

Experiment 5: Operating Modes, System Calls and Interrupts

This experiment further consolidates the programmer's view of computer architecture. It does this by giving you details of the ARM processor's operating modes and exceptions. This experiment also shows how you can interface to input/output devices using system calls and interrupts, two types of exceptions found on the ARM processor.

Aims

This experiment aims to:

- teach you the ARM processor operating modes,
- show how system calls can be made and handled using the `swi` instruction,
- give details of external interrupts and how to service them, and
- illustrate how I/O interfacing can be done efficiently using interrupt service routines.

Preparation

It is important that you prepare for each laboratory experiment, so that you can use your time (and your partner's time) most effectively. For this particular experiment, you should do the following *before* coming in to the Laboratory:

- read through this experiment in detail, trying to understand what you will be doing,
- quickly read through the relevant material from your lecture notes for this course,
- if you haven't already done so, read through *An Introduction to Komodo*, which you can find in an Appendix or on your CD-ROM,
- refresh your memory by quickly reading through Experiment 4 again, and
- skim through the *DSLUMU Microcontroller Board Hardware Reference Manual*, paying particular attention to the description of the IRQ Status and IRQ Enable ports. You can find this document in an Appendix or on your CD-ROM.

If you are keen (and you should be!), you could also:

- read through the *Hardware Reference Manual* in greater detail,
- look at the programming examples in the *examples/intro* directory on your CD-ROM, and
- type up or modify the necessary files in this experiment, to save time in class.

Getting Started

Once you arrive at the Laboratory, find a spare workbench and log into the Host PC. Next, create a new directory for this experiment. Then, copy all of the files in the directory `~elec2041/unswelec2041/labs-src/exp5` on the Laboratory computers into this new directory. You can do all of this by opening a Unix command-line shell window and entering:

```
mkdir ~/exp5
cd ~/exp5
cp ~elec2041/cdrom/unswelec2041/labs-src/exp5/* .
```

Be careful to note the `“.”` at the end of the `cp` command!

If you are doing this experiment at home, you will not have a `~elec2041` directory, of course. You should use the `unswelec2041/labs-src/exp5` directory on your CD-ROM instead.

The ARM Processor Operating Modes

The ARM processor has seven *processor operating modes*, as shown in Table 1. Each operating mode is used for a particular purpose; only one mode is in use at any one time:

Mode	Privileged	Purpose
User	No	Normal operating mode for most programs (tasks)
Fast Interrupt (FIQ)	Yes	Used to handle a high-priority (fast) interrupt
Interrupt (IRQ)	Yes	Used to handle a low-priority (normal) interrupt
Supervisor	Yes	Used when the processor is reset, and to handle the software interrupt instruction swi
Abort	Yes	Used to handle memory access violations
Undefined	Yes	Used to handle undefined or unimplemented instructions
System	Yes	Uses the same registers as User mode

Table 1: ARM Processor Operating Modes

Among other things, the operating modes shown in Table 1 define the registers that can be used (also called the *register map*) and the operating *privilege level*.

The ARM processor has a simple privilege model: all modes are privileged apart from User mode. *Privilege* is the ability to perform certain tasks that cannot be done from User mode. For example, changing the operating mode is a privileged operation.

In a system with *memory management*, only privileged modes have access to certain areas of the address space, such as memory used by the operating system, or to I/O devices. User programs are then run from User mode, which does not have such privileges. This means that such tasks cannot directly interfere with the hardware—a good thing when running *untrusted code*. Furthermore, since User mode cannot change the operating mode, user tasks cannot escape these restrictions.

Systems with memory management are not usually needed for an embedded environment; this experiment will only deal with the User, FIQ, IRQ and Supervisor modes of operation.

The ARM processor has a total of 37 registers: 31 general-purpose registers (including the Program Counter R15) and 6 status registers. These registers are shown in Figure 1:

General-purpose Registers and Program Counter

User	System	Fast Interrupt	Interrupt	Supervisor	Abort	Undefined
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8_fiq	R8	R8	R8	R8
R9	R9	R9_fiq	R9	R9	R9	R9
R10	R10	R10_fiq	R10	R10	R10	R10
R11	R11	R11_fiq	R11	R11	R11	R11
R12	R12	R12_fiq	R12	R12	R12	R12
R13 (SP)	R13 (SP)	R13_fiq	R13_irq	R13_svc	R13_abt	R13_und
R14 (LR)	R14 (LR)	R14_fiq	R14_irq	R14_svc	R14_abt	R14_und
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)

Program Status Registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_fiq	SPSR_irq	SPSR_svc	SPSR_abt	SPSR_und

Register indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the mode of operation.

Figure 1: ARM Processor Registers

As you can see from Figure 1, each processor mode has its own R13 and R14 registers. This allows each mode to maintain its own stack pointer and return address. In addition, the Fast Interrupt (FIQ) mode has additional registers: R8–R12. This means that when the ARM processor switches into FIQ mode, the software does not need to save the normal R8–R12 registers, as FIQ mode has its own set that can be modified.

The Current Program Status Register (CPSR) is used to store condition code flags, interrupt disable bits, the current processor mode and other status and control information. This register is depicted in Figure 2:

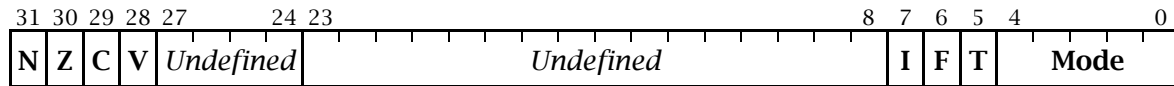


Figure 2: The Current Program Status Register

The Current Program Status Register is defined in the following way:

- Bits 24–31 can be modified in any mode, and are used to store the *condition code flags* (often just called *flags*). Only four condition code flags are available: N for Negative, Z for Zero, C for Carry and V for Overflow; the other bits are undefined. The condition code flags are set or cleared as a by-product of certain arithmetic instructions. For example, “`cmp r0, r1`” sets the Z (Zero) flag if R0 and R1 are equal.
- Bits 6 and 7 (F and I respectively) are the *interrupt disable* bits: setting one of these bits to 1 disables that interrupt; bit 6 disables the Fast Interrupt (FIQ), bit 7 disables the normal Interrupt (IRQ). These bits can only be modified in a privileged mode.
- Bit 5 (the T bit) determines whether the processor runs in ARM state or in Thumb state. Thumb state uses a different, more compact, instruction set when compared to ARM. You must never set this bit; doing so will make the processor enter an unpredictable state. This bit can only be modified in a privileged mode.
- Bits 0–4 set the processor mode; Table 2 shows the individual bit patterns needed to use a particular mode. These bits can only be modified in a privileged mode.
- Bits 8–27 are undefined and reserved for future or more advanced ARM processors. You should never alter the contents of these bits; instead, use a read-modify-write cycle (as explained in Experiment 4) to preserve them. These bits can only be modified in a privileged mode.

As mentioned above, bits 24–31, the condition code flags, can be modified in any mode. Bits 0–23 can only be modified in a *privileged* mode (ie, any mode other than User mode). Table 2 shows the individual bit patterns needed in bits 0–4 to use a particular mode:

Mode Bits		Processor Mode (Abbreviation)	Accessible Registers
Bin	Hex		
10000	10	User (usr)	PC, R14–R0, CPSR
10001	11	Fast Interrupt (fiq)	PC, R14_fiq–R8_fiq, R7–R0, CPSR, SPSR_fiq
10010	12	Interrupt (irq)	PC, R14_irq, R13_irq, R12–R0, CPSR, SPSR_irq
10011	13	Supervisor (svc)	PC, R14_svc, R13_svc, R12–R0, CPSR, SPSR_svc
10111	17	Abort (abt)	PC, R14_abt, R13_abt, R12–R0, CPSR, SPSR_abt
11011	1B	Undefined (und)	PC, R14_und, R13_und, R12–R0, CPSR, SPSR_und
11111	1F	System (sys)	PC, R14–R0, CPSR

Table 2: ARM Processor Modes

Please note that the five Saved Program Status Registers (SPSRs) have the same format as the Current Program Status Register; these registers save the contents of CPSR when an exception occurs.

System Initialisation

When an ARM-based system is switched on, a large amount of its *state* is undefined. In other words, the contents of volatile memory (ie, not including non-volatile storage such as the Flash ROMs) and, more importantly, most of the registers in the processor itself, will have random (undefined) values. However, two registers *are* well-defined at reset: the Program Counter and the Current Program Status Register:

- The Program Counter PC (also called R15) is set to 0x00000000. This allows the processor to execute instructions at that address; most systems are designed in such a way that the Flash ROM resides at address 0x00000000 at reset.
- The Current Program Status Register CPSR is set to 0x000000D3 (0b11010011 in binary). This disables the Fast and normal Interrupts, selects the normal ARM state (instead of Thumb state) and switches to Supervisor mode.

Once the ARM processor resets PC and CPSR, it usually begins executing code at the new address in register PC (0x00000000). This code (usually called the *boot code*) performs any further initialisation as required. This includes setting up the various stack pointers (the R13 registers in each mode), initialising the exception handlers (including those that handle interrupts) and setting up any peripheral devices in the system. After doing all this, the boot code usually enters User mode.

Understanding what the boot code must do in a real system is beyond the scope of this course. However, in the Laboratory, the ARM processor under the Komodo ARM Environment gives you a “blank machine” in a system that does not require much initialisation. In this environment, you can switch the processor from Supervisor mode to User mode (with Fast and normal Interrupts disabled) by modifying the CPSR appropriately and branching to the user code:

```
.set ARM_PSR_i,      0b10000000 ; I bit in CPSR/SPSR
.set ARM_PSR_f,      0b01000000 ; F bit in CPSR/SPSR
.set ARM_PSR_mode_usr, 0b10000   ; User mode

mov  r14, #(ARM_PSR_i | ARM_PSR_f | ARM_PSR_mode_usr) ; 0b11010000 = 0xD0
msr  cpsr, r14 ; CPSR = User mode, no interrupts
ldr  pc, =User_code_start ; Load start address of user code into PC
```

By the way, note that the `mov` instruction uses the GNU Assembler to work out the bit pattern needed to initialise the CPSR; the syntax to do so is very similar to C. In this case, the expression uses the C “|” (“OR”) operator and evaluates to 0b10000000 OR 0b01000000 OR 0b10000, ie, 0b11010000.

Exceptions

During the ordinary flow of execution in a user program, the Program Counter usually increases sequentially through the address space, with perhaps a branch here or there to nearby labels, or with branch-and-links to subroutines and functions.

An *exception* causes this normal flow of execution to be diverted. Exceptions are generated by sources internal or external to the processor. This allows the processor to handle events generated by these sources; such events include:

- interrupts generated by some peripheral device,
- an attempt to execute an undefined or unimplemented instruction,
- a software-generated interrupt, via the `swi` instruction.

The ARM processor supports seven types of exceptions. These are listed in Table 3, along with the processor mode that is used to handle it. When an exception occurs, the processor branches to a fixed address that corresponds to that exception. This fixed address, called the *exception vector address*, is located in the bottom 32 bytes of the memory map. These 32 bytes are called the *exception vector table*.

You will note, by looking at Table 3, that there is just enough room at each vector address for *one* instruction (4 bytes). This is usually initialised to be a branch instruction or something like “ldr pc, [pc, #24]”.

Exception Type	Processor Mode	Vector Address
Reset	Supervisor	0x00000000
Undefined instructions	Undefined	0x00000004
Software Interrupt (swi)	Supervisor	0x00000008
Prefetch Abort (instruction fetch memory abort)	Abort	0x0000000C
Data Abort (data access memory abort)	Abort	0x00000010
Interrupt (IRQ)	Interrupt (IRQ)	0x00000018
Fast Interrupt (FIQ)	Fast Interrupt (FIQ)	0x0000001C

Table 3: ARM Processor Exceptions

Handling an exception requires the *processor state* to be preserved: in general, the contents of all registers (especially the registers PC and CPSR) must be the same after an exception as they were before it. Imagine the chaos that would occur in your program if this was not done correctly!

The ARM processor uses the additional (banked) registers associated with each processor mode (as shown in Figure 1) to help save the processor state. To handle an exception, the ARM processor:

1. copies the address of the next instruction (the *return address*), or the return address plus some offset, into the appropriate LR register,
2. copies the CPSR into the appropriate SPSR,
3. sets the CPSR mode bits to the processor mode corresponding to the exception,
4. enforces ARM state by setting bit 5 (the T bit) of CPSR to zero,
5. possibly disables fast interrupts by setting bit 6 of CPSR to one (only for FIQ exceptions),
6. disables normal interrupts by setting bit 7 (the I bit) of CPSR to one, and
7. loads the address of the exception vector into the Program Counter PC.

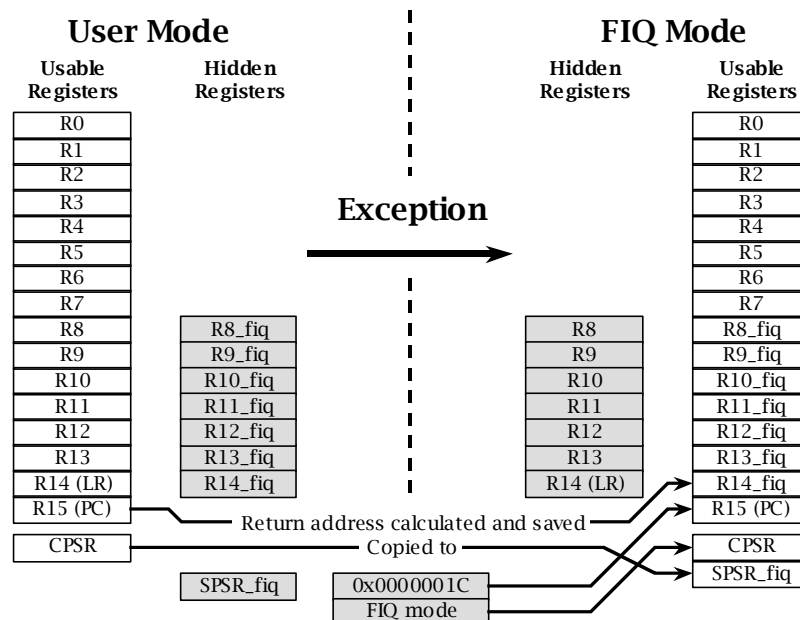


Figure 3: Switching from User Mode to FIQ Mode on a Fast Interrupt Exception

At this point, the ARM processor executes the code at the exception vector address. Please note that, for the duration of the exception (or, more correctly, while in that specific processor mode), the banked registers replace the ordinary ones. For example, while the processor

is in FIQ (Fast Interrupt) mode, the ordinary registers R8–R14 *cannot* be seen directly: only the registers R8_fiq–R14_fiq can be seen. This is shown in Figure 3 on the previous page.

Once the exception handler does what is necessary to handle the exception, the handler:

1. moves the contents of register LR, or LR less some offset, into PC, and
2. copies SPSR back to CPSR.

Both of these steps can be done in one instruction. Doing so has the effect of returning to the original task, running under the original processor mode and with the CPSR as it was originally before the exception.

Software Interrupts

A *software interrupt* is a type of exception that is initiated entirely by software. On the ARM processor, the relevant instruction that does this is `swi`. When this instruction is executed, it causes the processor to switch into Supervisor mode and branch to the relevant exception vector address, `0x00000008`. In other words, `swi` causes an exception, but one that is foreseen by the program.

Software interrupts are useful because they allow a program running in User mode to switch to a privileged mode; the code that handles the software interrupt can then do whatever is needed on behalf of the user program. An operating system providing input/output routines is a classic example of this.

To the ARM processor, `swi` is just another type of exception. When the processor executes this instruction, it:

1. copies the address of the next instruction following the `swi` into the `LR_svc` (`R14_svc`) register. This return address is actually `PC - 4`; the `swi` instruction can be found at `PC - 8`,
2. copies the CPSR into `SPSR_svc` (the Supervisor mode SPSR),
3. sets the CPSR mode bits to Supervisor mode. This has the effect of “swapping in” `R13_svc` and `R14_svc` and “swapping out” the previously-visible `R13` and `R14`,
4. enforces ARM state by setting bit 5 (the T bit) of CPSR to zero,
5. disables normal interrupts by setting bit 7 (the I bit) of CPSR to one. This means that normal interrupts cannot cause exceptions during the `swi` call, unless bit 7 is later set to zero in the exception handler’s code. Fast interrupts are *not* disabled and can still occur;
6. loads the address of the exception vector, `0x00000008`, into the Program Counter PC.

Once the software interrupt handler has finished its task, it returns control to the calling program by:

1. moving the contents of register `LR_svc` (`R14_svc`) into PC, and
2. copying `SPSR_svc` back to CPSR.

The following single instruction performs both of these steps:

```
movs    pc, lr    ; Copy current LR to PC and copy current SPSR to CPSR
```

Note that the instruction is `movs`, not `mov`: the `movs` instruction automatically copies SPSR to CPSR, but *only* when the destination register is PC (`R15`) and the instruction is executed in a privileged mode. **Question:** What would happen if “`mov pc, lr`” was used instead of “`movs pc, lr`”?

Question: Why must the two return steps be done using a single instruction? Why can’t you do something like the following? What happens if you do?

```
str    r0, [sp, #-4]!    ; Save the value of R0 to the stack
mrs    r0, spsr         ; Copy current SPSR to R0
msr    cpsr, r0         ; Copy R0 to CPSR
ldr    r0, [sp], #4     ; Retrieve original value of R0 from the stack
mov    pc, lr           ; Copy current LR to PC
```

You can essentially treat `swi` as a fancy “`b1 0x00000008`” that not only branches-and-links to `0x00000008`, but also changes the processor mode and does some other things besides.

A Very Simple Software Interrupt Handler

The DSLMU Microcontroller Board in the Laboratory does not support memory management and the associated protection of resources. In other words, all peripherals and memory can be accessed equally freely in any processor mode, whether privileged or unprivileged.

For the purposes of this experiment, however, pretend that you cannot access the microcontroller’s I/O ports from User mode. In other words, pretend that you must switch to a privileged mode to access the microcontroller’s Port A and so on.

Carefully examine the program contained in Figures 4-7. This program reimplements the LED-flashing program that you have already met in Experiments 1 and 4 as a program that uses the `swi` instruction. Please note that some of the comments have been removed to conserve space; if possible, you should use **kate** to examine the files on your CD-ROM instead:

```
.set ARM_PSR_i, 0b10000000 ; I bit in CPSR/SPSR
.set ARM_PSR_f, 0b01000000 ; F bit in CPSR/SPSR
.set ARM_PSR_t, 0b00100000 ; T bit in CPSR/SPSR

.set ARM_PSR_mode_mask, 0b11111 ; Processor modes mask
.set ARM_PSR_mode_usr, 0b10000 ; User mode
.set ARM_PSR_mode_fiq, 0b10001 ; Fast Interrupt mode
.set ARM_PSR_mode_irq, 0b10010 ; Interrupt mode
.set ARM_PSR_mode_svc, 0b10011 ; Supervisor mode
.set ARM_PSR_mode_abt, 0b10111 ; Abort mode
.set ARM_PSR_mode_und, 0b11011 ; Undefined mode
.set ARM_PSR_mode_sys, 0b11111 ; System mode

.set portA, 0x10000000 ; Microcontroller Port A address
.set value1, 0b11111111 ; Value to turn the LEDs on
.set value2, 0b00000000 ; Value to turn the LEDs off
```

Figure 4: Program header file *header-v1.s*

```
.global _start ; "_start" is where the code starts running
.extern main ; "main" is defined in another file
.extern swi_handler ; "swi_handler" is also defined elsewhere
.include "header-v1.s" ; Include various definitions

; -----
; The following code will be placed into the ".zeropage" section, NOT into the
; ".text" section. The GNU Linker will place the ".zeropage" section at address 0x0.
.section .zeropage, "awx" ; For code located at address 0x00000000
_start: ; Start of the entire program
; ARM processor exception vector table
ev00: b init ; Reset exception
ev04: nop ; Undefined Instruction exception
ev08: b swi_handler ; Software Interrupt exception
ev0C: nop ; Prefetch Abort exception
ev10: nop ; Data Abort exception
ev14: nop ; (Not used)
ev18: nop ; Interrupt exception
ev1C: nop ; Fast Interrupt exception

; -----
; The following code will be placed into the ".ospage" section, NOT into the ".text"
; section. The GNU Linker will place the ".ospage" section at address 0x1000.
```

(Continued on the next page...)

(Continued from the previous page...)

```
.section .ospage, "awx" ; For "operating system" code
init:
  mrs    ip, cpsr          ; Change to User mode using read-modify-write
  bic    ip, ip, #ARM_PSR_mode_mask ; Mask out the bottom 5 bits
  orr    ip, ip, #(ARM_PSR_i | ARM_PSR_f | ARM_PSR_mode_usr)
  msr    cpsr, ip          ; Disable both interrupts, set User mode
  b      main              ; Actually make the changes to CPSR
                          ; Now in User mode; jump to the main program
.end
```

Figure 5: Program source file *boot-swi-v1.s*

```
; -----
; The following code will be placed into the ".ospage" section, NOT into the ".text"
; section. The GNU Linker will place the ".ospage" section at address 0x1000.
.section .ospage, "awx" ; For "operating system" code
.global swi_handler    ; Make this label visible to other modules
swi_handler:           ; Software Interrupt handler
                      ; This handler runs in Supervisor mode
  strb   r1, [r0]      ; Write the byte in R1 to "protected port" R0
  movs   pc, lr        ; Return to caller (end of exception)
.end
```

Figure 6: Program source file *swi-v1.s*

```
.text                ; Ordinary program code follows
.include "header-v1.s" ; Include various definitions
.global main          ; Make this label visible to other modules
main:
  ldr    r0, =portA   ; Main program, running in User mode
                      ; Load address of Port A into R0
main_loop:
  mov    r1, #value1  ; R1 = value to turn the LEDs on
  swi                                ; Generate a software interrupt
  mov    r1, #value2  ; R1 = value to turn the LEDs off
  swi                                ; Generate a software interrupt
  b      main_loop    ; Repeat forever (or until stopped)
.end
```

Figure 7: Program source file *flash-v1.s*

You should note a few things about this program:

- The program has been written as a set of *modules*. Each file contains a section of code that does just one thing. Doing this makes it easier to understand the complete program. Thus, *boot-swi-v1.s* (Figure 5) provides the initialisation routines (“boot code”), *swi-v1.s* (Figure 6) provides the software interrupt handler and *flash-v1.s* (Figure 7) provides the main program.
- Important (“global”) constants have been placed in their own file, *header-v1.s* (Figure 4). This file can be inserted into other source code files by using the **.include** assembler directive. This directive is the assembly-language equivalent of **#include** in C.
- The instructions at labels *ev00* to *ev1C* in Figure 5 form the ARM exception vector table, stored at address *0x0*. Notice *ev08*: this provides the branch to the *swi* handler; this branch instruction is executed by the ARM processor as if it were the *next* instruction after *swi*. The *nop* (“do nothing”) instruction simply means that some exceptions will *not* be handled correctly at all. This is not recommended in real life, of course!

- Notice the four instructions at the label `init` in Figure 5: these instructions use a read-modify-write cycle to change the CPSR to User mode. Although a read-modify-write cycle is not strictly needed here, it is the recommended way of doing things; it makes it easier to port your code to newer ARM processors at a later date. You should remember from Experiment 4 that `bic` clears bits to zero and `orr` sets bits to one.
- A new assembler directive has been used: `.section`. This directive allows you to place blocks of code into separate areas in the memory map. This program uses three such areas (called *sections*): the standard `.text` section, the `.zeropage` section and the `.ospage` section.

This last point is an important one to understand. Up until now, all of the assembly language programs in these experiments have placed everything into just one section, `.text`. This makes things quite simple: all of the code and data is placed together in memory; the actual *load address* is determined by the GNU Linker (and is `0x8000` by default).

This program is different, however: some parts of the program simply *must* be placed at address `0x0`, as that is where the ARM processor expects them to be. Other parts are logically part of what might later become an “operating system”; ideally, these should be placed into an area of memory that is separate from the ordinary User mode program.

All this can be done with *sections*. A section is simply an area of memory that has a name; that section can be loaded at a particular address, as determined by the programmer or the GNU Linker. In the case of this program, the GNU Linker will place the `.zeropage` section at address `0x0`, the `.ospage` section at address `0x1000` and the `.text` section at address `0x8000`.

Program sections are a somewhat advanced topic; Figure 8 should help you understand more clearly how the different blocks of code are placed into memory. Note in particular how the two blocks of code, Code B and Code C, are merged into the single `.ospage` section:

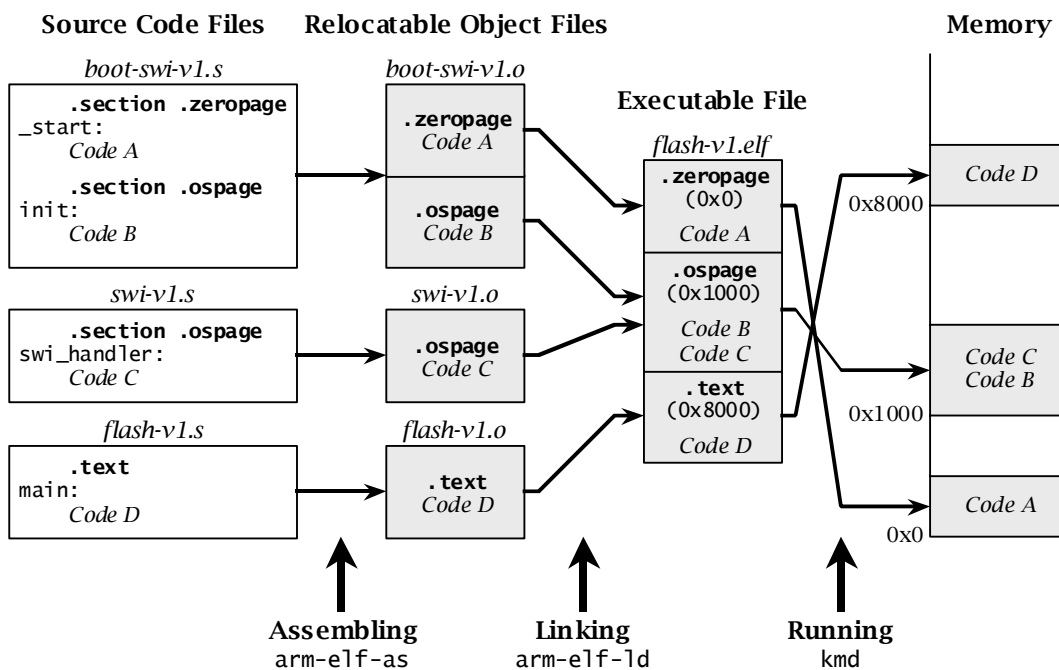


Figure 8: From Source Code to Memory for the program `flash-v1.elf`

Creating the executable `flash-v1.elf` requires a number of steps, as shown in Figure 8. The first is to *assemble* the source code files. This is very similar to what you have already been doing:

```
arm-elf-as -marm7tdmi --gdwarf2 boot-swi-v1.s -o boot-swi-v1.o
arm-elf-as -marm7tdmi --gdwarf2 swi-v1.s -o swi-v1.o
arm-elf-as -marm7tdmi --gdwarf2 flash-v1.s -o flash-v1.o
```

The second step is to *link* all of the *.o* object files into an executable. Doing this determines where each section will eventually appear in memory:

```
arm-elf-ld misc.ls boot-swi-v1.o swi-v1.o flash-v1.o -o flash-v1.elf
```

Notice that an additional file was specified on the command line: *misc.ls*. This file contains directives that tell the GNU Linker to place the *.zeropage* and *.ospage* sections in the right places; although this file is essential, you don't need to understand its contents to use it. By the way, if you are wondering how the GNU Linker "knows" how to place the *.text* section at 0x8000, look at the file */usr/local/arm-elf/lib/ldscripts/armelf.x* and satisfy your curiosity!

Task 1: Single-stepping Through the Program

Assemble and link the program *flash-v1.elf* using the commands listed above. If you prefer, you can use the **make** command instead; the make-file is called *flash-v1.make*, and can be used by typing:

```
make -f flash-v1.make
```

Use the Komodo debugger to download the program to the DSLMU Microcontroller Board. Single-step through the program, noting in particular when the processor mode changes. Be ready to explain what is happening to the Laboratory assessor.

Checkpoint 1: Signature:

Allowing Multiple System Call Functions

You should have noticed by now that the software interrupt handler in Figure 6 can only perform one task. This is adequate as an example from which to learn, but is unrealistic in the "real world". For example, an operating system needs to provide a large number of services to its clients. The ARM processor only provides a *single* *swi* instruction, however. This means that the software interrupt handler must be written in such a way that it can determine which service the client (caller) requires and act accordingly. One common technique for doing this is to use a particular register, such as R0, as a function number specifier. This technique is illustrated in Figure 9:

```
swi_handler:                ; Software Interrupt handler
                            ; R0 = SWI function number
    cmp    r0, #swi_num_funcs ; Check if R0 >= swi_num_funcs
    bhs   swi_outofrange     ; If so, this indicates an error
    cmp    r0, #0           ; Does the client want function 0?
    beq   swi_f0            ; Yes, handle it in swi_f0
    cmp    r0, #1           ; Does the client want function 1?
    beq   swi_f1            ; Yes, handle it in swi_f1
    ...                    ; and so on... By the way, each "..."
                            ; can represent multiple lines of code
swi_f0: ...                 ; Code to implement SWI function 0
    b     swi_end           ; Return to the caller
swi_f1: ...                 ; Code to implement SWI function 1
    b     swi_end           ; Return to the caller
swi_end:                    ; Return to the client program
    ...                    ; Whatever needs to be done...
    movs  pc, lr           ; Restore PC and CPSR
```

Figure 9: A Typical Multi-function Software Interrupt Handler

The program *flash-v2.elf* in Figures 10-14 implements two software interrupt functions: one to turn the LEDs on and off, the other to pause the program in a delay loop. Please note that the only difference between *boot-swi-v2.s* in Figure 12 and *boot-swi-v1.s* in Figure 5 is that the line containing the `.include` assembler directive has changed:

```
.set    swi_num_funcs, 2           ; Number of SWI functions
.set    swi_set_LEDs, 0           ; Function 0: set the LEDs
.set    swi_delay, 1             ; Function 1: delay the program

.set    value1, 0b11111111       ; Value to turn the LEDs on
.set    value2, 0b00000000       ; Value to turn the LEDs off
.set    waitval, 10000           ; Number of loops to wait
```

Figure 10: Program header file *header-v2-pub.s*

```
.set    ARM_PSR_i, 0b10000000    ; I bit in CPSR/SPSR
.set    ARM_PSR_f, 0b01000000    ; F bit in CPSR/SPSR

.set    ARM_PSR_mode_mask, 0b11111 ; Processor modes mask
.set    ARM_PSR_mode_usr, 0b10000 ; User mode

.set    portA, 0x10000000        ; Microcontroller Port A address
```

Figure 11: Program header file *header-v2-int.s*

```
.global _start           ; "_start" is where the code starts running
.extern main             ; "main" is defined in another file
.extern swi_handler     ; "swi_handler" is also defined elsewhere

.include "header-v2-int.s" ; Include various definitions

; -----
; The following code will be placed into the ".zeropage" section, NOT into the
; ".text" section. The GNU Linker will place the ".zeropage" section at
; address 0x0.

.section .zeropage, "awx" ; For code located at address 0x00000000
_start:                   ; Start of the entire program
; ARM processor exception vector table
ev00: b    init           ; Reset exception
ev04: nop                ; Undefined Instruction exception
ev08: b    swi_handler    ; Software Interrupt exception
ev0C: nop                ; Prefetch Abort exception
ev10: nop                ; Data Abort exception
ev14: nop                ; (Not used)
ev18: nop                ; Interrupt exception
ev1C: nop                ; Fast Interrupt exception

; -----
; The following code will be placed into the ".ospage" section, NOT into the
; ".text" section. The GNU Linker will place the ".ospage" section at address
; 0x1000.

.section .ospage, "awx" ; For "operating system" code
init:
mrs    ip, cpsr          ; Change to User mode using read-modify-write
bic    ip, ip, #ARM_PSR_mode_mask ; Get current value of CPSR into IP (R12)
; Mask out the bottom 5 bits
orr    ip, ip, #(ARM_PSR_i | ARM_PSR_f | ARM_PSR_mode_usr)
; Disable both interrupts, set User mode
msr    cpsr, ip          ; Actually make the changes to CPSR
b      main              ; Now in User mode; jump to the main program

.end
```

Figure 12: Program source file *boot-swi-v2.s*

```

.section .ospage, "awx" ; For "operating system" code
.include "header-v2-pub.s" ; Include various public definitions
.include "header-v2-int.s" ; Include various internal definitions

.global swi_handler ; Make this label visible to other modules
swi_handler: ; This code runs in Supervisor mode
; R0 contains the SWI function code. A slight optimisation: no check is done
; for out-of-range errors here, as the code simply "falls through" to the error
; handler if need be, which then "falls through" to the exit code.

    cmp    r0, #swi_set_LEDs ; Client wants to set the LEDs?
    beq    swi_set_LEDs_func ; Yes, handle it
    cmp    r0, #swi_delay ; Client wants to delay the program?
    beq    swi_delay_func ; Yes, handle it

swi_outofrange: ; Out of range error handler
    mov    r0, #0xFFFFFFFF ; Signal an error by returning 0xFFFFFFFF

swi_end: ; Return to the client program
    movs   pc, lr ; Restore PC and CPSR

swi_set_LEDs_func: ; Function 0: Set the LEDs
    ldr    r0, =portA ; Load "protected port" address into R0
    strb   r1, [r0] ; Set the LEDs to the value in R1
    b     swi_end ; Return to the caller

swi_delay_func: ; Function 1: Delay the program
    cmp    r1, #0 ; R1 = number of loops to delay (unsigned)
    beq    swi_end ; If R1 = 0, no loops, just end
swi_delay_func_1:
    subs   r1, r1, #1 ; Decrement the number of loops to go
    beq    swi_end ; Return to caller if finished
    b     swi_delay_func_1 ; Otherwise, repeat the loop

.end

```

Figure 13: Program source file *swi-v2.s*

```

.text ; Ordinary program code follows
.include "header-v2-pub.s" ; Include various public definitions
.global main ; Make this label visible to other modules
main: ; Main program; this code runs in User mode
main_loop:
    mov    r0, #swi_set_LEDs ; Request the SWI handler to set the LEDs
    mov    r1, #value1 ; R1 = value to turn the LEDs on
    swi ; Generate a software interrupt

    mov    r0, #swi_delay ; Request the SWI handler to delay program
    ldr    r1, =waitval ; R1 = number of delay loops
    swi ; Generate a software interrupt

    mov    r0, #swi_set_LEDs ; Request the SWI handler to set the LEDs
    mov    r1, #value2 ; R1 = value to turn the LEDs off
    swi ; Generate a software interrupt

    mov    r0, #swi_delay ; Request the SWI handler to delay program
    ldr    r1, =waitval ; R1 = number of delay loops
    swi ; Generate a software interrupt

    b     main_loop ; Repeat forever (or until stopped)

.end

```

Figure 14: Program source file *flash-v2.s*

Please note that the actual implementation of the software interrupt handler in Figure 13 is slightly different from the "typical" example in Figure 9. The main change is that a range

check is *not* performed; an illegal value in R0 simply will not match any of the `cmp` comparison instructions and will thus “fall through” to the error handler. This is a reasonably common way of writing this style of *dispatch handling* code.

Another difference that you should have noticed is that it is now the software interrupt handler that loads the address of Port A, not the main program. This is a recommended practice; that way, only the “operating system” (swi handler, in this case) needs to know how “set the LEDs” works. The user program can be blissfully unaware of the implementation details: all it has to do is call the appropriate swi service. Abstraction (ie, separating the interface from the implementation) is always a Good Thing!

In this program, abstraction is further enhanced by splitting the original single header file into two. Thus, the file *header-v2-pub.s* (Figure 10) now provides the public interface (ie, everything the user program needs to know), and *header-v2-int.s* (Figure 11) provides the internal implementation-specific details that only the swi handler and boot code needs.

Task 2: Stepping Through Multiple System Calls

Assemble and link the program *flash-v2.elf*. You may use the following command lines:

```
arm-elf-as -marm7tdmi --gdwarf2 boot-swi-v2.s -o boot-swi-v2.o
arm-elf-as -marm7tdmi --gdwarf2 swi-v2.s -o swi-v2.o
arm-elf-as -marm7tdmi --gdwarf2 flash-v2.s -o flash-v2.o

arm-elf-ld misc.lis boot-swi-v2.o swi-v2.o flash-v2.o -o flash-v2.elf
```

Alternatively, simply type “**make -f flash-v2.make**”.

Use the Komodo debugger to download the program to the DSLMU Microcontroller Board. Make sure you press the **Reset** button within Komodo *before* you start running the program! Step through the program, noting in particular when the processor mode changes. Demonstrate your flashing LEDs to the Laboratory assessor.

Checkpoint 2: Signature:

Note: the advantage of passing the swi function number in a general-purpose register, such as in R0, is that it makes writing the software interrupt handler a little easier. However, it also “wastes” a register that could have been used for some other purpose.

An alternative technique is to use a *numbered swi call*. This utilises the fact that the swi instruction can encode an optional 24-bit number; this number is ignored by the ARM processor. You can retrieve the number by decoding the swi instruction; this instruction always has the hexadecimal format *yFnnnnnn*, where *y* is the condition code and *nnnnnn* is the desired number. For example:

```
swi    0x123456
```

is encoded as 0xEF123456. You can use code similar to the following within your software interrupt handler to retrieve the number 0x123456:

```
ldr    ip, [lr, #-4]      ; Retrieve the "swi" instruction
bic    ip, ip, #0xFF00000 ; Mask off the top 8 bits
```

You can now use the value in register IP (R12) instead of the previous method of using R0. Of course, you might want to save R12 to the stack before using it in this fashion. Just remember to *restore* it afterwards...

Using Jump Tables

The method shown in Figure 9 is not particularly suitable for handling a large number of software interrupt calls. It is simply not efficient to check for the function number for each and every case: imagine if there are thousands of functions! A far more efficient way of doing things is to use a *jump table*. This method is shown in Figure 15:

```
swi_handler:                                ; Software Interrupt handler
                                           ; R0 = SWI function number

    cmp    r0, #swi_num_funcs              ; Check if R0 >= swi_num_funcs
    bhs    swi_outofrange                  ; If so, this indicates an error

    adr    ip, swi_jump_table              ; Register IP = address of jump table
    ldr    pc, [ip, r0, lsl #2]           ; Load PC with the value stored at address
                                           ; IP + R0 * 4; ie, jump to the code pointed
                                           ; to by the value stored at IP + R0 * 4

swi_jump_table:                             ; Table of addresses for the SWI functions
    .word  swi_f0                          ;   for function 0
    .word  swi_f1                          ;   for function 1
    ...                                     ; and so on for all other functions. Note:
                                           ; each "..." can stand for multiple lines

swi_f0: ...                                ; Code to implement SWI function 0
    b      swi_end                          ; Return to the caller

swi_f1: ...                                ; Code to implement SWI function 1
    b      swi_end                          ; Return to the caller

swi_end:                                    ; Return to the client program
    ...                                     ; Whatever needs to be done...
    movs   pc, lr                          ; Restore PC and CPSR
```

Figure 15: Using a Jump Table to Handle SWI Functions

The code that actually implements the jump table (also called the *dispatch table*) with error checking is contained in the four instructions at `swi_handler`:

```
    cmp    r0, #swi_num_funcs              ; Register R0 must be between 0 and
    bhs    swi_outofrange                  ; swi_num_funcs-1 inclusive

    adr    ip, swi_jump_table              ; Load address of swi_jump_table into IP
    ldr    pc, [ip, r0, lsl #2]           ; and jump to the code pointed to by the
                                           ; value stored at IP + R0 * 4
```

Note that most of the work is done by the `ldr` instruction: it takes the function number in register `R0` and shifts it to the left by 2 bits (ie, multiplies it by 4). This gives a value that can be used as a *word offset* into the `swi_jump_table` table. Next, the instruction adds this word offset to the address of the table (which has been previously loaded into register `IP`, ie, `R12`): this gives $IP + R0 \times 4$. Finally, the `ldr` instruction loads the word stored at $IP + R0 \times 4$ into register `PC`; this has the effect of jumping to the code handling that function number.

The four instructions shown above are still slightly inefficient in that they use an additional register, `IP`. The following *three* instructions can replace them; they rectify this inefficiency, at the cost of requiring the jump table to be stored directly after them:

```
    cmp    r0, #swi_num_funcs              ; Check that R0 is in the correct range
    ldrlo  pc, [pc, r0, lsl #2]           ; If it is, jump to code pointed to by the
                                           ; value stored at PC + R0 * 4
    b      swi_outofrange                  ; Otherwise, jump to the error handler
                                           ; Jump table MUST appear at this point
```

These three instructions work by exploiting an idiosyncrasy of the ARM instruction architecture: at the point where the `ldrlo` (“load if unsigned less-than”) instruction is executed, the value in register `PC` is the address of the instruction *plus eight*. And in this code, $PC + 8$ happens to be the address of the jump table.

Note: By convention, the *ARM Thumb Procedure Call Standard* described in Experiment 3 applies to software interrupt calls as well. In other words, the swi handler should follow the ATPCS to know where to expect its parameters and return its results, to preserve the contents of the appropriate registers, to maintain the stack frame and so on.

Task 3: Using a Jump Table in Practice

Write a new version of the function `swi_handler` so that it uses a jump table; call your new source code file `swi-jt.s` (you can use the file `swi-v2.s` as a basis for this new file). You do *not* need to modify any other source code files. Compile your program using the following command lines:

```
arm-elf-as -marm7tdmi --gdwarf2 boot-swi-v2.s -o boot-swi-v2.o
arm-elf-as -marm7tdmi --gdwarf2 swi-jt.s -o swi-jt.o
arm-elf-as -marm7tdmi --gdwarf2 flash-v2.s -o flash-v2.o

arm-elf-ld misc.lis boot-swi-v2.o swi-jt.o flash-v2.o -o flash-jt.elf
```

Alternatively, simply type “**make -f flash-jt.make**”. In either case, the resulting executable is called `flash-jt.elf`.

Use the Komodo debugger to download the program to the DSLMU Microcontroller Board. Make sure you press the **Reset** button within Komodo *before* you start running the program! Step through the program, noting in particular when the processor mode changes. Be ready to explain how your code works to the Laboratory assessor.

Checkpoint 3: Signature:

Hardware Interrupts

Hardware interrupts are a mechanism that allows an external signal (called an *interrupt request*, or IRQ for short) to interrupt the normal execution of code in a processor. If a processor honours the interrupt request, it suspends whatever it is currently running and jumps to some other code located at a predetermined fixed address; this “other code” is called the *interrupt service routine*. In effect, it is as if the processor had inserted a branch instruction *between* two instructions of whatever it had been running up to that point.

The reason interrupts are useful is that it allows the processor to handle peripherals in the most efficient manner. Without interrupts, the processor would have to check each input/output device periodically to see if that device needed attention. This method, called *polling*, wastes much time that could have been used to do something more useful. With interrupts, on the other hand, the input/output device can signal the processor to indicate that it needs attention; at other times, the processor can simply ignore the device.

The following analogy may help you better understand interrupts: think of the humble telephone. If the telephone did not have a bell (or any other indicator), you would have to pick up the tube every few minutes to see if someone was calling you. In other words, you would have to *poll* the telephone—an extremely inefficient use of your time! And there would always be the possibility that you would miss an important call while doing something else. On the other hand, a telephone with an audible bell would ring every time someone tried to call you. In other words, the bell would *interrupt* whatever you were doing at the time; you would then pick up the tube to “service the call” at that time. Using this method, you would not need to poll the telephone every few minutes. The result: more work can be done!

The following are a few examples where interrupts can be used:

- For *input*. Real-world input devices are often irregular and unpredictable in their timing. Interrupts allow the processor to read the input from such devices, such as the input generated by a user pressing a key or moving the mouse.
- For *waiting*. Real-world input/output devices are *much* slower than the processor; an 8-page-per-minute laser printer might be well over 25,000,000 times slower than the Pentium IV processor driving it! Interrupts allow the device to indicate that it is now ready to receive more data.
- For *timing*. An external timer/counter generates a fixed-frequency timing signal that can be used to provide a regular source of interrupts. Modern operating systems use these interrupts to make sure that all tasks are allocated a fair share of processor time.

Task 4: Polling the Push-button Switches

Examine the source code in Figure 16 and Figure 17; this program uses polling to determine whether push-button switches S2 and/or S3 are being pressed and, if so, lights one or the other side of the on-board LEDs:

```
.set    iobase,      0x10000000 ; Base of Microcontroller I/O space
.set    portA,       0x00        ; Microcontroller Port A offset
.set    portB,       0x04        ; Microcontroller Port B offset

.set    portB_pbs2,  0b10000000 ; Port B bit 7 = Push-button switch S2
.set    portB_pbs3,  0b01000000 ; Port B bit 6 = Push-button switch S3

.set    left_leds,   0b00000111 ; Value to turn on the left LEDs
.set    right_leds,  0b01110000 ; Value to turn on the right LEDs
```

Figure 16: Program header file *header-v3.s*

```
.include "header-v3.s" ; Include definitions needed for this program
.text                  ; Executable code follows
_start: .global _start ; "_start" is required by the linker
        .global main   ; "main" is our main program
        b          main ; Start running the main program
main:
        ; Entry to the function "main"
        ldr        r0, =iobase ; R0 = base of the Microcontroller I/O space
        mov        r2, #left_leds ; Use R2 when turning on the left set of LEDs
        mov        r3, #right_leds ; Use R3 when turning on the right set of LEDs
poll_loop:
        ldrb       r1, [r0, #portB] ; Read the microcontroller's Port B
        tst        r1, #portB_pbs2 ; Check if push-button S2 is pressed
        strneb     r2, [r0, #portA] ; If it is, turn on the left LEDs
        tst        r1, #portB_pbs3 ; Check if push-button S3 is pressed
        strneb     r3, [r0, #portA] ; If it is, turn on the right LEDs
        b          poll_loop ; Do this forever (or until stopped)
.end
```

Figure 17: Program source file *pb-poll.s*

Assemble and link this program as usual; the easiest way to do so is to use the make-file *pb-poll.make* by typing “**make -f pb-poll.make**”. Use the Komodo debugger to download the resulting program *pb-poll.elf* to the DSLMU Microcontroller Board and to run it. Verify that pressing the push-buttons S2 and/or S3 on the Expansion Board turns on the on-board LEDs. Be ready to show and explain your running program to the Laboratory assessor.

Hints: The `tst` (“bitwise test”) instruction is very similar to the `ands` instruction; the only difference is that `tst` discards the logical result of the AND operation and only changes the condition code flags. The `strneb` instruction is “store byte if not equal”.

Question: What happens when *both* S2 and S3 are pressed? Is there any moment when all six LEDs are turned on at the same time? Why or why not?

Checkpoint 4: Signature:

Hardware Interrupts on the ARM Processor

The ARM processor core provides two signals that are used by peripherals to request interrupts: the Interrupt signal `nIRQ` and the Fast Interrupt signal `nFIQ`; both of these signals are *active-low* and *level-sensitive*. Pulling one of these signals low generates the corresponding processor exception, as shown in Table 3—as long as that interrupt has not been disabled in the Current Program Status Register, as explained on page 86.

The Fast Interrupt request is designed to handle high-priority and/or high-speed peripherals in the least possible time. The following factors make this possible:

- The `nFIQ` signal has a higher priority than `nIRQ`. This means that if *both* signals are pulled low, the fast interrupt is serviced first.
- Servicing a Fast Interrupt request causes ordinary interrupts to be disabled in the CPSR (see page 86 for an explanation of the I bit). Thus, an ordinary interrupt will not preempt (interrupt) the fast interrupt handler. (Naturally, this does not apply if the fast interrupt handler re-enables ordinary interrupts by setting the I bit to zero in CPSR).
- The Fast Interrupt processor mode has five additional registers (when compared with other processor modes). These registers, `R8_fiq`–`R12_fiq`, can be used to store the status of the handler between FIQ exceptions; it also means that the handler does not need to save the user’s `R8`–`R12` registers.
- The FIQ exception vector occupies the last entry in the exception vector table, as shown in Table 3 on page 88. This means that the code for the fast interrupt handler may be placed directly at address `0x0000001C` without the need for an intermediate branch instruction with its associated delays.

The ARM processor handles requests for interrupts by generating an exception of the corresponding type, assuming the relevant interrupt has not been disabled in the CPSR. Thus, a Fast Interrupt request would generate a Fast Interrupt exception, and an ordinary Interrupt request would generate an Interrupt exception. You should reread page 88 onwards to see a list of steps taken by the processor to handle any exception.

In particular, assume that the `nIRQ` signal has just been asserted and that bit 7 (the I bit) in CPSR is set to zero. This being the case, the ARM processor waits until the current instruction has finished executing, then:

1. copies the address of the next instruction to be executed, plus 4, into the `LR_irq` register. This means that `LR_irq` now points to the second instruction beyond the point of the interrupt request;
2. copies the CPSR into `SPSR_irq` (the Interrupt mode SPSR),
3. sets the CPSR mode bits to Interrupt mode. This has the effect of “swapping in” `R13_irq` and `R14_irq` and “swapping out” the previously-visible `R13` and `R14`,
4. enforces ARM state by setting bit 5 (the T bit) of CPSR to zero,
5. disables normal interrupts by setting bit 7 (the I bit) of CPSR to one. This means that further normal interrupts will not cause Interrupt exceptions to be generated, unless

bit 7 is later set to zero in the exception handler's code. Fast interrupts are *not* disabled and can still occur; and

6. loads the address of the Interrupt exception vector, 0x00000018, into the PC register. This will usually contain a branch instruction to the actual handler's code.

Essentially, the net effect of handling the Interrupt exception is as if the ARM processor inserted an "exception procedure call" into the instruction stream.

Once the interrupt handler (also called the *interrupt service routine*) has finished its task, it returns to whatever the processor was doing before by:

1. moving the contents of register LR_irq (R14_irq) less 4 into PC, and
2. copying SPSR_irq back to CPSR.

The following single instruction performs both of these steps:

```
subs    pc, lr, #4    ; Copy LR_irq-4 to PC and copy current SPSR to CPSR
```

Note that the instruction is *subs*, not *sub*: the *subs* instruction automatically copies SPSR to CPSR, but *only* when the destination register is PC (R15) *and* the instruction is executed in a privileged mode.

The ARM processor handles Fast Interrupt exceptions in much the same way as ordinary Interrupt exceptions. The differences are that a different set of registers are swapped in and out, that *both* normal and fast interrupts are disabled (ie, both bits 6 and 7 of CPSR are set to one), and that the exception vector address is 0x0000001C. See Figure 3 on page 88 for a visual representation of how the processor handles this type of exception.

Interrupts on the DSLMU Microcontroller Board

Two interrupt request signals is almost never enough for all of the hardware present in an ARM-based system. For this reason, most systems (including the DSLMU Microcontroller Board) provide an *interrupt controller*. This device controls whether other peripherals can interrupt the ARM processor or not. Essentially, the interrupt controller acts as a large AND-OR gate, as shown in Figure 18:

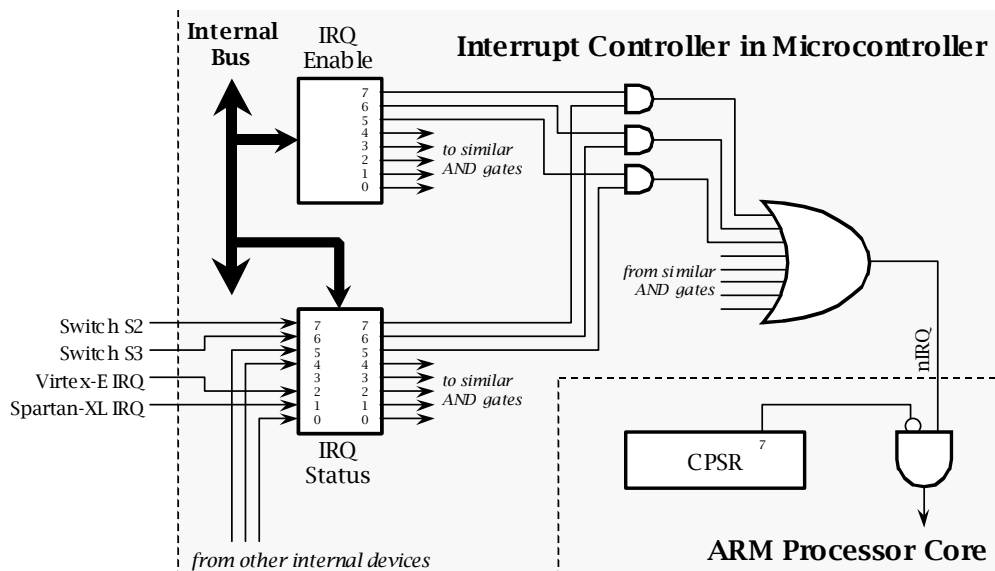


Figure 18: Interrupt Controller on the DSLMU Microcontroller Board

As shown in Figure 18, the interrupt controller on the DSLMU Microcontroller Board has two 8-bit ports: the IRQ Status port and the IRQ Enable port. The IRQ Status port, at address 0x10000018, indicates whether or not peripherals are trying to interrupt the ARM processor. The IRQ Enable port, at address 0x1000001C, controls whether or not those peripherals

are actually allowed to interrupt the processor. These ports are described in detail in the *DSLMU Microcontroller Board Hardware Reference Manual*; you can find this document in an Appendix or on your Companion CD-ROM.

The bit definitions for the IRQ Status and IRQ Enable ports are shown in Table 4:

Bit	Mode	Function
7	R/W	Push-button switch S2 on the Expansion Board
6	R/W	Push-button switch S3 on the Expansion Board
5	R/W	Serial port transmitter ready
4	R/W	Serial port receiver ready
3	—	(Reserved)
2	R/W	Xilinx Virtex-E interrupt request
1	R/W	Xilinx Spartan-XL interrupt request
0	R/W	Timer Compare interrupt request

Table 4: IRQ Status and IRQ Enable Ports Bit Definitions

Note that a peripheral will only be allowed to interrupt the ARM processor if its corresponding bit is enabled in the IRQ Enable port *and* normal interrupts are enabled by clearing the I bit in the Current Processor Status Register. For example, pressing push-button switch S3 will set bit 6 of the IRQ Status port to 1; this happens whether or not interrupts are enabled. However, an interrupt will only be generated for this switch if the bit 6 of the IRQ Enable port is set to 1 *and* bit 7 (the I bit) of CPSR is set to 0.

You should note that both Fast and normal Interrupts are *disabled* when the processor is reset. This is because the hardware is almost always in an undefined state when the system is initialised, and so may generate spurious (unwanted) interrupts. Once the boot code has set up the hardware, it may enable the interrupts by setting bits 6 and 7 of CPSR to zero.

Once the ARM processor has generated an Interrupt exception, the interrupt service routine must *acknowledge* the interrupt. In other words, the interrupt handler code must clear the appropriate bit in the IRQ Status port. If this is not done, the peripheral responsible for the interrupt will continue to request attention, causing another exception to be generated as soon as interrupts are re-enabled.

Interrupts in Practice: the Push-button Switches

Carefully examine the program *pb-irq-v1.elf* shown in Figures 19–22. This program does the same thing as the program in Task 4, except that it uses an interrupt handler to do its work. (Remember, you should use **kate** to examine the files on your CD-ROM if at all possible, as many comments have been removed from these figures to conserve paper):

```

.set    ARM_PSR_i, 0b10000000    ; I bit in CPSR/SPSR
.set    ARM_PSR_f, 0b01000000    ; F bit in CPSR/SPSR

.set    ARM_PSR_mode_mask, 0b11111    ; Processor modes mask
.set    ARM_PSR_mode_usr, 0b10000    ; User mode

.set    iobase,    0x10000000      ; Base of Microcontroller I/O space
.set    portA,    0x00            ; Microcontroller Port A offset
.set    portB,    0x04            ; Microcontroller Port B offset
.set    irq_status, 0x18          ; IRQ Status port offset
.set    irq_enable, 0x1C          ; IRQ Enable port offset

.set    irq_pbs2, 0b10000000      ; IRQ ports bit 7 = Push-button S2
.set    irq_pbs3, 0b01000000      ; IRQ ports bit 6 = Push-button S3

.set    left_leds, 0b00000111     ; Value to turn on the left LEDs
.set    right_leds, 0b01110000    ; Value to turn on the right LEDs

```

Figure 19: Program header file *header-v3.s*

```

.global _start          ; "_start" is where the code starts running
.extern main            ; "main" is defined in another file
.extern swi_handler     ; "swi_handler" is also defined elsewhere
.include "header-v3.s" ; Include various definitions
; -----
.section .zeropage, "awx" ; For code located at address 0x00000000
_start:                ; Start of the entire program
; ARM processor exception vector table
ev00: b      init      ; Reset exception
ev04: nop                    ; Undefined Instruction exception
ev08: nop                    ; Software Interrupt exception
ev0C: nop                    ; Prefetch Abort exception
ev10: nop                    ; Data Abort exception
ev14: nop                    ; (Not used)
ev18: b      irq_handler ; Interrupt exception
ev1C: nop                    ; Fast Interrupt exception
; -----
.section .ospage, "awx" ; For "operating system" code
init:                   ; Initialise hardware and change to User mode

ldr    r0, =iobase      ; R0 = base of I/O space
mov    r1, #(irq_pbs2 | irq_pbs3) ; R1 = enable IRQs mask for S2/S3
strb   r1, [r0, #irq_enable] ; Actually enable the IRQs

mrs    ip, cpsr         ; Get current value of CPSR into IP (R12)
bic    ip, ip, #(ARM_PSR_i | ARM_PSR_f | ARM_PSR_mode_mask)
; Mask out bottom 5 bits. Also clear I and F
; bits; this enables the interrupts

orr    ip, ip, #ARM_PSR_mode_usr ; Set User mode
msr    cpsr, ip         ; Actually make the changes to CPSR

b      main             ; Now in User mode; jump to the main program

.end

```

Figure 20: Program source file *boot-pbirq-v1.s*

```

.section .ospage, "awx" ; For "operating system" code
.include "header-v3.s" ; Include definitions needed for this program
.global irq_handler    ; Make this label visible to other modules
irq_handler:           ; This code runs in Interrupt mode

ldr    r3, =iobase      ; R3 = base of Microcontroller I/O space
ldrb   r0, [r3, #irq_status] ; Read the IRQ Status register into R0

tst    r0, #irq_pbs2    ; Check if push-button S2 generated the IRQ,
; ie, is currently being pressed
movne  r1, #left_leds  ; If it is, prepare to turn on the left LEDs
strneb r1, [r3, #portA] ; Now actually do so

tst    r0, #irq_pbs3    ; Check if push-button S3 generated the IRQ
movne  r1, #right_leds ; If it did, prepare to turn on the right LEDs
strneb r1, [r3, #portA] ; Now actually do so

bic    r1, r0, #(irq_pbs2 | irq_pbs3) ; Clear the IRQs for S2 and S3
strb   r1, [r3, #irq_status] ; Acknowledge the interrupts

subs  pc, lr, #4        ; Return to whatever the processor was doing
; at the time of the interrupt

.end

```

Figure 21: Program source file *pb-irq-v1.s*

```

        .text                ; Ordinary program code follows
        .global main         ; Make this label visible to other modules
main:
    mov     r0, #0           ; Main program, running in User mode
                                ; Initialise a trivial counter in R0
count_loop:
    add     r0, r0, #1       ; Main program loop
                                ; Increment the counter; this gives the main
                                ; program something to do...
    nop
    b       count_loop       ; Waste a machine cycle doing nothing
                                ; Do this forever (or until stopped)
        .end

```

Figure 22: Program source file *null-main.s*

You should note a few things about this program:

- The program is written as a set of modules to make it easier to understand. In particular, *boot-pbirq-v1.s* (Figure 20) contains the initialisation routines, *pb-irq-v1.s* (Figure 21) contains the interrupt handler and *null-main.s* (Figure 22) contains the main program.
- The instructions at labels *ev00* to *ev1C* in *boot-pbirq.v1.s* form the ARM exception vector table that is located at address *0x0*.
- The first three instructions at the label *init* initialise the interrupt controller to allow the push-button switches *S2* and *S3* to potentially interrupt the ARM processor. This is done by setting the relevant bits in the IRQ Enable port to 1.
- The next four instructions switch the processor into User mode and enable Fast and ordinary interrupts.
- The interrupt service routine technically starts at the label *ev18*; this address contains a branch to the real handler at *irq_handler*. It is run every time the ARM processor processes an Interrupt exception; the code runs in the Interrupt processor mode.
- Accesses to the Microcontroller I/O space are all done using a “base + offset” method. In this case (in the file *pb-irq-v1.s* in Figure 21), the base address is contained in the register *R3* and the offset is specified directly in the *ldrb* and *strb* instructions (using equates defined in *header-v3.s*). Accessing the Microcontroller I/O space in this way is more efficient than loading individual addresses into multiple registers or into the same register multiple times.
- The following instructions, found towards the end of the interrupt handler, are worth considering:

```

        bic     r1, r0, #(irq_pbs2 | irq_pbs3)
        strb    r1, [r3, #irq_status]

```

These instructions acknowledge the interrupt by clearing the appropriate bits in the IRQ Status port. Ideally, a read-modify-write cycle should be used, but this program does not do so for simplicity.
- The main program, starting at the label *main*, implements a trivial counter using the register *R0*. This code represents the “real work” that a program might want to do instead of spending its time polling the hardware devices. This code runs in User mode.

Task 5: Debugging Interrupt Handlers

Use the GNU Tools to assemble and link the program *pb-irq-v1.elf* shown in Figures 19–22. The easiest way to do so is to type “**make -f pb-irq-v1.make**”. Use the Komodo debugger to download the program to the DSLMU Microcontroller Board; make sure you press **Reset** in Komodo after doing so! Use the debugger to trace through the program; take note, in particular, the effect of every instruction in the interrupt handler. Be ready to show the program running at full speed to the Laboratory assessor; pressing the push-buttons should show the LEDs being turned on.

Hint: The easiest way to trace through the interrupt handler is to set a breakpoint at the label `ev18`, then run the code at full speed by pressing **Run**. You might find the **Walk** button useful as well. Please consult *An Introduction to Komodo* for more information; this document appears in an Appendix or on your CD-ROM.

Question: What happens to the trivial counter running in the main program (in the file `null-main.s`)? Why does this happen?

Checkpoint 5: Signature:

Preserving the State of Execution

You should have discovered a major problem with the program `pb-irq-v1.elf` in Task 5: the interrupt service routine uses the registers R0-R3 in its code *without* taking into consideration the fact that these registers are almost certainly being used in the interrupted code! In other words, the interrupted code finds that its registers are mysteriously being corrupted. Such bugs are *extremely* difficult to find in real life, since they depend on timing issues that are hardware dependent.

The solution is to make sure that the interrupt handler *preserves the state of execution*. In particular, it must save the original values of all registers used in its code, and it must restore those registers to their original values on exit.

The ARM processor helps the interrupt handler do this for some of the registers. As shown in Figure 1 on page 85, each mode has its own R13 and R14 registers. This means that R13 and R14 in other processor modes are automatically preserved, since the interrupt handler code does not even get to access them directly. In the same way, other modes' CPSR is automatically transferred to `SPSR_irq`.

This does not help save the state of other registers, however. The best way to preserve their state is by saving them to the stack. Since the Interrupt processor mode has its own stack pointer, `R13_irq`, this can be done using code similar to the following:

```
irq_handler:
    str    r0, [sp, #-4]!    ; Save R0 to the stack (pointed to by SP_irq)
    str    r1, [sp, #-4]!    ; Similarly for R1-R3. NB: This assumes the stack
    str    r2, [sp, #-4]!    ; pointer has already been set up!
    str    r3, [sp, #-4]!
    ...
    ; Code to handle the interrupt
    ldr    r3, [sp], #4      ; Retrieve original value of R3 from the stack
    ldr    r2, [sp], #4      ; Similarly for R2-R0, in reverse order
    ldr    r1, [sp], #4
    ldr    r0, [sp], #4
    subs   pc, lr, #4        ; Return from IRQ handler; restores CPSR
```

The individual `str` and `ldr` instructions can be replaced by single `stm` and `ldm` instructions, respectively:

```
irq_handler:
    stmfd  sp!, {r0-r3}      ; Save R0-R3 to the stack (pointed to by SP_irq)
    ...
    ; Code to handle the interrupt
    ldmsd  sp!, {r0-r3}      ; Retrieve original values of R0-R3 from the stack
    subs   pc, lr, #4        ; Return from IRQ handler; restores CPSR
```

This can be further optimised by saving a *corrected* value of `LR_irq` to the stack and restoring it directly into PC later. Doing this allows you to call other (internal) functions using the `b1` instruction; you must remember that such functions will be run in Interrupt mode. The following code fragment shows how to save and restore `LR_irq`:

```

irq_handler:
    sub    lr, lr, #4      ; Calculate the correct return address in LR_irq
    stmfd  sp!, {r0-r3, lr} ; Save R0-R3 and return address to the stack
    ...                ; Code to handle the interrupt; the "bl" instruc-
                        ; tion is now permitted (since LR is saved)
    ldmsd  sp!, {r0-r3, pc}^ ; Retrieve original values of R0-R3 from the
                        ; stack, move return address into PC (ie, return
                        ; from IRQ handler), restore CPSR

```

Note the circumflex “^” that is part of the `ldm` instruction: whenever PC appears in the list of registers loaded by this instruction, specifying the circumflex makes the ARM processor automatically restore the Current Program Status Register.

Some hardware systems require the interrupt handler to be *reentrant*: the handler must be written in such a way that it itself can be interrupted by another IRQ of the same priority. This is often needed in systems (such as the ARM) that have many sources of IRQs but only one interrupt handler. Such a reentrant interrupt handler must save the corrected value of `LR_irq`, as above. In addition, the handler must clear the I bit in the CPSR at some appropriate point.

One important requirement for using the Interrupt mode stack pointer `SP_irq` is that that register must be initialised. This is usually done at initialisation; the boot code should initialise `SP_irq` to point to some location in memory set aside for the stack.

Figure 23 and Figure 24 show the previous program rewritten to preserve the state of execution. Carefully examine this program; as before, you should read the files on your CD-ROM, as those have more comments than are shown here. You will also need to refer to Figure 19 for the header file *header-v3.s* and to Figure 22 for the main program *null-main.s*. Modified or added instructions have been highlighted in **bold**:

```

.global _start          ; "_start" is where the code starts running
.extern main            ; "main" is defined in another file
.extern swi_handler     ; "swi_handler" is also defined elsewhere
.include "header-v3.s" ; Include various definitions
; -----
.section .zeropage, "awx" ; For code located at address 0x00000000
_start:                ; Start of the entire program
; ARM processor exception vector table
ev00:  b      init      ; Reset exception
ev04:  nop             ; Undefined Instruction exception
ev08:  nop             ; Software Interrupt exception
ev0C:  nop             ; Prefetch Abort exception
ev10:  nop             ; Data Abort exception
ev14:  nop             ; (Not used)
ev18:  b      irq_handler ; Interrupt exception
ev1C:  nop             ; Fast Interrupt exception
; -----
.section .ospage, "awx" ; For "operating system" code
init:                ; Initialise hardware and change to User mode
                    ; This code initially runs in Supervisor mode

    ldr    sp, =svc_stack_top      ; Initialise SP_svc
    mrs    r0, cpsr                ; Get current value of CPSR into R0
    bic    r0, r0, #ARM_PSR_mode_mask ; Mask out the bottom 5 bits
    orr    r1, r0, #ARM_PSR_mode_irq ; R1 = CPSR + IRQ mode
    msr    cpsr_c, r1              ; Switch into IRQ mode
    ldr    sp, =irq_stack_top      ; and set SP_irq appropriately

```

(Continued on the next page...)

(Continued from the previous page...)

```
    orr    r1, r0, #ARM_PSR_mode_fiq ; R1 = CPSR + FIQ mode
    msr    cpsr_c, r1                ; Switch into FIQ mode
    ldr    sp, =fiq_stack_top        ; and set SP_fiq appropriately

    orr    r1, r0, #ARM_PSR_mode_sys ; R1 = CPSR + System mode
    msr    cpsr_c, r1                ; Switch into System mode
    ldr    sp, =usr_stack_top        ; and set SP appropriately
    ; Note that User and System modes share the same registers

    ldr    r0, =iobase                ; R0 = base of I/O space
    mov    r1, #(irq_pbs2 | irq_pbs3) ; R1 = enable IRQs mask for S2/S3
    strb   r1, [r0, #irq_enable]     ; Actually enable the IRQs

    mrs    ip, cpsr                  ; Get current value of CPSR into IP (R12)
    bic    ip, ip, #(ARM_PSR_i | ARM_PSR_f | ARM_PSR_mode_mask)
    ; Mask out bottom 5 bits. Also clear I and F
    ; bits; this enables the interrupts

    orr    ip, ip, #ARM_PSR_mode_usr  ; Set User mode
    msr    cpsr, ip                  ; Actually make the changes to CPSR
    b      main                       ; Now in User mode; jump to the main program

; -----
; Stack space for the different processor modes
    .bss ; Use uninitialised memory for the stack
    .align ; Make sure the stack is word-aligned
    .skip 1024 ; Allow a 1KB stack for the User/System modes
usr_stack_top: ; "usr_stack_top" points to top of this stack
    .skip 512 ; Allow 512 bytes stack for IRQ mode
irq_stack_top: ; "irq_stack_top" points to top of this stack
    .skip 512 ; Allow 512 bytes stack for FIQ mode
fiq_stack_top: ; "fiq_stack_top" points to top of this stack
    .skip 512 ; Allow 512 bytes stack for Supervisor mode
svc_stack_top: ; "svc_stack_top" points to top of this stack
    .end
```

Figure 23: Program source file *boot-pbirq-v2.s*

```
.section .ospage, "awx" ; For "operating system" code
#include "header-v3.s" ; Include definitions needed for this program
.global irq_handler ; Make this label visible to other modules
irq_handler: ; This code runs in Interrupt mode
    sub    lr, lr, #4 ; Calculate the correct return address
    stmfid sp!, {r0-r3, lr} ; Save registers to Interrupt mode stack
    ldr    r3, =iobase ; R3 = base of Microcontroller I/O space
    ldrb   r0, [r3, #irq_status] ; Read the IRQ Status register into R0
    tst    r0, #irq_pbs2 ; Check if push-button S2 generated the IRQ,
    ; ie, is currently being pressed
    movne  r1, #left_leds ; If it is, prepare to turn on the left LEDs
    strneb r1, [r3, #portA] ; Now actually do so
    tst    r0, #irq_pbs3 ; Check if push-button S3 generated the IRQ
    movne  r1, #right_leds ; If it did, prepare to turn on the right LEDs
    strneb r1, [r3, #portA] ; Now actually do so
    bic    r1, r0, #(irq_pbs2 | irq_pbs3) ; Clear the IRQs for S2 and S3
    strb   r1, [r3, #irq_status] ; Acknowledge the interrupts
    ldmfd  sp!, {r0-r3, pc}^ ; Return to whatever the processor was
    ; doing at the time of the interrupt;
    ; restore registers R0-R3 and CPSR

.end
```

Figure 24: Program source file *pb-irq-v2.s*

Task 6: Interrupt Handlers and Stacks

Use the GNU Tools to assemble and link the program *pb-irq-v2.elf* shown in Figures 23–24. The easiest way to do so is to type “**make -f pb-irq-v2.make**”. Use the Komodo debugger to download the program to the DSLMU Microcontroller Board; make sure you press **Reset** in Komodo after doing so! Use the debugger to trace through the program; take note, in particular, how the initialisation code sets up the various stack pointers in the different processor modes, and how the interrupt service routine saves and restores the registers. Be ready to show the program running at full speed to the Laboratory assessor; pressing the push-buttons S2 and S3 should show the LEDs being turned on.

Question: What happens this time to the trivial counter running in the main program (in the file *null-main.s*)?

Checkpoint 6: Signature:

Task 7: Timer Interrupts

In Experiment 4 Task 4, you wrote a program *slower-flash.s* to flash the LEDs on the DSLMU Microcontroller Board at a frequency of 0.5 Hz (ie, the LEDs were to be on for one second, then off for one second). In that program, you used the Timer port to determine how long a delay should be.

Your final task for this experiment is to rewrite that program so that it uses hardware interrupts. You will need to consult the description of the Timer, Timer Compare and IRQ Status ports in the *DSLMU Microcontroller Board Hardware Reference Manual*; you can find that document in an Appendix or on your CD-ROM.

In particular, you will need to write the interrupt service routine; you should give this code the label `irq_handler` and place it in the source file *timer-irq.s*. The rest of the program has already been written for you; you should examine the source files *header-v3.s*, *boot-timer.s* and *null-main.s* in your `~/exp5` directory. Use the supplied make-file *timer-irq.make* to create the executable file *timer-irq.elf*. You can do this by typing:

```
make -f timer-irq.make
```

Make sure your interrupt service routine preserves the state of execution! Measure the frequency of the flashing LEDs using the oscilloscope. Show your running program to the Laboratory assessor; make sure you are ready to explain how your code works.

Hint: The Timer Compare port at address 0x1000000C allows interrupts to be generated at a frequency of between 1000 times a second to just under 4 times a second: still too fast for this task! The best way to solve this problem is to keep a separate count of the number of interrupts seen, in software. You should use the variable `irq_count` for this particular task, as the code in *boot-timer.s* automatically resets that variable to zero at initialisation. You can do this by including the following line in your file *timer-irq.s*:

```
.extern irq_count
```

Checkpoint 7: Signature:

Task 8: (No Credit) Undefined Instruction Exception

Write an undefined exception routine that that execute the undefined instruction “div10 Rd, Rm, #10”, with action as “Rd = Rm/10”. Use the binary encoding 0xFEE(Rm)(Rd)00A to represent your new instruction.

Checkpoint 8: (No Extra Credit) Signature:

Task 9: Mini Project- (Extra Credit × 5) Task Scheduling

Build a scheduler that is invoked via a system call from two or more application programs (a “*cooperative scheduler*”). The scheduler should work in a round robin fashion. Each program during the course of its execution invokes the system scheduler via a swi call. The scheduler suspends the current process, saves its state on the process stack, and invokes the next process on the queue to run. The current active process will eventually invoke the scheduler, which in turn suspend it and pass control to the next process in the circular list.

Checkpoint 9: (Extra Credit × 5)..... Signature: