Real-time system

The controlling system interacts with its environment, the controlled system, via sensors and actuators. It is imperative that the percepted state of the environment, as seen by the controlling system, is consistent with its actual state. Otherwise, the controlling systems' activities may have disastrous effects. Hence, periodic monitoring and timely processing of the information is necessary.

1.1 Motivation

Knowing the Worst Case Execution Time (WCET) is of prime importance for the timing analysis of hard real-time systems. WCET analysis is becoming more difficult as modern computer systems become more complex. Features like deep execution pipelines, branch prediction, caches and virtual memory, all lack deterministic behaviour, and must be included in the WCET analysis. Estimating the WCET of any non-trivial code is in general very difficult, and when taking into account the unpredictable hardware, WCET analysis requires unmanageably high efforts [20]. Also, real-time systems are often subjected to unbounded external stimuli, known as asynchronous events or interrupts, which may drive the system into overload. To lower the chances of missing deadlines, systems have to be over-dimensioned to reduce the risk of an unexpected temporal resource shortage during operation. Because of this, there is a need for a dynamic measurement and control of the execution times of tasks and asynchronous events.

This thesis will focus on how to make a hardware unit, called a Time Management Unit (TMU), able to improve the predictability of real-time systems based on standard processor architectures with unpredictable timing behaviour. The TMU will limit the execution times of tasks as well as the occurrences of interrupts. The unit shall preferably be architecture independent and not depend on any special programming language feature.

1.2 Work in the same field

Hardware platforms for real-time system have gained little interest compared to that of the software, where programming language features and scheduling algorithms have been thoroughly studied. An accurate Time-Management Unit for real-time processors was proposed by Kailas and Agrawala [19]. It is a fine granularity clock embedded in CPUs solving problems with the software clock approach, updating a memory location with a new time value at regular intervals, controlled by an external interrupt. The unit is not intended for measuring or controlling execution times. Others have studied processor support for temporal predictability; The SPEAR design example [7] is a processor design with predictable timing and interrupt response aimed at hard real-time systems. Colnaric and Halang [6] presents an asymmetrical multiprocessor architecture for hard real-time systems, whose temporal behaviour is fully predictable. As for software, Puschner and Koza [26] created new programming language features making WCET analysis easier.

1.3 Thesis organisation

This thesis is organised as follows:

- Chapter 2 presents the necessary background literature regarding real-time systems, scheduling theory, FPGA development and the basic software tools used in this project.
- Chapter 3 describes the system development phase. This includes the proposed TMU's theory of operation, implementation choices, host and target development platforms, HDL design and implementation of the TMU, hardware simulations and the modifications to eCos for TMU support.
- Chapter 4 presents the final tests performed on the implemented system.
- Chapter 5 gives a discussion of the project and the work that has been done.
- Chapter 6 presents the conclusion of this project.
- Chapter 7 gives an overview of further work.

Chapter 2

Background literature

The purpose of this chapter is to present real-time systems and RTOSs, basic concepts of scheduling, and to introduce FPGAs and soft-core processors. And finally to describe the basic tools and utilities used in this project.

2.1 Real-time system definition

There are several definitions of a real-time system; however, they all have in common the notion of response time – the time taken for the system to generate output from some associated input. The Oxford Dictionary of Computing gives the following definition:

> Any system in which the time at which the output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to that same movement. The lag (delay) from the input time to the output time must be sufficiently small for acceptable timeliness.

Krishna and Shin [20] have another definition:

> A real-time system is anything that we, the authors of this book, consider to be a real-time system. This includes embedded systems that control things like aircraft, nuclear reactors, chemical power plants, jet engines, and other objects where Something Very Bad will happen if the computer does not deliver its output in time. These are called *hard* real-time systems. There is another category called (not surprisingly) *soft* real-time systems, which are systems such as multimedia, where nothing catastrophic happens if some deadlines are missed, but where the performance will be degraded below what is generally considered acceptable. In general, a real-time system is one in which a substantial fraction of the design effort goes into making sure that task deadlines are met.

2.2 Real-Time Operating System

A Real-Time Operating System (RTOS) is a multitasking operating system intended for real-time applications. The RTOS provides an abstraction layer that hides the hardware details from the application software. The abstraction layer typically consists of the following five categories:

1. Task management
2. Intertask communication and synchronisation
3. Timers
4. Dynamic memory allocation
5. Device I/O

The most basic job of the RTOS kernel is task management. This set of services allows application software developers to design their software as a number of separate pieces of software known as *tasks* – each handling a distinct topic, a distinct goal, and perhaps its own real-time deadline. Services in this category include the ability to create and control tasks and assign priorities to them. The main RTOS service in this category is the scheduling of tasks as the embedded system is in operation.

The scheduler controls the execution of application software tasks, and is a component of RTOSs that have been studied thoroughly. Most RTOSs do their scheduling of tasks using a scheme called priority-based preemptive scheduling. Each task in a software application must be assigned a priority, with higher priority values representing the need for quicker responsiveness. Very quick responsiveness is made possible by the preemptive nature of the task scheduling. Preemptive means that the scheduler is allowed to stop any task at any point in its execution, if it determines that another task needs to run immediately. The basic rule that governs priority-based preemptive scheduling is that at every moment in time, the highest priority task that is ready to run, will be the task that must be running. In other words, if both a low-priority task and a higher-priority task are ready to run, the scheduler will allow the higher-priority task to run first. The low-priority task will only get to run after the higher-priority task has finished with its current work. But sometimes, the task with the highest priority depends on a resource held by a lower priority task. This scenario is called *priority inversion*. This causes the execution of the high priority task to be blocked until the low priority task has released the resource, effectively inverting the relative priorities of the two tasks. If some other medium priority task, that does not depend on the shared resource, attempts to run in the interim, it will take precedence over both the low priority task and the high priority task. Solutions to this problem are disabling all interrupts to protect critical sections, the priority ceiling protocol or priority inheritance.

The second category of kernel services is intertask communication and synchronisation. These services make it possible for tasks to pass information from one to another, without danger of that information ever being damaged. They also make it possible for tasks to coordinate, so that they can productively cooperate with one another. Without the help of these RTOS services, tasks might communicate corrupted information or otherwise interfere with each other. Message passing, mutexes, semaphores, event flags and signals are examples of services in this category.

Since many embedded systems have stringent timing requirements, most RTOS kernels also provide some basic timer services, such as task delays, event counters and alarms.

Some RTOSs provide dynamic memory allocation and device drivers.

2.2.1 What makes an OS real-time?

Many general purpose operating systems provide similar kernel services as real-time operating systems. The key difference between general-computing operating systems and real-time operating systems is the need for deterministic timing behaviour in the real-time operating systems. Formally, deterministic timing means that operating system services consume only known and expected amounts of time. In theory, these service times could be expressed as mathematical formulas. These formulas must be strictly algebraic and not include any random timing components. Random elements in service times could cause random delays in application software and could then make the application randomly miss real-time deadlines – a scenario clearly unacceptable for a real-time embedded system.

An RTOS does not necessarily have high throughput; rather, an RTOS provides facilities which, if used properly, guarantee deadlines can be met generally (soft real-time) or deterministically (hard real-time). Designers of general purpose OSs on the other hand, will strive for better throughput, not determinism. An RTOS is valued more for how quickly and/or predictably it can respond to a particular event than for the given amount of work it can perform over time.

2.3 Scheduling

It is the job of the scheduling algorithm and operating system scheduler to provide predictability to the system and to coordinate resources to meet the timing constraints of the physical system. Traditionally, real-time systems have used the cyclical executives approach for scheduling. Cyclical executives provide a deterministic schedule for all tasks and resources in a real-time system by creating a static timeline upon which tasks and resources are assigned specific time intervals. While such an approach is manageable for simple systems, it quickly becomes unmanageable for large systems. It is a painful process to develop application code that fit in the time slots of a cyclical executive while ensuring that the critical sections of different tasks do not interleave. This approach is typically expensive to create, verify, and update.

An alternative approach uses preemptive and priority-driven scheduling algorithms to schedule tasks. A scheduling algorithm is a set of rules that determines the task to be executed at a particular moment. This means that whenever there is a request for a task of higher priority than the one currently being executed, the running task is immediately interrupted and the newly requested task is started. Thus the specification of such algorithms amounts to the specification of the method of assigning priorities to tasks. Using well defined algorithms to schedule tasks in a real-time system yields an understandable scheduling solution.

The preemptive real-time scheduling algorithms can be broadly classified into two categories: *static priority* and *dynamic priority*. This classification is based on the manner in which priorities are assigned to tasks. A scheduling algorithm is said to be static if priorities are assigned to tasks a priori and they do not change during run-time. A static algorithm is also called a fixed priority scheduling algorithm, an example of which is the Rate Monotonic (RM) algorithm. A scheduling policy is said to be dynamic if priorities of a task might change from request to request. The Earliest Deadline First (EDF) algorithm falls under the category of dynamic priority scheduling policy. Before discussing these algorithms in detail, some of the terms associated with real-time scheduling theory will be explained.

A task is a thread of execution performing a specific function. For example, a task could be a simple thread polling the serial port to check if any data has arrived. A real-time task can be classified as periodic or aperiodic depending on its arrival pattern or as soft or hard based on the consequences of a missed deadline.

2.3.1 Scheduling periodic tasks

Tasks with regular arrival times are called periodic. A common use of periodic tasks is to process sensor data. For example, a temperature monitor of a nuclear reactor should be read periodically to detect any changes promptly. Tasks with irregular arrival times are aperiodic tasks and are used to handle the processing requirements of random events such as operator requests.

Each of the tasks must complete execution before some fixed time has elapsed since its request. This fixed time is known as the deadline of the task. If meeting a given task's deadline is critical to the system's operation the task is called a hard real-time task. If missing occasional deadlines of a particular task does not adversely affect the system's performance it is a soft real-time task.

2.3.2 Rate Monotonic Scheduling

Liu and Layland presented an optimal fixed priority algorithm known as the Rate Monotonic (RM) algorithm [22]. In this algorithm, priorities are assigned according to the request rate (frequency) of tasks. Specifically, tasks with higher frequency will have higher priority. The Rate Monotonic priority assignment is optimum in the sense that no other fixed priority assignment rule can schedule a task set which cannot be scheduled by the RM algorithm.

One drawback of the RM algorithm is that the schedulable bound is less than 100 %. Schedulable bound is the maximum value of the CPU utilisation for a set of tasks up to which all tasks will be guaranteed to meet their deadline. Since C_i/T_i is the fraction of processor time spent in executing task τ_i, the total CPU utilisation for a set of n tasks is:

$$U = \sum_{i=0}^{n} C_i/T_i \tag{2.1}$$

where C_i is the worst-case execution time and the period is T_i.

Liu and Layland [22] proved that the worst-case schedulable bound W_n for n tasks is:

$$W_n = n\left(2^{1/n} - 1\right) \tag{2.2}$$

This worst case utilisation bound decreases monotonically from 0.83 when $n = 2$ to 0.693 as n approaches infinity. This shows that any periodic task set of any size will meet deadlines all the time if the RM algorithm is used and the total utilisation is not greater than 0.693. Note that this is a sufficient but not necessary schedulability test. The task set may still be schedulable even if the processor utilisation is above the schedulable bound.

2.3.3 Earliest Deadline First

Liu and Layland [22] also present a dynamic scheduling algorithm called the Earliest Deadline First (EDF) because the task with the nearest deadline for its current request has the highest priority. At any instant, the task with the highest priority and an unfulfilled request gets hold of the CPU. This contrasts with the RM algorithm in which priorities do not change with time. One drawback of the EDF algorithm is the higher scheduling overhead with respect to the RM algorithm, as it has to dynamically compute priorities of tasks.

The necessary and sufficient condition for the feasibility of a task set with EDF is given below:

$$U = \sum_{i=0}^{n} C_i/T_i \leqslant 1 \tag{2.3}$$

where, n is the total number of tasks.

The ratio of C_i/T_i is the CPU utilisation by the ith task. The above condition is necessary because the tasks should not overload the CPU if they have to meet all the deadlines. EDF is an optimal algorithm in the sense that if a task set can be scheduled by any algorithm, it can also be scheduled by EDF.

Dynamic priority scheduling with the EDF algorithm has a distinct advantage over fixed priority scheduling: the schedulable bound for EDF is 100 % for all task sets. This means that you can fully utilise the computing power of the CPU. The major problem with the EDF algorithm is that there is no way to guarantee which tasks will fail in a transient overload. With RM, low priority tasks are always the first to fail. However, no such priority assignment exists with EDF.

2.3.4 Scheduling Aperiodic Tasks

A task with random arrival times is called an aperiodic task. A sporadic task is a subset of aperiodic tasks, having with a minimum interarrival time. Aperiodic tasks result from operator actions or asynchronous events. Non-critical aperiodic tasks can be executed as background tasks. On the other hand, if the aperiodic task is critical it can be incorporated into the RM algorithm through the use of a periodic server, which is a periodic task whose function is to service one or more aperiodic tasks. The simplest periodic server is the Polling Server (PS) and is illustrated in Figure 2.1. The PS is activated every period. If no aperiodic tasks are available, the server suspends itself until the next period. The major disadvantage of this approach is unpredictable response times for aperiodic tasks. An aperiodic task arriving just after the PS has been invoked, has to wait for the server's next period.

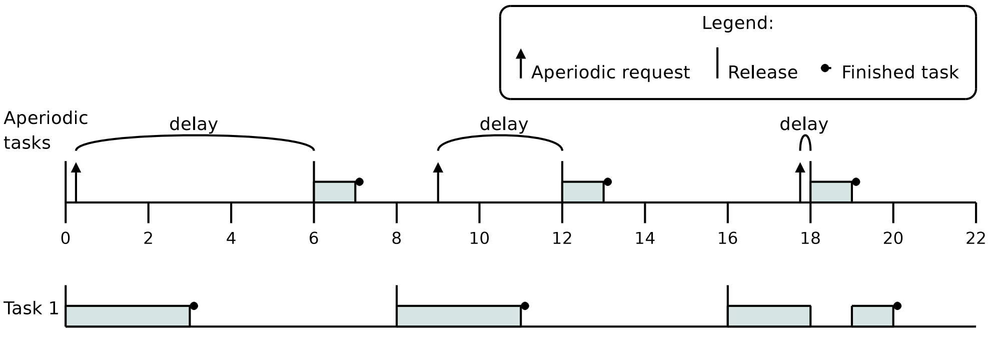


Figure 2.1: A polling server for handling aperiodic tasks

An efficient scheduler should be able to meet the hard deadlines of periodic tasks and provide a fast average response time for aperiodic tasks. Two of the algorithms that strive to achieve this goal are: the Deferrable Server (DS) algorithm and the Sporadic Server (SS) algorithm. These algorithms overcome the major limitation of the polling method where the aperiodic task arrives after the polling instant.

Deferrable Server Algorithm

Lehoczky, Sha and Strosnider proposed an algorithm for scheduling aperiodic tasks called the Deferrable Server (DS) algorithm [38]. This algorithm creates a periodic server τ_i with C_i execution time, or execution *budget*, with a priority defined by the server's period T_i. The DS has the entire period to use its C_i budget at priority P_i. When the server has consumed its budget, its priority is set to background – the lowest priority of the system, only allowing it to run when no periodic tasks are available. At the beginning of each period, the budget is set to C_i. If the DS has any remaining budget from the previous period, it will be discarded.

The difference between the PS and the DS is that a DS can service an aperiodic task anytime during its period provided that it still has some unused execution time. On the other hand, a polling server can only service an aperiodic task that is pending at the start of its period.

Figure 2.2 illustrates a schedule of two periodic tasks and a DS which handles aperiodic tasks. Vertical arrows indicate an aperiodic request and vertical lines represent the release of periodic tasks. The release of the DS means that its budget is replenished. The tasks are ordered by priority, although once the DS has spent its budget, it will receive the lowest priority.

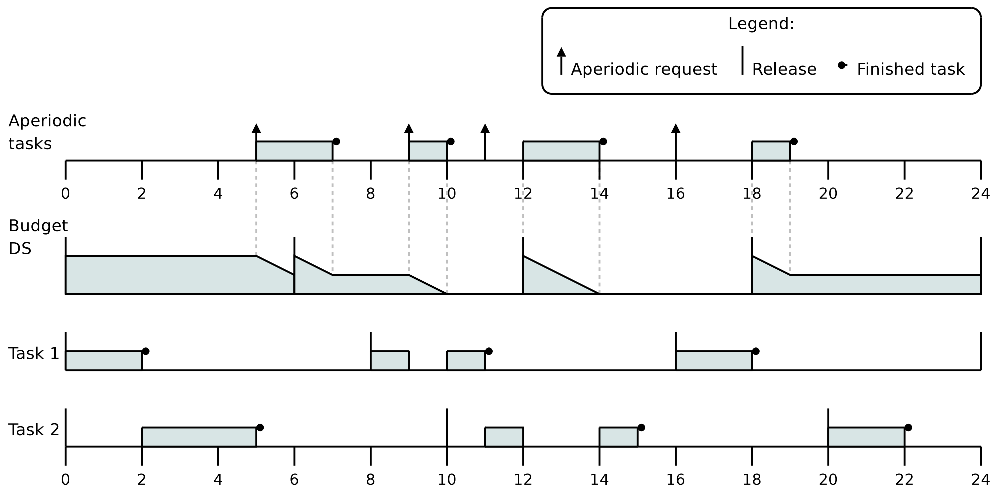


Figure 2.2: An example schedule with a Deferrable Server

Sporadic Server Algorithm

Another algorithm for scheduling soft deadline aperiodic tasks was given by Sprunt, Sha and Lehoczky [34], called the Sporadic Server (SS). The SS algorithm creates a high priority task for servicing aperiodic tasks and preserves its server execution time at its high priority level until an aperiodic request occurs. It differs from the DS algorithm in the way it replenishes its server execution time. Thus, the sporadic server does not recover its capacity to its full value at the beginning of each new period, but only after it has been consumed by aperiodic task execution. More precisely, the sporadic server replenishes its capacity each time TA it becomes active and its capacity is greater than 0. The replenishment time is set to TA plus the server period. The replenishment amount is set to the capacity consumed within the interval TA and the time when the sporadic server becomes idle or its capacity has been exhausted. The sporadic server is considered to be *active* if the priority at which the system is currently executing is equal to or greater than its priority, otherwise it is *idle*. Figure 2.3 illustrates an example schedule with a medium priority sporadic server and two periodic tasks.

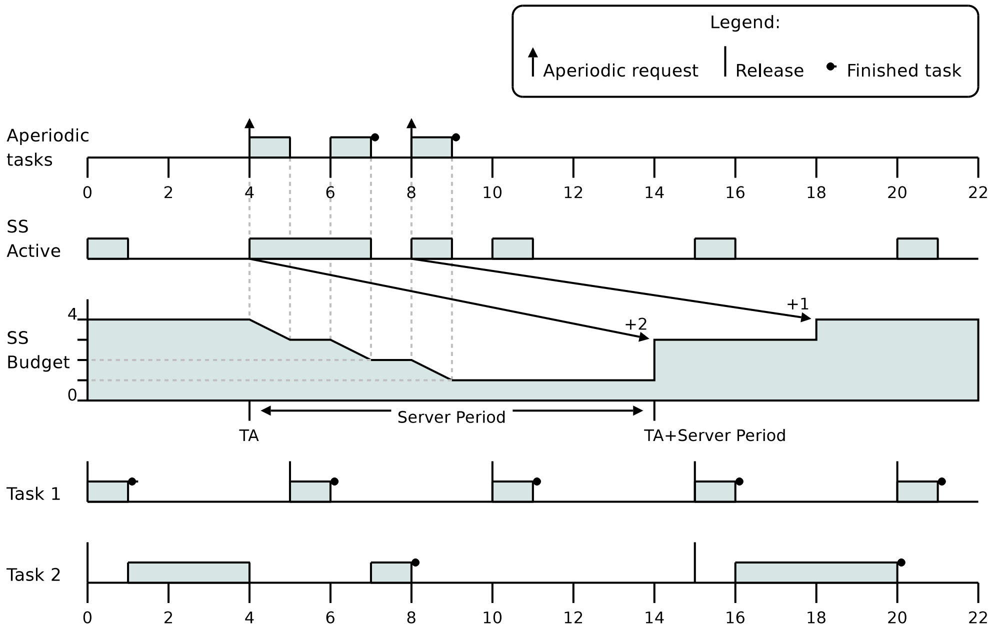


Figure 2.3: A medium priority sporadic server for handling aperiodic tasks

According to Sprunt et al. [34], from a scheduling point of view, a sporadic server can be treated as a standard periodic task with the same period and execution time as the sporadic server. They proved that a periodic task set that is schedulable with a task, τ_i, is also schedulable if τ_i is replaced by a sporadic server with the same period and execution time.

2.4 FPGA

A Field Programmable Gate Array (FPGA) is a semiconductor device containing cells of programmable logic known as Configurable Logic Blocks (CLB) and programmable interconnections. Each CLB has a Look-Up Table (LUT) that can be configured to give a specific type of logic function when programmed. Additionally, there are clocked flip-flops in the CLBs, allowing an optionally synchronous operation and a basic memory element. A typical FPGA has thousands of CLBs and often have special features such as dedicated hardware multipliers and block RAM. Modern FPGAs contain enough logic blocks to implement a number of 32-bit processors on a single device [40]. Most FPGAs are volatile and lose their configuration upon power loss. Thus the use of non-volatile memory, i.e. flash, for holding the configuration bits is necessary. Often a Complex Programmable Logic Device (CPLD)[1] is used to read the non-volatile memory and program the FPGA when powering up. To define the behaviour of the FPGA the user provides schematics or Hardware Description Language (HDL) source code. The most common HDLs are VHDL and Verilog.

2.4.1 IP Cores

To simplify the design of complex systems in FPGAs, there exist libraries of predefined complex functions and circuits that have been tested and optimised to ease the design process. These predefined circuits are commonly called Intellectual Property cores, or IP cores, and are available from FPGA vendors and third-party IP suppliers. These cores are rarely free, and typically released under proprietary licenses. Other cores are available from developer communities such as OpenCores [24], which are typically free and released under GNU General Public License (GPL) or similar license.

Soft-core processor

A soft-core processor is a processor IP core that can be wholly implemented using logic synthesis. It can be implemented in different semiconductor devices containing programmable logic, e.g. FPGAs and CPLDs. Because of this implementation, the processor will not operate at the speeds or have the performance of a hard-core solution. In many embedded applications, the high performance achieved by the hard-core processor is not required, and performance can be traded for expanded functionality and flexibility. With a soft-core processor, its functionality can be tweaked and peripheral devices can be added and removed very easily.

[1] CPLDs are also programmable logic devices, similar to FPGAs, but are non-volatile and typically has less logic resources than FPGAs

2.4.2 Design Flow Overview

The general design flow of FPGAs consists of the system design entry where the user supplies HDL files or schematics providing a description of the desired hardware functionality. Then functional simulation may be performed until the result is satisfactory. The next step is to synthesise the design, that is, the synthesis tool will analyse the HDL files and will extract RAM, counters, multiplexers and arithmetic blocks out of the code. When synthesis is done, it is time for place and route. The place and route tool generates a netlist and a timing file. With the timing file it is possible to perform a timing simulation again, in the same simulation environment as before. This timing simulation will be more accurate than the one from synthesis, since it involves timing for routing as well. The last step is to generate a bit-file and download it into the device.

2.5 VHDL

VHDL is the VHSIC Hardware Description Language. VHSIC is an abbreviation for Very High Speed Integrated Circuit. VHDL is used for describing the structure and behaviour of digital electronic hardware designs, such as ASICs and FPGAs as well as conventional digital circuits.

VHDL was originally developed in order to document the behaviour of digital systems, that is to say, VHDL was developed as an alternative to huge, complex manuals which were subject to implementation-specific details. Later, logic simulators were developed that could read the VHDL files. The next step was the development of logic synthesis tools that read the VHDL, and output a definition of the physical implementation of the circuit. Modern synthesis tools can extract RAM, counter, and arithmetic blocks out of the code, and implement them according to what the user specifies. Thus, the same VHDL code could be synthesised differently for lowest cost, highest power efficiency, highest speed, or other requirements.

VHDL is strongly-typed, is case insensitive and has many similarities with the Ada programming language.

In VHDL, a design consists at a minimum of an *entity* which describes the interface and an *architecture* which contains the actual implementation. In addition, most designs import library modules. Some designs also contain multiple architectures and configurations. The following example is a counter with parameterised register width, asynchronous reset and parallel load.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity counter is
  generic (
```

```
    width : integer := 4
  );
  port (
    clk,rst,load : in  std_ulogic;
    data         : in  unsigned(width-1 downto 0);
    count        : out unsigned(width-1 downto 0)
  );
end counter;

architecture behaviour of counter is
begin
  process(rst,clk)
    variable q : unsigned(width-1 downto 0);
  begin
    if rst = '1' then
      q := (others => '0');
    elsif rising_edge(clk) then
      if load = '1' then
        q := data;
      else
        q := q + 1;
      end if;
    end if;
    count <= q;
  end process;
end behaviour;
```

2.6 GHDL

GHDL is a complete free software simulator for VHDL using GCC technology. It works by compiling VHDL files into a binary which simulates, or executes, the design. GHDL is strictly a simulator; it does not do synthesis and it cannot translate a design into a netlist.

Below is an example of how to simulate a simple design with GHDL. The Device Under Test (DUT) is in the file `design.vhd` and the testbench file is `design_tb.vhd` in which the top entity is called `design_tb`. A simple testbench is built by instantiating the DUT, generating a sequence of input patterns and comparing the output to what output the system should produce. The synopsys library specified on the command line enables the use of the standard textio package in the testbench, needed for printing diagnostics to the standard ouput, stdout. The first step is to *analyse* the design, the next step is to *elaborate* the design. Note how the name of the entity is used instead of the filename here. The last step is to execute the compiled binary file.

```
$ ghdl -a --ieee=synopsys design.vhd design_tb.vhd
$ ghdl -e --ieee=synopsys design_tb
$ ./design_tb
```

GHDL can also create VCD or GHW files, which may be visually inspected with a waveform viewer such as GTKWave. VCD format files are defined in the Verilog HDL standard which limits its use with VHDL. The GHW format is specially designed for the use with GHDL, enabling inspection of all VHDL datatypes.

```
$ ./design_tb --vcd=dump.vcd
$ ./design_tb --wave=dump.ghw
```

2.7 GTKWave

GTKWave is a fully featured waveform viewer which reads many dumpfile formats, i.e. GHW and VCD, and allows their viewing. GTKWave is developed for Linux, with ports for various other operating systems. Figure 2.4 shows a screenshot of GTKWave, displaying a dumpfile. GTKWave support save files, enabling window and signal display settings to be restored – as shown below.

```
$ gtkwave dump.ghw gtkwave.sav
```

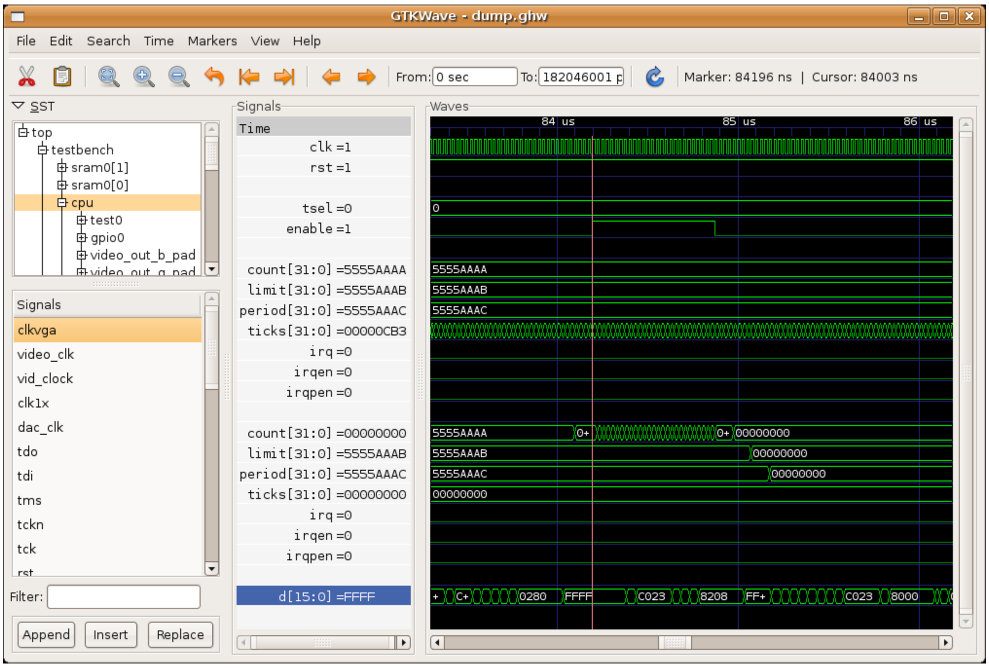


Figure 2.4: A screenshot of GTKWave

Chapter 3

System development

This chapter presents the theory of operation of the TMU system and the whole development process; from the choice of host and target platforms to the HDL model of the TMU, hardware simulations and the modifications to the eCos RTOS for TMU support.

3.1 TMU theory of operation

The TMU is a hardware unit that is integrated in, or in close proximity to, the CPU. Its purpose is to measure and control the execution time of both tasks and Interrupt Service Routines (ISR). For each interrupt level the unit has an execution timer called *IRQ Timer* and for the ordinary execution state the unit has a single execution timer called the *Task Timer*, as illustrated in Figure 3.1.

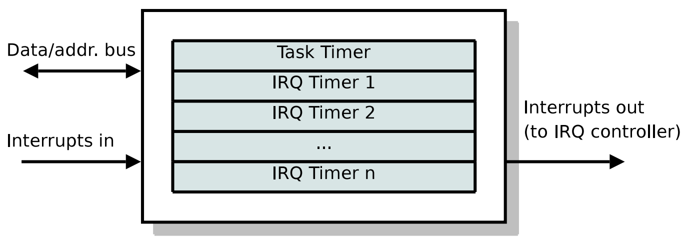


Figure 3.1: A block diagram of the TMU

The Task Timer, shown in Figure 3.2, has three processor accessible registers; *count*, *limit* and *control*. The count register is incremented every clock cycle and compared with the limit register. When they equal, an interrupt is generated, signalling that the current task has exceeded its given budget. The control register manages the interrupt generation capabilities of the timer.

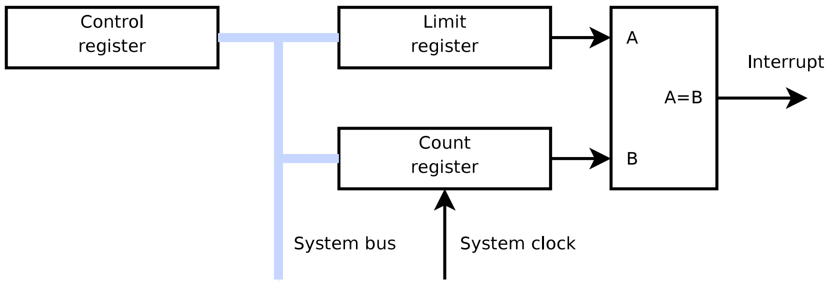


Figure 3.2: A block diagram of the TMU Task Timer

Each IRQ Timer has an extra register, the *period* register, as seen in Figure 3.3. The period register stores the interval at which the count register shall be cleared. Thus, the IRQ Timers support the DS algorithm presented in section 2.3.4 by replenishing the execution budget at regular intervals. Actually, the IRQ Timer requires another register too, the *ticks* register. Ticks always increments and is compared with the period register, but is not accessible to the processor. When ticks and period equals, both ticks and count will be cleared. Interrupt signals previously connected directly to the IRQ controller are routed via the IRQ Timers. While there is execution budget left, all IRQ signals are passed through. But when the budget is spent, no more signals will pass until the next replenish period.

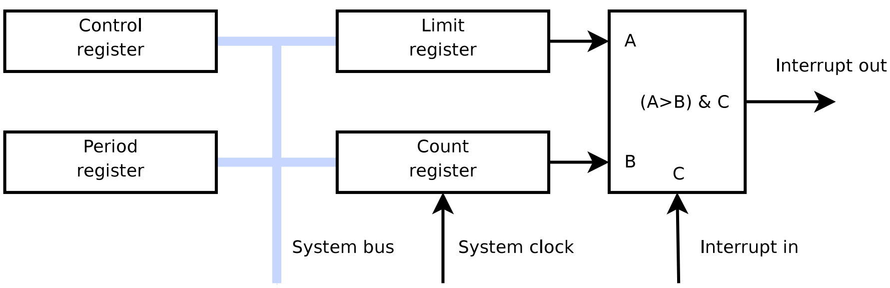


Figure 3.3: A TMU IRQ Timer block diagram

The TMU also has a *configuration* register and a *timer select* register. The configuration register has a global enable bit for the TMU and some read only information about the current implementation, helping software determine its underlying platform. The timer select register sets the active TMU timer and is controlled by software. Only one timer is running at any given time.

Count, limit and period registers are preferably 64-bit wide, but may be less. The minimum register width is the one with a register overflow period of more than the longest relative deadline for any task in the system.

3.1.1 OS support for the TMU

The TMU must be incorporated into the Operating System (OS) for proper handling. The Task Timer registers shall become a part of the task, or thread, context. That is the count, limit and control registers of the Task Timer. That way, all threads share the same hardware timer, but are under the impression of having their own timer. The following operations must be performed by the OS:

- When a thread is created, default values for the TMU registers must be provided.
- During a context switch, the OS must save the current threads TMU registers to the threads stack, and load the next threads TMU registers from its stack and into the TMU hardware.
- When an interrupt occurs, the OS must change the active timer of the TMU to that of the corresponding interrupt by writing to the Timer Select register. And when the Interrupt Service Routine (ISR) has completed, the Timer Select register must be written again to reselect the Task Timer.

Further, a suitable Application Programming Interface (API) to the TMU must be provided by the system for application program use.

3.2 TMU implementation alternatives

The objective is to find the implementation giving the lowest overhead, but at the same time is flexible enough to allow the TMU being implemented with various processor architectures.

3.2.1 Peripheral device

For a System-on-Chip (SoC), functions such as timers, General Purpose Input/Output (GPIO), system timers and UARTs are commonly known as peripheral devices. These devices are on the same chip as the CPU and are accessed through a system bus. Being a peripheral device, the TMU benefits of the standardised system bus and processor architecture independence. On the negative side, accessing a device through a bus means slower access times.

3.2.2 Coprocessor

A coprocessor is a processor that operates under the supervision of the primary processor, the CPU. Typically, it has no means of accessing memory without the use of the CPU and it supports a limited instruction set. The purpose of a coprocessor is to alleviate the CPU load by handling some computational intensive tasks. Typical application areas are floating point arithmetic, graphics and signal processing. As for the TMU, being implemented as a coprocessor means tight integration with the CPU. This shortens the register access times and gives high flexibility as the coprocessor has its own instruction set. On the other hand, many systems only allow for a single coprocessor, and thus using the TMU comes at the expense of other coprocessor functions. Further, the TMU has no benefit of having its own instruction set, and it does not do anything computational intensive; it only counts and compares its registers. Additionally, the coprocessor interface is architecture specific and would make it harder to implement the TMU on other processor systems.

3.2.3 Register File

The register file is a set of registers that the CPU uses for temporarily storing data. These registers are addressed by the operand fields of processor instructions which are defined by the Instruction Set Architecture (ISA) for that particular architecture. Since the registers are addressed in the instruction itself, it is the fastest memory access times of the computer memory hierarchy. Because the register file is closely related to the ISA, adding registers to the register file means expanding the ISA. In turn, the assembler for the given architecture has to be modified to support the new instruction format, making it an intrusive approach. Thus, implementing the TMU inside the register file means that the ISA and assembler have to be modified, making it a highly architecture specific and complicated implementation.

3.2.4 Chosen implementation

Based on the above given aspects of the various TMU implementations, the decision fell on peripheral device implementation. As a peripheral device, the TMU is very easy to implement compared to the coprocessor and register file alternatives. It also means that the TMU will be easy to adapt to new systems and architectures. There is no need to modify the compiler/assembler or in any way modify the ISA. The only negative side is that accessing the device through a bus means slower access times.

3.3 Host development platform

The host development platform is a standard desktop computer with a parallel port for FPGA programming and a serial port or ethernet port for communicating and debugging the target. Linux is opted as the host platform OS because it is a convenient platform for development with GNU tools. Using GNU tools on MS Windows often involve programs like Cygwin or MinGW, emulating a UNIX environment. The Ubuntu 7.04 Linux distribution was used as the host OS.

To be utilised as a development platform, additional software must be installed on the host. The following sections describe the installation procedure of the basic tools used throughout this project.

3.3.1 Installation of Xilinx software and cable drivers

A user account was created at the Xilinx home page, `http://www.xilinx.com`, before the download of the Xilix WebPACK could take place. When the download had finished, the installer was started like this:

```
$ cd /path/to/dl/
$ unzip <name>.zip
$ cd <name>
$ sudo ./setup
```

When using Xilinx JTAG software like Impact, Chipscope and XMD on Linux, the proprietary kernel module windrvr from Jungo is needed to access the parallel- or USB-cable. As this module does not work with current linux kernel versions (versions above 2.6.18) a library was developed by users on the Internet, which emulates the module in userspace and allows the tools to access the JTAG cable without the need for a proprietary kernel module. Thus, “Install cable drivers” in the install procedure was deselected and the third-party cable driver was installed instead. The cable driver was downloaded from `http://rmdir.de/~michael/xilinx/`. and is called *libusb-driver*.

The library was built like this:

```
$ cd /path/to/dl/
$ tar xzf usb-driver-HEAD.tar.gz
$ cd usb-driver
$ make
```

The built library is named libusb-driver.so and exists in the working directory. Although the library is called lib*usb*, it also handles the parallel programming cables.

The Xilinx software depends on several environment variables defined in the file `settings.sh` in the installation directory. `settings.sh` must be sourced in the shell before starting any Xilinx utility. The example below starts Xilinx ISE Project Navigator.

```
$ source /path/to/Xilinx92i/settings.sh
$ ise
```

When using the FPGA download utility Impact, the path to libusb-driver.so must be set:

```
$ su
$ source /path/to/Xilinx92i/settings.sh
$ export LD_PRELOAD=/path/to/libusb-driver.so
```

```
$ impact
```

Note that using the cable drivers requires root privileges by default. Ordinary user access to the parallell port device used for FPGA programming was enabled by adding the user to the "lp" group:

```
$ sudo usermod -a -G lp username
```

When trying to program the FPGA with iMPACT, the error "Programming failed" and "DONE did not go high" appeared. This was fixed by setting the option "Use HIGHZ" instead of "BYPASS" in Edit → Preferences → iMPACT → Configuration Preferences.

Now, most of the Xilinx IDE was running correctly, but when trying to run i.e. Floorplanner, the following error appeared:

```
$ floorplanner
Wind/U X-toolkit Error: wuDisplay: Can't open display
```

This behaviour was corrected by setting the environment variable DISPLAY to `:0`.

To sum up, the following entries were added to the shell resource file[1] after installation of the Xilinx WebPACK, so that all utilities would work.

```
source /path/to/Xilinx92i/settings.sh
export LD_PRELOAD=/path/to/libusb-driver.so
export DISPLAY=:0
```

3.3.2 GHDL

The VHDL simulator GHDL can be installed through the package management system of Ubuntu:

```
$ sudo apt-get install ghdl
```

3.3.3 GTKWave

The waveform viewer GTKWave can be installed through the package management system of Ubuntu:

```
$ sudo apt-get install gtkwave
```

3.4 FPGA development board

There are two market leading FPGA manufacturers, Xilinx and Altera. Both provide free tools for MS Windows platforms. Xilinx also provide free development tools for Linux, while Linux tools from Altera must be purchased.

[1] The shell resource file is named, on Ubuntu systems at least, `.bashrc`, and exists in the home directory of the user

Because Linux has been chosen as the host platform OS, Xilinx FPGA development boards were opted. In the next sections, the following Xilinx boards will be presented:

1. Spartan-3 Starter Board
   This board was readily available at project start and was subject to the first implementation tests, presented in the next section.

2. Spartan-3A DSP 1800A Development Board
   This board was ordered when it became clear that the first board had insufficient resources. As the supplier could not deliver the board in time, no actual implementation was performed on this platform. But while the order was being processed, an implementation was prepared and simulations were performed.

3. Virtex-4 ML401 Evaluation Platform
   This board was obtained when the supplier of the Spartan-3A DSP 1800A Development Board could not deliver in time. The Virtex-4 ML401 Evaluation Platform is the board in which the final design is implemented.

3.4.1 Spartan-3 Starter Board

The Spartan-3 Starter Board, Figure 3.4, was readily available at project start and was subject to the first implementation tests. As will be discussed in this text, the board was an unsuitable platform for testing the TMU and another board had to be obtained.

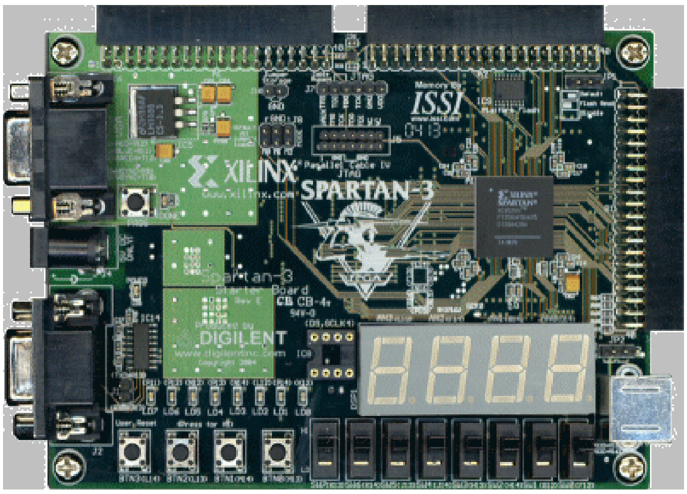


Figure 3.4: Xilinx Spartan-3 Starter Board, source `http://www.xilinx.com`

The Xilinx Spartan-3 Starter Board features:

- Xilinx Spartan-3 XC3200 FPGA with 1 920 slices[2], 30 Kib distributed RAM and 216 Kib of block RAM
- On-board 2 Mib Platform Flash (XCF02S)
- 8 slide switches, 4 pushbuttons, 9 LEDs, and 4-digit seven-segment display
- Serial port, VGA port, and PS/2 mouse/keyboard port
- 1 MiB[3] on-board 10ns SRAM (256 Kib[4] x 32)
- Three 40-pin expansion connectors

Trying out an 8-bit soft-core processor

The idea behind the first implementation attempt was to keep the system as simple as possible, finding a good match of an 8-bit soft-core MCU running a suitable RTOS in which the TMU could be embedded. Table 3.1 shows a selection of 8-bit MCU IP cores found through OpenCores [24] and the Internet. Listed compilers and RTOSs are also free. Note that the quality of cores found on the Internet varies greatly. Some cores are totally equivalent to a particular device, implementing the same instruction set and set of peripherals, while others are only partially compatible, either lacking some peripheral devices, not implementing the full instruction set or have not been tested in hardware. The cores listed below provide an acceptable quality standard, suitable for implementation in an FPGA.

Table 3.1: A selection of free 8-bit MCU IP cores

Arch.	**Name**	**Compatible**	**HDL**	**Compiler**	**RTOS**
AVR	avr_core	ATmega103	VHDL	GCC	FreeRTOS
8051	8051	-	Verilog	SDCC	FreeRTOS
PIC16	CQPIC	PIC16F84	VHDL	SDCC[1]	-
PIC18	ae18	PIC18	Verilog	SDCC[1]	FreeRTOS

Of these cores, the AVR Core was chosen because of its support by the well known GCC compiler and that it has been thoroughly tested. The AVR Core is written by Ruslan Lepetenok in synthesisable VHDL and is an Atmel ATmega103 equivalent device, having the same instruction set and instruction timing. The core features 32 8-bit general purpose registers, 23 interrupt vectors and support for up to 128 KiB of program and up to 64 KiB of data

[2]On Spartan-3 architectures: 1 CLB = 4 slices = 8 LUTs
[3]mebibyte = 2^{20} bytes = 1,024 kibibytes (MiB)
[4]kibibit = 2^{10} bits = 1,024 bits (Kib)
[1]SDCC support for PIC16 and PIC18 MCUs is a work in progress as of March 2008

memory. It also has a programmable UART, two 8-bit timers, eight external interrupt sources and two 8-bit parallell ports.

The AVR Core is supported by FreeRTOS – a minimal RTOS featuring:

- Multitasking capabilities
- Preemptive, cooperative and hybrid kernel configuration options
- Support for co-routines
- Intertask communication and synchronisation
- Priority inheritance
- Scalable

FreeRTOS is licensed under a modified GPL and can be used in commercial applications under this license, without having to open source user application code.

A new project in the Xilinx ISE Project Navigator was created and all AVR Core VHDL files were copied into the project. The program memory of the AVR Core is hardcoded in the file `PROM.vhd`, and must be specified before the design can be synthesised. The AVR Core distribution comes with an utility called, `hex2vhd`, reading Intel Hex format files and generating a PROM VHDL entity in a file named `PROM.vhd` containing the program instructions and static data. An example of compiling a program for the AVR Core and generating `PROM.vhd` is shown below.

```
$ avr-gcc -mcall-prologues -mmcu=atmega103 -Os -g prog.c -o prog.elf
$ avr-objcopy -j .text -O ihex prog.elf prog.hex
$ hex2vhd prog.hex
```

`avr-gcc` and `avr-objcopy` is a part of the AVR development tools that was installed through the package management system of Ubuntu:

```
$ sudo apt-get install avr-libc binutils-avr \
> gcc-avr gdb-avr simulavr
```

Having specified the location of the generated `PROM.vhd` in Project Navigator, the design can be implemented. Test programs were written, testing the parallel ports by flashing LEDs and writing to the UART. These test programs can be found in Appendix A. When building the FreeRTOS demo design, updating `PROM.vhd` and implementing the design, it became clear that the generated bit-file was too big for the XC3S200 FPGA. Thus, an implementation of the TMU with an RTOS for proper testing would be impossible. Even if the design could fit into the FPGA, the AVR Core has no debug unit, making single stepping through programs impossible. Also, the process of updating the PROM memory with a new program takes a significant amount of time, as the FPGA design cycle has to be performed each time.

Instantly updating block RAM contents in a .bit file

In designs like the AVR Core, the contents of the program memory is hard-coded into a HDL file. When loading a new program, the whole FPGA design flow has to be performed. This process is very time consuming, and when only the memory contents of a design is changing, it is also unnecessary. Xilinx provide a tool called Data2Mem, which decreases development time by orders of magnitude by directly replacing block RAM data in bit files without the intervention of any other Xilinx implementation tools.

Data2Mem needs a .bmm (Block Memory Map) file to direct the translation of data into the proper initialisation form. A .bmm file is a text file that has syntactic descriptions of how individual Block RAMs constitute a contiguous logical data space. However, as Data2Mem is a part of Xilinx automated design flow for embedded processors, proper syntax of the .bmm file is poorly documented for stand-alone use. As a result, usage of this tool failed.

3.4.2 Spartan-3A DSP 1800A Development Board

As the XC3S200 FPGA on the Spartan-3 Starter Board had too little resources, a new development board was ordered. Although the supplier could not deliver the board in time, and the Virtex-4 ML401 Evaluation Platform was used for the implementation, the board is still presented here. The time between ordering the board and obtaining the Virtex-4 ML401 Evaluation Platform was spent simulating and preparing a LEON3 design for actual implementation on the Spartan-3A DSP 1800A Development Board. Because there was no demo design in GRLIB for this board, a lot of time was spent on making one. The Spartan-3A DSP 1800A development board, Figure 3.5, features:

- Spartan-3A XC3SD1800A DSP FPGA with 16 640 slices, 260 Kib distributed RAM and 1 512 Kib of block RAM
- 128 MiB DDR2 SDRAM (32 Mib x 32), 16 Mib x 8 parallel flash, 64 Mib SPI serial flash
- 10/100/1000 Ethernet PHY
- RS232 serial port
- Video RGB Port
- 8 user LEDs, 8-position user DIP switch, 4 user push button switches, reset push button switch
- JTAG programming/configuration port

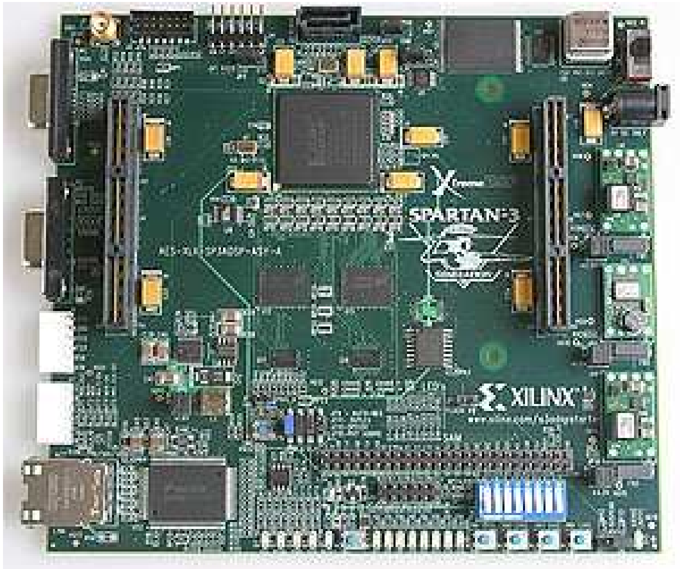


Figure 3.5: Xilinx partan-3A DSP 1800A Development Board, source `http://www.xilinx.com`

3.4.3 Virtex-4 ML401 Evaluation Platform

As the XC3S200 FPGA on the Spartan-3 Starter Board had too little logic resources, another development board was obtained. The Virtex-4 ML401 Evaluation Platform, Figure 3.6, featuring:

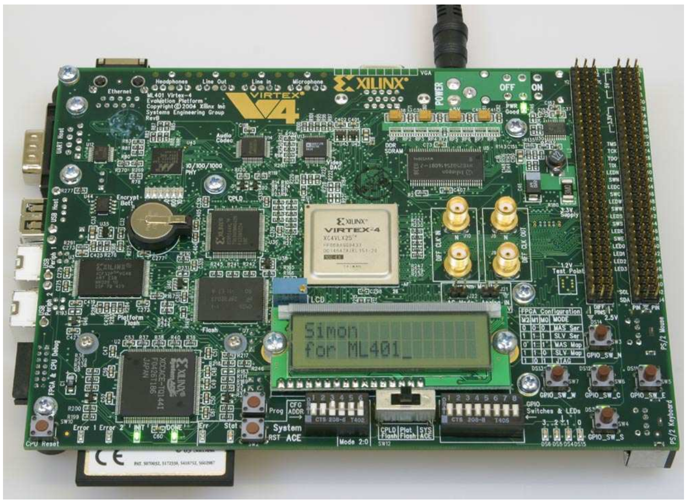


Figure 3.6: Xilinx Virtex-4 ML401 Evaluation Platform, source `http://www.xilinx.com`

- Xilinx Virtex-4 XC4VLX25 FPGA with 10 752 slices, 168 Kib distributed RAM and 1 296 Kib of block RAM

- 64 MiB DDR SDRAM, 8 Mib ZBT SRAM, 64 Mib Flash, 4 kb I^2C EEPROM
- 16-character x 2-line LCD
- General purpose DIP switches LEDs, and push buttons
- Stereo AC97 audio codec with line-in, line-out, 50-mW headphone, and microphone-in (mono) jacks
- RS-232 serial port
- USB Ports (2 Peripheral/1 Host)
- VGA port with 24-bit video DAC
- PS/2 mouse and keyboard connectors
- On-board 32 Mib Platform Flash (XCF32P)
- System ACE CompactFlash controller with Type I/II CompactFlash connector for FPGA configuration from CF-cards

3.5 Soft-core processor alternatives

The 8-bit AVR Core MCU test proved to be a difficult development platform, as discussed in Section 3.4.1. In the second design, larger processors with debug facilities and rapid program downloading functionalities was opted. Table 3.2 lists a selection of free 32-bit soft-core processors suitable for implementation. As before, listed processors and RTOSs are free and all architectures use the GCC. All processors are supported by Linux and uClinux, but as they have poor real-time capabilities, only RTOSs are listed.

Table 3.2: A selection of free 32-bit soft-core processors

Name	**ISA**	**HDL**	**RTOS**
LEON	SPARC V8	VHDL	RTEMS, eCos
OpenRISC	OpenRISC	Verilog	RTEMS
AEMB	MicroBlaze	Verilog	FreeRTOS

3.5.1 LEON

The LEON processor is a 32-bit synthesisable processor core written in VHDL based on the SPARC V8 architecture. The core is highly configurable, and particularly suitable for System-on-Chip (SoC) designs. The LEON3 core is a newest implementation of the LEON family, with a 7 stage integer pipeline and multi-processor support. It is distributed as part of the GRLIB IP library under the GNU GPL.

3.5.2 OpenRISC

OpenRISC, or more specifically OpenRISC 1000, is a processor family based on an open source 32/64-bit RISC ISA design by the OpenCores team [24] released under the GNU Lesser General Public License (LGPL). The only finished implementation is the 32-bit OpenRISC 1200 which is a 32-bit scalar RISC with Harvard microarchitecture, 5 stage integer pipeline and virtual memory support (MMU), written in Verilog HDL. Being Wishbone compliant[5], OpenRISC will easily integrate with many other open source cores that use the same open bus specification.

3.5.3 AEMB

The AEMB is an open source 32-bit microprocessor core written in Verilog HDL. It is instruction compatible with the Xilinx Microblaze and comes with several architectural enhancements. It will easily run C/C++ code and has been independently proven in hardware. AEMB is Wishbone compliant and will easily integrate with many other open source cores that use the same open bus specification. The AEMB core is fully parameterisable, without using any unwieldy define files. This includes the address space and optional functional units, such as the barrel-shifter and multiplier. This allows customisation in the design without having to make any changes to the original AEMB files.

3.5.4 Chosen soft-core processor

Of the given processors, LEON3 was chosen as it seemed to be the easiest processor to start out with, having a large user base, lots of peripherals, included testbench and demo designs for many FPGA boards. Also, Kjetil Svarstad, a professor dealing with system level design and analysis of digital systems at NTNU, recommended it.

3.6 RTOS alternatives

Suitable RTOSs for the LEON3 soft-core processor are RTEMS and eCos, as listed in Table 3.2. Both are presented here before the choice is made.

[5]Wishbone is an open bus standard for SoC designs

3.6.1 RTEMS

RTEMS (Real-Time Executive for Multiprocessor Systems) is a free open source RTOS designed for embedded systems. RTEMS is designed to support various open API standards including POSIX and uITRON. The basic features of the RTEMS kernel include:

- Multitasking capabilities
- Event-driven, priority-based preemptive scheduling
- Optional rate-monotonic scheduling
- Intertask communication and synchronisation
- Priority inheritance
- Dynamic memory allocation
- High level of user configurability

Additionally, RTEMS provide networking, filesystems and debugging support.

RTEMS is distributed under a modified GPL licence, allowing linking RTEMS objects with other files without requiring the full executable to be covered by the GPL. This license is based on the GNAT Modified General Public License with the language modified to not be specific to the Ada programming language.

3.6.2 eCos

Embedded Configurable Operating System, or eCos, is an open source Real-Time Operating System (RTOS) intended for embedded and real-time systems.

It can be customised to precise application requirements, with hundreds of options, delivering the best possible run-time performance and minimised hardware needs. It is written in the C and C++ programming languages, and its standard application interface is C. eCos also has compatibility layers and APIs for POSIX and uITRON. eCos was designed for devices with memory size in the tens to hundreds of kilobytes, and it can be used on hardware with too little RAM to support embedded Linux, which currently needs a minimum of about 2 MiB of RAM, not including application and service needs. The basic features of the eCos kernel include:

- Multitasking capabilities
- Event-driven, priority-based preemptive scheduling – choose between the bitmap scheduler and the multi-level queue (MLQ) scheduler
- Intertask communication and synchronisation

- Priority inheritance and priority ceiling protocol
- Integration with the system's support for interrupts and exceptions
- High level of user configurability

eCos also provide device drivers, filesystem support and network capabilities. eCos is distributed under the GPL license with an exception which permits proprietary application code to be linked with eCos without itself being forced to be released under the GPL. It is also royalty and buyout free.

3.6.3 Chosen RTOS

Both RTEMS and eCos seemed like equally good alternatives, but only one could be chosen. eCos was chosen as the system RTOS because it is highly configurable and simple, both important aspects when implementing the TMU.

3.7 LEON3 development tools

3.7.1 GRLIB IP Library

GRLIB is a collection of reusable IP cores distributed under GNU GPL. It is based on the AMBA AHB and APB on-chip buses, which is used as the standard interconnect interface. The implementation of the AHB/APB buses is compliant with the AMBA-2.0 specification [1], with additional "sideband" signals for automatic address decoding, interrupt steering and device identification. The AHB and APB signals are grouped according to functionality into VHDL records, declared in the GRLIB VHDL library. All GRLIB cores use the same data structures to declare the AMBA interfaces, and can then easily be connected together.

GRLIB is downloaded from the home page of Gaisler Research, `http://www.gaisler.com`. It can be installed anywhere, just extract the downloaded archive:

```
$ tar xzvf grlib-gpl-1.0.17-b2710.tar.gz
```

3.7.2 GRMON debug monitor

GRMON is a general debug monitor for the LEON processor, and for SoC designs based on the GRLIB IP library. GRMON features:

- Read/write access to all system registers and memory
- Built-in disassembler and trace buffer management
- Downloading and execution of LEON applications

- Breakpoint and watchpoint management
- Remote connection to GNU debugger (gdb)
- Support for USB, JTAG, RS232, PCI, Ethernet and SpaceWire debug links

GRMON can be run in stand-alone mode, or connected through a network socket to the GNU GDB debugger. In stand-alone mode, a variety of debugging commands are available to allow manipulation of memory contents and registers, breakpoint/watchpoint insertion and performance measurement. Connected to GDB, GRMON acts as a remote target and supports all GDB debug requests. The communication between GDB and GRMON is performed using the GDB extended-remote protocol.

GRMON is released under both a professional and an evaluation license. The evaluation version may be used during a limited period without purchasing a license. GRMON can be downloaded from `http://www.gaisler.com`. Untar the download and add the the path to the GRMON executable in the search path:

```
$ tar xzvf grmon-eval-1.1.27b.tar.gz
$ echo "export PATH=/path/to/grmon-eval/linux:$PATH" >> $HOME/.bashrc
```

3.7.3 TSIM LEON simulator

TSIM is an instruction-level simulator capable of emulating LEON-based computer systems. TSIM features:

- Accurate and cycle-true emulation
- High performance: up to 30 MIPS on high-end PC (Xeon@3.2GHz)
- Standalone operation and remote connection to GNU debugger (gdb)
- Instruction trace buffer
- Stack backtrace with symbolic information
- Unlimited number of breakpoints and watchpoints

TSIM can be downloaded from `http://www.gaisler.com` and installed like this:

```
$ tar xzvf tsim-eval-2.0.10.tar.gz
$ echo "export PATH=/path/to/tsim-eval/tsim/linux:$PATH" >> $HOME/.bashrc
```

TSIM can be run in stand-alone mode, or connected through a network socket to the GNU GDB debugger. It acts very similarly to the GRMON debugger and has a variety of debugging commands allowing manipulation of memory contents and registers, breakpoint/watchpoint insertion and performance measurement.

A stand-alone TSIM session:

```
$ tsim-leon3
tsim> load hello
tsim> go
resuming at 0x40001114
Hello, eCos world!
```

A session with TSIM and gdb, using two terminals:
Terminal 1:

```
$ tsim-leon3 -gdb
gdb interface: using port 1234
connected
Hello, eCos world!
```

Terminal 2:

```
$ sparc-elf-gdb
(gdb) file hello
(gdb) target extended-remote localhost:1234
(gdb) load
(gdb) c
Continuing.

Program exited normally.
(gdb)
```

3.7.4 BCC

The Leon Bare-C Cross Compilation System (BCC), a free C/C++ cross-compiler system for LEON2 and LEON3 processors based on GCC and the Newlib embedded C-library. BCC includes a small run-time system with interrupt support and Pthreads library. BCC consists of the following packages:

- GNU C/C++ cross-compiler (3.2.3 and 3.4.4)
- GNU Binutils-2.16.1 (assembler, linker ...)
- Newlib 1.13.0 Embedded C-library
- Bare-C run-time system with interrupt support and tasking
- Pthreads library
- Boot-prom builder
- GNU debugger (gdb)
- DDD graphical front-end for gdb
- Insight graphical front-end for gdb
- Windows (Cygwin) and linux host
- Optional Eclipse-based IDE

BCC can be downloaded from the homepage of Gaisler Research, `www.gaisler.com`. The procedure for binary package installation of BCC on a Linux host is shown below. BCC is recommended to be installed in `/opt`:

```
$ su
# mv sparc-elf-3.4.4-1.0.30.tar.bz2 /opt
# cd /opt
# tar xjvf sparc-elf-3.4.4-1.0.30.tar.bz2
# exit
$ echo "export PATH=/opt/sparc-elf-3.4.4/bin:$PATH" >> $HOME/.bashrc
```

3.8 Implementing a LEON3 system

The best way to implement a LEON3 system is to start out with one of the many demo designs available in the GRLIB distribution. In the following text, the demo design for the Virtex-4 ML401 Evaluation Platform is used. If the design needs modifications, GRLIB provides a graphical configuration tool which is started like this:

```
$ cd <grlib>/designs/leon3-avnet-ml401
$ make xconfig
```

Figure 3.7 shows the GRLIB configuration tool.

The design can then be simulated or implemented in the FPGA. GRLIB supports several simulators, i.e. GHDL, ModelSim. GRLIB also support several synthesis tools, but for this project, only Xilinx tools have been used. To perform synthesis, place and route and generation of bitfiles, all that is needed is:

```
$ make ise
```

To program the FPGA or PROM, type:

```
$ make ise-prog-fpga
$ make ise-prog-prom
```

3.8.1 Running applications on target

To download and debug applications on the target board, the GRMON debug monitor is used. GRMON can be connected to the target using RS232, JTAG, ethernet or USB. Ethernet was opted for its high speed. To connect using the ethernet port, do:

```
$ grmon-eval -eth -u -gdb

 initialising ...........
 detected frequency:  41 MHz

 Component                            Vendor
 LEON3 SPARC V8 Processor             Gaisler Research
 GR Ethernet MAC                      Gaisler Research
```

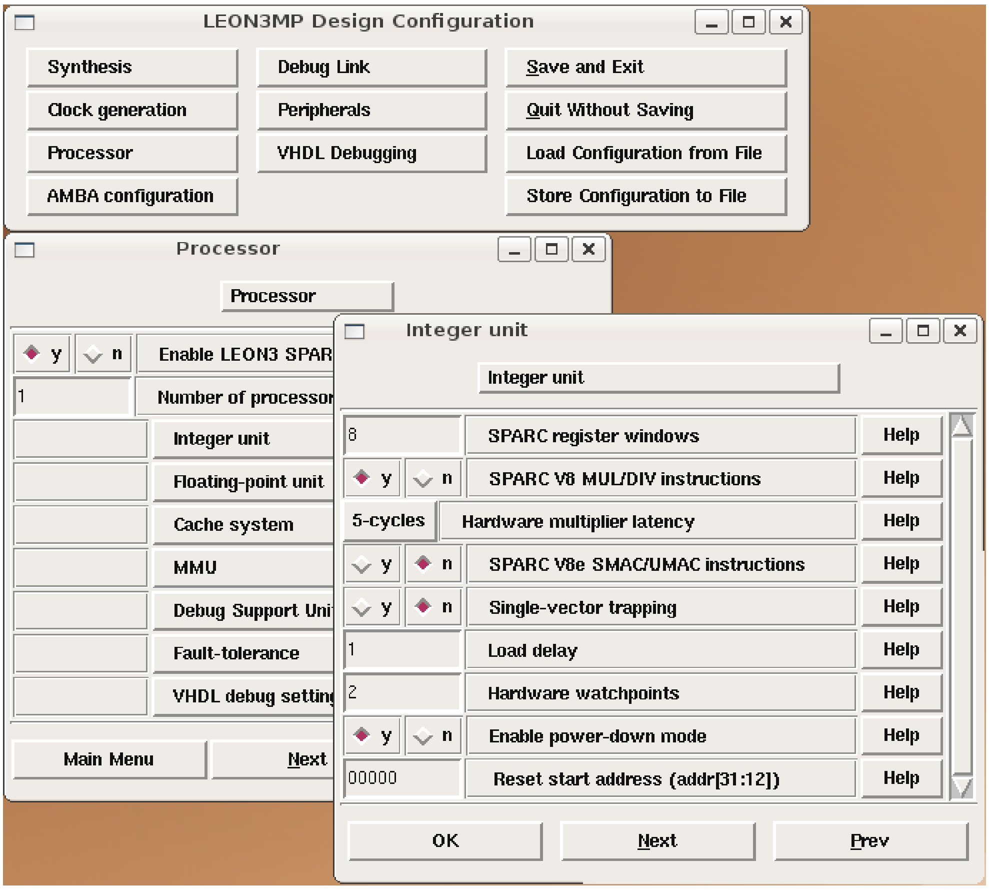


Figure 3.7: The GRLIB configuration tool

```
 DDR266 Controller                    Gaisler Research
 AHB/APB Bridge                       Gaisler Research
 LEON3 Debug Support Unit             Gaisler Research
 LEON2 Memory Controller              European Space Agency
 Generic APB UART                     Gaisler Research
 Multi-processor Interrupt Ctrl       Gaisler Research
 Modular Timer Unit                   Gaisler Research
 General purpose I/O port             Gaisler Research
 Unknown device                       Unknown vendor
 AHB status register                  Gaisler Research

 Use command 'info sys' to print a detailed report of attached cores

gdb interface: using port 2222
```

The unknown device is the TMU. It is unknown because its plug and play codes are not defined in GRMON. Open another terminal, start gdb and connect to the target:

```
$ sparc-elf-gdb
(gdb) file program-name
(gdb) target extended-remote localhost:2222
(gdb) load
(gdb) c
Continuing.
```

To reload the program:

```
(gdb) monitor reset
(gdb) load
(gdb) c
Continuing.
```

Commands that start with *monitor* is passed to the debug monitor, which in this case is GRMON. That way, all commands available in GRMON can be accessed in GDB by using the monitor prefix.

To run programs with GRMON in stand-alone mode, do:

```
$ grmon-eval -eth -u
grlib> load program-name
grlib> run
```

3.9 Configuring eCos and building applications

This section will demonstrate how to configure and build eCos as well as compiling user applications and running them on the LEON simulator TSIM.

The eCos build process involves three separate directory trees: *source*, *build*, and *install*.

- The source tree is the source code repository, which is located under the packages directory of an eCos distribution.

- The build tree is generated by the configuration tools and contains intermediate files, such as makefiles and object files. The structure of the build tree might differ between system builds. Typically, each package in the configuration has its own directory in the build tree, which is used to store that package's makefiles and object files.

- The install tree is the location of the eCos main library file and the exported header files, which are used when the application is built. The library files are located under the *lib* subdirectory, and the header files are contained under the *include* subdirectory.

For this example, the eCos port for LEON3 will be used. Either download eCos from CVS or download the latest version with applied patches from `www.gaisler.com`. Untar the download at a suitable location and export the eCos repository path through the ECOS_REPOSITORY environment variable:

```
$ tar xjvf ecos-rep-1.0.8.tar.gz
$ export ECOS_REPOSITORY=/path/to/ecos-rep-1.0.8/packages
```

To permanently have the the environment variable set, add it to the shell resource file like this:

```
$ echo "export ECOS_REPOSITORY=/path/to/ecos-rep-1.0.8/packages" >> $HOME/.bashrc
```

The kernel will be configured and built in a separate directory called the *build* tree. To start out with a standard configuration, do:

```
$ cd <build>
$ ecosconfig new sparc_leon3
```

This will create a text file called `ecos.ecc` that contains the default configuration for the LEON3 target. The configuration file may be altered by hand, or preferably by using the graphical user interface called eCos Configuration Tool:

```
$ configtool ecos.ecc
```

Figure 3.8 shows a screenshot of the eCos Configuration Tool, showing some of the many options.

The next step after the configuration phase is to fill the eCos build tree. This can be performed either within the Configuration Tool or in the shell as will be done here.

```
$ ecosconfig tree
```

The build tree is a set of makefiles, header files and source files that will be used for building the eCos install tree. Sometimes it may be necessary to make the build tree after some configuration changes. Now, to build eCos, simply type make:

```
$ make
```

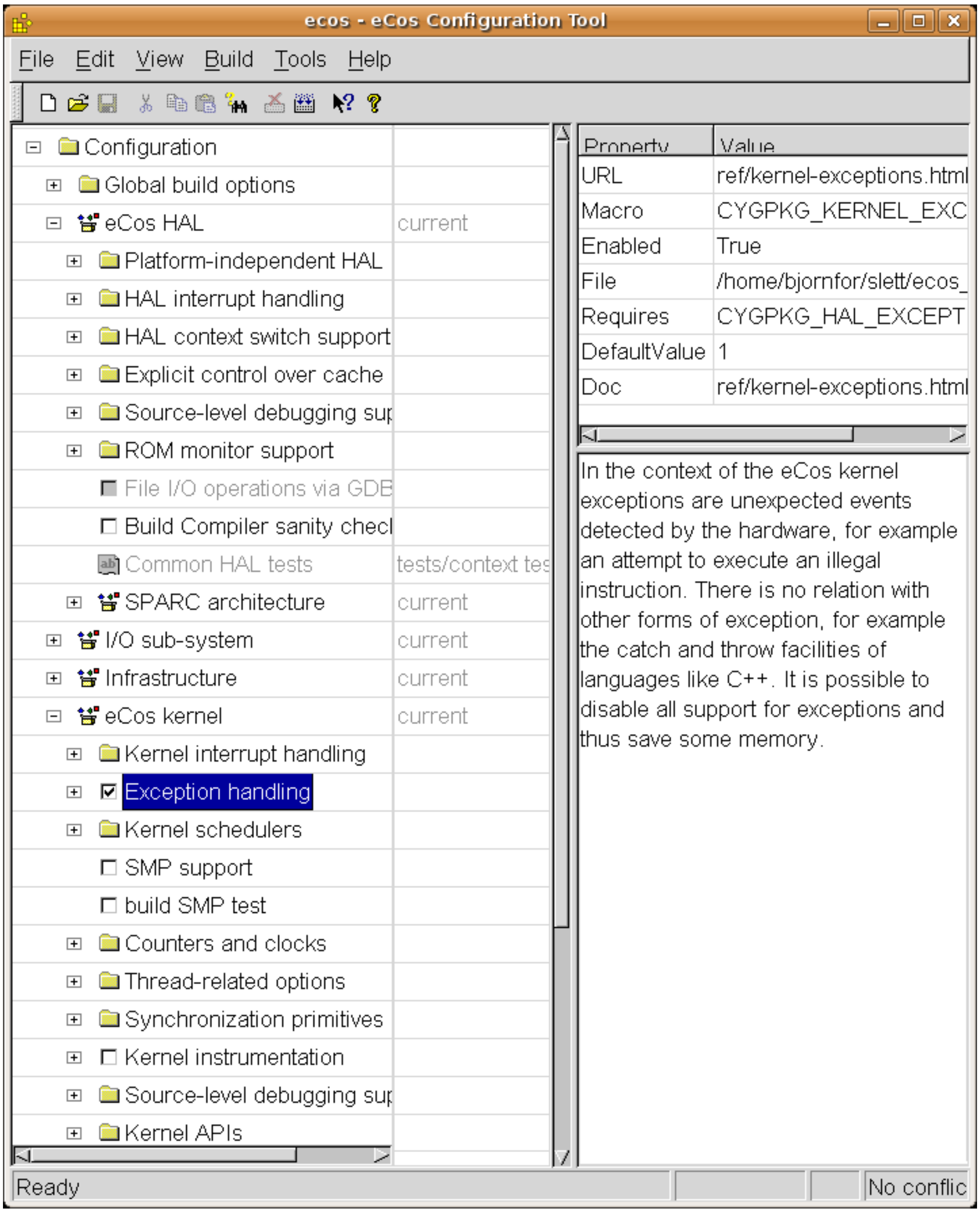


Figure 3.8: A screenshot of the eCos configuration tool

Within the build tree there is now a directory called *install*. The install tree is the location of the eCos main library file and the exported header files, which are used when the application is built. The library files are located under the lib subdirectory, and the header files are contained under the include subdirectory.

Building applications is preferably performed in yet another directory.

```
$ cd <apps>
$ sparc-elf-gcc -g -I/path/to/ecos-build/install/include \
> -L/path/to/ecos-build/install/lib -Ttarget.ld -nostdlib hello.c
```

Applications can be tested with the TSIM simulator:

```
$ tsim-leon3 a.out
tsim> go
resuming at 0x40001114
Hello, eCos world!
```

3.10 HDL design of the TMU

As the LEON3 processor is written in VHDL, it was reasonable to code the TMU in VHDL as well. The TMU is written in a structured VHDL coding style called “two-process method” proposed by Jiri Gaisler [11]. The method is applicable to any synchronous single-clock design, which represents the majority of all designs. The coding style has three simple rules:

- Only use two processes per entity
- Use record types in all port and signal declarations
- Use high-level sequential statements to code the algorithm

In order to improve readability and provide a uniform way to encode the algorithm of a VHDL entity, the two-process method only uses two processes per entity: one process that contains all combinational (asynchronous) logic, and one process that contains all sequential logic (registers). Using this structure, the complete algorithm can be coded in sequential (non-concurrent) statements in the combinational process while the sequential process only contains registers, i.e. the state.

Signals are grouped together in VHDL records, according to functionality, which simplifies the connection of modules. By using record types to group associated signal, the port list becomes both shorter and more readable.

The biggest difference between a program in VHDL and standard programming language such as C, is that VHDL allows concurrent statements and processes that are scheduled for execution by events rather than in the order they are written. This reflects indeed the dataflow behaviour of real hardware, but becomes difficult to understand and analyse when the number of concurrent statements passes some threshold. On the other hand, analysing the behaviour of programs written in sequential programming languages does not become a problem even if the program tends to grow, since there is only one thread of control and the execution is done sequentially from top to bottom. The two-process method asserts this problem by using only high-level sequential statements to code the algorithm.

3.10.1 TMU VHDL model

The component declaration of the TMU is given in Listing 3.1.

Listing 3.1: The component declaration of the TMU, `apbtmu.vhd`

```
component apbtmu
  generic (
    pindex   : integer := 0;          -- APB device select signal
    paddr    : integer := 0;          -- 12 MSb of APB address
    pmask    : integer := 16#fff#;    -- APB address mask
    pirq     : integer := 0;          -- which APB irq to generate
    ntimers  : integer range 1 to MAXTIMERS := 16;
    tbits    : integer range 1 to 32        := 32;  -- timer bits
    );
  port (
    rst      : in  std_logic;
    clk      : in  std_logic;
    apbi     : in  apb_slv_in_type;
    apbo     : out apb_slv_out_type;
    tmui     : in  tmu_in_type;
    tmuo     : out tmu_out_type
    );
end component;
```

The first four generics are common to all APB devices. They are used for configuring which APB select signal will be used to access the unit, its base address, address range and the interrupt line that the unit shall use. The TMU further has a generic specifying how many timers to implement, where the first timer is the Task Timer and subsequent timers will be IRQ Timers. The last generic selects how many bits each timer register will occupy. As for now, the maximum register width is 32 bits, sufficient for testing. The port signals, apart for clock, reset and APB in and out, are signals for connecting to the IRQ controller and setting the unit in debug mode. The declaration of the VHDL records `tmu_in_type` and `tmu_out_type` can be seen below:

```
type tmu_in_type is
  record
    dhalt : std_ulogic;                -- halt timers in debug mode
  end record;

type tmu_out_type is
  record
    apbi       : apb_slv_in_type;         -- to irq controller
    morebudget : std_logic_vector(MAXTIMERS-1 downto 1); -- only for debug
  end record;
```

As can be seen, the record declaration of `tmu_out_type` contains a `apbi` signal for the IRQ controller. This `apbi` is equivalent to the `apbi` in the port declaration, except that the IRQ signals are controlled by each implemented IRQ Timer. The record `tmu_in_type` contains a signal `dhalt` that freezes the TMU when asserted. `dhalt` should be connected to the LEON3 Debug Support Unit (DSU), stopping the TMU whenever the system is in debug mode. The TMU conforms to the theory of operation given in Section 3.1 and the complete VHDL model is available in Appendix B.

3.10.2 TMU register details

Section 3.1 introduced the processor accessible registers of the TMU. This section will describe these registers in detail, starting with the address mapping of all TMU registers in Table 3.3.

Table 3.3: The Time Management Unit (TMU) registers

Address offset	**Register name**
0x00	Configuration
0x04	Timer select
0x08	Pending IRQs
0x0C	Unused
0x10	Timer 1 Counter
0x14	Timer 1 Limit
0x18	Timer 1 Replenish Period
0x1C	Timer 1 Control
0xn0	Timer n Counter
0xn4	Timer n Limit
0xn8	Timer n Replenish Period
0xnC	Timer n Control

Table 3.4: TMU Configuration register

31 – 13	12	11 – 6	5 – 1	0
"00..0"	D	TBITS	NTIMERS	E

0 : Enable or disable the TMU. The current timer will only increment if this bit is set.

1 – 5 : The number of timers implemented. Read only.

6 – 11 : How many bits that are used for timer registers; count, limit and period. Read only.

12 : Indicates whether the TMU is in debug mode. Debug mode is controlled by the LEON3 Debug Support Unit (DSU). Read only.

13 – 31 : Unused

Table 3.5: TMU Timer Select register:

31 – 5	4 – 0
"00..0"	TSEL

0 – 4 : Value selects which timer to enable
5 – 31 : Unused

Table 3.6: TMU Pending IRQ register:

31 – 0
IPEN

0 – 31 : Bit i is set by hardware when timer i's budget is spent. Software must clear these bits.

Table 3.7: TMU Timer n Count register:

tbits-1 – 0
COUNT

0 – tbits-1 : Value.

Table 3.8: TMU Timer n Limit register:

tbits-1 – 0
LIMIT

0 – tbits-1 : Value.

Table 3.9: TMU Timer n Period register:

tbits-1 – 0
PERIOD

0 – tbits-1 : Value.

Table 3.10: TMU Timer n Control register:

31 – 2	1	0
"00..0"	IP	IE

0 : Enable interrupt generation for this timer. For the Task Timer, it means an interrupt is requested when count equals limit. For IRQ Timers, it means that the IRQ signal associated with that timer is decoupled from the IRQ controller while its budget is empty and waiting for a replenish. When the budget is replenished, interrupts will be forwarded to the IRQ controller again.
1 : Indicates a pending interrupt, and is the same bit as in the Pending Interrupt register. Must be cleared by software.
2 – 31 : Unused.

3.10.3 TMU logic utilisation

The logic resource utilisation of the TMU is listed in Table 3.11, with different settings of the VHDL generic `ntimers`. When '1', only the Task Timer is synthesised. The number of IRQ Timers is `ntimers` - 1. All TMU registers are 32-bits wide. The table data is taken from the map report generated by the Xilinx implementation tools after synthesis, translation and mapping. The target device is a Xilinx Virtex-4 XC4VLX25.

Table 3.11: Time Management Unit (TMU) logic utilisation on a Virtex-4 FPGA

Timers	**LUTs**
1	262
2	683
4	1471
8	3099
12	4713
16	6348

Table 3.11 can be compared to the logic utilisation of some GRLIB IP cores in Table 3.12, taken from the GRLIB IP Core User's Manual [12].

Table 3.12: Approximate logic utilisation of some GRLIB IP cores on a Virtex-2 FPGA

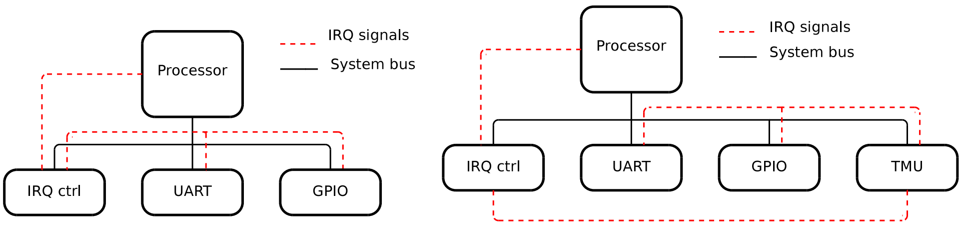
IP	**LUTs**
APBUART	200
DDRCTRL	1600
GRGPIO, 16-bit configuration	100
IRQMP (1 processor)	300
GRETH 10/100 Mbit Ethernet MAC with EDCL	2600
LEON3, 8 + 8 Kbyte cache	4300
GPTIMER (16-bit scaler + 2x32-bit timers)	250

The Task Timer, requiring about 262 LUTs, takes about the same amount of resources as the GPTIMER with a 16-bit prescaler and 2 x 32-bit timers. An approximate number of LUTs required by an IRQ Timer is found by subtracting the number of LUTs used for implementing the Task Timer and one IRQ Timer and when only implementing the Task Timer: $683 - 262 = 421$. IRQ timers use about 60 % more resources than the Task Timer, because of the added complexity by the period register and its surrounding logic.

In the overall demo design used for implementing the TMU, using the TMU with four timers increased the number of LUTs by 8 %.

3.11 Adding the TMU to a LEON3-design

The TMU is implemented as a peripheral device on the AMBA APB bus. This setup is illustrated in in Figure 3.9(a) where the processor is connected to its peripheral devices via a common system bus. Interrupt signals are routed to the IRQ controller which forwards the highest priority unmasked interrupt to the processor. Figure 3.9(b) shows the same system with an added TMU, connected to the same system bus. Interrupt signals from the peripheral devices are now routed to the TMU. For each incoming interrupt signal, the TMU passes it on to the IRQ controller while its corresponding IRQ Timer has not timed out.



(a) Standard (b) Added TMU

Figure 3.9: A System-on-Chip (SoC) design with a processor and peripheral devices

Using the proposed TMU in an existing LEON3 design is easy. First, the TMU VHDL file, `apbtmu.vhd`, must be specified in the makefile of the design by adding it to the list of synthesisable files:

```
VHDLSYNFILES=config.vhd ahbrom.vhd apbtmu.vhd leon3mp.vhd
```

The files are compiled in order from left to right, requiring the top design file, `leon3mp.vhd`, to be specified at the end, as it depends on the other files.

Now the TMU package defined in `apbtmu.vhd`, will be available in the VHDL library `work` and can be used in the top design file, `leon3mp.vhd`, by specifying a use clause, before the entity declaration like this:

```
use work.tmupkg.all;     -- Time Management Unit (TMU) package/library
```

In the architecture declaration part of the top design, two signals for the TMU are listed as well as an integer constant enabling conditional instantiation of the TMU:

```
constant CFG_TMU_ENABLE : integer := 1;
signal tmui : tmu_in_type;
signal tmuo : tmu_out_type;
```

The next step is to instantiate the TMU in the top design. Note the use of conditional generate statements.

```
tmugen : if CFG_TMU_ENABLE /= 0 generate
  tmu0 : entity work.apbtmu
    generic map (pindex => 9, paddr => 16#010#, pmask => 16#fff#,
      pirq => 10, ntimers => 4, tbits => 32)
    port map (rst => rstn, clk => clkm, apbi => apbi, apbo => apbo(9),
      tmui => tmui, tmuo => tmuo);
end generate;

tmui.dhalt <= dsuo.tstop;

tmungen : if CFG_TMU_ENABLE = 0 generate
  tmuo.apbi <= apbi;
end generate;
```

The value of generic `pindex` must not equal any other instantiated peripheral. The same applies to `apbo(9)`. `paddr` sets the 12 most significant bits of the units base address relative to that of the AHB/APB bridge unit. The AHB/APB unit address range start at `0x80000000` giving the TMU its base address at `0x80001000`. `pmask` specifies the address range of the unit, and when set to `0xfff` the unit has 256 bytes address range – enough for 16 timers. The *debug halt* signal of the TMU, `tmui.dhalt`, is connected to the Debug Support Unit (DSU), setting the TMU in debug mode at the control of the DSU.

The last step is to connect the APB signals from the TMU to the IRQ controller by using `tmuo.apbi` instead of `apbi` in the port map of the IRQ controller:

```
irqctrl : if CFG_IRQ3_ENABLE /= 0 generate
  irqctrl0 : irqmp            -- interrupt controller
  generic map (pindex => 2, paddr => 2, ncpu => NCPU)
  port map (rstn, clkm, tmuo.apbi, apbo(2), irqo, irqi);
end generate;
```

3.12 Hardware simulation

The TMU has been simulated by using the VHDL simulator GHDL and the wave viewer GTKWave. GRLIB comes with VHDL testbenches for each LEON3 processor demo design, and makefiles for building a simulation model of the design with many VHDL simulators. Building the testbench executable with GHDL is accomplished by typing `make ghdl` at the shell prompt. When running the testbench, it first starts inspecting the system configuration by reading the AMBA "plug and play" registers of all IP cores and printing information to the terminal. Then, the testbench starts the processor which executes the contents of the simulated memory. The memory contents can be updated by typing `make soft`. `make soft` builds a library of test procedures for all parts of the system from source files in GRLIB, compiles the file `systest.c` in the local design and finally links the code and store the binary in `.srec` file formats ready for the testbench. An example `systest.c` is displayed below.

```
main()
{
  report_start();
  base_test(); // leon3_test(), irqtest(), gptimer_test(), apbuart_test()
  tmu_test(0x80001000, 10);
  report_end();
}
```

`base_test()` performs tests on the LEON3 processor, the IRQ controller, the general purpose timer and the UART. A source file was written to exercise the TMU for simulation purposes, and can be found as part of the GRLIB patch in Appendix A. The TMU test procedure makes sure that all TMU registers can be read or written, testing that the timers will start counting when they are activated and testing the IRQ of the Task Timer. All that is needed is to call `tmu_test()` with the base address of the TMU and its IRQ number.

The procedure for simulating the design is thus:

```
$ cd /path/to/design
$ make soft
$ make ghdl
$ ./testbench --wave=dump.ghw
$ gtkwave dump.ghw
```

If only the software changes, the testbench need not be rebuilt. If some VHDL files are changed, the next run of `make ghdl` might fail and the remedy is `make ghdl-clean; make ghdl`.

Below is the output of a simulation.

```
$ ./testbench --wave=dump.ghw
LEON3 Digilent XC3S1000 Demonstration design
GRLIB Version 1.0.17, build 2710
Target technology: spartan3  ,  memory library: spartan3
ahbctrl: AHB arbiter/multiplexer rev 1
ahbctrl: Common I/O area disabled
ahbctrl: AHB masters: 2, AHB slaves: 8
ahbctrl: Configuration area at 0xfffff000, 4 kbyte
ahbctrl: mst0: Gaisler Research        Leon3 SPARC V8 Processor
ahbctrl: mst1: Gaisler Research        JTAG Debug Link
ahbctrl: slv0: European Space Agency   Leon2 Memory Controller
ahbctrl:       memory at 0x40000000, size 1024 Mbyte, cacheable, prefetch
ahbctrl: slv1: Gaisler Research        AHB/APB Bridge
ahbctrl:       memory at 0x80000000, size 1 Mbyte
ahbctrl: slv2: Gaisler Research        Leon3 Debug Support Unit
ahbctrl:       memory at 0x90000000, size 256 Mbyte
ahbctrl: slv4: Gaisler Research        Test report module
ahbctrl:       memory at 0x20000000, size 1 Mbyte
ahbctrl: slv6: Gaisler Research        Generic AHB ROM
ahbctrl:       memory at 0x00000000, size 1 Mbyte, cacheable, prefetch
apbctrl: APB Bridge at 0x80000000 rev 1
apbctrl: slv0: European Space Agency   Leon2 Memory Controller
```

```
apbctrl:       I/O ports at 0x80000000, size 256 byte
apbctrl: slv1: Gaisler Research        Generic UART
apbctrl:       I/O ports at 0x80000100, size 256 byte
apbctrl: slv2: Gaisler Research        Multi-processor Interrupt Ctrl.
apbctrl:       I/O ports at 0x80000200, size 256 byte
apbctrl: slv3: Gaisler Research        Modular Timer Unit
apbctrl:       I/O ports at 0x80000300, size 256 byte
apbctrl: slv5: Gaisler Research        PS2 interface
apbctrl:       I/O ports at 0x80000500, size 256 byte
apbctrl: slv6: Gaisler Research        VGA controller
apbctrl:       I/O ports at 0x80000600, size 256 byte
apbctrl: slv8: Gaisler Research        General Purpose I/O port
apbctrl:       I/O ports at 0x80000800, size 256 byte
apbctrl: slv9: NTNU ITK                Time Management Unit
apbctrl:       I/O ports at 0x80001000, size 256 byte
ahbrom6: 32-bit AHB ROM Module,  108 words, 7 address bits
apbtmu9: Time Management Unit rev 0, with 4 32-bit timers, irq 10
clkgen_spartan3e: spartan3/e sdram/pci clock generator, version 1
clkgen_spartan3e: Frequency 50000 KHz, DCM divisor 4/5
leon3_0: LEON3 SPARC V8 processor rev 0
leon3_0: icache 1*8 kbyte, dcache 1*8 kbyte
dsu3_2: LEON3 Debug support unit + AHB Trace Buffer, 2 kbytes
ahbjtag AHB Debug JTAG rev 0
apbuart1: Generic UART rev 1, fifo 4, irq 2
irqmp: Multi-processor Interrupt Controller rev 3, #cpu 1
gptimer3: GR Timer Unit rev 0, 8-bit scaler, 2 32-bit timers, irq 8
apbps2_5: APB PS2 interface rev 0, irq 5
apbvga6: APB VGA module rev 0
grgpio8: 18-bit GPIO Unit rev 0
testmod4: Test report module

**** GRLIB system test starting ****
Leon3 SPARC V8 Processor
  CPU#0 register file
  CPU#0 multiplier
  CPU#0 radix-2 divider
  CPU#0 cache system
Multi-processor Interrupt Ctrl.
Modular Timer Unit
  timer 1
  timer 2
  chain mode
Generic UART
Test passed, halting with IU error mode

testbench.vhd:117:6:@932123ns:(assertion failure): *** IU in error mode,
    simulation halted ***
./testbench:error: assertion failed
./testbench:error: simulation failed
```

Note that the error at the end is correct behaviour; the only way to stop a simulation from within the design is to generate an error [13].

3.12.1 Problems

A few problems occured while trying to simulate the design.

- Simulation files for GHDL are generated from a master makefile within GRLIB with statements like `@echo -e \\tmkdir gnu > compile.ghdl`. The `-e` option enables interpretation of backslash escapes, but for some reason fails when used in a makefile[6]. The result is that the `-e` is written to the file and no "tab" character. The solution is to remove all `-e`'s from these statements. See Appendix A for the patch that fix this.
- GRLIB has a graphical user interface for configuration and simulation which can be started with `make xgrlib`. Using this Graphical User Interface (GUI) for simulation fails.
- The testbench for the Xilinx ML401 development board fails. Thus, a different (arbitrary) demo design was used for simulation[7].

3.12.2 Results

By inspecting the signals in GTKWave, the TMU has been proved to perform according to its specification; IRQ generation of the Task Timer works and so does IRQ blocking of the IRQ Timers as well as the replenishment of budgets.

Figure 3.10 illustrates the workings of the Task Timer. The Timer Select register, shown as signal `tsel`, is initially set to zero. That way the Task Timer is selected. The Limit register is written with the value 150 and interrupt generation is enabled by setting `irqen`. Then the global enable bit is set, which resumes the active timer. Count increments each cycle and when equal to Limit, a short interrupt pulse is generated and the pending interrupt bit, `irqpen` is set. The testbench incorporates a minimal interrupt handler for the TMU IRQ which clears Count and the pending interrupt bit `irqpen` – as seen about $t = 104us$. Yet another interrupt is generated before the Timer Select register is written to select the first IRQ Timer and the Task Timer stops.

3.13 Changes to eCos for TMU support

This section describes what has been done to accomplish support for the TMU in eCos. The modified parts of eCos includes:

- The HAL
  - Added the Task Timer registers to the thread context

[6] At least on a Ubuntu 7.04 system with GNU Make 3.81
[7] Digilent XC3S1000, known as `leon3-digilent-xc3s1000` in GRLIB

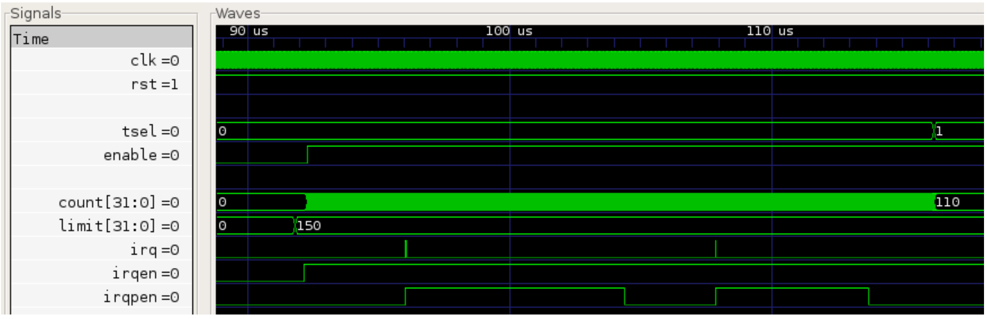


Figure 3.10: GTKWave displaying signals of the Task Timer

  – The trap handling code – changing the active TMU timer when runnning ISRs and added termination model exception handlers

- Added a return value to the exception handler prototypes in the kernel to support termination model exceptions
- Added a TMU API
- Changed the eCos configuration system to allow easy TMU configuration when building eCos

LEON3 is a SPARC processor, thus all references to the HAL means the SPARC HAL. All modified eCos source code is available in Appendix A.

3.13.1 Changes to the HAL

The struct `HAL_SavedRegisters` in file `hal_arch.h` contains the definition of the register layout on the stack, that is, the saved thread context. `HAL_SavedRegisters` has been modified to incorporate the additional TMU registers as illustrated in Figure 3.11. There are only three Task Timer registers, Count, Limit and Control, but due to a simplification of the TMU VHDL design, the Task Timer also has a Period register, making a total of four TMU registers in the thread context. The Period register of the Task Timer is a dummy register not performing any task. It can be read and written though, and has been used for debug.

The file `icontext.c` contains the function `hal_thread_init_context()` which initialises a thread context. A pointer variable called `regs` of type `HAL_SavedRegisters *` is used to initialise the thread context in memory, given the thread's stack base address. The Task Timer initialisation code in this file is listed below. The only register that has a meaningful value at this point is the control register. When set to zero, it ensures that all threads start out with its Task Timer disabled.

```
#ifdef CYGPKG_HAL_HAS_TMU // TMU Time Management Unit
    // init TMU task timer regs
    regs->count = 0xF00DF00D; // lower addr
```

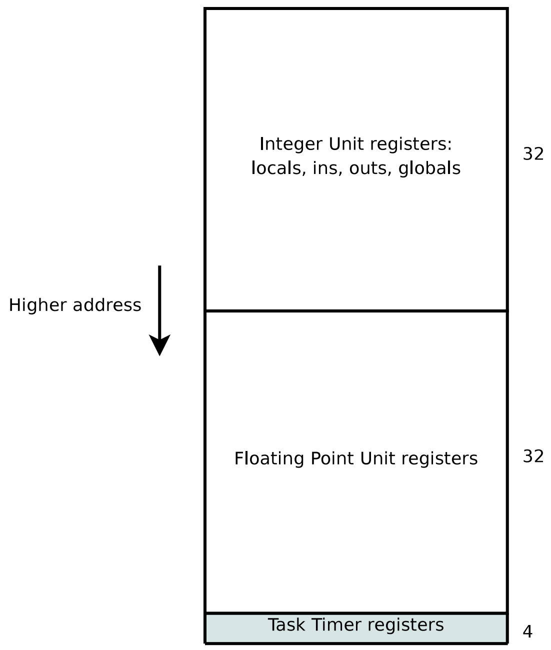


Figure 3.11: The layout of saved thread context on the stack

```
    regs->limit = 0xCAFEBABE;
    regs->period = 0xDEADBEEF;
    regs->control = 0;          // higher addr
#endif // !CYGPKG_HAL_HAS_TMU
```

As for the context switch code, `context.S`, there are two modified functions, `hal_thread_switch_context` and `hal_thread_load_context`. Actually, the upper half of `hal_thread_switch_context` saves the context and then drops through to `hal_thread_load_context`. The saving and loading of TMU registers have been placed as close as possible, to reduce the amount of "lost" cycles as illustrated in Figure 3.12.

The SPARC architecture defines traps as the union of interrupts and exceptions. Figure 3.13 is a flowdiagram of the eCos trap handling code with modifications marked by the shaded elements. When a trap occurs, eCos calls the Vector Service Routine (VSR) that has been registered for that trap vector. The interrupt VSR is located in `vec_ivsr.S` and the exception VSR in `vec_xvsr.S`. Both VSRs set up a context for which the ISR or Exception Service Routine (XSR) is called, but there are some important differences between the interrupt handler and exception hander system that will be explained next.

Interrupts are asynchronous events caused by external devices. They may arrive at any time and are not associated in any way with the thread that is currently running. ISRs may not perform any system call or anything other than what is directly related to the interrupt subsystem and hardware. If

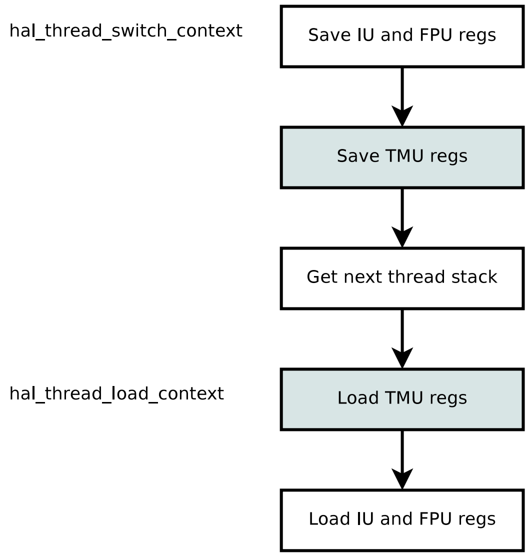


Figure 3.12: Flowchart of the context switch code

further processing is needed, the ISR may specify that a Deferred Service Routine (DSR) is run as soon as possible. If the scheduler is unlocked, the DSR is run immediately after the ISR. If not, the DSR is delayed until the scheduler is unlocked. DSRs can perform a subset of system calls, the non-blocking ones. Because DSRs have the ability to release threads, a re-schedule is performed after any DSR.

An exception is a synchronous event caused by the execution of a thread. These include both the machine exceptions raised by hardware such as divide-by-zero, memory fault and illegal instruction. XSRs in eCos is called from thread context and may use any system call. No re-scheduling is performed at the end of an XSR.

The TMU Task Timer generates asynchronous interrupts, but are interpreted as exceptions associated with the current thread. Handling the event as an exception is a more correct abstraction, and because interrupts are not disabled as in ISRs and the scheduler is not locked as in DSRs, the exception handler may use any system call. There are two models for handling exceptions; the resumption model and the termination model. In the resumption model, the program returns to the place where the exception was raised after the handler had been run. In the termination model, the program does not return to the point where the exception occurred. An example of resumption model exceptions are POSIX signals, and for the termination model; Ada exceptions.

eCos provide only resumption model exception handlers, and as discussed in Section 3.14, termination exception handlers for the TMU was opted. In order to make termination model exception handlers in eCos, the first step was to add a non-void return value to the exception handler prototypes. The next step was to modify the exception VSR to check, and possibly jump to, the

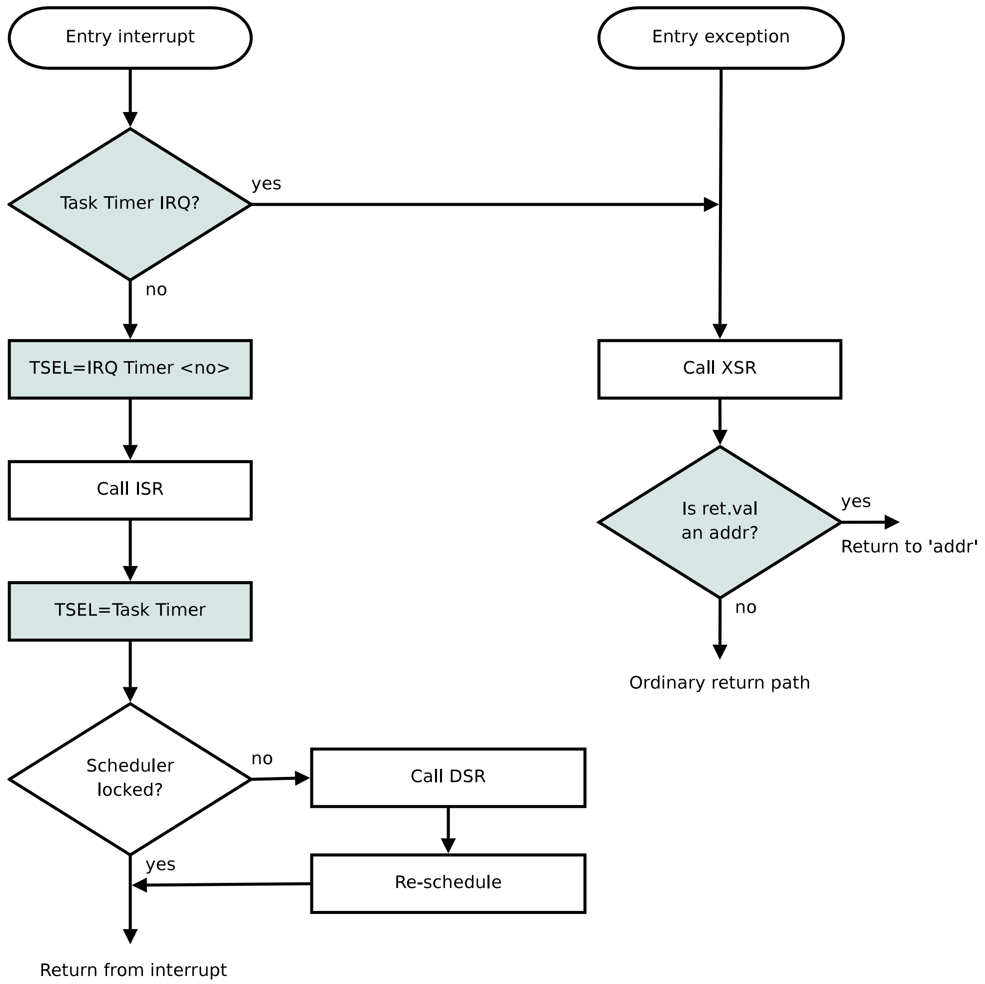


Figure 3.13: A flowchart of the eCos interrupt and exception VSRs. The shaded blocks are added for TMU support

value of the return argument. If the return argument is below or equal to a small number, say 10, the handler returns to the instruction that caused the exception – the resumption model. If the return argument is above 10, the argument is treated as an address and the VSR will jump to this address instead – the termination model. Thus, the exception handler simply returns a function pointer which the VSR will jump to instead of the saved program counter from when the exception occurred.

3.13.2 The TMU API

The files hal_tmu.h and hal_tmu.c were written as the TMU API, and is available in Appendix A. User applications must include the header file to get access to the API:

```
#include <cyg/hal/hal_tmu.h>          // the TMU API header
```

The API consist of the following functions:

```
// Set the TMU to the state it enters upon reset: all regs set to
// zero except Limit & Period regs which is set to 0xffffffff
void tmu_reset(void);

// Enable counters - and in effect, interrupt generation/blocking
void tmu_enable(void);

// Freeze timers
void tmu_disable(void);

// Get the number of implemented timers
unsigned int tmu_ntimers(void);

// Get the number of bits used for timer registers
unsigned int tmu_tbits(void);

// Enable/disable IRQ generation if Task Timer,
// or IRQ blocking if IRQ Timer
void tmu_enable_irq(int timer);
void tmu_disable_irq(int timer);

// Read/clear the Pending Interrupt register
unsigned int tmu_read_irqpen(void);
void         tmu_clear_irqpen(void);

// Setup any timer, preferrably IRQ timers
void tmu_tmr_setup(int timer, int limit, int period, int irqen);


// Setup the Task Timer for the current thread, must be called in
// its context! 'period' is not actually used - debug
void tmu_task_tmr_setup(int limit, int period, int irqen);

unsigned int tmu_task_tmr_get_count(void);
void         tmu_task_tmr_set_count(unsigned int val);
```

```
unsigned int tmu_task_tmr_get_limit(void);
void         tmu_task_tmr_set_limit(unsigned int val);

unsigned int tmu_task_tmr_get_irqen(void);
void         tmu_task_tmr_set_irqen(int irqen);

unsigned int tmu_task_tmr_get_irqpen(void);
void         tmu_task_tmr_set_irqpen(int irqpen);
```

3.13.3 Changes to the configuration system

All TMU code for eCos is written within a guard macro called `CYGPKG_HAL_HAS_TMU`. If the TMU is not used, its processing overhead can be removed completely at compile time by not defining that macro. If the TMU is used, `CYGPKG_TMUADDR` defines the base address of the unit, and `CYGPKG_HAL_TMU_IRQ` the IRQ number of the TMU. All three macros can be set in the eCos Configuration Tool under eCos HAL → SPARC architecture.

To incorporate these options into the configuration tool, the SPARC HAL package configuration file, `hal_sparc.cdl`, was modified. The following TMU component definition was added, allowing the user of the eCos Configuration Tool to manipulate the TMU macros.

```
cdl_component CYGPKG_HAL_HAS_TMU {
    display       "Has TMU"
    default_value 1
description   "This option will add support for the Time Management Unit
    (TMU). This means that the Task Timer registers of the TMU
    will be added to the thread context. The user may then specify execution
    budgets for ordinary tasks and interrupts (ISRs). For tasks, the user
    can register an exception handler (CYGNUM_HAL_EXCEPTION_OTHERS) to
    handle tasks that exceed their execution budget. API: hal_tmu.h"

    cdl_option CYGPKG_TMUADDR {
        display       "Base address of the TMU"
        flavor        data
        default_value 0x80001000
    }

    cdl_option CYGPKG_HAL_TMU_IRQ {
        display       "IRQ of the TMU"
        flavor        data
        default_value 10
    }
}
```

The TMU API `hal_tmu.c`, see Section 3.13.2, must be included in the eCos build process to allow its use by application programs. This was performed by adding the file name to the list of files in the `hal_sparc.cdl` used for compilation of the HAL:

```
compile       hal_intr.c hal_boot.c callcons.S memcpy.S hal_tmu.c
```

3.14 Handling the Task Timer timeout

POSIX signal handlers and the standard exception handlers in eCos are based on the resumption model. After a resumption handler has run, control is transferred back to the instruction that raised the exception. This concept is illustrated in Figure 3.14 with a flowdiagram of a simple thread using a resumption model TMU exception handler. But the resumption model makes

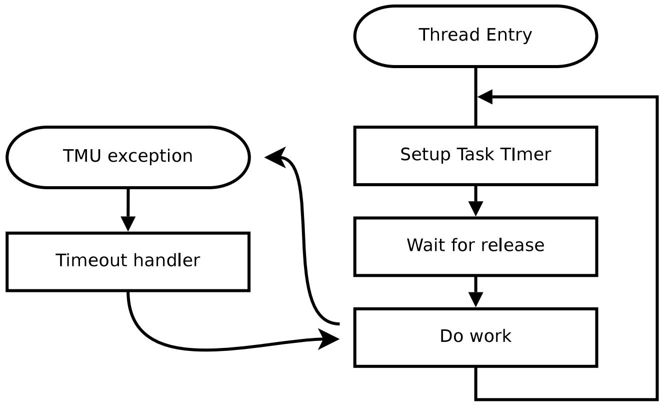


Figure 3.14: Flowchart of a thread with a resumption model Task Timer exception handler

little sense for the Task Timer exception. When the TMU raises an exception, because the current thread has exceeded its budget, an *alternative* algorihm or code sequence must be provided, because the current algorithm clearly did not produce a result on time. There is no point in continuing execution at the instruction that caused the exception and therefore the termination model was opted. The termination model exception handler system was implemented in eCos as described in Section 3.13.1.

Figure 3.15 shows a flowdiagram of a simple thread using the termination model exception handler for the Task Timer as well as `setjmp/longjmp` for non-local jumps or control flow. The thread performs a call to `setjmp`, saving its execution environment in a jump buffer. Later when calling `longjmp`, the execution environment is restored to the saved state, effectively transferring the control back to the `setjmp` line. When executing the `setjmp` call, it returns zero, but when jumped to from `longjmp` it returns the value specified in the second argument of `longjmp`. The curly lines in Figure 3.15 denotes a special transfer of control. Line one represents the control flow taken when the thread's Task Timer raises an exception. The exception handler only specifies where to continue executing when the handler returns, shown as line two. The dummy function has only one purpose; to perform the `longjmp` call, causing a jump to `setjmp` shown by line three, so that the thread's timeout handler can be placed within the thread, having access to all necessary data and information about the thread. Each time the thread is about to do some work, it resets the count register of the Task Timer. While the thread is waiting for its release and is in the blocked state, its budget is preserved.

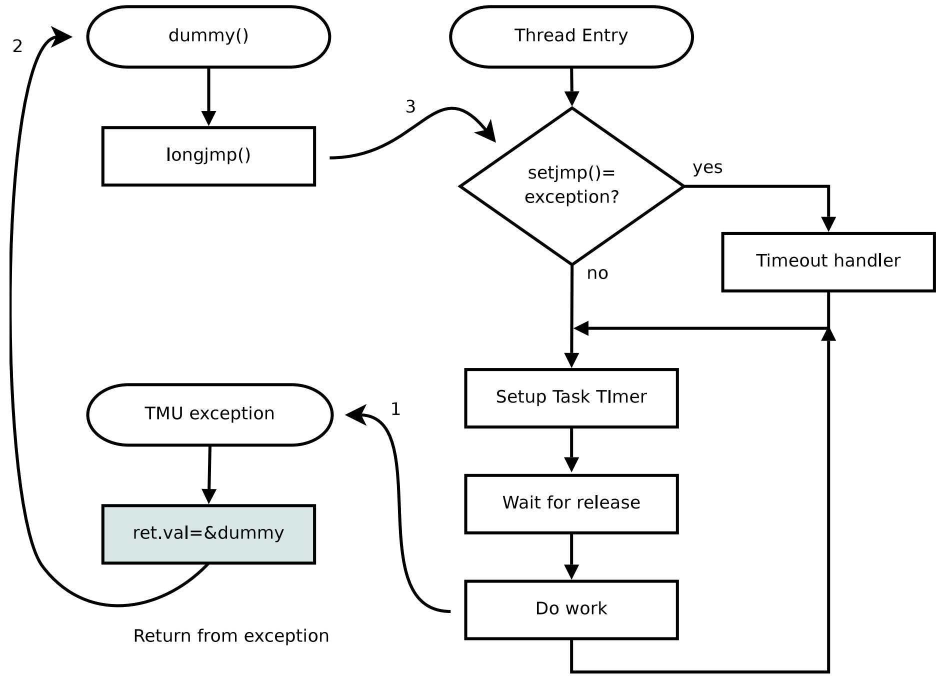


Figure 3.15: Flowchart of a thread with termination model Task Timer exception handler

3.14.1 Problems

Asynchronous Transfer of Control (ATC) must be used carefully so that the state of the system is not corrupted. Imagine a thread that has just aquired a mutex and immediately receives the TMU exception. The thread asynchronously jumps to the timeout handler and then back to the thread's entry, not releasing the aquired mutex. On the next run it will try to aquire the same mutex again, causing problems in the OS kernel if the system does not support recursive mutexes. eCos for one, does not support recursive mutexes. Thus, to use the TMU with Asynchronous Transfer of Control (ATC), care must be taken to do a proper tidy up in the timeout handler.

Chapter 4

Tests

This chapter presents the final tests on the target system. This includes a test of the Task Timer and an IRQ Timer.

4.1 Setup

The target system is based on the LEON3 demo design for the Xilinx ML401 Evaluation Platform, which in GRLIB is called `leon3-avnet-ml401`, with an added TMU as described in Section 3.11. The demo design almost fully utilises the resources of the FPGA, leaving only room for 4 TMU timers. To make room for more timers, the GRLIB configuration had to be stripped. Table 4.1 lists changes to the standard demo design which enabled up to 14 TMU timers.

Table 4.1: Changes to the ML401 demo design

What	**Old value**	**New value**
System clock	65 MHz	40 MHz
Power down mode	enabled	disabled
Cache	8 + 8 KiB	disabled
MMU	enabled	disabled
DSU instruction trace	enabled	disabled
DSU JTAG	enabled	disabled
I2C	enabled	disabled

The system clock setting must be accompanied with a modification of the ucf-file (User Constraint File) of the design, specifying timing constraints:

```
#TIMESPEC "TS_rclk270b_clkml_rise" = FROM "rclk270b_rise" TO "clkml_rise" 3.500;
TIMESPEC "TS_rclk270b_clkml_rise" = FROM "rclk270b_rise" TO "clkml_rise" 4.000;#new
```

It might seem strange that a reduction of the operating frequency allows for more logic. But the process of implementing an HDL description of a design into actual hardware in an FPGA is an optimisation problem of speed versus area. When the FPGA becomes almost full, the speed optimal placements of logic blocks and routing cannot be done. The result is a reduction in speed and the ability to utilise more logic.

The system is running eCos with the modifications to the HAL and the exception subsystem of the kernel as presented in Section 3.13.

4.2 Test 1: Sporadic events

This test demonstrates how the system handles an overload of sporadic events, or interrupts, via the TMU. To make the test more reproducible, two threads are incorporated:

Thread 1: This thread manipulates the GPIO to generate a hardware interrupt. The interrupt line is routed through the TMU. After each generated interrupt, the thread sleeps for a variable amount of time. Although the interrupt interval is not actually sporadic, controlled interval times will display the system behaviour better.

Thread 2: This thread is released by the DSR that runs when the processor is interrupted by the GPIO unit. It has the highest priority of the two threads enabling it to preempt Thread 1.

The source code can be found in Appendix A.

The Task Timer of the TMU is not used in this test, but there is no problem in mixing the use of the Task Timer and an IRQ Timer. The TMU is set up to accept interrupts from the GPIO unit at a limited interarrival time, thus limiting the effective release frequency of Thread 2. This is accomplished by setting the limit register to '1', and the period register to the desired budget replenish interval. The TMU can also be set up to accept bursts of interrupts by setting the limit register, and possibly the period register, to a greater value.

4.2.1 Results

```
grlib> load test-tmu-irq-timer
grlib> run
Entering cyg_user_start() function
Beginning execution; thread 2
Beginning execution; thread 1
  --> Thread 2: released!
Thread 1: generated irq and now a delay of 5 clock ticks
Thread 1: generated irq and now a delay of 6 clock ticks
Thread 1: generated irq and now a delay of 7 clock ticks
Thread 1: generated irq and now a delay of 8 clock ticks
```

```
Thread 1: generated irq and now a delay of 9 clock ticks
Thread 1: generated irq and now a delay of 10 clock ticks
Thread 1: generated irq and now a delay of 12 clock ticks
Thread 1: generated irq and now a delay of 14 clock ticks
Thread 1: generated irq and now a delay of 16 clock ticks
Thread 1: generated irq and now a delay of 19 clock ticks
Thread 1: generated irq and now a delay of 22 clock ticks
Thread 1: generated irq and now a delay of 26 clock ticks
  --> Thread 2: released!
Thread 1: generated irq and now a delay of 31 clock ticks
Thread 1: generated irq and now a delay of 37 clock ticks
Thread 1: generated irq and now a delay of 44 clock ticks
Thread 1: generated irq and now a delay of 52 clock ticks
Thread 1: generated irq and now a delay of 62 clock ticks
  --> Thread 2: released!
Thread 1: generated irq and now a delay of 74 clock ticks
Thread 1: generated irq and now a delay of 88 clock ticks
  --> Thread 2: released!
Thread 1: generated irq and now a delay of 105 clock ticks
Thread 1: generated irq and now a delay of 126 clock ticks
  --> Thread 2: released!
Thread 1: generated irq and now a delay of 151 clock ticks
  --> Thread 2: released!
Thread 1: generated irq and now a delay of 181 clock ticks
  --> Thread 2: released!
Thread 1: generated irq and now a delay of 217 clock ticks
  --> Thread 2: released!
Thread 1: generated irq and now a delay of 260 clock ticks
  --> Thread 2: released!
Thread 1: generated irq and now a delay of 312 clock ticks
  --> Thread 2: released!
Thread 1: generated irq and now a delay of 374 clock ticks
  --> Thread 2: released!
Thread 1: generated irq and now a delay of 448 clock ticks
  --> Thread 2: released!
Thread 1: generated irq and now a delay of 5 clock ticks
Thread 1: generated irq and now a delay of 6 clock ticks
Thread 1: generated irq and now a delay of 7 clock ticks
Thread 1: generated irq and now a delay of 8 clock ticks
Thread 1: generated irq and now a delay of 9 clock ticks
Thread 1: generated irq and now a delay of 10 clock ticks
Thread 1: generated irq and now a delay of 12 clock ticks
  --> Thread 2: released!
Thread 1: generated irq and now a delay of 14 clock ticks
Thread 1: generated irq and now a delay of 16 clock ticks
Thread 1: generated irq and now a delay of 19 clock ticks
Thread 1: generated irq and now a delay of 22 clock ticks
Thread 1: generated irq and now a delay of 26 clock ticks
Thread 1: generated irq and now a delay of 31 clock ticks
Thread 1: generated irq and now a delay of 37 clock ticks
Thread 1: generated irq and now a delay of 44 clock ticks
  --> Thread 2: released!
Thread 1: generated irq and now a delay of 52 clock ticks
Thread 1: generated irq and now a delay of 62 clock ticks
Thread 1: generated irq and now a delay of 74 clock ticks
  --> Thread 2: released!
Thread 1: generated irq and now a delay of 88 clock ticks
```

```
68 Thread 1: generated irq and now a delay of 105 clock ticks
69
70 Interrupt!
71 stopped at 0x40005e24
72 grlib>
```

Thread 2 is initially released as the IRQ Timer has a full budget. Summing the delays of Thread 1, lines 7-18, adds up to 154 ticks. At this time, the IRQ Timer has had its budget replenished and allows interrupts to pass through to the processor which schedules Thread 2, on line 19. Jumping to line 53, Thread 2 is released after only 57 ticks since its last release. Since the IRQ timers conform to the Deferrable Server (DS) algorithm, that is, replenishing the budget at regular intervals, this behaviour is correct. What happens is that at the previous release, the timer is very close to its next replenish period.

The result is that when interrupts arrive more often than the TMU accepts, the interrupt is stopped at the TMU and the processor usage is preserved. If it is important not to miss interrupts, the Interrupt Pending bit can be read.

4.3 Test 2: Tasks with variable execution times

This test will show how the system handles periodic tasks which execute for too long. Two threads are used to make sure that the single Task Timer hardware unit works for more than one thread. The threads are identical and used ATC TMU exception handler. Both threads perform a CPU heavy *for*-loop, iterating over `i` from zero to `work`, where `work` cycles between 100, 150 and 200. Both threads use the same TMU setup, allowing almost work 191 to be performed until the TMU interrupts. The source code can be found in Appendix A.

4.3.1 Results

```
grlib> load test-tmu-task-timer
grlib> run
Entering cyg_user_start() function
Beginning execution; thread 1
Beginning execution; thread 2
Thread 1: used  783785 TMU cycles work=100
Thread 2: used  788489 TMU cycles work=100
Thread 1: used 1173512 TMU cycles work=150
Thread 2: used 1174685 TMU cycles work=150
  --> Thread 1 timeout: TMU count=1501535 work=200 i=191
  --> Thread 2 timeout: TMU count=1501604 work=200 i=191
Thread 1: used  783857 TMU cycles work=100
Thread 2: used  789306 TMU cycles work=100
Thread 1: used 1173595 TMU cycles work=150
Thread 2: used 1179100 TMU cycles work=150
  --> Thread 1 timeout: TMU count=1501534 work=200 i=191
  --> Thread 2 timeout: TMU count=1501595 work=200 i=191
Thread 1: used  783929 TMU cycles work=100
Thread 2: used  789275 TMU cycles work=100
```

```
Thread 1: used 1173736 TMU cycles work=150
Thread 2: used 1178989 TMU cycles work=150
  --> Thread 1 timeout: TMU count=1501537 work=200 i=191
  --> Thread 2 timeout: TMU count=1501583 work=200 i=191
Thread 1: used  783878 TMU cycles work=100
Thread 2: used  789308 TMU cycles work=100
Thread 1: used 1173630 TMU cycles work=150
Thread 2: used 1179229 TMU cycles work=150
  --> Thread 1 timeout: TMU count=1501536 work=200 i=191
  --> Thread 2 timeout: TMU count=1501590 work=200 i=191

Interrupt!
stopped at 0x4000669c
grlib>
```

As can be seen, both threads succeed in computing work=100 and work=150, but when they try to perform work=200, the Task Timer fires at i=191 and the thread's exception handler is called.

Chapter 5

Discussion

5.1 Results

The test of the IRQ timers of the TMU showed that the unit can limit the number of interrupts forwarded to the processor within a given period. ISRs are assumed to be short and have almost constant execution time, thus the limitation of interrupt occurrences have almost the same effect as limiting the execution times of the ISRs themselves. Once an IRQ Timer is set up, it is autonomous, requiring no more processor intervention causing overhead.

The test of the Task Timer of the TMU showed that the unit accurately measures the execution times of threads running on the system and generates an interrupt when a given execution limit is specified. The unit measures execution time so accurately that variations between thread runs become apparent. The Task Timer is fully setup once for a thread, but before each cycle, the Task Timer must be reset by clearing the Count register.

5.2 Hardware choices

An early attempt on building a platform for TMU implementation, involved using the 8-bit AVR IP core, `avr_core`, from OpenCores [24] on a small FPGA development board, the Spartan-3 Starter Board (XC3S200), which was readily available at project start. The use of this board, and 8-bit MCU, turned out to be a dead end, because of three reasons. First, the `avr_core` has no debug unit, making debugging very hard. Second, the program memory had to be hardcoded into the bit-file, a slow approach compared to having a debug unit downloading a program to the device online. The last reason was that the FPGA was too small to fit a FreeRTOS application. A new board was ordered, the Spartan-3A DSP 1800A Development Board, which is an embedded HW/SW development board intended for soft-core use. The supplier of the board was later found to be unable to deliver on time and the Virtex-4 ML401 Evaluation Platform was obtained instead. As for the platform soft-core processor, the 32-bit LEON3 processor was opted. LEON3 can be set up with a debug unit enabling debugging and online downloading of applications. The LEON3 processor is a part of the GRLIB IP library and has many demo designs for FPGA boards, one of which is the Virtex-4 ML401 Evaluation Platform. Because this board was already supported, getting the first LEON3 design up and running went fairly quickly. This saved valuable project time, as the failed delivery of the Spartan-3A DSP 1800A board had set the project back. GRLIB, and the LEON3 processor, was well documented and enabled easy system configuration.

5.3 Software choices

The choice of software is basically the choice of RTOS for the target platform. The chosen RTOS, eCos, is a highly configurable system, and integrating the TMU into eCos' configuration system was an intuitive process. eCos is well documented and comes with examples making the development process rapid.

5.4 Design of the TMU

The TMU was designed as a peripheral device for the LEON3 soft-core processor. Being a peripheral device, the TMU is easily ported to other architectures with minimal effort, in contrast to a coprocessor or register file implementation. The negative side is that there is more overhead when the processor has to access registers on a system bus, compared to the register file, where access is instant. The TMU is designed by the VHDL "two-process" model, which makes its algorithm easy to understand. And its AMBA APB interface makes it easy to incorporate it into new LEON3 designs. The use of VHDL generics enables rapid configuration of the number of timers, the timer register widths and which interrupt line to use.

Chapter 6

Conclusion

The main objective in this thesis has been to develop a TMU capable of improving the predictability of real-time systems by measuring and controlling the execution times of tasks and prevent overloading of interrupts. The TMU is designed, simulated and implemented in an FPGA as a peripheral device connected to a LEON3 soft core processor. The system is running the eCos RTOS with a modified HAL for TMU support, and tests have been performed, verifying that the unit is working correctly. This suggests that the TMU can prevent real-time systems from being overloaded and miss deadlines, even if the underlying architecture is largely non-deterministic. The TMU uses little to moderate amounts of FPGA logic resources, depending on how many timers are being implemented. The Task Timer uses about 250 LUTs, and each IRQ Timer uses about 400 LUTs.

Earlier research has either been on how to make the hardware platform itself predictable, by making each operation deterministic, or on how to calculate the WCET of tasks or creating new programming languages features capable of easier WCET analysis. None have investigated the dynamic approach of measuring execution time as done here. A favourable aspect of using the proposed TMU is that it is architecture independent and can be implemented on any processor design by only modifying its bus interface. Another positive side is that existing code need not be modified unless the services of the TMU are needed.

Chapter 7

Further work

Although the proposed TMU is fully functional, there are some improvements worth mentioning:

- Add a VHDL generic mask specifying which timers to implement. This has the advantage of not having to implement more timers than absolutely necessary. With the current version of the TMU, having a task timer and a timer for IRQ level 10, the timers 0 to 10 have to be implemented. This wastes FPGA resources.
- Make the TMU registers 64-bit for "infinite" overflow periods. This is trivial for 64-bit systems. If the system is less than 64-bit, there will be at least twice as many memory accesses during context switches, but as the TMU processing overhead is very low, doubling the memory accesses is still affordable.
- Build the TMU IRQ Timers according to the Sporadic Server (SS) algorithm. The TMU IRQ Timers are now operating according to the Deferrable Server (DS) algorithm, which periodically replenishes the execution budget to full. If a budget is exhausted at the same time as a replenish occurs, the DS can use two consecutive budgets. Depending on the application, this might be unfortunate.
- Add a prescaler register, reducing the frequency of the TMU timer ticks. The TMU has a very fine granularity, running at the system clock frequency, and by adding a prescaler register, one may balance the timer granularity against its overflow period.
- Add a *soft limit* register for the Task Timer, giving the system the ability to be warned when a task's limit is approaching, and perhaps kill the task if it has not completed before the actual limit is reached.

Bibliography

[1] ARM Limited. *AMBA Specification*, 2.0 edition, 1999. Available from the Internet: `http://www.gaisler.com/doc/amba.pdf`.

[2] Peter J. Ashenden. *Digital Design - An Embedded Systems Approach Using VHDL*. Morgan Kaufmann, 2008.

[3] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages*. Pearson Education Limited, Essex, England, 3rd edition, 2001.

[4] Giorgio C. Buttazzo and Fabrizio Sensini. Optimal deadline assignment for scheduling soft aperiodic tasks in hard real-time environments. *IEEE Trans. Comput.*, 48(10):1035–1052, 1999.

[5] Joseph J. F. Cavanagh. *Verilog HDL - Digital Design and Modeling*. CRC Press, 2007.

[6] Matjaz Colnaric and Wolfgang A. Halang. Architectural support for predictability in hard real-time systems, 1993.

[7] Martin Delvai, Wolfgang Huber, Peter P. Puschner, and Andreas Steininger. Processor Support for Temporal Predictability - The SPEAR Design Example. In E [8], pages 169–176.

[8] *15th Euromicro Conference on Real-Time Systems (ECRTS 2003), 2-4 July 2003, Porto, Portugal, Proceedings*. IEEE Computer Society, 2003.

[9] eCos developer community. *eCos Reference Manual*, 2.0 edition. Available from the Internet: `http://ecos.sourceware.org/docs-2.0/pdf/ecos-2.0-ref-a4.pdf`.

[10] eCos developer community. *eCos User Guide*, 2.0 edition. Available from the Internet: `http://ecos.sourceware.org/docs-2.0/pdf/ecos-2.0-user-guide-a4.pdf`.

[11] Jiri Gaisler. A structured VHDL design method. Available from the Internet: `http://www.gaisler.com/doc/vhdl2proc.pdf`.

[12] Gaisler Research. *GRLIB IP Core User's Manual*, 1.0.17 edition, 2007. Available from the Internet: `http://www.gaisler.com/products/grlib/grip.pdf`.

[13] Gaisler Research. *GRLIB IP Library User's Manual*, 1.0.17 edition, 2007. Available from the Internet: `http://gaisler.com/products/grlib/grlib.pdf`.

[14] Brinda Ganesh. Architectural support for embedded operating systems. Master's thesis, University of Maryland, 2002.

[15] Michael Gonzalez Harbour, J. J. Gutierrez Garcia, and J. C. Palencia Gutierrez. Implementing application-level sporadic server schedulers in ada 95. In *Ada-Europe*, pages 125–136, 1997.

[16] John L. Hennessy and David A. Patterson. *Computer Architecture; A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2007.

[17] SPARC International Inc. FAQ. Web page: `http://www.sparc.org/aboutFAQ.html`, [cited 2008-05-21].

[18] Damir Isović and Gerhard Fohler. Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints.

[19] Krishnan K. Kailas and Ashok K. Agrawala. An Accurate Time-Management Unit for Real-Time Processors. Technical Report CS-TR-3768, University of Maryland, USA, 1997.

[20] C. M Krishna and Kang G. Shin. *Real-Time Systems*. McGraw-Hill, 1997.

[21] Jonathan Larmour. How eCos can be shrunk to fit. Embedded Systems, May 2005. Available from the Internet: `http://i.cmpnet.com/embedded/europe/esemay05/esemay05p32.pdf`.

[22] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[23] Anthony J. Massa. *Embedded Software Development with eCos*. Prentice Hall, 2002.

[24] Opencores.org. Home page: `http://www.opencores.org`.

[25] David A. Patterson. Reduced instruction set computers. *Commun. ACM*, 28(1):8–21, 1985.

[26] P. Puschner and Ch. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Syst.*, 1(2):159–176, 1989.

[27] K. Ramamritham and J. Stankovic. Scheduling algorithms and operating systems support for real-time systems, 1994.

[28] Christine Rochange and Pascal Sainrat. A time-predictable execution mode for superscalar pipelines with instruction prescheduling. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 307–314, New York, NY, USA, 2005. ACM.

[29] Soft processor comparison. Web page: http://ews.uiuc.edu/~pdabrows/soft_processor_comparison.html, [cited 2008-06-07].

[30] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computer*, 39(9):1175–1185, 1990.

[31] Lui Sha and John B. Goodenough. Real-time scheduling theory and ada. *Computer*, 23(4):53–62, 1990.

[32] SPARC International Inc., Menlo Park, CA. *The SPARC Architecture Manual Version 8*. Available from the Internet: www.sparc.org/standards/V8.pdf.

[33] Brinkley Sprunt. *Aperiodic Task Scheduling for Real-Time Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, August 1990. Available from the Internet: http://beru.univ-brest.fr/~singhoff/cheddar/publications/sprunt90.pdf.

[34] Brinkley Sprunt, Lui Sha, and John Lehoczky. Aperiodic task scheduling for hard-real-time systems. *The Journal of Real-Time Systems*, 1(1):27–60, June 1989.

[35] J. Stankovic and K. Ramamritham. What is predictability for real-time systems, 1990.

[36] John A. Stankovic. Real-time and embedded systems. Available from the Internet: http://www-ccs.cs.umass.edu/sdcr/rt.ps.

[37] John A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *IEEE Computer Society Press Los Alamitos, CA, USA*, 21(10), October 1988.

[38] Jay K. Strosnider, John P. Lehoczky, and Lui Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Trans. Comput.*, 44(1):73–91, 1995.

[39] Lonnie Vanzandt. Scheduling Sporadic Events. *Embedded Systems Design*, 11 2002. Available from the Internet: http://www.embedded.com/9900812, [cited 2008-05-21].

[40] Peter R. Wilson. *Design Recipes for FPGAs*. Newnes, 2007.

[41] Xilinx. *Spartan-3 FPGA Family: Complete Data Sheet*, 2007. Available from the Internet: http://www.xilinx.com/support/documentation/data_sheets/ds099.pdf.

[42] Xilinx. *Virtex-4 Family Overview*, 2007. Available from the Internet: http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf.

[43] Xilinx. *Virtex-4 Family Overview*, 2008. Available from the Internet: `http://www.xilinx.com/bvdocs/userguides/ug070.pdf`.

[44] Mark Zwoliński. *Digital System Design with VHDL*. Pearson Education Limited, Essex, England, 2nd edition, 2004.

Appendix A

CD

A.1 Contents

This report is accompanied by a CD containing the following folders:

- **code** contains the source code written throughout this project. This includes:
  - The synthesisable TMU model written in VHDL.
    `[code/apbtmu.vhd]`
  - Test programs for eCos and the TMU written in the C Programming Language.
    `[code/ecos/*.*]`
  - The complete modified version of the eCos SPARC HAL for use with the TMU.
    `[code/ecos/repository-packages-hal-sparc/]`
  - Two patches for eCos which add a return value to the exception handler prototypes and default exception handlers, and one patch for GRLIB which fixes the a GHDL simulation problem and defines constants for TMU identification in the VHDL testbench. The order in which the patches are applied does not matter.
    `[code/patch/*.*]`
  - The TMU demo design. Must be built against GRLIB with the above GRLIB patch applied.
    `[code/leon3-avnet-ml401/]`
- **report** contains the LaTeX source files of this report, including images, and the report itself as a Portable Document File (PDF).

A.2 CD

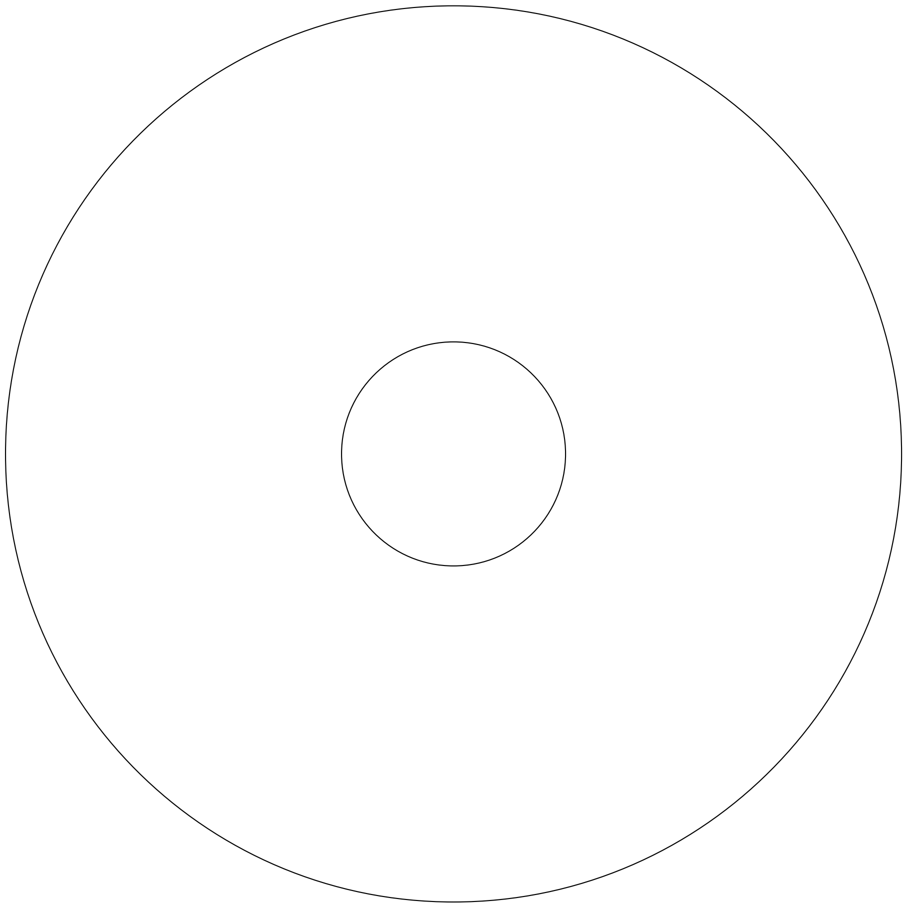


Figure A.1: CD

Appendix B

Source code

B.1 TMU

```
--------------------------------------------------------------------------------
-- Entity:      apbtmu
-- File:        apbtmu.vhd
-- Author:      Bjorn Forsman
-- Description: TMU with AMBA APB interface (Leon3 peripheral device)
--
-- Revision list
-- Date         By      Changes
-- 2008-04-04   BF      initial design
-- 2008-04-06   BF      added interrupt handling
--
--
--
--
--
--
-- ghdl -a --ieee=synopsys --workdir=gnu/work  --work=work -Pgnu -Pgnu/opencores -Pgnu
    /gaisler -Pgnu/work apbtmu.vhd
-- ghdl -e --ieee=synopsys --workdir=gnu/work  --work=work -Pgnu/grlib apbtmu
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
library grlib;
use grlib.amba.all;                     -- apb_slv_{in,out}_type

package tmupkg is

  constant MAXTIMERS : integer := 32;
  constant TSELBITS  : integer := 5;    -- log2(MAXTIMERS)

  type tmu_in_type is
    record
      dhalt : std_ulogic;               -- halt timers in debug mode
    end record;

```

```
  type tmu_out_type is
    record
      apbi       : apb_slv_in_type;          -- to irq controller
      morebudget : std_logic_vector(MAXTIMERS-1 downto 1); -- only for debug
    end record;

  component apbtmu
    generic (
      pindex   : integer := 0;           -- APB device select signal
      paddr    : integer := 0;           -- 12 MSb of APB address
      pmask    : integer := 16#fff#;     -- APB address mask
      pirq     : integer := 0;           -- which APB irq to generate
      ntimers  : integer range 1 to MAXTIMERS := 16;
      tbits    : integer range 1 to 32        := 32;  -- timer bits
      );
    port (
      rst      : in  std_logic;
      clk      : in  std_logic;
      apbi     : in  apb_slv_in_type;
      apbo     : out apb_slv_out_type;
      tmui     : in  tmu_in_type;
      tmuo     : out tmu_out_type
      );
  end component;

end tmupkg;



library ieee;
use ieee.std_logic_1164.all;
library grlib;
use grlib.amba.all;                     -- apb_slv_{in,out}_type
use grlib.stdlib.all;                   -- report_version and "+" fcn. for std_logic_vector!
use grlib.devices.all;                  -- lib/grlib/amba/devices.vhd
                                        -- (configuration constants)
--pragma translate_off
use std.textio.all;
--pragma translate_on
use work.tmupkg.all;

entity apbtmu is
  generic (
    pindex  : integer := 0;              -- APB device select signal
    paddr   : integer := 0;              -- 12-bit MSB APB address
    pmask   : integer := 16#fff#;        -- APB address mask
    pirq    : integer := 0;              -- which APB interrupt to generate
    ntimers : integer range 1 to MAXTIMERS := 16;
    tbits   : integer range 1 to 32        := 32;  -- timer bits
    );
  port (
    rst     : in  std_logic;
    clk     : in  std_logic;
    apbi    : in  apb_slv_in_type;
    apbo    : out apb_slv_out_type;
    tmui    : in  tmu_in_type;
```

```
    tmuo    : out tmu_out_type
    );
end;

architecture rtl of apbtmu is

  constant REVISION : integer := 0;

  constant pconfig : apb_config_type := (
    0 => ahb_device_reg(VENDOR_NTNU, NTNU_TMU, 0, REVISION, pirq),
    1 => apb_iobar(paddr, pmask));

  type timer_reg is
    record
      irq    : std_ulogic;                          -- interrupt pulse
      irqen  : std_ulogic;                          -- interrupt enable
      irqpen : std_ulogic;                          -- interrupt pending
      ticks  : std_logic_vector(tbits-1 downto 0);  -- always increment
      count  : std_logic_vector(tbits-1 downto 0);  -- inc when tmr selected
      limit  : std_logic_vector(tbits-1 downto 0);  -- interrupt when cnt>lim
      period : std_logic_vector(tbits-1 downto 0);  -- replenish period
    end record;

  type timer_reg_vector is array (natural range <> ) of timer_reg;

  type registers is
    record
      enable : std_ulogic;                          -- enable TMU
      tsel   : integer range 0 to ntimers-1;        -- timer select
      timers : timer_reg_vector(0 to ntimers-1);
    end record;

  signal r, rin : registers;

begin

  comb : process(rst, r, apbi, tmui)
    variable v          : registers;
    variable readdata   : std_logic_vector(31 downto 0);
    variable tmpirq     : std_logic_vector(NAHBIRQ-1 downto 0);
    variable morebudget : std_logic_vector(ntimers-1 downto 1);
    variable toirqctrl  : apb_slv_in_type; -- tmp var for irq controller
  begin

    v := r;                             -- copy regs to working var.

    -- increment count for the selected timer (when not in debug mode)
    if tmui.dhalt = '0' then    -- halt timers in debug mode
      if v.enable = '1' then
        v.timers(r.tsel).count := v.timers(r.tsel).count + 1;
      end if;

      -- deferred server model: replenish budget every period
      -- increment ticks, reset count and replenish budgets for irq tmrs
      for i in 1 to ntimers-1 loop
        v.timers(i).ticks   := v.timers(i).ticks + 1;
        if v.timers(i).ticks >= v.timers(i).period then
```

```
          v.timers(i).count := (others => '0');
          v.timers(i).ticks := (others => '0');
        end if;
      end loop;

    end if; -- if tmui.dhalt


    -- internal interrupt signal generation
    for i in 0 to ntimers-1 loop
      if (v.timers(i).count = v.timers(i).limit) then
        if (v.timers(i).irqen = '1') then
          v.timers(i).irq    := '1';
          v.timers(i).irqpen := '1';    -- must be reset by software
        end if;
      else
        v.timers(i).irq      := '0';
      end if;
    end loop;

    -- external interrupt lines
    toirqctrl := apbi;                -- default assignment
    for i in 1 to ntimers-1 loop
      if v.timers(i).count < v.timers(i).limit then
        morebudget(i) := '1';
      else
        morebudget(i) := '0';
      end if;

      --into(i) <= inti(i) and not
      toirqctrl.pirq(i) := apbi.pirq(i) and not
                  (not morebudget(i) and v.timers(i).irqen and v.enable);
    end loop;

    -- Task TMU interrupts overrides
    toirqctrl.pirq(pirq) := (v.timers(0).irq and v.timers(0).irqen) and v.enable;


    -- read registers
    readdata := (others => '0');
    case apbi.paddr(8 downto 2) is
      when "0000000" =>                 -- control reg
        readdata(0)               := v.enable;
        readdata(5 downto 1)      := conv_std_logic_vector(ntimers, TSELBITS);
        readdata(11 downto 6)     := conv_std_logic_vector(tbits, 6);
        readdata(12)              := tmui.dhalt;
      when "0000001" =>                 -- timer select reg
        readdata(TSELBITS-1 downto 0) := conv_std_logic_vector(v.tsel, TSELBITS);
      when "0000010" =>                 -- irqpen
        for i in 0 to ntimers-1 loop
          readdata(i) :=  v.timers(i).irqpen;
        end loop;
      when others =>                    -- timer registers
        for i in 0 to ntimers-1 loop
          if conv_integer(apbi.paddr(8 downto 4))-1 = i then
            case apbi.paddr(3 downto 2) is
              when "00"   =>            -- count reg
```

```
            readdata(tbits-1 downto 0) := r.timers(i).count;
          when "01"   =>            -- limit reg
            readdata(tbits-1 downto 0) := r.timers(i).limit;
          when "10"   =>            -- period reg
            readdata(tbits-1 downto 0) := r.timers(i).period;
          when "11"   =>            -- timer n control reg
            readdata(0)                := r.timers(i).irqen;
            readdata(1)                := r.timers(i).irqpen;
          when others =>
        end case;
      end if;
    end loop;
end case;


-- write registers
if (apbi.psel(pindex) and apbi.penable and apbi.pwrite) = '1' then
  case apbi.paddr(8 downto 2) is
    when "0000000" =>              -- control reg
      v.enable            := apbi.pwdata(0);
    when "0000001" =>              -- timer select reg
      v.tsel              := conv_integer(apbi.pwdata(TSELBITS-1 downto 0));
    when "0000010" =>              -- irqpen reg
      for i in 0 to ntimers-1 loop
        v.timers(i).irqpen := apbi.pwdata(i);
      end loop;
    when others =>
      for i in 0 to ntimers-1 loop
        if conv_integer(apbi.paddr(8 downto 4))-1 = i then
          case apbi.paddr(3 downto 2) is
            when "00"   =>          -- count reg
              v.timers(i).count  := apbi.pwdata(tbits-1 downto 0);
            when "01"   =>          -- limit reg
              v.timers(i).limit  := apbi.pwdata(tbits-1 downto 0);
            when "10"   =>          -- period reg
              v.timers(i).period := apbi.pwdata(tbits-1 downto 0);
            when "11"   =>          -- timer n control reg
              v.timers(i).irqen  := apbi.pwdata(0);
              v.timers(i).irqpen := apbi.pwdata(1);
            when others =>
          end case;
        end if;
      end loop;
  end case;
end if;


-- reset
if rst = '0' then
  for i in 0 to ntimers-1 loop
    v.timers(i).irq    := '0';
    v.timers(i).irqen  := '0';
    v.timers(i).irqpen := '0';
    v.timers(i).ticks  := (others => '0');
    v.timers(i).count  := (others => '0');
    v.timers(i).limit  := (others => '1');
    v.timers(i).period := (others => '1');
```

```
      end loop;
      morebudget           := (others => '1'); -- irq timers
      v.enable             := '0';       -- disable counting
      v.tsel               := 0;         -- start with the task timer selected
    end if;


    -- outputs
    rin         <= v;                    -- copy working variable to signal
    apbo.prdata <= readdata;             -- drive apb read bus
    tmuo.apbi   <= toirqctrl;
    tmuo.morebudget(ntimers-1 downto 1) <= morebudget;
  end process;


  -- concurrent assignments
  apbo.pindex  <= pindex;
  apbo.pconfig <= pconfig;


  -- registers
  regs : process(clk)
  begin
    if rising_edge(clk) then
      r <= rin;
    end if;
  end process;


-- boot message

-- pragma translate_off
  bootmsg          : report_version
    generic map ("apbtmu" & tost(pindex) &
                 ": Time Management Unit rev " & tost(REVISION) &
                 ", with " & tost(ntimers) &
                 " " & tost(tbits) & "-bit timers" & ", irq " & tost(pirq));
-- pragma translate_on

end;
```