# MSc Wireless Embedded Systems, Newcastle University 2014

# EEE8068 Coursework Report

Implementation of a TTCS on an ARM development board

### Michael Walker (student no. 130623935)

# Abstract

A Time Triggered Cooperative Scheduler (TTCS) was implemented in C on an ARM microcontroller that coordinated 8 simple but simultaneous PWM tasks with overrun protection, user-mode restrictions with system calls & Phase correction. Understanding of the concept of real-time scheduling was gained through the practical experience of implementing this system.

# Introduction

## Scheduler

The scheduler that was to be required by the coursework specification is a Time Triggered Cooperative Scheduler (TTCS). This means that a periodic clock is used to generate an interrupt that (re)starts the processing of tasks by a dispatcher routine. Cooperative Scheduling means that there is no mechanism employed to pre-empt tasks making the schedule design comparatively simple. Overruns are handled implicitly by the interrupt repopulating the task queue every 1ms.

## Hardware & Toolchain

### Hardware

The project was implemented on an ARM-XILINX Laboratory Board developed by the University of Manchester. The XILINX Spartan 3 FPGA chip is left unused, but the Atmel AT91SAM9261 SoC was principal hardware component of the project.

The AT91SAM9261 chip contains an ARM926EJ-S 32-bit RISC CPU Core running at 190 MHz. The ARM9E family use the ARMv5 Architecture; a 5-stage pipelined, Little-Endian, RISC design incorporating DSP-specific extensions and an integrated MMU. It is targeted at multi-tasking applications[[1](#_ENREF_1)].

Scheduling activity was facilitated through a periodic Timed Interrupt at a rate of 1 kHz. Brief investigations suggested that a 32,768 MHz Crystal Oscillator on the board would be involved in generating this. Programming is performed through a standard DE-9 Serial Interface to a PC running the requisite software to handle the likely JTAG interface.

### GNU Cross-Compilation Development Toolchain

The programming environment was broadly UNIX command-line shell based, thus requiring the use of Cygwin on the Windows Computer in the Lab. C is the programming language used to describe the scheduler design. The C compiler used is the GNU C Compiler (*gcc*) set for cross-compilation targeting the ARM platform that together with the GNU linker (*ln*) and assembler (*as*) generates an ARM compatible *ELF* (Executable and Linkable Format) image file suitable for loading to the board, the initial project files include a *Makefile* for easy compilation and linking, together with some assembler initialization files. Most of the later were left untouched. The *make* process is explained in the diagram on the next page.

Once new versions of the scheduler are written testing was carried out using the Komodo Manchester Debugger. This controlled the loading of executables to the board and debugging tasks. Low-level debugging facilities are provided which allow all Memory contents, CPU registers & CPU status flags to be examined (with disassembly of the machine code) and modified. Program flow can be suspended, stepped-through, and resumed both manually and through the use of breakpoints and conditional watchpoints. Further debugging was facilitated by the GNU object dumper (*objdump*) disassembling the executable code generated into assembler useful for programmer inspection.

Dump File

(*ex.dmp*)

Executable File

(*ex*)

Object Dumper (objdump)

assembler (as)

assembler (as)

compiler (gcc)

Assembler File

(ex.s)

Assembler Helper File

(startup.s)

Header file

(ex.h)

Main C Code File

(ex.c)

Linker (ld)

Object File

(ex.o)

Object File

(startup.0)

Linker Options File

(*armfpga.ld*)

# Figure 1: The Make Process

Notes:

**gcc** is used with option –S to not invoke the assembler and output assembler code (.s) file

**ld** is used with –L options specifying the arm-elf cross-compiler ‘libgcc.a & ‘libc.a’ libraries

**objdump** is run with –d option to disassemble the executable file into assembler code.

# Implementation

## Initial Simple Scheduler

The development of the project can be broadly split into two phases. The first involved getting the most basic functionality working. This involved the control of only a single task (one LED) to ensure familiarity with the project platform. It would be helpful to explain the Implementation of this phase first to explain many of the concepts used in the final version without getting lost in the extra functionality added later.

### Naked Functions

All functions in the scheduler are implemented as ‘naked functions’. Generally C functions implicitly include prologue and epilogue code to allow for the previous state of registers and stack to be saved prior, and restored after the function proper is run. This allows for function calls and nesting, but is an unnecessary overhead in designing this scheduler. In fact it would cause stack overflow.

### Interrupts

There are two types of interrupt used in the initial system. Both interrupts must be handled by a C function called from a vector table setup by the *startup.s* file. The first is the hardware interrupt that is raised from a clocked source outside the system every 1 millisecond. The function - called *irq()* - is used to reset the *doTask* variable, clear the *irq* flag in the status register and start the dispatcher in the simple setup. The second interrupt is the software interrupt (**swi**), this is handled in the code through the *swi()* function. In the simple case it is called by the task when completed and calls the dispatcher in turn.

### Dispatcher, Task & Sleep

The Dispatcher started as a simple naked function that checked if the ‘doTask’ variable was set, if it was it unset it and started the task() function, if not it called the sleep() function which contained inline assembler code that performed an endless branch loop to do nothing until the next *irq().*

The task function processed just one LED directly in the simple version. Using global ‘counter’, ‘level’ & ‘updown’ variables to keep track of what the LED state should be. The counter was incremented on every task but kept modulo 20 to keep track of the 20ms Pulse Width Modulation (PWM) period. (20ms is required for human persistence of vision). Upon each return of counter to 0, level was in turn incremented up to 20 then decremented back to 0 using the updown state variable to control the direction. The level was used to set the duty cycle of the PWM as the led was only set on when the counter was less than the value, hence 100% Duty cycle at level = 0, 50% at level=10 and 0% at level=20.

counter**++**;

**if** (counter **==** modulus) {

counter **=** 0;

*// If at boundaries of pwm cycle, need to switch direction*

*// nb. in updown variable '1' means upwards, '0' means downwards*

**if** (level **==** modulus) updown **=** 0;

**if** (level **==** 0) updown **=** 1;

*// change to new level*

level **=** updown **==** 1 **?** **++**level **:** **--**level;

}

Leds **=** counter **<** level **?** 0xFF **:** 0;

Figure 2: Task code determining LED switching in intial scheduler implementation

## Full Scheduler Development

Task Queue Structure

Modes USER/SUPERVISOR – Operating System Kernal User Mode Seperation – Memory Protection

It implicitly switches the system into supervisor mode after running tasks in user mode to protect the memory and memory mapped hardware (such as the leds) by acting as a hardware driver to

High Priority Interupts – Button – Overrun Testing

Phase Correction

Finished Code Explanation and walkthrough

Main() – Initialization

Irq() – Update

Dispatcher() – Dispatcher

Task() – Tasks (User Mode not OS)

swi() – hardware drivers

phase\_correction() – Timing reference data and phase recovery

*Hardware driver*

*Toggle overrun test variable*

*Initialise system*

START

*Overrun if test variable set*

# Figure 3: Scheduler Flowchart Diagram

**Notes:**

* Solid lines indicate function calls, dotted lines indicate interrupts.
* The Button press initiates a **fiq** interrupt which means dotted lines should run from every function to the fiq() function, but these are omitted for clarity.
* The two decisions markers are part of the **irq()** and **dispatch()** functions but are extracted out to explain scheduler flow more completely.

*Overrun on*

*Timer*

*Interrupt*

*Timer interrupt*

BUTTON PRESS

fiq()

No

correct\_phase()

Yes

DETECT

OVERRUN?

task()

task()

task()

Yes

No

STILL

TASKS?

swi()

dispatch()

irq()

sleep()

main()

# Testing & Results

Confirmation of Timing correctness was performed by recording video of the LED output over a duration of 1 minute. The video was analysed repeated to tally the number of flashes performed by each of the LEDs.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| LED Colour | **LED Side** | **Reg. Position** | **Count** | **Duration Measured** | **Duration Specified** |
| Red | Left | 2 | 60 | **1.00** | 1.0 |
| Yellow | Left | 1 | 53 | **1.13** | 1.1 |
| Green | Left | 0 | 50 | **1.20** | 1.2 |
| Blue | Left | 3 | 45 | **1.33** | 1.3 |
| Red | Right | 6 | 43 | **1.40** | 1.4 |
| Yellow | Right | 5 | 40 | **1.50** | 1.5 |
| Green | Right | 4 | 38 | **1.58** | 1.6 |
| Blue | Right | 7 | 35 | **1.71** | 1.7 |

Table 1: Tally of timing analysis

Figure 4: Flash Duration Measured Vs Specified shows close match

Results of overrun testing and overrun protection provision – explain additional code used for intentionally causing overruns, fiq and button – note advanced toggle behaviour!

Results of Phase correction testing… Still ongoing development – write last

# Conclusion

‘It worked’ – expand on this….

What Did I learn – how can this knowledge be applied to the real-world etc.

Further experiments and alterations that would be interesting to undertake

# References

[[2-6](#_ENREF_2)]

[1] Atmel, “AT91SAM ARM-based Embedded MPU,” 2011.

[2] J. Blazewicz, *Scheduling Computer and Manufacturing Processes*: Springer, 2001.

[3] G. C. Buttazzo, *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications*: Springer, 2005.

[4] B. W. Kernighan, and D. M. Ritchie, *The C Programming Language*: Prentice Hall, 1988.

[5] C. Shore, “The ARM Architecture,” 2003.

[6] UNSW, “An Introduction to Komodo.”

# Appendix

## Initial Simple Scheduler Source Code:

**#include "ex.h"**

**#define NUM\_OF\_TASKS 1**

**#define CHAR\_MASK 0xFFFFFF00**

**int** Leds **=** 0;

**int** doTask **=** 1;

**int** modulus **=** 20;

**int** counter **=** 0;

**int** level **=** 10;

**int** updown **=** 1;

*// a structure that contains the information passed to the task function*

**struct** task\_inst {

*// Function pointer (why?)*

**void** (**\***func) ();

*// Task Parameters*

**int** run\_phase;

**int** n;

**int** pwm\_period;

**int** flash\_half\_period;

*// Task Variables*

**int** flash\_count;

**int** pwn\_count;

**int** brightness;

**int** updown;

};

**void** **\_\_attribute\_\_** ((**naked**)) task (**struct** task\_inst **\***d) {

d.n **=** 0;

counter**++**;

**if** (counter **==** modulus) {

counter **=** 0;

*// If at boundaries of pwm cycle, need to switch direction*

*// nb. in updown variable '1' means upwards, '0' means downwards*

**if** (level **==** modulus) updown **=** 0;

**if** (level **==** 0) updown **=** 1;

*// change to new level*

level **=** updown **==** 1 **?** **++**level **:** **--**level;

}

Leds **=** counter **<** level **?** 0xFF **:** 0;

asm("swi 0"); *// Perform SWI - If task overruns, will perform IRQ instead...*

}

**struct** task\_inst task\_list[NUM\_OF\_TASKS] **=** {

{task, 3, 0, 20, 500, 0, 0, 0, 1}

};

**void** **\_\_attribute\_\_** ((**naked**)) sleep() {

asm("sleepmode: b sleepmode"); *// ASM to prevent compiler eliminating loops*

}

**void** **\_\_attribute\_\_** ((**naked**)) dispatch() {

**if** (doTask **!=** 0) {

doTask **=** 0; *// Remove task from queue*

USR\_MODE; *// Goto User Mode*

task(**&**task\_list[0]); *// Run the Task*

} **else** {

sleep(); *// Tasks done - Wait for next irq*

}

}

**void** **\_\_attribute\_\_** ((**naked**)) irq() {

**\***INT\_RAW\_P **&=** **~**INT\_MASK\_TIMER; *// Reset IRQ request*

**\***TIMER\_COMPARE\_P **=** **\***TIMER\_P **+** 1; *// Setup next interupt*

doTask **=** 1; *// TODO: Update Task Queue State Machine thingy with new tasks*

dispatch(); *// Call Dispatcher*

}

**void** **\_\_attribute\_\_** ((**naked**)) swi() {

IRQ\_ENABLE; *// IRQ enable required in SWI*

**\***LEDS\_P **=** Leds; *// Set Physical LEDs based on user mode modified variable*

dispatch(); *// Call Dispatcher*

}

**int** **main** () {

*// Initial state - invert current LED State - debugging sanity checker*

**\***LEDS\_P **=** **~\***LEDS\_P **^** CHAR\_MASK;

*// Initialise timer for sometime off, while setting things up*

**\***TIMER\_COMPARE\_P **=** **\***TIMER\_P **+** 128;

*// IRQ input unmask*

**\***INT\_ENABLE\_P **|=** INT\_MASK\_TIMER;

*// reset IRQ request*

**\***INT\_RAW\_P **&=** **~**INT\_MASK\_TIMER;

IRQ\_ENABLE; *// IRQ enable required in main*

*// sleep until first interrupt*

sleep();

}