# Real-time processing with the Philips LPC ARM microcontroller; using GCC and the MicroC/OS-II RTOS.

Philips 05: Project Number AR1803
D. W. Hawkins (dwh@ovro.caltech.edu)

May 10, 2006

## Contents

# 1 Introduction

The ARM processor is a reduced instruction set computer (RISC) intellectual property (IP) core defined by Advanced RISC Machines, Ltd (ARM). The ARM CPU architectures widely available today are based on the version 4 and 5 architectures [12]. ARM processor cores are used by Intel (StrongARM and XScale processors), Sharp, Atmel, Philips, Analog Devices, and many other semiconductor manufacturers.

The ARM processor can operate with two instruction sets; ARM mode, and THUMB mode. The ARM mode uses a 32-bit instruction set, while THUMB mode uses a 16-bit instruction set. The use of THUMB mode reduces the execution speed of the code, but reduces the memory requirements of the code, so finds use in the microcontroller applications of the processor core.

$\mu$COS-II is a real-time operating system (RTOS) written by Jean Labrosse and supported by his company Micrium. The RTOS is well described in his book [6]. The RTOS defines a standard set of operating system (OS) primitives and the book defines how to port the RTOS to different processor architectures. This document describes a port for the ARM processor operating in 32-bit mode for the GNU GCC compiler.

The following references provide additional resources on ARM processors and $\mu$COS-II RTOS:

- "ARM system-on-chip architecture", S. Furber [5].
- "ARM Architecture Reference Manual", D. Seal [12]. Chapters A1 and A2 provide an overview of the ARM architecture and programming model.
- "ARM System Developer's Guide", A. Sloss *et al* [13]
- "MicroC/OS-II: The real-time kernel", J. Labrosse [6].

**Author's Note:** May 10, 2006.

This document and the associated code were submitted to the Circuit Cellar Philips ARM 2005 contest. The project was selected for a Distinctive Excellence award. At some point Circuit Cellar are going to put the project files up on their web site.

Prior to the ARM 2005 contest I'd never used the ARM processor. My initial objective was to understand the code generated and required by GCC to link microcontroller applications, and then use that knowledge to port the uCOS-II RTOS to the processor. I'd played with the Atmel AVR and WinAVR for the Circuit Cellar Atmel AVR 2004 contest, but had simply used WinAVR, not appreciating the task done by the startup files and the AVR standard library. Many of the examples in this project are stand-alone, in that the code provides the start-up routines and the application code (some of the code in subfolders is repeated for the sake of simplification).

Please excuse the poor makefiles and anything else you find over-simplified, I was just playing and didn't really anticipate too many people looking at the code. However, it seems alot of the questions asked on the LPC2000 news group could be answered by this document, so feel free to provide feedback, or modified code, and I'll update the original source and re-release the code as it is updated. I plan to go though and add more sections, and get newlib-lpc up-and-running, but for now, this will have to do.

Feel free to post comments to the LPC2000 news group, I read it.

Cheers,
Dave Hawkins, Caltech.
dwh@ovro.caltech.edu.

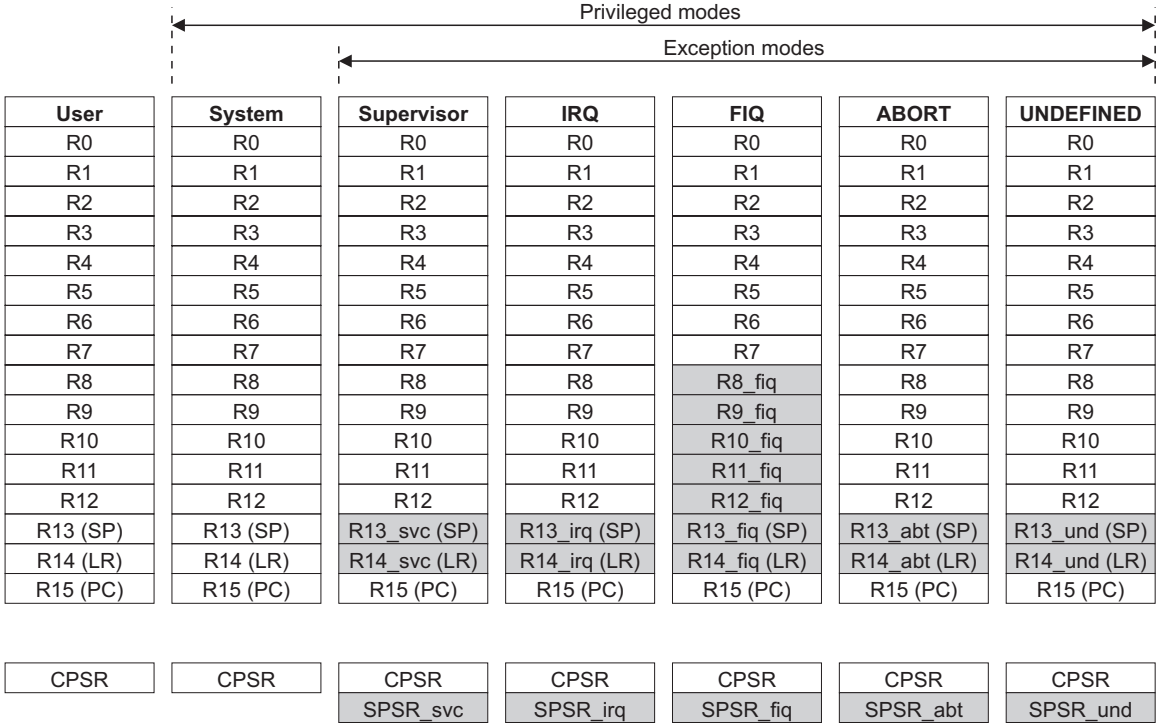



Figure 1: ARM programming modes. In ARM-mode the processor can switch between seven operating modes. The processor has a set of banked registers, i.e., the actual register an instruction accesses is dependent on the operating mode. The greyed registers in the figure show the physically different registers in each operating mode.

# 2 Programmers Model

Figure 1 shows the ARM programming model (Chapter A2 [12], p39 [5], p7 [7]), and the seven ARM operating modes. A general purpose operating system such as Linux uses the *User* mode of the processor for user-space processes, and the *Supervisor* mode for the operating system kernel. For a real-time OS, such as $\mu$COS-II, the kernel and application tasks run in *Supervisor* mode. Exception modes need to be dealt with appropriately in either a general purpose OS (by kernel routines) or in an RTOS. The seven processor modes are (pA2-3 [12], pA2-11 [12] has the 5-bit values for each mode);

| Mode | Description |
|---|---|
| User | Normal program execution code |
| System | Runs privileged operating system tasks |
| Supervisor | A protected mode for the operating system |
| IRQ | General-purpose interrupt handling |
| FIQ | Fast-interrupt handling |
| Abort | Used to implement virtual memory or memory protection |
| Undefined | Supports software emulation of coprocessors |

In any of the seven operating modes shown in Figure 1, code has access to 16 general-purpose registers, R0 through R15, and a current program status register (CPSR). In exception modes there is an additional register, called the saved program status register (SPSR), which has identical bits to the CPSR. The processor has a set of banked registers, where dependent on the operating mode

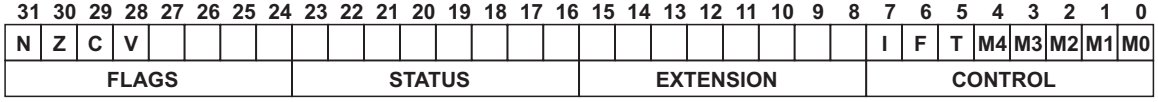



Figure 2: Control and program status register (CPSR) bits. The defined bits are the flags; negative, zero, carry, and overflow, and the control bits; IRQ disable, FIQ disable, ARM/THUMB instruction mode, and the 5-bit processor operating mode (where the modes are shown in Figure 1).

the physical register accessed can be different. For example in fast interrupt mode (FIQ) registers R8 through R14 are unique for that mode so do not need saving through interrupt context switches. Register R13 is conventionally used as the *Stack Pointer* (pA2-6 [12]), while registers R14 and R15 have special roles as the *Link Register* (return address), and *Program Counter* (pA1-3 [12]). The ARM procedure calling standard (APCS) defines the recommended use of the other registers for passing arguments and return variables.

**Stack pointer**
The stack grows from high-to-low.

**Link register**
The link register holds the address of the next instruction after a Branch and Link (BL) instruction which is the instruction used to make a subroutine call. At all other times, R14 can be used as a general-purpose register (pA1-3 [12]). To return from a subroutine call, the link register is copied into the program counter register (pA1-4 [12]). If nested of interrupts is used, special care of the link register contents is required (pA2-6 [12]).

**Program counter**
When an instruction reads the program counter, the value read is the address of the instruction plus 8 (4 bytes if operating in THUMB mode). The program counter is 32-bit aligned (bits 1 and 0 are always zero) in ARM mode, and 16-bit aligned in THUMB mode (bit 0 is zero) (pA1-3 [12]).

**Status registers**
The CPSR (and SPSR) contains four sections; flags, status, extension, and control. These sections and bits are shown in Figure 2. There are specific instructions for transferring the status registers to and from the general purpose registers.

# 3 ARM GCC

An embedded systems C-coded application consists of assembly-coded startup code containing processor and run-time environment (eg. stacks) initialization, the C-coded application, statically-linked library code (newlib and user libraries), and a linker script defining the device memory map and code load and run addresses. The GNU Compiler Collection (GCC) for the ARM processor can be downloaded from `www.gnuarm.com`, or can be built from source as described in Appendix A. The following sections walk-through increasingly complex examples to demonstrate the GCC tools.

The examples in the following sections are developed for a Keil MCB2130 development board containing a Philips LPC2138 ARM microcontroller. The Keil MC2130 evaluation board contains a set of LEDs connected to pins `P1.[16..23]`. The examples use the LEDs to provide visual feedback that the example program operates correctly.

## 3.1 Example 1: Basic startup assembler

```
/* ex1.s */

/* ------------------------------------------------------------------------
 * Exception vectors
 * ------------------------------------------------------------------------
 */
    .text
    .arm
    .global _start
_start:
    /* Vectors (8 total) */
    b reset     /* reset */
    b loop      /* undefined instruction */
    b loop      /* software interrupt */
    b loop      /* prefetch abort */
    b loop      /* data abort */
    nop         /* reserved for the bootloader checksum */
    b loop      /* IRQ */
    b loop      /* FIQ */

/* ------------------------------------------------------------------------
 * Test code
 * ------------------------------------------------------------------------
 */
reset:
    ldr r0, IODIR1
    ldr r1, IODIR1_VALUE
    str r1, [r0]
    ldr r0, IOCLR1
    str r1, [r0]
    ldr r0, IOSET1
    ldr r1, IOSET1_VALUE
    str r1, [r0]

loop:   b   loop
```

```
/* ------------------------------------------------------------------------
 * Constants
 * ------------------------------------------------------------------------
 */
/* LED control registers */
IOSET1:             .word   0xE0028014
IODIR1:             .word   0xE0028018
IOCLR1:             .word   0xE002801C
IODIR1_VALUE:       .word   0x00FF0000
IOSET1_VALUE:       .word   0x00550000

    .end
```

Example 1 is a short assembler program that sets up basic exception vectors (they drop into an infinite loop), and then turns the MCB2130 even LEDs on (where indexing is `LED[7:0]`) and the odd LEDs off (a high value on an I/O pin turns an LED on). The example does not setup stacks (since we do not need them). The successful compilation and download of this example checks the compiler and linker setup.

A linker script is used to define the memory map of a processor. If GCC is not passed a linker script as part of the command line arguments, a default linker script is used. The linker script can be displayed by passing `-Wl,--verbose` to `arm-elf-gcc` or `--verbose` to `arm-elf-ld`. The default linker script defines an entry point symbol called `_start` and loads to a default address of 0x8000. The example assembler program uses the expected entry symbol `_start`, and since the Philips LPC microcontroller reset vector is address 0, a command line option, `-Ttext=0`, is needed to link the `.text` section to address 0 (note that this just overrides the linker script `.text` section start address, it does not replace the linker script). The example needs to setup the exception vector table correctly, as the Philips LPC microcontroller expects the reserved exception vector (the old 26-bit ARM processor address exception) to contain a checksum word. The Philips FlashUtils downloader fills in this checksum.

The example is compiled using

```
arm-elf-gcc -mcpu=arm7tdmi -c ex1.s
```

and is linked using

```
arm-elf-ld -Ttext=0 ex1.o -o ex1.elf
```

The linker `-T` option links the code starting at physical address 0.

The compilation and link step can be combined using

```
arm-elf-gcc -mcpu=arm7tdmi -nostartfiles -Ttext=0 ex1.s -o ex1.elf
```

The `-nostartfiles` option tells the compiler not to use its startup assembler routine.

The executable can be disassembled using

```
arm-elf-objdump -d ex1.elf
```

which simply shows the original source. The disassembled code covers addresses 0 through 54h (inclusive), i.e., 58h bytes = 88 bytes. The amount of Flash and SRAM needed by a program can be summarized using

```
arm-elf-size ex1.elf
text    data     bss     dec     hex filename
  88       0       0      88      58 ex1.elf
```

A key point to note about this example is that there is only a text section, i.e., the example will only use Flash RAM.

The FlashUtils downloader expects an Intel hex format file. This file is created using

```
arm-elf-objcopy -O ihex ex1.elf ex1.hex
```

Start `LPC210x_ISP.exe` and download the program; the even LEDs should be on, the odd off (`LED[0]` is closest to the corner of the board).

## 3.2 Example 2: A simple C program

The following startup routine sets up exception vectors, with the reset vector jumping directly to the C coded main application

```
/* ex2_start.s */
    .text
    .arm
    .global main
    .global _start
_start:
    /* Vectors (8 total) */
    ldr pc, main_addr    /* reset */
    ldr pc, loop_addr    /* undefined instruction */
    ldr pc, loop_addr    /* software interrupt */
    ldr pc, loop_addr    /* prefetch abort */
    ldr pc, loop_addr    /* data abort */
    nop                  /* reserved for the bootldr checksum */
    ldr pc, loop_addr    /* IRQ */
    ldr pc, loop_addr    /* FIQ */

loop_addr:  .word   loop
main_addr:  .word   main
loop:   b   loop
    .end
```

The exception vectors are setup slightly differently than those in example 1. Instead of branching to the loop address, they load the program counter with the address of the loop.

The main C code is

```
/* ex2_main.c */

#define IOSET1 (*((volatile unsigned long *) 0xE0028014))
#define IODIR1 (*((volatile unsigned long *) 0xE0028018))
#define IOCLR1 (*((volatile unsigned long *) 0xE002801C))

int main (void)
{
    /* Define the LED pins P1.[16..23] as output */
    IODIR1 = 0x00FF0000;

    /* Clear all pins */
    IOCLR1 = 0x00FF0000;

    /* LED[7:0]; even on (high), odd off (low) */
```

```
    IOSET1 = 0x00550000;

    while(1);
    return 0;
}
```

Compilation and disassembly of just the main code using

```
arm-elf-gcc -mcpu=arm7tdmi -c ex2_main.c
arm-elf-objdump -d ex2_main.o
```

will show assembly code that uses the stack pointer. However, a stack pointer is not setup by `ex2_start.s`, so instead the code should be compiled with optimization level 2, `-O2`, as that eliminates the stack references for this particular example. The application can be compiled using

```
arm-elf-gcc -O2 -mcpu=arm7tdmi -nostartfiles -Ttext=0 \
      ex2_start.s ex2_main.c -o ex2.elf
```

The order of the files here is important, the startup code must come first. Disassembly of the code using

```
arm-elf-objdump -d ex2.elf
```

shows the startup code followed by the main code. The code covers addresses 0 through 48h (4Ch bytes). The size of the standard sections in the file are (all the sections can be viewed using `arm-elf-objdump -h ex2.elf`)

```
arm-elf-size ex2.elf
 text    data     bss     dec     hex filename
   76       0       0      76      4c ex2.elf
```

As with example 1, the code produces only a text section, so only Flash RAM is used.

Conversion of the elf file into hex format, and then download using FlashUtils produces the same result as example 1.

## 3.3 Examples 3(a) and (b): C program stack setup

The LPC2138 contains 32Kbytes (i.e., 8000h) SRAM starting at address 40000000h. The ARM stack grows down, so the `ex3_start.s` file initializes the stack pointer to the end of SRAM, i.e., 40008000h, and then jumps to main

```
/* ex3_start.s */
    .text
    .arm
    .global main
    .global _start
_start:
    /* Vectors (8 total) */
    b reset  /* reset */
    b loop   /* undefined instruction */
    b loop   /* software interrupt */
    b loop   /* prefetch abort */
    b loop   /* data abort */
    nop      /* reserved for the bootloader checksum */
    b loop   /* IRQ */
```

```
    b loop    /* FIQ */

/* Setup the stack pointer and then jump to main */
reset:
    ldr sp, stack_addr
    bl  main

/* Catch return from main */
loop:   b   loop

/* Constants */

/* LPC SRAM starts at 0x40000000, and there is 32Kb = 8000h */
stack_addr: .word   0x40008000
    .end
```

The Example 3(a) main application turns on the MCB2130 LEDs as in the previous examples, but uses a function call. A function call requires the use of a stack (the usuage of the stack by the main code in the final executable is shown shortly).

```
/* ex3a_main.c */
#include "led.h"

int main (void)
{
    led_init();
    led(0x55);
    while(1);
    return 0;
}
```

Since the LED routines are used in multiple programs, they are placed in a separate files; a header

```
/* led.h */
#ifndef LED_H
#define LED_H

/* Initialize the LEDs */
void led_init();

/* Control the LEDs */
void led(unsigned long val);

/* Set LEDs */
void led_set(unsigned long set);

/* Clear LEDs */
void led_clr(unsigned long clr);

#endif
```

and an implementation

```
/* led.c */
#include "led.h"

#define IOSET1 (*((volatile unsigned long *) 0xE0028014))
#define IODIR1 (*((volatile unsigned long *) 0xE0028018))
#define IOCLR1 (*((volatile unsigned long *) 0xE002801C))

void led_init()
{
    /* Define the LED pins P1.[16..23] as output */
    IODIR1 = 0x00FF0000;

    /* Clear all pins */
    IOCLR1 = 0x00FF0000;
}

void led(unsigned long val)
{
    /* LEDs off */
    IOCLR1 = (~val & 0xFF) << 16;

    /* LEDs on */
    IOSET1 =  (val & 0xFF) << 16;
}

void led_set(unsigned long set)
{
    IOSET1 =  (set & 0xFF) << 16;
}


void led_clr(unsigned long clr)
{
    IOCLR1 = (clr & 0xFF) << 16;
}
```

Board control functions, such as LED control, are reusable in multiple projects, and they are usually collected into a library referred to as a *board-support package* (BSP). In the following examples, the compiler arguments are simplified by just linking directly with the LED functions, but keep in mind that the creation of a BSP is preferred.

The startup, main, and LED code (if located in the current directory) can be compiled using

```
arm-elf-gcc -O2 -mcpu=arm7tdmi -nostartfiles -Ttext=0 \
    ex3_start.s ex3a_main.c led.c -o ex3a.elf
```

and then disassembled using

```
arm-elf-objdump -d ex3a.elf
```

to give the main code assembler

```
00000030 <main>:
  30:   e1a0c00d        mov     ip, sp
  34:   e92dd800        stmdb   sp!, {fp, ip, lr, pc}
  38:   e24cb004        sub     fp, ip, #4      ; 0x4
  3c:   eb000002        bl      4c <led_init>
  40:   e3a00055        mov     r0, #85 ; 0x55
  44:   eb000006        bl      64 <led>
  48:   eafffffe        b       48 <main+0x18>
```

The main routine starts by loading the inter-procedure call register with the stack pointer, and stores four registers to the stack. The registers are; the frame pointer, the inter-procedure call pointer, the link register, and the program counter. The ARM Procedure Calling Standard (APCS) [3], and the GCC *Using as* manual have details on the registers and their use in a C environment. The main code then adjusts the frame pointer, and calls the LED initialization routine (which sets the LED I/O pins to output mode, and outputs a logic low on each pin, turning the LEDs off). The value 0x55 is then moved into register `r0`, as the LED function argument, and a branch to the LED routine is made. When the LED call returns, an infinite loop occurs to end the program. The example contains only a `.text` section containing 188 bytes (BCh bytes).

To add a little more interest to the LED examples, Example 3(b) adds a delay loop to the main application that blinks the MCB2130 LEDs at approximately once per second. The loop count was determined to be 35000h (using an oscilliscope to measure the LED blink period). The startup routine used for this example does not setup the LPC2138 phase-locked-loop (PLL), so the ARM core operates at 12MHz. Example 5 sets up the PLL and uses an appropriately larger loop delay count.

## 3.4 Examples 4(a), (b), and (c): C programs with .bss, .data, and .rodata sections

Example 4(a) modifies Example 3(b) to add a static integer vector of length 2 to hold the LED blink values. The uninitialized vector is then initialized in main with the two LED blink values. The main application then drops into a while loop that uses the LED values.

Compilation of the main application

```
arm-elf-gcc -O2 -mcpu=arm7tdmi -c ex4a_main.c
```

followed by a dump of the sections gives

```
arm-elf-objdump -h ex4a_main.o

ex4a_main.o:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000054  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  00000000  00000000  00000088  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000008  00000000  00000000  00000088  2**2
                  ALLOC
  3 .comment      00000012  00000000  00000000  00000088  2**0
                  CONTENTS, READONLY
```

or alternatively

```
arm-elf-size ex4a_main.o
   text    data     bss     dec     hex filename
     84       0       8      92      5c ex4a_main.o
```

shows that the main code for the application contains an 8-byte .bss section.

Example 4(b) modifies Example 4(a) to use a static integer vector of length 2, and initializes the vector with the two LED blink values. The sections dump shows that there is an 8-byte .data (initialized data) section. Example 4(c) modifies Example 4(a) to use an initialized static const integer vector, this causes an 8-byte .rodata (read-only data) section to be generated (you need to use arm-elf-objdump -h ex4c_main.o to see the .rodata section).

The LPC2138 ARM microcontroller contains 512kB of Flash RAM and 32kB of SRAM. Applications are typically linked to execute from Flash, while modifiable variables and stacks always use SRAM. Executable code is linked to the .text section, while read-only variables are linked to the .rodata section. Initialized variables (that can also be modified) are linked to the .data section, while uninitialized variables are linked to the .bss section. Although an applications .data section uses SRAM, an area of Flash RAM of equal size is also required, to store the initial values assigned to the SRAM area. The startup routine copies the initial values from Flash to the SRAM .data section addresses before the main application executes and accesses those variables. The startup routine also needs to zero the range of SRAM addresses used by the .bss section.

Examples 4(a), (b), and (c) require a startup routine that performs the C environment setup; copying the .data section initial values from Flash, zeroing the .bss section, and setting up the stack pointer. A linker script is also required. The linker script is used to define the memory map of the processor, eg. location and length of Flash RAM and SRAM, and to define symbols for the .data section Flash and SRAM addresses, and the .bss section size. The following linker script can be used to compile the examples

```
/* lpc2138_flash.ld
 *
 * Linker script for Philips LPC2138 ARM microcontroller
 * applications that execute from Flash.
 */

/* The LPC2138 has 512kB of Flash, and 32kB SRAM */

MEMORY
{
    flash (rx) : org = 0x00000000, len = 0x00080000
    sram  (rw) : org = 0x40000000, len = 0x00008000
}


SECTIONS
{
    /* ---------------------------------------------------------------
     * .text section (executable code)
     * ---------------------------------------------------------------
     */
    .text :
    {
        *start.o (.text)
        *(.text)
        *(.glue_7t) *(.glue_7)
    } > flash
    . = ALIGN(4);

    /* ---------------------------------------------------------------
     * .rodata section (read-only (const) initialized variables)
     * ---------------------------------------------------------------
     */
    .rodata :
    {
        *(.rodata)
    } > flash
    . = ALIGN(4);

    /* End-of-text symbols */
    _etext = . ;
    PROVIDE (etext = .);

    /* ---------------------------------------------------------------
     * .data section (read/write initialized variables)
     * ---------------------------------------------------------------
     *
     * The values of the initialized variables are stored
     * in Flash, and the startup code copies them to SRAM.
     *
     * The variables are stored in Flash starting at _etext,
```

```
 * and are copied to SRAM address _data to _edata.
 */
 .data : AT (_etext)
 {
     _data = . ;
     *(.data)
 } > sram
 . = ALIGN(4);

 _edata = . ;
 PROVIDE (edata = .);

/* ----------------------------------------------------------------------
 * .bss section (uninitialized variables)
 * ----------------------------------------------------------------------
 *
 * These symbols define the range of addresses in SRAM that
 * need to be zeroed.
 */
.bss :
{
   _bss = . ;
   *(.bss)
   *(COMMON)
} > sram
. = ALIGN(4);
_ebss = . ;

_end = .;
PROVIDE (end = .);

/* Stabs debugging sections.  */
.stab          0 : { *(.stab) }
.stabstr       0 : { *(.stabstr) }
.stab.excl     0 : { *(.stab.excl) }
.stab.exclstr  0 : { *(.stab.exclstr) }
.stab.index    0 : { *(.stab.index) }
.stab.indexstr 0 : { *(.stab.indexstr) }
.comment       0 : { *(.comment) }
/* DWARF debug sections.
   Symbols in the DWARF debugging sections are relative to the beginning
   of the section so we begin them at 0.  */
/* DWARF 1 */
.debug          0 : { *(.debug) }
.line           0 : { *(.line) }
/* GNU DWARF 1 extensions */
.debug_srcinfo  0 : { *(.debug_srcinfo) }
.debug_sfnames  0 : { *(.debug_sfnames) }
/* DWARF 1.1 and DWARF 2 */
.debug_aranges  0 : { *(.debug_aranges) }
.debug_pubnames 0 : { *(.debug_pubnames) }
```

```
    /* DWARF 2 */
    .debug_info     0 : { *(.debug_info .gnu.linkonce.wi.*) }
    .debug_abbrev   0 : { *(.debug_abbrev) }
    .debug_line     0 : { *(.debug_line) }
    .debug_frame    0 : { *(.debug_frame) }
    .debug_str      0 : { *(.debug_str) }
    .debug_loc      0 : { *(.debug_loc) }
    .debug_macinfo  0 : { *(.debug_macinfo) }
    /* SGI/MIPS DWARF 2 extensions */
    .debug_weaknames 0 : { *(.debug_weaknames) }
    .debug_funcnames 0 : { *(.debug_funcnames) }
    .debug_typenames 0 : { *(.debug_typenames) }
    .debug_varnames  0 : { *(.debug_varnames) }
}
```

Executable code (`.text`), and read-only data (`.rodata`) are linked to Flash addresses. Initialized variables (`.data`) are linked to SRAM at addresses between symbols `_data` and `_edata`, but the values of the initialized variables are stored in Flash after the `.text` and `.rodata` sections, at address `_etext`. The symbols are defined by the linker and are refereed to in the startup routine. The startup routine defines variables (storage) initialized to the linker symbol values. The startup routine then uses the data section symbols to copy values from Flash to SRAM. The application refers to the SRAM versions of the variables. The `.bss` section contains uninitialized data. The linker combines the `.bss` sections from the various object files that make up an application, and the linker script defines start (`_bss`) and end (`_ebss`) addresses for the uninitialized variables in the final linked application image. The startup code must zero the SRAM address range between `_bss` and `_ebss`. The startup code used for the examples is

```
/* ex4_start.s */

    .global main
    .global _start

    /* Symbols defined by the linker script */
    .global _etext
    .global _data
    .global _edata
    .global _bss
    .global _ebss

    .text
    .arm
_start:
    /* Vectors (8 total) */
    b reset  /* reset */
    b loop   /* undefined instruction */
    b loop   /* software interrupt */
    b loop   /* prefetch abort */
    b loop   /* data abort */
    nop      /* reserved for the bootloader checksum */
    b loop   /* IRQ */
    b loop   /* FIQ */
```

```
/* Setup C runtime:
 * - copy .data section to SRAM
 * - clear .bss
 * - setup stack pointer
 * - jump to main
 */
reset:
    /* Copy .data */
    ldr r0, data_source
    ldr r1, data_start
    ldr r2, data_end
copy_data:
    cmp   r1, r2
    ldrne r3, [r0], #4
    strne r3, [r1], #4
    bne   copy_data

    /* Clear .bss */
    ldr r0, =0
    ldr r1, bss_start
    ldr r2, bss_end
clear_bss:
    cmp   r1, r2
    strne r0, [r1], #4
    bne   clear_bss

    /* Stack pointer */
    ldr sp, stack_addr
    bl  main

/* Catch return from main */
loop:   b   loop

/* Constants */

/* LPC SRAM starts at 0x40000000, and there is 32Kb = 8000h */
stack_addr: .word   0x40008000

/* Linker symbols */
data_source:    .word   _etext
data_start:     .word   _data
data_end:       .word   _edata
bss_start:      .word   _bss
bss_end:        .word   _ebss

    .end
```

Example 4(b) uses an initialized vector, so uses the `.data` section

```
/* ex4b_main.c */
#include "led.h"

static int led_value[2] = {0x55, 0xAA};

int main (void)
{
    int i;
    led_init();
    while(1) {
        led(led_value[0]);
        for (i = 0; i < 0x50000; i++);
        led(led_value[1]);
        for (i = 0; i < 0x50000; i++);
    }
    return 0;
}
```

Example 4(b) can be compiled using

```
arm-elf-gcc -O2 -mcpu=arm7tdmi -nostartfiles -T../lpc2138_flash.ld \
    ex4_start.s ex4b_main.c led.c -o ex4b.elf
```

(where it is assumed that the LED functions are in the same directory as the example source) and then the sections dumped using

```
arm-elf-objdump -h ex4b.elf

ex4b.elf:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000012c  00000000  00000000  00008000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000008  40000000  0000012c  00010000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  40000008  00000134  00010008  2**0
                  ALLOC
  3 .comment      00000024  00000000  00000000  00010008  2**0
                  CONTENTS, READONLY
```

The `.data` section LMA (load memory address) in Flash is the address at which the initial values are stored, while the VMA (virtual memory address) in SRAM is the start address at which the application refers to initialized variables.

Table 1: LPC213x PLL control registers (p17 [10])

| Name | Description | Access | Reset Value | Address |
|---|---|---|---|---|
| PLLCON | PLL Control | R/W | 0 | 0xE01F C080 |
| PLLCFG | PLL Configuration | R/W | 0 | 0xE01F C084 |
| PLLSTAT | PLL Status | RO | 0 | 0xE01F C088 |
| PLLFEED | PLL Feed | WO | N/A | 0xE01F C08C |

## 3.5 Example 5: LPC2138 processor initialization

Processor initialization code for all ARM-based processors follows a similar sequence of steps, however each processor has processor-specific steps. The Philips LPC21xx series of microcontrollers require the following processor initialization steps;

1. Exception vector setup
2. Phase-locked loop setup
3. Memory accelerator module setup
4. Setup stack pointers for each processor mode
5. Copy .data section to SRAM
6. Clear .bss
7. Jump to main

The initialization sequence starts with exception vector setup, since those vectors are located starting at address zero. The phase-locked loop (PLL) and memory accelerator module (MAM) are then setup so that the remaining code runs at the full processor clock speed. The memory accelerator module allows the LPC-microcontroller processor core to fetch instructions efficiently from slower on-chip flash RAM. The LPC213x User Manual [10] describes the PLL and MAM.

### 3.5.1 PLL setup

The MCB2130 evaluation board contains an LPC2138 microcontroller connected to a 12MHz crystal. The PLL control registers are shown in Table 1, the crystal oscillator is described in Section 3.4 (p18 [10]) and the PLL is described in Section 3.7 (p26 [10]). The LPC2138 can operate with a processor clock frequency of up to 60MHz, so with a 12MHz crystal the PLL needs to be configured to effectively multiply the crystal by 5. The phase-locked loop consists of a phase-detector, a Current Controller Oscillator (CCO), a divide-by-2×P output divider, and a divide-by-M feedback divider that follows the divide-by-2×P divider (so the PLL feedback division for the CCO is 2×M×P) (p28 [10] has a block diagram). The CCO operates over the frequency range 156MHz to 320MHz, and the output divider generates the processor clock frequency. The output divider options are 2, 4, 8, and 16, so for a desired 60MHz processor clock, the CCO can be operated at 240MHz with a divide-by-4 divider setting. The feedback multiplier required to get from 60MHz down to 12MHz is 5. The PLLCFG register settings are then `PSEL[1:0] = PLLCFG[6:5] = 01b` (P = 2) and `MSEL[4:0] = PLLCFG[4:0] = 00100b` (M = 5), i.e., `PLLCFG = 0100100b = 24h` (p29 [10]).

Programming of the PLL requires a special unlock (or feed) sequence, to avoid erroneous programming of the PLL. The PLL takes some time to lock, and so a status bit needs to be polled to

check for lock (or an interrupt can also be generated). Once the PLL is locked, it can be used to clock the processor core. The sequence for programming the PLL (without using an interrupt) is;

1. Write the PLL PSEL and MSEL settings to the PLLCFG register (eg. 24h for the MCB2130).
2. Write 1 to the PLL enable bit 0 (PLLE) in the PLL control register (PLLCON) (p29 [10]).
3. Enable the PLL by writing the feed sequence to the PLLFEED register, i.e., 0xAA then 0x55 (p30 [10]). The feed sequence causes the values written during the first two steps to activate the PLL to lock and generate a 60MHz processor clock.
4. Poll bit 10 (PLOCK) in the PLL status register (PLLSTAT) until it becomes 1 (p30 [10]).
5. Write 1 to the PLL connect bit 1 (PLLC) in the PLL control register (PLLCON) (p29 [10]) (the enable bit, bit 0, should also remain set).
6. Connect the PLL clock to the processor core by writing the feed sequence to the PLLFEED register, i.e., 0xAA then 0x55.

Since the PLL is to be configured prior to stack setup, the PLL initialization sequence needs to be coded in assembler. An alternative processor initialization would be to setup the stacks first, and then call an `_init` function coded in C, prior to jumping to main. A C-coded PLL initialization sequence is

```
#define PLLCON  (*(volatile unsigned int *)0xE01FC080)
#define PLLCFG  (*(volatile unsigned int *)0xE01FC084)
#define PLLSTAT (*(volatile unsigned int *)0xE01FC088)
#define PLLFEED (*(volatile unsigned int *)0xE01FC08C)

#define PLLCON_PLLE   (1 << 0)
#define PLLCON_PLLC   (1 << 1)
#define PLLSTAT_PLOCK (1 << 10)
#define PLLFEED1      0xAA
#define PLLFEED2      0x55

#define PLLCFG_VALUE  0x24

void pll_init(void)
{
    PLLCFG  = PLLCFG_VALUE;
    PLLCON  = PLLCON_PLLE;
    PLLFEED = PLLFEED1;
    PLLFEED = PLLFEED2;
    while ((PLLSTAT & PLLSTAT_PLOCK) == 0);
    PLLCON  = PLLCON_PLLC|PLLCON_PLLE;
    PLLFEED = PLLFEED1;
    PLLFEED = PLLFEED2;
}
```

This function can be compiled to assembler, and the output hand-optimized. An assembly-coded version of the PLL initialization is

```
/* Constants (and storage, used in ldr statements) */
PLLBASE:       .word   0xE01FC080
```

```
/* Constants (used as immediate values) */
    .equ PLLCON_OFFSET,   0x0
    .equ PLLCFG_OFFSET,   0x4
    .equ PLLSTAT_OFFSET,  0x8
    .equ PLLFEED_OFFSET,  0xC

    .equ PLLCON_PLLE,    (1 << 0)
    .equ PLLCON_PLLC,    (1 << 1)
    .equ PLLSTAT_PLOCK,  (1 << 10)
    .equ PLLFEED1,        0xAA
    .equ PLLFEED2,        0x55

    .equ PLLCFG_VALUE,    0x24

pll_init:
    /* Use r0 for indirect addressing */
    ldr r0, PLLBASE

    /* PLLCFG = PLLCFG_VALUE */
    mov r3, #PLLCFG_VALUE
    str r3, [r0, #PLLCFG_OFFSET]

    /* PLLCON = PLLCON_PLLE */
    mov r3, #PLLCON_PLLE
    str r3, [r0, #PLLCON_OFFSET]

    /* PLLFEED = PLLFEED1, PLLFEED2 */
    mov r1, #PLLFEED1
    mov r2, #PLLFEED2
    str r1, [r0, #PLLFEED_OFFSET]
    str r2, [r0, #PLLFEED_OFFSET]

    /*  while ((PLLSTAT & PLLSTAT_PLOCK) == 0); */
pll_loop:
    ldr r3, [r0, #PLLSTAT_OFFSET]
    tst r3, #PLLSTAT_PLOCK
    beq pll_loop

    /* PLLCON = PLLCON_PLLC|PLLCON_PLLE */
    mov r3, #PLLCON_PLLC|PLLCON_PLLE
    str r3, [r0, #PLLCON_OFFSET]

    /* PLLFEED = PLLFEED1, PLLFEED2 */
    str r1, [r0, #PLLFEED_OFFSET]
    str r2, [r0, #PLLFEED_OFFSET]
```

The code uses one word of storage for the address of the PLL base address register, and then uses 8-bit immediate values for the remaining constants. The immediate values become coded as part of the assembly instruction, so do not require additional storage (see Ch. 5 of the ARM-ARM, eg. pA5-4 to A5-7 for `mov` and `orr` encoding [12]).

### 3.5.2 MAM setup

The access times of on-chip Flash memories usually limit the maximum speed of microcontrollers. Reference [11] explains how Philips solved this problem for the LPC21xx microcontroller family with the Memory Accelerator Module (MAM), and contains a nice introduction to the microcontroller features. Chapter 4 of the User Manual (p42 [10]) details the MAM. The MAM includes three 128-bit buffers called the Prefetch Buffer, the Branch Trail Buffer and the Data buffer. The 128-bit buffers allow Flash memory accesses to deliver four 32-bit ARM-instructions or eight 16-bit Thumb instructions. Nevertheless the CPU still must wait for the first instruction until the memory access is finished. Only then can the next three (ARM) or seven (Thumb) instructions be made available without further delay [11]. Reference [11] shows benchmark results of operation the MAM; disabled, partially enabled, and fully enabled (p44 [10] explains the three modes).

The MAM registers consist of a control register and a timing control register (p44 [10]). Two configuration bits select the three MAM operating modes. The configuration mode can be changed at any time, so the startup code fully enables the MAM (`MAM_mode_control = 10b`). The MAM Timing register determines how many processor core clock cycles are used to access the Flash memory. This allows tuning MAM timing to match the processor operating frequency. There is no code fetch penalty for sequential instruction execution when the CPU clock period is greater than or equal to one fourth of the Flash access time (p42 [10]). For a system clock slower than 20MHz (50ns period) the MAMTIM register can be set to 1 (p47 [10]). At 60MHz, the clock period is 16.7ns, four times this is 66.7ns, which is greater than 50ns, so MAMTIM can be set to 4. The MAM initialization code is;

```
/* Constants (and storage, used in ldr statements) */
MAMBASE:       .word   0xE01FC000

/* Constants (used as immediate values) */
    .equ MAMCR_OFFSET,   0x0
    .equ MAMTIM_OFFSET,  0x4

    .equ MAMCR_VALUE,    0x2  /* fully enabled */
    .equ MAMTIM_VALUE,   0x4  /* fetch cycles  */

mam_init:
    /* Use r0 for indirect addressing */
    ldr r0, MAMBASE

    /* MAMCR = MAMCR_VALUE */
    mov r1, #MAMCR_VALUE
    str r1, [r0, #MAMCR_OFFSET]

    /* MAMTIM = MAMTIM_VALUE */
    mov r1, #MAMTIM_VALUE
    str r1, [r0, #MAMTIM_OFFSET]
```

### 3.5.3 Stacks setup

Figure 1 shows the seven ARM operating modes. The figure shows that there are 6 different stack pointers; user/system mode, supervisor mode, IRQ mode, FIQ mode, abort mode, and undefined mode. The ARM processor resets to supervisor mode, a privileged mode (pA2-13, pA2-14 [12]). The control and program status register (CPSR) `M[4:0]` bits can be modified from within a privileged mode to switch between processor modes and setup the different stacks.

The size of the stack required for each processor mode is application dependent. When an exception occurs, the banked versions of the link register (LR or R14) and the saved processor status register (SPSR) for the exception mode are used to save state (see pA2-13 [12]). The ARM processor does not use the stack during the exception entry, it is only the handler code that uses the stack. If the default handler uses a branch or load instruction to 'lock-up', then no stack setup is required. If a more complex handler is installed, eg. an abort handler that writes a console message and then locks-up, then the stack size is determined by the function call requirements. The stacks that are generally required are the system and supervisor mode stacks for operating system usage, the user stack for task usage, the IRQ and FIQ stacks for interrupt handlers, and optionally the abort and undefined handlers.

An example of stack initialization code for the LPC2138 is;

```
/* Constants (and storage, used in ldr statements) */
STACK_START:       .word   0x40008000

/* Constants (used as immediate values) */
    /* Processor modes (see pA2-11 ARM-ARM) */
    .equ FIQ_MODE,       0x11
    .equ IRQ_MODE,       0x12
    .equ SVC_MODE,       0x13  /* reset mode */
    .equ ABT_MODE,       0x17
    .equ UND_MODE,       0x1B
    .equ SYS_MODE,       0x1F

    /* Stack sizes */
    .equ FIQ_STACK_SIZE, 0x00000080    /*  32x32-bit words */
    .equ IRQ_STACK_SIZE, 0x00000080
    .equ SVC_STACK_SIZE, 0x00000080
    .equ ABT_STACK_SIZE, 0x00000010    /*   4x32-bit words */
    .equ UND_STACK_SIZE, 0x00000010
    .equ SYS_STACK_SIZE, 0x00000400    /* 256x32-bit words */

    /* CPSR interrupt disable bits */
    .equ IRQ_DISABLE,    (1 << 7)
    .equ FIQ_DISABLE,    (1 << 6)

/* Setup the stacks */
    ldr r0, STACK_START

    /* FIQ mode stack */
    msr CPSR_c, #FIQ_MODE|IRQ_DISABLE|FIQ_DISABLE
    mov sp, r0
    sub r0, r0, #FIQ_STACK_SIZE

    /* IRQ mode stack */
    msr CPSR_c, #IRQ_MODE|IRQ_DISABLE|FIQ_DISABLE
    mov sp, r0
    sub r0, r0, #IRQ_STACK_SIZE

    /* Supervisor mode stack */
    msr CPSR_c, #SVC_MODE|IRQ_DISABLE|FIQ_DISABLE
    mov sp, r0
```

```
sub r0, r0, #SVC_STACK_SIZE

/* Undefined mode stack */
msr CPSR_c, #UND_MODE|IRQ_DISABLE|FIQ_DISABLE
mov sp, r0
sub r0, r0, #UND_STACK_SIZE

/* Abort mode stack */
msr CPSR_c, #ABT_MODE|IRQ_DISABLE|FIQ_DISABLE
mov sp, r0
sub r0, r0, #ABT_STACK_SIZE

/* System mode stack */
msr CPSR_c, #SYS_MODE|IRQ_DISABLE|FIQ_DISABLE
mov sp, r0

/* Leave the processor in system mode */
```

The initialization code sets up the system mode stack last, and leaves the processor in system mode. The processor mode is not left in supervisor mode, since a software interrupt (SWI) exception causes the processor to change to supervisor mode (pA2-13 [12]). Interrupts should not be enabled while the processor is in an exception mode, otherwise the link register can be over-written (pA2-6 [12]). The stack sizes are guesses, and will need to be checked for specific examples.

Example 5 repeats the LED blinking code from Example 3(b). The startup code was modified to setup the PLL, fully enable the MAM, and setup stacks for all modes. The delay loops in the main application had to be increased by a factor of 35 to obtain one second LED blink rate. A factor of 5 in speed-up was expected by enabling the PLL since that causes the core to be clocked at 60MHz, and a factor of 4 was expected due to enabling of the MAM, however, the observed improvement was a factor of 7.

To confirm the source of the unexpected increase in performance, the reset label was moved around in the startup code. First the reset label was moved such that it skipped the PLL setup and the MAM setup; the resulting period was 35 seconds. Next the label was moved to enable the MAM; the resulting period was 5 seconds. Moving the label back to its original location, enabling the PLL, put the period back at 1 second. So the source of the speed-up was the MAM.

To get an alternative measurement of the increase in performance between Example 5 and Example 3(b), the delay loops were commented out, and an oscilliscope was used to probe the first LED (`P1.16`). Example 3(b) produced a 40kHz square-wave (10.0$\mu$s high-time and 15.0$\mu$s low-time), while Example 5 produced a 518kHz square-wave (0.60$\mu$s high-time and 1.33$\mu$s low-time); an increase in frequency of about 13 times.

## 3.6 Example 6: Exception handling

Figure 1 shows the seven ARM operating modes and the five exception modes (pA1-3 [12]);

- fast interrupt (FIQ)
- normal interrupt (IRQ)
- memory aborts, which can be used to implement memory protection or virtual memory
- attempted execution of an undefined instruction
- software interrupt (SWI) instruction which can be used to make a call to an operating system

Figure 1 shows that each exception mode has banked versions of the stack-pointer (R13) (each exception has a separate stack) and link-register (R14). The fast interrupt mode has additional banked registers to reduce the context save and restore time for fast interrupts. When an exception handler is entered, the link-register holds the return address for exception processing. The address is used to return from the exception, or determine the address that caused the exception. The saved program status register (SPSR) register saves the state of the current program status register (CPSR) at the time of the exception. Exceptions are described in detail in the ARM-ARM [12] ppA2-13 to 21 and in Chapter 9 of the ARM System Developer's Guide [13]. The ARM7TDMI-S Technical Reference Manual [1] pp2-19 to 27 details exceptions for the ARM core used in the Philips LPC2138 microcontroller.

The recommended entry and exit sequence for an interrupt (FIQ or IRQ) is (pA2-14 [12]);

```
sub lr, lr, #4
stmfd sp!, {<other_registers>, lr}

... interrupt handler ...

ldmfd sp!, {<other_registers>, pc}^
```

The adjustment to the link register value required to determine an exception return address can be found in the ARM7TDMI-S manual (pp2-19 to 27 [1]).

An exception handler can be coded directly in ARM assembler, or C-compiler specific keywords can be used to generate the appropriate prolog and epilog code. The GCC compiler has a set of non-ANSI extensions to declare exception handlers from C code. The declaration syntax for an IRQ handler is

```
void irq_handler(void) __attribute__ ((interrupt("IRQ")));
```

The exception source keywords are; IRQ, FIQ, SWI, ABORT, and UNDEF (see Chapter 5 *Extensions to the C Language Family*, *Declaring attributes of functions*, in any recent GCC manual eg. the 3.4.4 or 4.0.1 manual on `www.gnu.org`).

The empty interrupt handler (with no exception source attribute):

```
/* handler.c */
/* Function declaration */
void handler(void) __attribute__((interrupt));

/* Function definition */
void handler(void)
{
    /* Handler body */
}
```

compiled to assembler using `arm-elf-gcc -mcpu=arm7tdmi -Wall -O2 -S handler.c` produces the (edited) assembler code

```
        .text
        .align 2
        .global handler
    handler:
        subs    pc, lr, #4
```

i.e., produces code appropriate for return from an FIQ, IRQ, or ABORT. Adding the FIQ, IRQ, or ABORT attribute causes no change in the assembler. The attribute SWI or UNDEF changes the return sequence to `movs pc, lr`. The ARM7TDMI-S manual pages 2-19 to 20 [1] shows the recommended return sequences for exceptions. The return sequences produced by the GCC compiler matches the recommendations for all but a data abort. The `interrupt` keyword changes the return sequence of the interrupt handler, it does not setup the interrupt vector table to point to the handler. The processor initialization code containing the exception vector table needs to be modified to point to the exception handler.

The ARM core contains an FIQ or IRQ interrupt pin, and most ARM processors include interrupt controllers that route external interrupt sources onto the FIQ or IRQ pins. Use of FIQ or IRQ interrupts requires setting up the interrupt controller prior to enabling the interrupt. The Philips LPC family uses the Vectored Interrupt Controller (VIC) defined by ARM.

The MCB2130 has a push button connected to the LPC2138 external interrupt pin (`EINT1`). Example 6(a) sets up the MCB2130 board so that on reset `LED[0]` is on, and each time the push button is pressed, an FIQ interrupt is generated. The interrupt handler moves the LED that is on to the next LED (eg. cycles through `LED[0]`, `LED[1]`, ..., `LED[7]`, and then starts back at `LED[0]`). Example 6(b) starts with `LED[7]` on, and button presses generate an IRQ interrupt which moves the LED on in the opposite direction to Example 6(a) The LPC213x User Manual [10] details the LPC2138 peripherals setup for this example;

- The startup file initializes the processor and leaves it in system mode with FIQ and IRQ enabled.
- The application code configures the `PINSEL0` register so that the `P0.14` pin is setup for EINT1 operation (p75 [10]).
- External interrupt configuration is detailed on p17, and pp20-24 [10]. The code sets up `EINT1` for falling-edge, edge-sensitive mode. The `EXTINT` register is written to after the mode change, and to clear the interrupt.
- The VIC select register is used to select `EINT1` as an FIQ interrupt in Example 6(a), and an IRQ interrupt in Example 6(b). Example 6(b) sets up the VIC for a priority interrupt from `EINT1` at VIC vector priority 0. The VIC enable register is then use to enable `EINT1` (Chapter 5 [10]).

Once the LPC2138 is configured, the main application drops into an infinite loop. After that point, button pushes generate FIQ or IRQ interrupts, and the interrupt handler updates the LEDs.

There are some minor changes to the startup file, `ex6_start.s`, relative to `ex5_start.s`. First, the IRQ and FIQ interrupt vectors are modified;

```
_start:
    b reset  /* reset */
    b loop   /* undefined instruction */
    b loop   /* software interrupt */
    b loop   /* prefetch abort */
```

```
    b loop   /* data abort */
    nop      /* reserved for the bootloader checksum */
    ldr pc, [pc, #-0x0FF0]     /* VicVectAddr */
    ldr pc, fiq_addr

    /* Address of the handler function */
fiq_addr: .word fiq_handler
```

The FIQ vector loads the program counter with the address of the FIQ handler, while the IRQ handler loads the address determined by the VIC. The second change is that when the system mode stack is setup, the FIQ and IRQ interrupts are left enabled.

Interrupt handling for Example 6(a) is fairly simple, since the interrupt vector loads the program counter with the address of the handler. The setup of an IRQ handler is slightly more complex. The VIC setup from Example 6(b) showing how to setup the VIC for EINT1 IRQs is;

```
void irq_init(void)
{
  /* Enable P0.14 EINT1 pin function: PINSEL0[29:28] = 10b */
  PINSEL0 = (2 << 28);

  /* Make EINT1 falling edge-sensitive
   * (level sensitive increments the LED count too fast)
   */
  EXTMODE  = 2;
  EXTPOLAR = 0;

  /* Clear register after mode change */
  EXTINT = EXTINT;

  /* Setup the VIC to have EINT1 generate IRQ
   * (EINT1 is interrupt source 15)
   */
  VICIntSelect   = 0;                          /* Select IRQ      */
  VICVectAddr0   = (unsigned long)irq_handler; /* Vector 0        */
  VICVectCntl0   = 0x20 | 15;                  /* EINT1 Interrupt */
  VICIntEnable   = (1 << 15);                  /* Enable          */
}
```

The IRQ initialization code sets up the EINT1 source and then the VIC. The VIC initialization sets up vector slot 0 for EINT1 interrupts. The IRQ handler has an additional step relative to the FIQ handler; an acknowledge to the VIC, i.e., `VICVectAddr = 0;`. Chapter 5 of the LPC213x user manual has a clear discussion on the VIC setup [10].

## 3.7 Example 7: I/O pin toggling

A simple technique for benchmarking operations, is to toggle an I/O pin around a block of code and measure the pulse time with an oscilliscope. Interrupt service routine (ISR) context save and restore routine times can also be determined using this technique. The measured I/O pin pulse time should be adjusted for the time it takes to simply toggle an I/O pin. The examples in this section demonstrate the fastest I/O toggle speed coded in assembler, and then the more practical case of toggle speed due to LED set and clear function calls from C-code.

Example 7(a) determines the maximum frequency an I/O pin can be toggled by; configuring the PLL for 60MHz operation, configuring the MAM, and configuring the peripheral bus clock divider (VPB divider) to 1. The code then drops into a loop that sets the LEDs high, then low, then loops back to high. The main loop from `ex7a.s` is

```
    /* LED register addresses and control value */
    ldr r0, IODIR1
    ldr r1, IOCLR1
    ldr r2, IOSET1
    ldr r3, IODIR1_VALUE

    /* Set pins as output */
    str r3, [r0]

loop:
    /* Set LEDs */
    str r3, [r2]

    /* Clear LEDs */
    str r3, [r1]

    b   loop
```

The high time will be slightly shorter than the low time due to the branch that occurs as part of the loop.

Figure 3 shows that a 3.5MHz square-wave is produced on the MCB2130 board; a high time of 119ns (about 7 clocks) and a low time of 164ns (10 clocks). If the VPB divider is left in its default state of divide-by-four, a 1.66MHz square-wave is produced; 266ns (16 processor clocks) high-time and 333ns (20 processor clocks) low-time.

Example 7(b) is similar to the code in Example 5. The Example 7(b) startup file initializes the PLL, sets up the MAM, sets up the C environment and jumps to main. The main application sets the peripheral bus divider to 1, and then falls into a while loop that toggles the LEDs high, and then low. Figure 4 shows that a 984Hz square-wave is produced; with a high time of 402ns (24 clocks) and a low time of 615ns (37 clocks). If the VPB divider is left in its default state of divide-by-four, a 789kHz square-wave is produced; 536ns (32 processor clocks) high-time and 731ns (44 processor clocks) low-time. A block of code benchmarked by pulsing an LED pin using the C-coded LED control functions, should adjust the measured pulse time by 402ns (for VPBDIV = 1) or 536ns (for VPBDIV = 0) to account for the LED pulsing overhead.

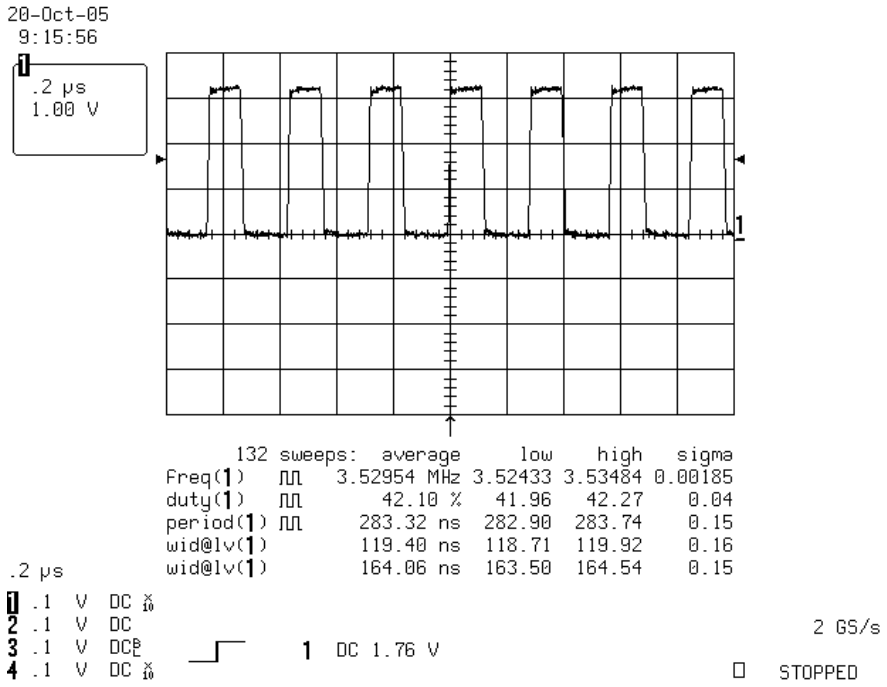



Figure 3: LPC2138 maximum I/O toggle speed; 3.5MHz. The oscilliscope screen capture shows the waveform frequency, duty cycle, period, high-time, and low-time.

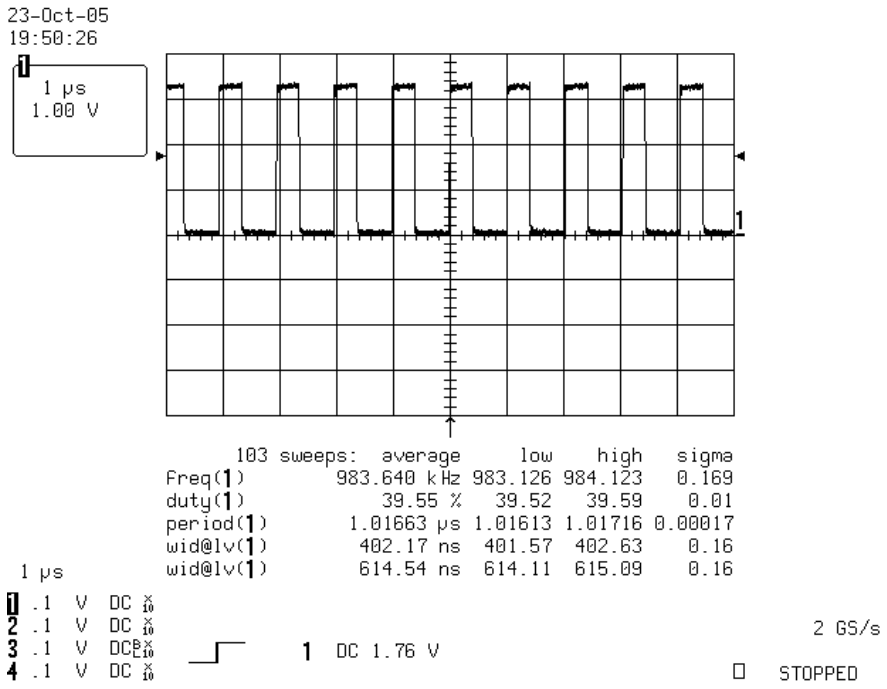



Figure 4: LPC2138 I/O toggle speed using C; 984kHz. The C-code uses a general purpose LED control function (making it slower).

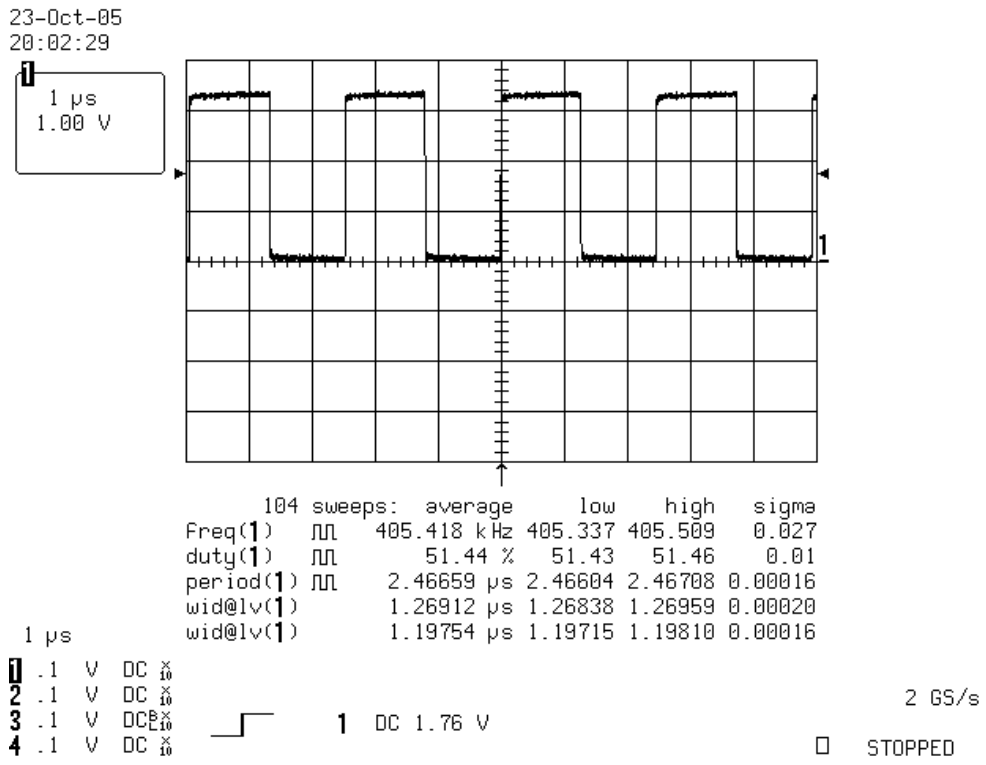



Figure 5: LPC2138 FIQ context save/restore benchmarking. The Example 8(a) test application toggles an output pin connected to an input pin configured as an EINT1 source. EINT1 is handled using an FIQ handler. The EINT1 interrupt is setup for rising-edge sensitivity, the main application toggles the pin high, and the FIQ handler toggles the pin low. The high-time of the waveform is 1.27$\mu$s (76 clocks), while the low time is 1.20$\mu$s (72 clocks).

## 3.8 Example 8: Interrupt context save/restore benchmarking

Example 8(a) takes the push-button FIQ handler code from Example 6(a) and modifies it so that EINT1 is generated from P0.3, and a jumper was placed between `LED[0]` (P1.16) and P0.3. The EINT1 interrupt was setup to be rising-edge sensitive. The main code in Example 8(a) sets all the LEDs low, enables FIQ interrupts, and then drops into a while loop that always sets the LEDs high. The rising-edge that occurs when the program starts triggers an FIQ interrupt, and the FIQ interrupt handler clears the LEDs. When the handler returns to the main application, the LEDs are set high again, and a FIQ interrupt is generated. The result is a square-wave on the LEDs. Figure 5 shows the waveform. The context save plus LED pulse high-time is 1.27$\mu$s, while the context restore and while loop time is 1.20$\mu$s.

This benchmark analysis indicates that an FIQ handler has a context save/restore time of approximately 2.5$\mu$s. So if the LPC was being used in a system processing a 1kHz FIQ interrupt, the FIQ context save/restore time represents a 0.25% CPU load. This benchmark represents the overhead of the save/restore sequence for a C-coded FIQ handler. Disassembly of the example code shows that the handler saves eight registers on entry (`r0-r3`, `fp`, `ip`, `lr`, `pc`), and restores seven registers on exit. When using an RTOS, an interrupt can cause a higher-priority task to become ready, and so additional context save or restore operations are required. For example, registers could need to be moved off the FIQ stack onto the task stack, and the new tasks registers moved onto the FIQ stack, or the context save routine might be setup to save registers directly to the task stack,

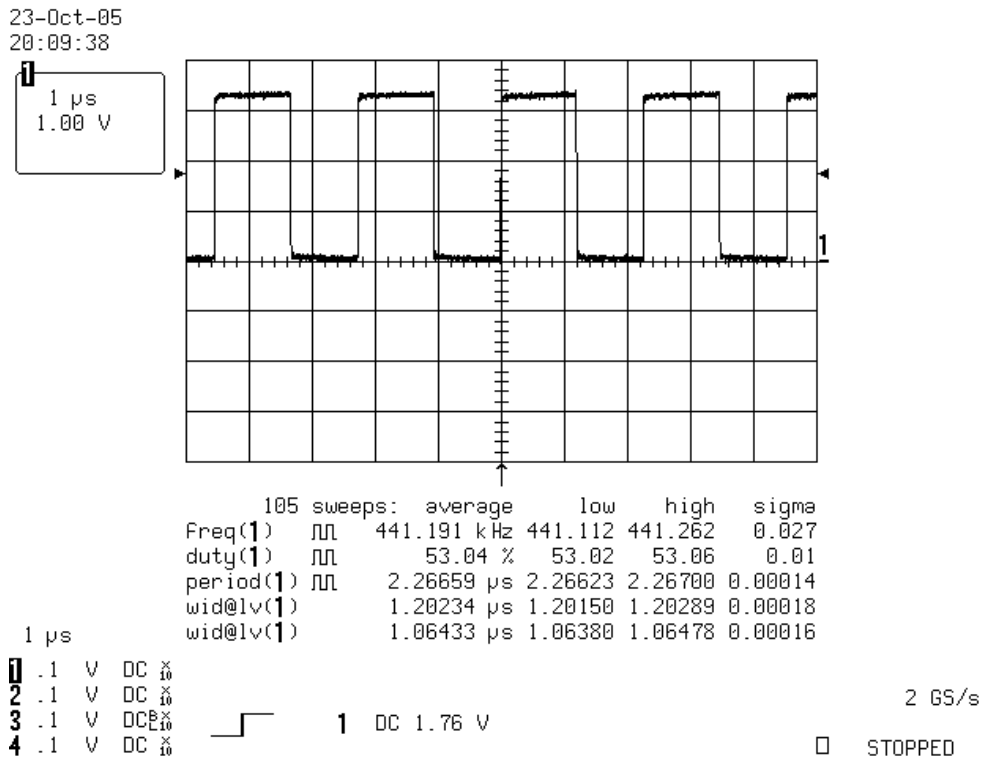



Figure 6: LPC2138 software generated FIQ context save/restore benchmarking. The Example 8(b) test application uses the vectored interrupt controller (VIC) to software generate an EINT1 interrupt. EINT1 is handled using an FIQ handler. The main application toggles the LED pins high, and the FIQ handler toggles the LED pins low. The high-time of the waveform is 1.20$\mu$s (72 clocks), while the low time is 1.06$\mu$s (64 clocks).

and make minimal use of the FIQ stack. Benchmarking of the uCOS-II RTOS is shown later.

Example 8(a) used an external interrupt pin to test interrupt latency. The Vectored Interrupt Controller in the LPC-series provides an alternative option for software testing of interrupts; the VIC software interrupt register, and software interrupt clear register (p49 [10]). Example 8(b) modifies Example 8(a) to use a software generated EINT1 interrupt. The main code still writes to the LEDs so that the code can be benchmarked using an oscilloscope, however, code is added to set and clear the software interrupt register (code to setup the external EINT1 interrupt is also removed). Figure 6 shows the waveform from the software generated FIQ interrupt. The high-time and low-time of the waveform are both slightly smaller than in Figure 5.

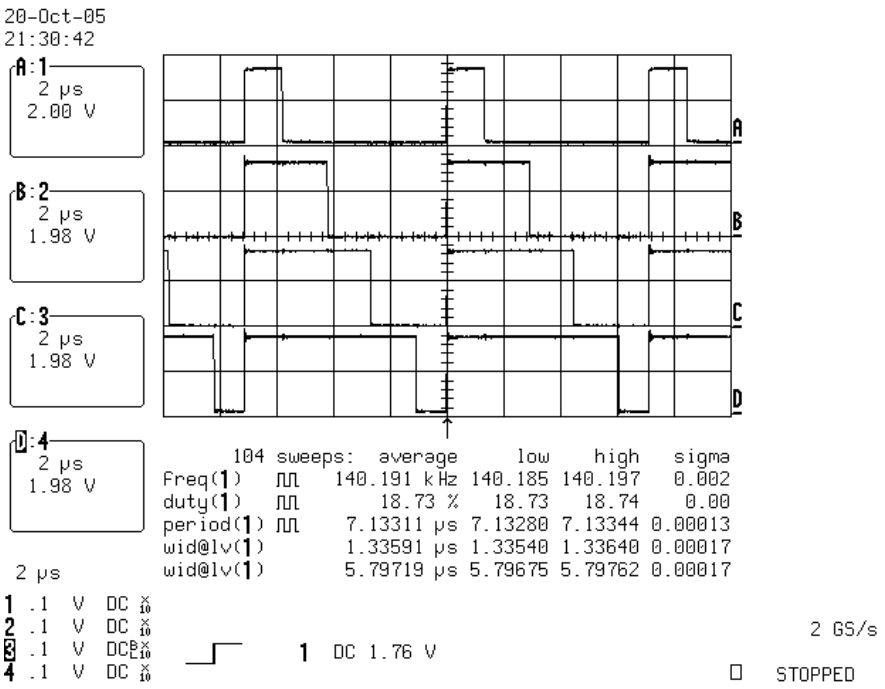



Figure 7: LPC2138 vectored IRQ prioritization. The Example 9(a) test application uses the vectored interrupt controller (VIC) to software generate EINT[0:3] interrupts which are enabled on the vectored IRQs 0 to 3. The main application toggles the LED pins 0 to 3 high, and the IRQ handlers each toggle one LED low. The figure shows how the EINT interrupts are serviced in order 0 to 3. The high-time of the EINT0 waveform is 1.34$\mu$s; similar to the previous tests. The high times of the other interrupts are progressively longer.

## 3.9 Example 9: Multiple interrupts

This section benchmarks interrupt handling when dealing with multiple interrupt sources. The ARM core has two interrupt lines; the FIQ and IRQ. The FIQ is generally expected to have a single interrupt source, while the IRQ line can have multiple interrupt sources. The VIC on the LPC-series can be used to divide the IRQ sources into vectored (prioritized) IRQs, and non-vectored IRQs. For the vectored IRQs, the VIC acts like a hardware multiplexer, causing the processor to jump to the address of the handler of the highest priority interrupt. For the non-vectored IRQs, the same handler is provided, and the handler code has to perform the demultiplexing for multiple non-vectored sources.

Example 9(a) follows on from Example 8(b) and uses the VIC to setup four software interrupts on `EINT[0:3]`. The interrupts are set up to generate IRQ interrupts. The interrupt handlers are placed in IRQ vector slots `VICVectAddr[0:3]`. The main application sets `LED[0:3]` high, and each EINT handler sets a single LED low. Figure 7 shows the resulting LED waveforms.

The VIC IRQ controller prioritizes when interrupts occur simultaneously. However, if an IRQ handler has already started, and a higher priority interrupt occurs, the higher priority handler will not run until after the current handler completes. Example 9(b) and Figure 8 demonstrate the problem. Example 9(b) starts by generating an interrupt on EINT3, and then EINT3's IRQ handler is used to set EINT2's LED high and generate an EINT2 interrupt. EINT2's handler does the same for EINT1, EINT1's handler does the same for EINT0, and EINT0 leaves it to main to restart the sequence. Its obvious from the figure that each lower priority interrupt is completing while a higher

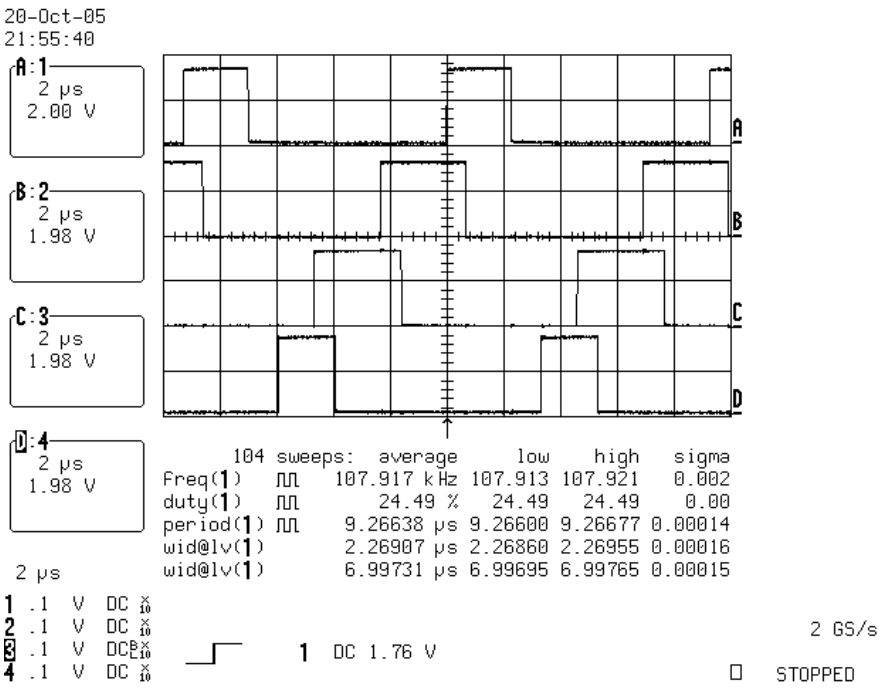



Figure 8: LPC2138 IRQ priority inversion. The Example 9(b) starts by generating a software interrupt on EINT3, and then each IRQ handler generates a software interrupt on the next higher-priority interrupt (except for EINT0, that handler leaves it to main to restart the process). Note how even though a higher priority interrupt has occurred, it does not get processed until the current handler completes.

priority interrupt is pending. The next section shows how interrupts can be made interruptible.

Examples 6(a), 6(b), 9(a) and 9(b) used the GCC keyword `interrupt` to define interrupt handler functions. Example 9(c) shows how interrupt handlers can be written without using the `interrupt` keyword. The example uses a short assembler coded sequence to save processor state, and then call a C-coded FIQ or IRQ handler. The C-coded IRQ handler reads from the VICVectAddr register to dispatch handler routines. Since those routines are called from a C function, they all need to be written as standard C-functions (without the interrupt keyword). The IRQ and FIQ exception vectors in the startup code were modified as follows;

```
_start:
    b reset  /* reset */
    b loop   /* undefined instruction */
    b loop   /* software interrupt */
    b loop   /* prefetch abort */
    b loop   /* data abort */
    nop      /* reserved for the bootloader checksum */
    ldr pc, irq_addr

    /* FIQ ISR */
fiq_isr:
    sub   lr, lr, #4
    stmfd sp!, {r0-r3, ip, lr}
    bl fiq_handler
```

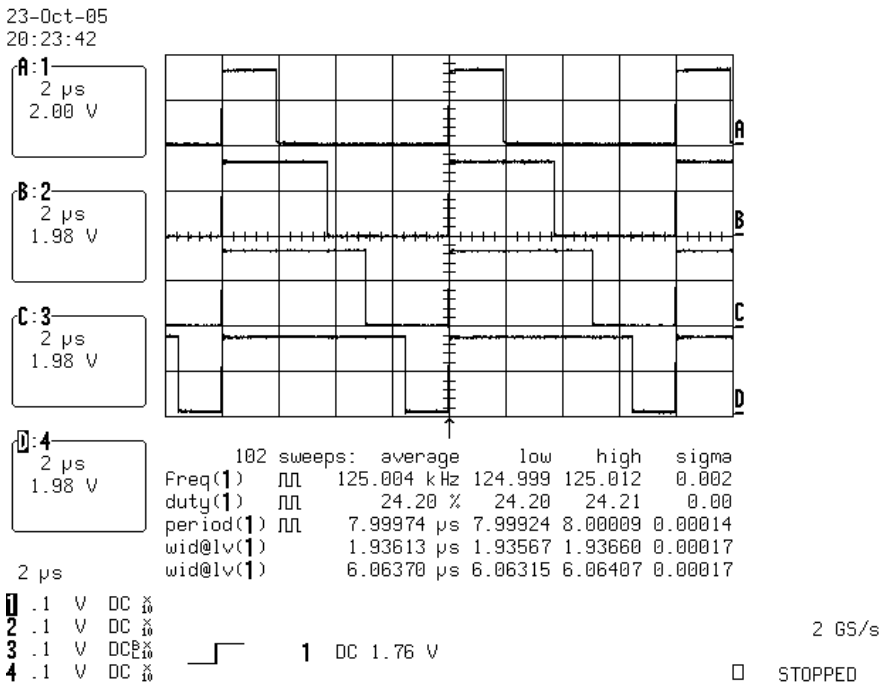



Figure 9: Example 9(c) waveforms. Example 9(c) uses assembler coded ISRs that then call C-coded functions. EINT0 is setup as a vectored IRQ, while EINT[1:3] are setup as non-vectored.

```
    ldmfd sp!, {r0-r3, ip, pc}^

irq_addr: .word irq_isr
irq_isr:
    sub   lr, lr, #4
    stmfd sp!, {r0-r3, ip, lr}
    bl irq_handler
    ldmfd sp!, {r0-r3, ip, pc}^
```

The FIQ vector service routine starts at its interrupt vector location, so the processor starts in the FIQ ISR when an FIQ interrupt occurs. The IRQ vector loads the program counter with the address of the IRQ assembler service routine, which is located just after the FIQ ISR. The FIQ ISR adjusts the link register address, saves it, along with the ARM Procedure Calling Standard (APCS) 'scratch' registers [3], to the stack, and calls the C-coded FIQ handler. When the handler returns, the FIQ IRQ performs a return-from-interrupt sequence. The IRQ ISR is similarly coded.

The reason for saving the APCS registers can be seen by disassembling the Example 9(c) code via `arm-elf-objdump -d ex9c.elf`. Looking at the code for `irq_handler` shows the prolog of that code saving several registers (not the ones saved by the ISR), and then the handler code uses `r0-3`. If the ISR code did not save these registers, then whatever was in those registers prior to the interrupt would be corrupted.

Figure 9 shows the waveforms from Example 9(c). The code is similar to Example 9(a), however, EINT0 is configured as a vectored IRQ, while EINT[1:3] are all configured as non-vectored. The example code provides details on how to setup the VIC. Comparison of Figure 7 to Figure 9 shows the increase in interrupt processing time caused by the implementation of Example 9(c).

## 3.10 Example 10: Interrupt nesting

The Philips LPC-series ARM microcontroller vectored interrupt controller (VIC) prioritizes interrupts under the condition of multiple interrupts occurring simultaneously, and it also prioritizes interrupts while interrupts are being serviced. When an interrupt occurs, the VIC modifies the interrupt enable state so that only higher-priority interrupts generate an IRQ to the processor core. This feature is not mentioned in the LPC2138 user manual (Chapter 5 [10]), but it is described in the ARM VIC PL190 documentation [2]. Without this feature, if IRQs were enabled to the core when a handler started, then any IRQ would interrupt the currently executing handler. The PL190 documentation also clarifies why you have to read from the `VICVectAddr` register, and then write to it once an interrupt has been serviced (p2-2 [2]);

> Reading from the Vector Interrupt Address Register, VICVECTADDR, provides the address of the ISR, and updates the interrupt priority hardware that masks out the current, and any lower priority interrupt requests. Writing to the VICVECTADDR Register indicates to the interrupt priority hardware that the current interrupt is serviced, enabling lower priority or the same priority interrupts to be removed, and for the interrupts to become active to go active.

To allow interrupts of higher priority to interrupt an interrupt handler requires re-enabling IRQ interrupts to the processor core. The generation of an IRQ to the core while in IRQ mode will overwrite the IRQ mode saved program status register (`SPSR_irq`) and any return address currently in the link register (`LR_irq`). To avoid corrupting the IRQ mode state by re-enabling IRQ interrupts, a nested interrupt handler needs to change processor modes when enabling IRQ interrupts; ARM recommends changing to the system-mode (the privileged version of user-mode). Philips application note AN10381 [9] gives two examples of the implementation of interrupt nesting; a version implemented using assembler coded prolog and epilog code, containing a C-coded handler function, and a version implemented using a C-compiler generated interrupt handler containing inline-assembler code to perform the additional steps required to implement nesting.

Interrupt nesting can utilize an important feature provided by the VIC hardware prioritization logic; that reading the `VICVectAddr` register adjusts the interrupt enable mask to allow interrupts of *higher-priority*. This feature results in an alternative implementation of the nesting entry sequence demonstrated in AN10381, i.e., for a specific handler loaded into the VIC address registers, the entry sequence is (see the full list in Section 3.1 in reference [9])

1. Save IRQ context.
2. Clear the interrupt source.
3. Switch to system-mode and enable IRQ.
4. Save SYS context.
5. ...

Since the handler function is executing as a consequence of the IRQ vector reading the `VICVectorAddr` register (via the statement `ldr pc, [#-0xFF0]` at the IRQ vector location), the currently executing IRQ priority has *already been masked* and can not generate another IRQ to the processor core. This means that the clearing of the interrupt source can be moved into the handler code. The consequence of that modification is that all handler prolog and epilog code is identical, so it can be replaced by a single assembler coded sequence, and all handler functions loaded into the VIC are simple C-functions. This improves the portability of the code, as compiler-specific interrupt handling keywords are no longer required.

Nested interrupts can be implemented by having the IRQ vector jump to a routine that performs the following sequence;

1. Save IRQ context; by adjusting the link register, and saving the APCS registers (`r0-3, ip`), work registers `r4-6`, and the link register to the IRQ mode stack.
2. Save the IRQ mode SPSR into `r4`.
3. Read the handler address from the `VICVectAddr` register into r6 (causing the VIC to mask the current interrupt).
4. Write to the CPSR to switch to system mode with IRQ interrupts enabled.
5. Save the system/user-mode link register to the user-mode stack.
6. Call the C-coded handler function.
7. Restore the system/user mode link register.
8. Write to the CPSR to switch to IRQ mode with IRQ interrupts disabled.
9. Restore the SPSR.
10. Acknowledge the interrupt to the VIC.
11. Restore IRQ mode saved context, and return from the IRQ.

The contents of the work registers `r4-6` are preserved across calls, so their contents can be loaded in the prolog code, and reused in the epilog code.

The nested IRQ assembler code is;

```
nested_irq_isr:
    /* (1) Save IRQ context, including the APCS registers, and r4-6 */
    sub   lr, lr, #4
    stmfd sp!, {r0-r6, ip, lr}

    /* (2) Save the SPSR_irq register */
    mrs r4, spsr

    /* (3) Read the VICVectAddr */
    ldr r5, VICVECTADDR
    ldr r6, [r5]

    /* (4) Change to SYS mode and enable IRQ */
    msr cpsr_c, #SYS_MODE

    /* (5) Save the banked SYS mode link register */
    stmfd sp!, {lr}

    /* (6) Call the C-coded handler */
    mov lr, pc
    ldr pc, r6

    /* (7) Restore SYS mode link register */
    ldmfd sp!, {lr}
```

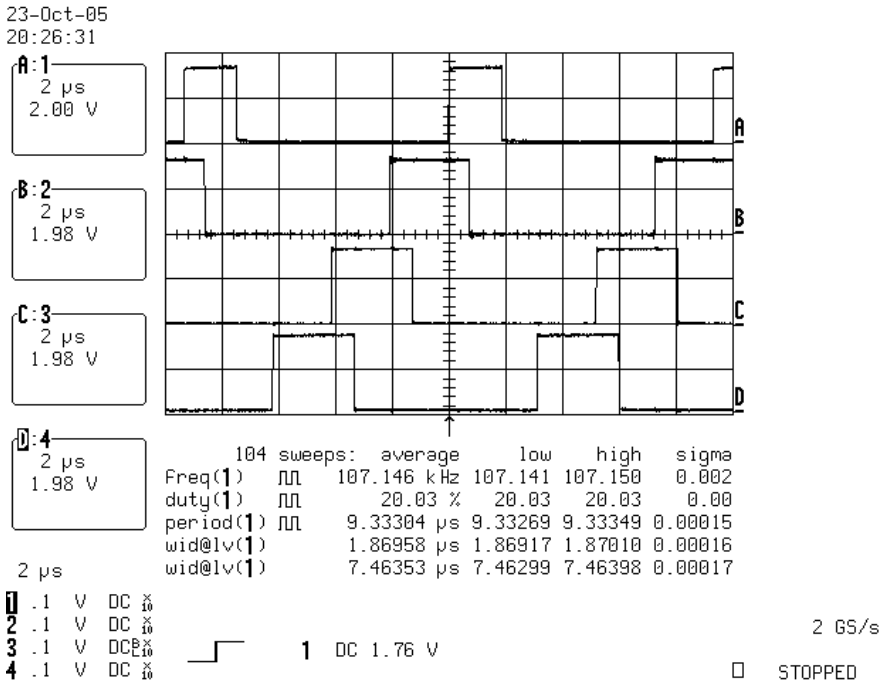
```
    /* (8) Change to IRQ mode and disable IRQ */
    msr cpsr_c, #IRQ_MODE|IRQ_DISABLE

    /* (9) Restore the SPSR */
    msr spsr, r4

    /* (10) Acknowledge the VIC */
    mov r0, #0
    str r0, [r5]

    /* (11) Restore IRQ context and return from interrupt */
    ldmfd sp!, {r0-r6, ip, pc}^
```

Comparing this to the non-nested IRQ ISR

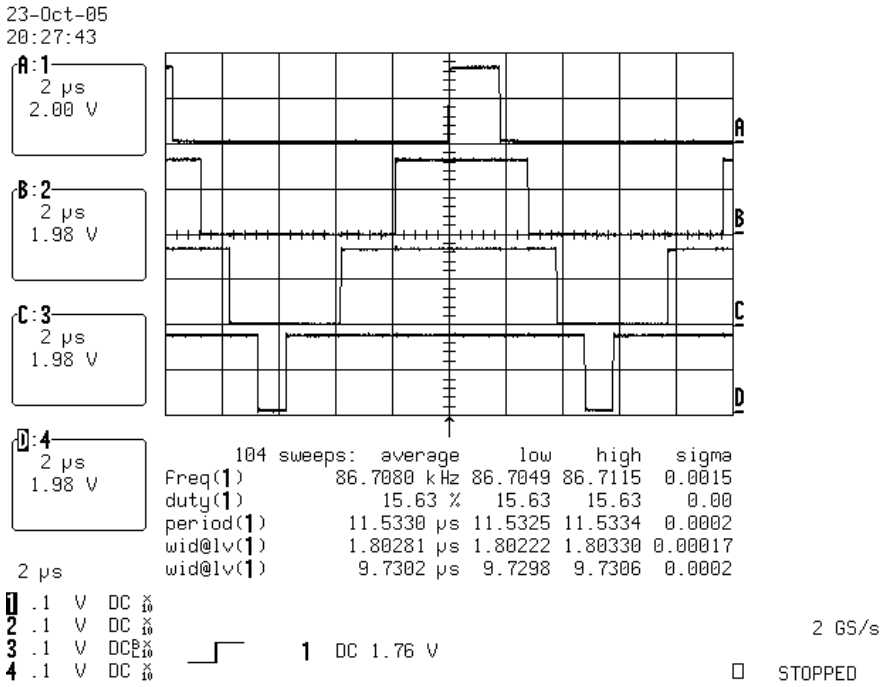
```
nonnested_irq_isr:
    sub   lr, lr, #4
    stmfd sp!, {r0-r3, ip, lr}
    bl irq_handler
    ldmfd sp!, {r0-r3, ip, pc}^
```

clearly shows the additional steps required to implement nesting!

The main application in Example 10 implements the same `EINT[0:3]` generation sequence as Example 9(b), but does not use the `interrupt` keyword in its handler definitions (as demonstrated by Example 9(c)). Example 10(a) links against an IRQ ISR that does not implement interrupt nesting (essentially repeating Example 9(b)), while Example 10(b) links against an IRQ ISR that implements nesting (i.e., Example 10(b) does not call the function `irq_handler()`). Figure 10 shows the waveforms from the two tests. Figure 10(b) shows that successive higher-priority interrupts successively interrupt the currently executing handler.




(a)




(b)

Figure 10: LPC2138 nested interrupt handling. Example 10 starts by generating a software interrupt on EINT3, and then each IRQ handler generates a software interrupt on the next higher-priority interrupt (except for EINT0, that handler leaves it to main to restart the process). Figure (a) used a IRQ ISR that does *not* implement interrupt nesting, while Figure (b) used a IRQ ISR that implements interrupt nesting.

# 4 $\mu$COS-II RTOS

The MicroC/OS-II or $\mu$COS-II real-time operating system (RTOS) was developed by Jean Labrosse for use in embedded systems such as microcontrollers and DSPs. The RTOS and methods for writing device drivers for it are covered in his two books; *MicroC/OS-II: The Real-Time Kernel* [6], and *Embedded Systems Building Blocks: Complete and Ready-to-Use Modules in C* [8]. The second edition of the RTOS book covers version 2.52 of the RTOS. The books contain the RTOS source code, and the RTOS can be used free-of-charge in university projects. The current commercial release of the RTOS is version 2.7x. The web site `www.micruim.com` contains additional resources, and ports for various processors.

Porting $\mu$COS-II version 2.52 is covered in Chapter 13 of *MicroC/OS-II: The Real-Time Kernel*, 2nd Ed [6]. A $\mu$COS-II port requires the definition of the data types on the processor, assembly language routines for critical section protection, interrupt handling, and context switching, and the definition of C coded hook functions. Table 13.1 on p289 [6] summarizes the porting requirements. The main effort involved in porting $\mu$COS-II is to determine the processor programming model, the calling conventions of the compiler, and servicing of interrupts. Earlier sections of this document developed this knowledge, so the port of $\mu$COS-II is now straightforward.

## 4.1 ARM-GCC port description

A task in $\mu$COS-II is defined as a function call of the form;

```
void task(void *pdata);
```

where `pdata` is a pointer that can be used to pass information to a task. Tasks start out life as if they were just interrupted at the entry of a call to their task function. The port-specific C-function `OSTaskStkInit()` is responsible for creating an appropriate initial stack.

Figure 11 shows the $\mu$COS-II ARM task initial stack context. The ordering of registers on the stack is as per the load/store multiple instruction format; registers with lower numbers are stored at lower addresses (higher on the stack in the figure) (pp60-63 of Furber's book has a nice description of the load/store multiple instructions [5]). Given this stack layout, with the processor in user/system mode, with the user/system mode stack pointer set to the task context, the state of the task can be restored using the sequence;

```
/* Copy the task CPSR to the CPSR register */
ldmfd sp!, {r0}
msr cpsr, r0

/* Restore task state (using load multiple) */
ldmfd sp!, {r0-r12, lr, pc}
```

Note that unlike the sequence used to return from an interrupt, this load multiple instruction does not end with a caret, '^', so the SPSR is not copied to the CPSR when the program counter is loaded (in fact, since you are in system mode, there is no SPSR and so that form of the instruction is illegal (p131 [5])). Another consequence of this choice of return sequence is that since the CPSR is loaded prior to loading the task registers, FIQ and IRQ interrupts will be enabled just prior to restoring the task state (this is fine though).

Most of the registers shown in Figure 11 can take on arbitrary values when an initial task context switch occurs. To aid in debugging, the registers are loaded with hexidecimal values that match their decimal register register numbers packed into a byte, and repeated four times (since debuggers often display the register contents in hexidecimal). The task argument `pdata` is placed on the stack at the location of `r0`, while the address of the task function is placed on the task stack at the location of

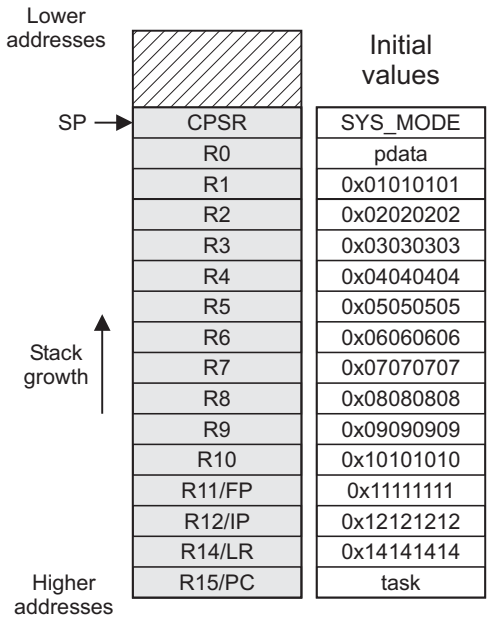



Figure 11: $\mu$COS-II ARM task initial stack context.

the program counter `pc`. The current program status register (CPSR) value is set to system mode, with FIQ and IRQ interrupts enabled.

The return address located at the link register location in the stack frame shown in Figure 11 would be used if the task function was ever returned from. Given the link-register value shown in Figure 11, the processor will almost certainly abort, so it can be useful to place the address of an exit handler on the stack in that location. The exit handler can log the fact that a task exited (when it probably should not have).

The Micrium web site hosts a number of ports for the ARM processor. The ports page can be accessed from the Micrium home page at `http://www.micrium.com`. The address of the ARM ports page was `http://www.micrium.com/contents/products/ucos-ii/ports-arm.html` when this document was written. The Micrium ARM-mode $\mu$COS-II port is described in Application note AN-1011 (revision D). The port is for the IAR compiler. Several other application notes apply the generic ARM port to specific processors. A port of the AN-1011 port to the GCC compiler is provided with the source code associated with this document.

The Micrium ARM ports do *not* support interrupt nesting; even for the case of an FIQ interrupt occurring during IRQ interrupt processing (the IRQ interrupt service routine, `OS_CPU_IRQ_ISR`, in `os_cpu_a.s` calls the handler function with both IRQ and FIQ interrupts disabled).

The ARM port described in the following sections implements interrupt nesting of IRQ interrupts by FIQ interrupts, and higher-priority IRQ interrupts. The assembler implementation of the port manipulates registers specific to the ARM vectored interrupt controller (VIC). Porting this code for a different interrupt controller would be fairly simple.

The only difference between the AN1011 source code and the code for this port is in the assembly file. The two versions of the assembler files are laid very similarly (in the style of the original AN1011 source). The following sections describe the port files.

Note that the $\mu$COS-II source code is not free, so it is not supplied with this document. The version of the the $\mu$COS-II source code used to test this port was version 2.52; the source provided with the 2nd edition of the Labrosse book.

### 4.1.1 Port header; os_cpu.h

**Critical section protection**

The ARM-GCC port uses OS critical section protection method #3; it defines a function for saving the processor status while disabling FIQ and IRQ interrupts, and another to restore the processor status. The function declarations and critical section macros are located in os_cpu.h;

```
OS_CPU_SR  OS_CPU_SR_Save(void);
void       OS_CPU_SR_Restore(OS_CPU_SR cpu_sr);

#define  OS_ENTER_CRITICAL()  {cpu_sr = OS_CPU_SR_Save();}
#define  OS_EXIT_CRITICAL()   {OS_CPU_SR_Restore(cpu_sr);}
```

and the function implementations are in os_cpu_a.s. The implementation of the OS_CPU_Save_SR() function is based on the recommendations in Atmel's ARM processor application note *Disabling Interrupts at Processor Level* [4].

**Task-level context switch**

The task-level context switch macro, OS_TASK_SW(), is defined as a call to OSCtxSw() (see the OS_CPU_A.ASM section.

### 4.1.2 Port C-functions; os_cpu_c.c

The only C function the port needed to define was OSTaskStkInit() to initialize the stack as shown in Figure 11.

### 4.1.3 Port assembler-functions; os_cpu_a.s

A port requires the implementation of four assembler routines; OSStartHighRdy (start multi-tasking), OSCtxSw (task-level context switch), OSIntCtxSw (interrupt-level context switch), and OSTickISR (time-tick ISR).

**Start multi-tasking**

OSStartHighRdy() is called at the end of OSStart() (in $\mu$COS-II source file OS_CORE.C), and is the exit point from main()'s context into the RTOS. OSStartHighRdy() implements the context restore of the registers shown in Figure 11. The function starts by ensuring that the processor is in user/system mode with FIQ and IRQ interrupts disabled (although having the interrupts disabled is not critical, as there should be no interrupt generating sources setup at this point). The OSTaskSwHook() function is then called, and the OSRunning flag set to true. The user/system mode stack pointer is then changed to that of the highest-priority (and only) task. The task CPSR is then copied into the CPSR register (which happens to enable FIQ/IRQ interrupts), and the task register context is restored.

**Task-level context switch**

OS_Sched() (OS_CORE.C) calls OS_TASK_SW() to implement a task-level context switch from inside a critical section (so both FIQ and IRQ are disabled when this function is called). The macro OS_TASK_SW() is a call to OSCtxSw() in this port, so on entry to the context switch function, the link register will contain the task return address. The job of OSCtxSw() is to save the current task context, switch over to the higher-priority task, and then restore context. The code saves the current tasks registers onto its stack as shown in Figure 11; the contents of link register is saved to both the link register and the program counter locations on the stack. The task stack-pointer is then saved to its task control block, the OSTaskSwHook() function is called, the higher-priority task stack is loaded, and the context of the higher-priority task is restored.

**Interrupt-level context switch**

The FIQ and IRQ ISRs start by saving the processor context, incrementing the `OSIntNesting` counter (and saving the current value of the stack pointer if required), and the IRQ ISR then re-enables IRQ interrupts. The ISR then calls handler code (written in C). When the handler returns, the ISR calls `OSIntExit()`, and then restores the processor state.

`OSIntExit()` (`OS_CORE.C`) checks to see if interrupt nesting is over, and then if a higher-priority task is ready. If interrupts are still nested, or the same task has the highest priority, then `OSIntExit()` returns, and the ISR runs to completion (i.e., performs the context restore of the task or interrupt it interrupted). If however, interrupt nesting is over, and a higher-priority task has been made ready, then a switch to the new task is required; that is the job of `OSIntCtxSw()`. `OSIntExit()` calls `OSIntCtxSw()` inside a critical section, so interrupts are disabled when this function is called.

The interrupt-level context switch code is similar to the task-level context switch code, except that the ISR has already done the work of saving the processor context to the task stack. `OSIntCtxSw()` starts by calling the `OSTaskSwHook()`, the higher-priority task stack is then loaded, and the context of the higher-priority task is restored.

**Interrupt service routines (ISRs)**

The FIQ and IRQ ISRs are setup to call C-coded handlers. It is up to the board-support package to decide where to call the OS function `OSTimeTick()`. For example, timer 0 can be setup to generate clock ticks and the VIC can be setup to generate an FIQ (for testing), or as an IRQ (a vectored interrupt would be recommended).

When an FIQ interrupt occurs, the ISR performs a partial context save (since the stack pointer is currently that of the FIQ, not the system mode task stack), and the processor is placed into system mode with interrupts disabled. The task context is then saved to the task stack. The interrupt nesting counter is then incremented, and if this is the first layer of nesting, the current value of the stack-pointer is saved to the task control block. The processor is then changed back to FIQ mode with interrupts disabled, and the FIQ handler function is called. After the handler returns, the processor is moved back to system mode, `OSIntExit()` is called, and the task context is restored. FIQ interrupts are not nested.

When an IRQ interrupt occurs, the ISR performs a partial context save (since the stack pointer is currently that of the IRQ, not the system mode task stack), and the processor is placed into system mode with interrupts disabled. The task context is then saved to the task stack. The interrupt nesting counter is then incremented, and if this is the first layer of nesting, the current value of the stack-pointer is saved to the task control block. The VIC vector address register is then read. The VIC vector address register returns the address of the IRQ handler, and triggers the VIC priority logic to only allow IRQ interrupts of higher-priority to interrupt the processor core. FIQ and IRQ interrupts are then enabled (with the processor left in system mode), and the handler function read from the VIC is called. After the handler returns, FIQ and IRQ interrupts are disabled, and the VIC is acknowledged by writing to the VIC vector address register. `OSIntExit()` is called, and the task context is restored.

### 4.1.4 Board-support package; `BSP.H,.C`

The port assembly language file `os_cpu_a.s` defines an FIQ interrupt service routine that calls a C-coded handler; that handler needs to be supplied as part of the board-support package, or the function needs to be defined in the user application. The minimal form of the handler is;

```
void OS_CPU_FIQ_ISR_Handler(void)
{
    return;
}
```

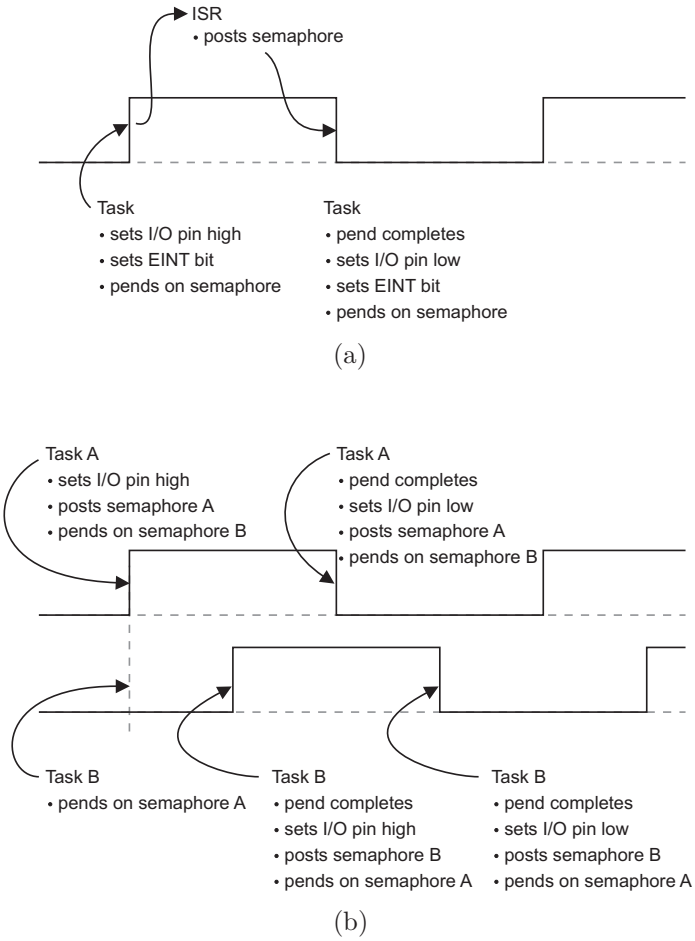



Figure 12: $\mu$COS-II ARM port testing. (a) task-to-ISR context switching, and (b) task-to-task context switching.

The port of the AN-1011 code requires a similar function defined for the IRQ ISR handler. The port described by this document accesses the IRQ ISR from the VIC vector address register, so does not require linking with an IRQ handler function. The BSP should also contain an initialization routine to setup a timer that generates an FIQ or IRQ interrupt, and a corresponding handler that calls the `OSTimeTick()` routine.

The example programs supplied with this document contain a minimal board support package containing timer setup, and LED control functions.

## 4.2 Port testing

This section presents test results from the AN-1011 ARM $\mu$COS-II port and the ARM nested-interrupts version presented in this document.

### 4.2.1 Test 1: Task-to-IRQ context switching

Figure 12(a) shows the sequence of a task-to-ISR test. A test application was written containing a single task, and an interrupt handler for `EINT0`. The task sets an I/O pin high, triggers an `EINT0`

interrupt, then pends on a semaphore. The interrupt handler posts a semaphore (an I/O pin is toggled while in the ISR). The task receives the semaphore, sets an I/O pinlow, triggers an `EINT0` interrupt, then pends on a semaphore. This sequence is repeated in a while loop.

Figure 13 shows the results of the task-to-ISR context switch testing for two ARM ports.

The time between the rising edge of the tasks LED to that of the ISR handler is 3.5$\mu$s for the AN-1011 port, and 2.7$\mu$s for the nested interrupts port (due to its use of load/store multiple instructions). The total task-to-ISR-to-task time is 13.0 and$\mu$s 11.7$\mu$s. Both ports can perform approximately 40,000 context switches per second.

#### 4.2.2 Test 2: Task-to-task context switching

Figure 12(b) shows the sequence of a task-to-task test. A test application was written containing two tasks, task A and task B. Task A posts semaphore A and then pends on semaphore B. Task B does the opposite, it pends on semaphore A, and posts semaphore B.

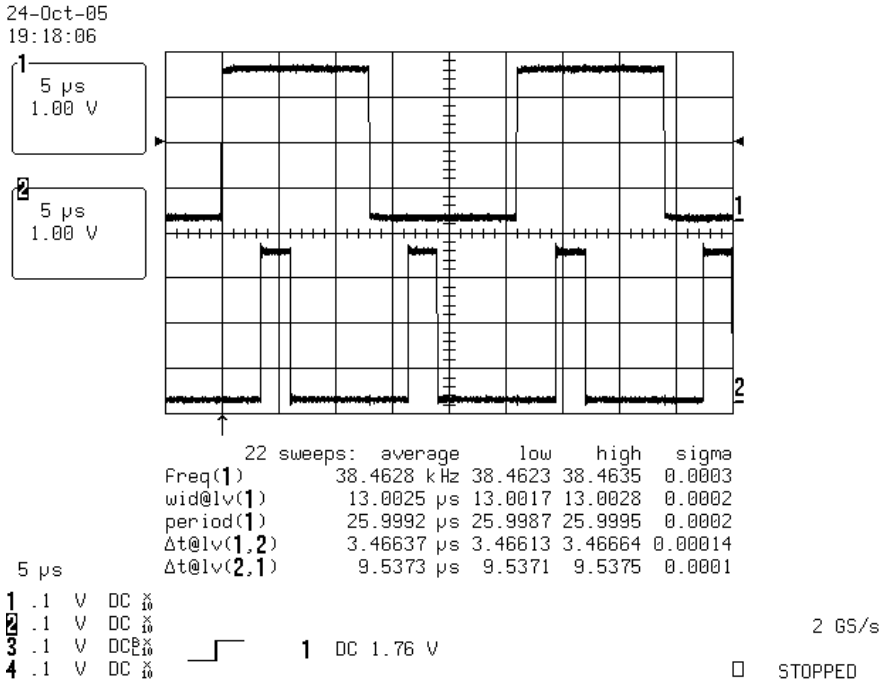
When Task A pends on semaphore B, it gives up the processor, and causes a task-level context switch to task B (task B is now ready, since task A posted the semaphore it was waiting for). The time between the rising-edge of the I/O pin toggled by task A, to the rising-edge of the I/O pin toggled by task B, is the time taken for a task-to-task context switch. Figure 14 shows the results of the task-to-task context switch testing for two ARM ports. The rising-edge to rising-edge time is about 13$\mu$, and both square waves have a frequency of around 20kHz, i.e., around 40,000 context switches per second occur.

#### 4.2.3 Test 3: IRQ-FIQ interrupt nesting

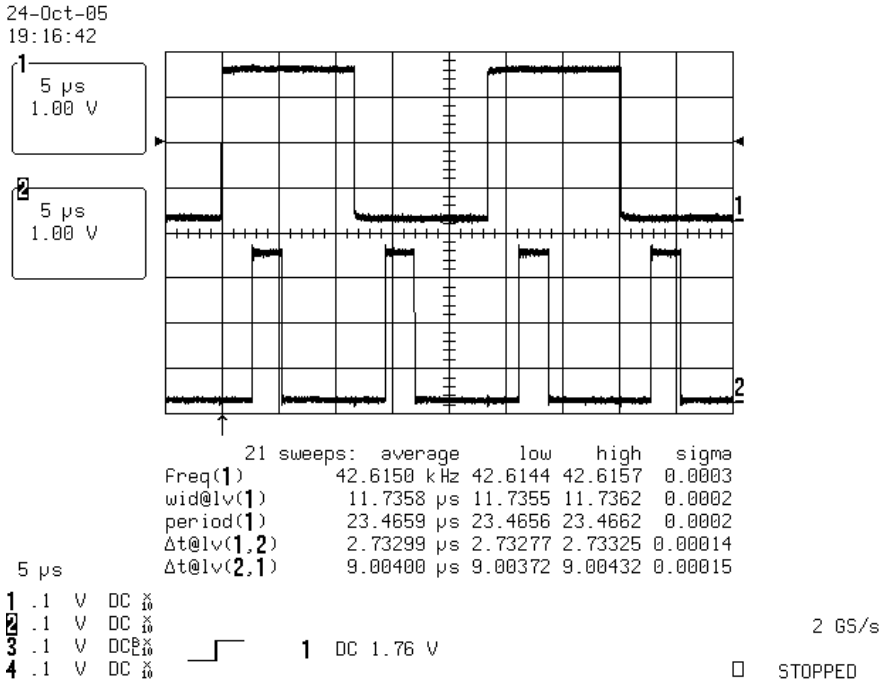
Figure 15 demonstrates the IRQ nesting feature of the port presented in this document relative to the AN-1011 port. A task triggers an EINT1 IRQ interrupt which triggers a higher priority EINT0 FIQ interrupt. In Figure 15(a) the EINT1 handler finishes before the higher-priority EINT0 handler, as the AN-1011 port does not implement IRQ-FIQ interrupt nesting. However, in Figure 15(b) the EINT1 handler is interrupted by the EINT0 handler.

#### 4.2.4 Test 4: IRQ interrupt nesting

Figure 16 demonstrates the IRQ nesting feature of the port presented in this document relative to the AN-1011 port. A task triggers an EINT1 interrupt which triggers a higher priority EINT0 interrupt. In Figure 16(a) the EINT1 handler finishes before the higher-priority EINT0 handler, as the AN-1011 port does not implement interrupt nesting. However, in Figure 16(b) the EINT1 handler is interrupted by the EINT0 handler.

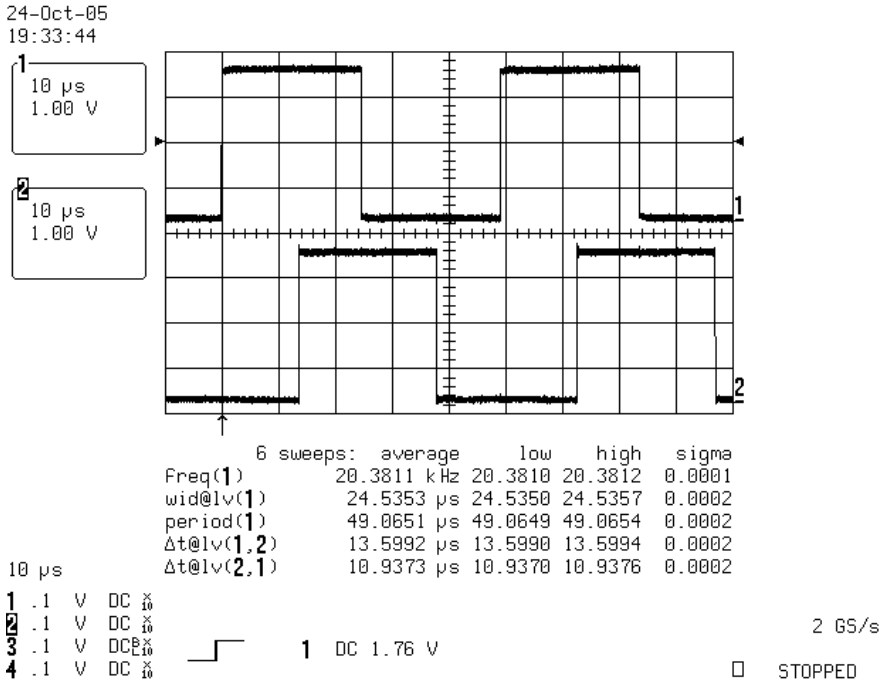



(a)

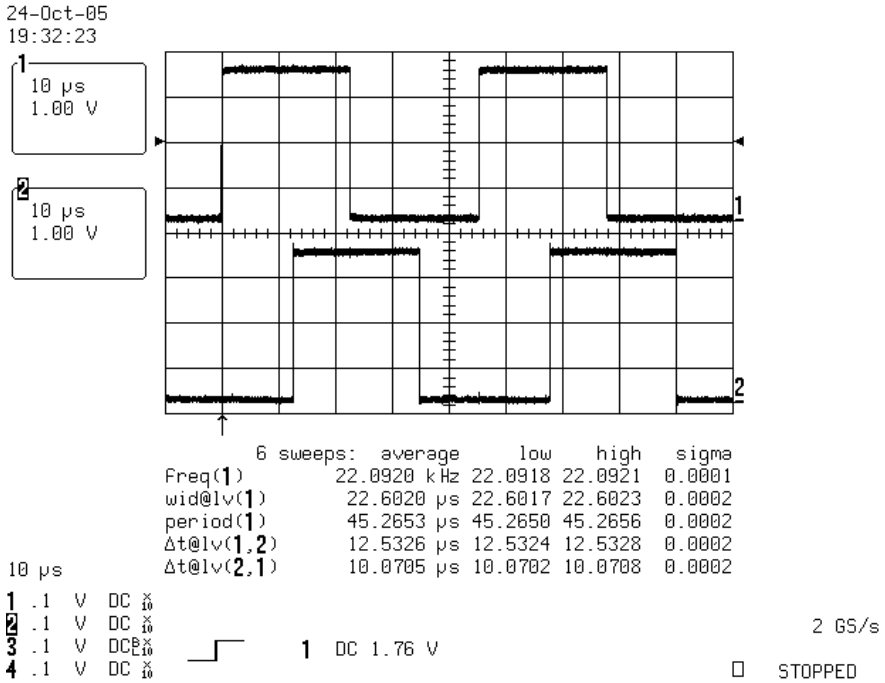



(b)

Figure 13: $\mu$COS-II ARM task-to-ISR context switch testing; (a) for AN-1011 port, and (b) the nested interrupts port.

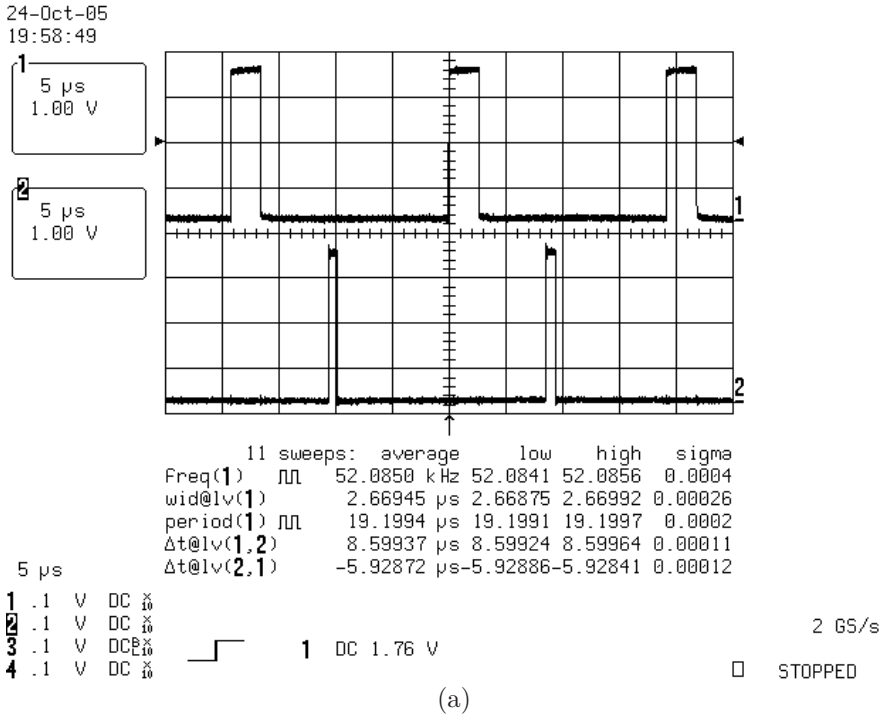



(a)

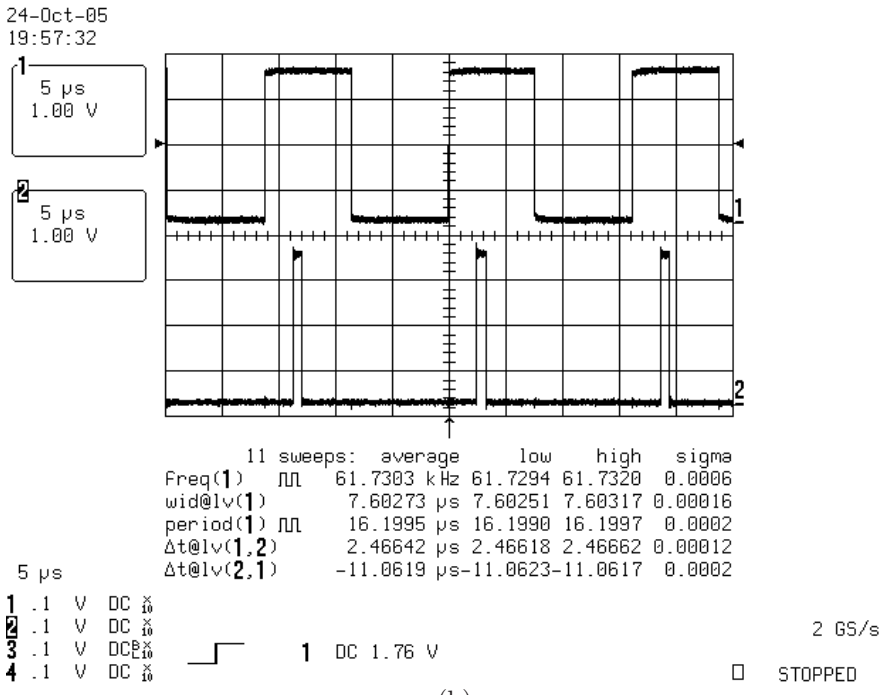



(b)

Figure 14: $\mu$COS-II ARM task-to-task context switch testing; (a) for AN-1011 port, and (b) the nested interrupts port.

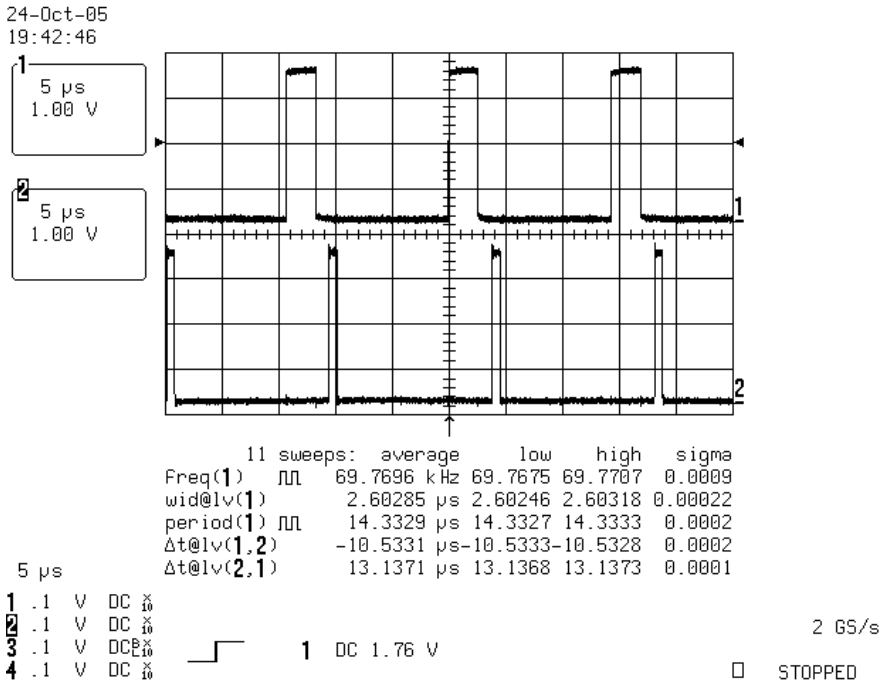



(a)

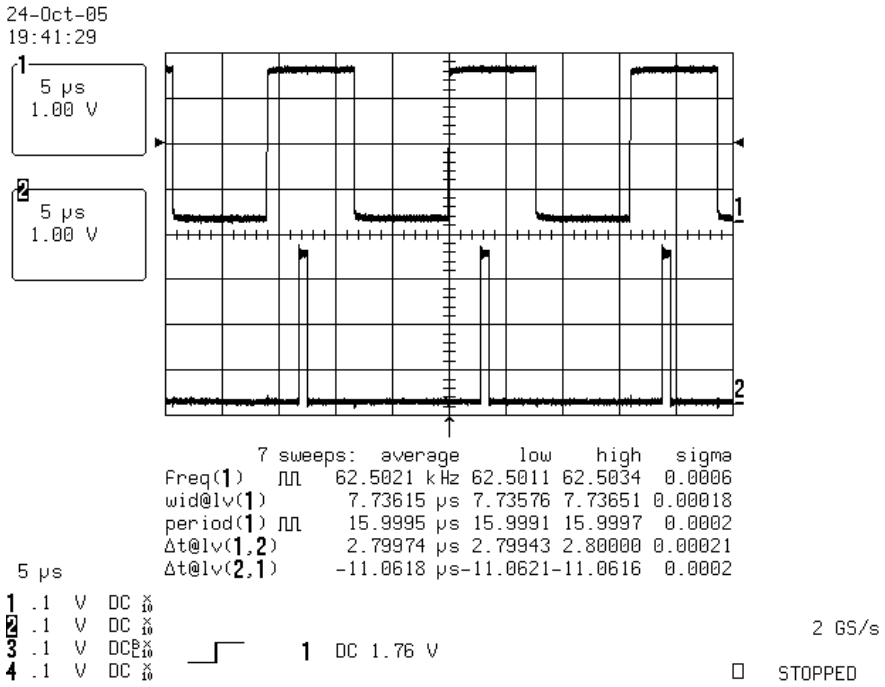



(b)

Figure 15: $\mu$COS-II ARM IRQ-FIQ nesting testing; a task triggers an EINT1 IRQ interrupt which triggers a higher priority EINT0 FIQ interrupt. The top trace is an I/O pulsed in the EINT1 handler, while the bottom trace is the EINT0 handler. Figure shows the waveform (a) for the AN-1011 port, and (b) for the nested interrupts port. Note that in (a) the EINT1 IRQ handler finishes before the higher-priority EINT0 FIQ handler, as the AN-1011 port does not implement IRQ-FIQ interrupt nesting.




(a)




(b)

Figure 16: $\mu$COS-II ARM IRQ nesting testing; a task triggers an EINT1 interrupt which triggers a higher priority EINT0 interrupt. The top trace is an I/O pulsed in the EINT1 handler, while the bottom trace is the EINT0 handler. Figure shows the waveform (a) for the AN-1011 port, and (b) for the nested interrupts port. Note that in (a) the EINT1 handler finishes before the higher-priority EINT0 handler, as the AN-1011 port does not implement interrupt nesting.

## 4.3 uCOS-II examples

### 4.3.1 Example 1: Blinking LEDs

The $\mu$COS-II example 1 program creates two tasks. Each task controls four LEDs. The first task rotates its four LEDs every 1s, while the second task rotates its LEDs every 250ms. The delays are implemented using OS delay functions.

### 4.3.2 Example 2: Serial port echo console

The $\mu$COS-II example 1 program creates a task that listens on a serial port and echos what is typed, and another task that blinks LEDs.

Download the program (eg. using FlashUtils), and connect to the other serial port on the board 115200-baud, and you will be greeted with the following message;

```
----------------------------------
Welcome to uCOS-II!
----------------------------------

This application will echo characters received.
When a line of text is received on the serial port,
one of four LEDs (LED[0:3]) is blinked on the MCB2130.
A second task rotates the other four LEDs (LED[4:7]) every 250ms.

uCOS-II>
```

The serial port driver included with the example is interrupt driven and uses the $\mu$COS-II OS to cause the task controlling the serial port to block appropriately.

I'd like to explain in mode detail, but I have simply run out of time, its now time to upload this entry!

# A ARM GCC

The GNUARM web site `www.gnuarm.com` contains pre-built binaries of the GCC compiler for ARM systems. The tools used at the time of writing of this document consisted of;

- `binutils-2.15.tar.bz2`
- `newlib-1.12.0.tar.gz`
- `gcc-3.4.3.tar.bz2`
- `insight-6.1.tar.gz`

The GNUARM binary tools can be used directly, or you can build them via the following instructions. Building the tools yourself is useful if you also want to build for other CPU architectures.

## A.1 Build procedure

The tools in this section were built in July 2005 under Cygwin on a Windows 2000 machine, and under Linux on a Red Hat 9.0 machine. Cygwin used gcc 3.4.4, while Linux used the default Red Hat gcc 3.2.2. The machine was a dual-boot HP Omnibook 6100 laptop (1GHz Pentium III-M with 512MB RAM).

Building `binutils` under Cygwin and Linux (bash shell syntax):

1. `export TARGET=arm-elf`
2. `export PREFIX=/opt/gnutools`
3. `tar -jxvf binutils-2.15.tar.bz2`
4. `mkdir binutils-build; cd binutils-build`
5. `../binutils-2.15/configure --target=$TARGET --prefix=$PREFIX --enable-interwork --enable-multilib`
6. `make`
7. `make install` (requires logging in as root under Linux)

Note: the first time I attempted to build binutils under Cygwin, it failed, as it could not find `lex`. So, be prepared to update and add tools to your Cygwin installation.

Building `gcc` and `newlib` under Cygwin and Linux:

1. `export TARGET=arm-elf`
2. `export PREFIX=/opt/gnutools`
3. `export PATH=$PREFIX/bin:$PATH`
4. `tar -jxvf gcc-3.4.3.tar.bz2`
5. `tar -zxvf newlib-1.12.0.tar.gz`
6. `cp t-arm-elf gcc-3.4.3/gcc/config/arm/`
   (this updated file is from the GNUARM site and sets up the multilib build)

7. `mkdir gcc-build; mkdir newlib-build; cd gcc-build`
8. `../gcc-3.4.3/configure --target=$TARGET --prefix=$PREFIX --enable-interwork --enable-multilib --with-float=soft --enable-languages=c,c++ --with-newlib --with-headers=../newlib-1.12.0/newlib/libc/include`
   Note: this step requires root privileges under Linux to copy the newlib headers into a subdirectory under `$PREFIX`.
9. `make all-gcc`
10. `make install-gcc`
11. `cd ../newlib-build`
12. `../newlib-1.12.0/configure --target=$TARGET --prefix=$PREFIX --enable-interwork --enable-multilib`
13. `make`
14. `make install`
15. `cd ../gcc-build`
16. `make`
17. `make install`

Note that the configure options for `gcc` used here are slightly different than those stated on the GNUARM web site. If you install the GNUARM tools, and look in the `arm-elf-gccbug` script, it contains the configure command line used to build those tools;

```
arm-elf-gccbug(349): configured with:
../gcc-3.4.3/configure --target=arm-elf --prefix=/c/gnuarm-3.4.3
--enable-interwork --enable-multilib --with-float=soft
--with-newlib --with-headers=../newlib-1.12.0/newlib/libc/include
--enable-languages=c,c++,java --disable-libgcj
```

So relative to the build instruction on the GNUARM web site, the GNUARM tools need the option `--with-float=soft`, and the binary version of the tools also have Java enabled.

Building `insight` (which includes `gdb`) under Cygwin and Linux:

1. `export TARGET=arm-elf`
2. `export PREFIX=/opt/gnutools`
3. `tar -jxvf insight-6.1.tar.gz`
4. `mkdir insight-build; cd insight-build`
5. `../insight-6.1/configure --target=$TARGET --prefix=$PREFIX --enable-interwork --enable-multilib`
6. `make`
7. `make install`

The build failed under Cygwin. The errors were related to linker errors in `newlib-6.1/tcl/win/tclWin32Dll.c` and several other files in that Windows-specific directory. The problem might be that gcc 3.4.4 is optimizing away functions that are only referred to inside inline assembler (and hence the compiler believes they are unused). The problem was not investigated, since a binary version was available, and it worked fine under Linux.

**What is a multilib?**

Multilib enables the building of the different libraries required to link against code compiled with different command lines options. For example, processors without a floating-point unit (FPU) require the `--with-float=soft` option to trigger the use of software-implemented floating-point, whereas a processor with an FPU can use floating-point hardware instructions. The multilibs compiled can be printed using `arm-elf-gcc -print-multi-lib` (Reference: p37 configure.pdf from GNUARM web site).

For the GNUARM binary installation, the multilibs are;

```
$ /gnuarm/bin/arm-elf-gcc -print-multi-lib
.;
thumb;@mthumb
be;@mbig-endian
fpu;@mhard-float
interwork;@mthumb-interwork
nofmult;@mcpu=arm7
fpu/interwork;@mhard-float@mthumb-interwork
fpu/nofmult;@mhard-float@mcpu=arm7
be/fpu;@mbig-endian@mhard-float
be/interwork;@mbig-endian@mthumb-interwork
be/nofmult;@mbig-endian@mcpu=arm7
be/fpu/interwork;@mbig-endian@mhard-float@mthumb-interwork
be/fpu/nofmult;@mbig-endian@mhard-float@mcpu=arm7
thumb/be;@mthumb@mbig-endian
thumb/interwork;@mthumb@mthumb-interwork
thumb/be/interwork;@mthumb@mbig-endian@mthumb-interwork
```

If you built gcc and newlib without copying the altered `t-arm-elf` file into the gcc source, then the build occurs relatively quickly, and the multilibs are;

```
$ /opt/gnutools/bin/arm-elf-gcc -print-multi-lib
.;
thumb;@mthumb
```

i.e., there are no big-endian, interwork, or hardware floating-point multilibs. Copying the `t-arm-elf` file into the gcc source, gives the same multilib output as the GNUARM binary.

# References

[1] ARM. ARM7TDMI-S Technical Reference Manual (Revision 4.3). Reference Manual, 2001. (www.arm.com).

[2] ARM. PrimeCell Vectored Interrupt Controller (PL190) (revision r1p2). Reference Manual (DDI 0181E), 2004. (www.arm.com).

[3] ARM. Procedure call standard for the ARM architecture. Application Note (GENC-003524), 2005. (www.arm.com).

[4] Atmel. Disabling interrupts at Processor Level. Application Note (DOC1156A-08/98), 1998. (www.atmel.com).

[5] S. Furber. *ARM system-on-chip architecture*. Addison-Wesley, 2nd edition, 2000.

[6] J. Labrosse. *MicroC/OS-II: The real-time kernel*. CMP Books, 2nd edition, 2002.

[7] J. Labrosse. MicroC/OS-II and the ARM processor. Micrium Application Note AN-1011 (Revision D), 2004. (www.micrium.com).

[8] J. J. Labrosse. *Embedded Systems Building Blocks: Complete and Ready-to-Use Modules in C*. CMP Books, 2nd edition, 2000. (www.micrium.com).

[9] Philips. Nesting of interrupts on the LPC2000. Application Note, 2005. (www.philips.com).

[10] Philips. Volume 1: LPC213x User Manual. User Manual, 2005. (www.philips.com).

[11] W. Schwartz. Enhancing Performance Using an ARM Microcontroller with Zero Wait-State Flash. *Information Quarterly*, 3(2), 2004.

[12] D. Seal. *ARM Architecture Reference Manual*. Addison-Wesley, 2nd edition, 2000.

[13] A. N. Sloss, D. Symes, and C Wright. *ARM System Developer's Guide*. Morgan Kaufman, 2004.