Experiment 4:
Interfacing to Input/Output Devices

This experiment further consolidates the programmer's view of computer architecture. It does this by showing you how microprocessors interface to the real world. This experiment guides you through some of the details of programming for microcontroller-based systems and interfacing to input/output devices on such systems.

Aims

This experiment aims to:

- show the support that the ARM instruction set architecture has for interfacing to input/output devices,
- illustrate the ARM logical instructions and how they can be used for bit manipulation,
- demonstrate simple peripherals, as well as the use of timers and counters in microcontroller-based systems, and
- give you more examples of writing and debugging assembly language programs for the ARM microprocessor.

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,
- skim-read the sections on Programming Style in Experiments 1 and 2,
- read through *An Introduction to Komodo*, which you can find either in an Appendix or on your CD-ROM,
- quickly read through the relevant material from your lecture notes for this course, and
- skim through the *DSLMU Microcontroller Board Hardware Reference Manual*. You can find this document in an Appendix or on your CD-ROM.

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

- browse through the Companion CD-ROM, if you haven't done so already, 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/unsw/elec2041/labs-src/exp4* 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 ~/exp4
cd ~/exp4
cp ~elec2041/cdrom/unsw/elec2041/labs-src/exp4/* .
```

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 *unsw/elec2041/labs-src/exp4* directory on your CD-ROM instead.

The Hardware

Take the time to examine the DSLMU Microcontroller Board in front of you. As you saw in Experiment 1, this board actually consists of two printed circuit boards, called the MU Board and the Expansion Board. These boards have been designed to show you the typical development environment used for embedded controllers in the early years of the 21st Century (and it surely can't get better than that!).

The basic system consists of a software-programmable processor (in this case, the ARM microcontroller) and a number of peripheral input/output systems. The microcontroller itself includes, among other things, some parallel input/output ports, serial interfaces, timer units and a simple interrupt controller.

Much greater input/output capabilities are provided by the two Field Programmable Gate Arrays (FPGAs) that are on the DSLMU Microcontroller Board. These FPGAs can be configured at will, to suit a particular application. This means that both the software and the hardware on the board are programmable and available for your use. By the way, the ability to use FPGAs for customisable hardware has had considerable influence on the design of embedded systems in the past decade. As FPGAs have increased in capacity, it has become feasible to use them for quite significant tasks and systems.

On the DSLMU Microcontroller Board, the particular FPGA that you will most likely use is the Xilinx Spartan-XL; this device has up to 10,000 system gates available. The board also contains a Xilinx Virtex-E FPGA with up to 412,000 system gates; this device is large enough to design a high-performance custom coprocessor.

The board in front of you has many peripherals available for your use; you can read about them in the *DSLMU Microcontroller Board Hardware Reference Manual*. However, this experiment will only use the eight Light Emitting Diodes (LEDs) in the top left-hand corner of the MU Board.

The Software

This experiment will use the Komodo debugger, a low-level debugger specifically written for the DSLMU Microcontroller Board. You have already used this debugger to download your program in Experiment 1.

Please note that you will need to read through (and understand!) *An Introduction to Komodo* **before** you can proceed any further! That *Introduction* will tell you how to start the Komodo debugger, how to load a program onto the Board and how to debug it.

You will still be using the GNU Assembler and the GNU Linker to create your programs. You can also use the **`make`** command, as before.

Address Space on the Board

The ARM microcontroller on the DSLMU Microcontroller Board communicates with its internal I/O ports, and with other peripheral devices, through what is known as a *memory-mapped input/output space*. The address space (also called a memory map) on this Board starts at address 0x00000000 and ends at address 0xFFFFFFFF (the highest address that can be accessed as a 32-bit word is 0xFFFFFFFC). Most of this address space is empty (not used and reserved for the future).

Read/write memory (RAM) occupies the space beginning at address 0x00000000 (ie, from the bottom of the address space). The DSLMU Microcontroller Boards in the Laboratory have at least 512 KB of RAM present, which means that the highest *byte* address of this RAM is 0x0007FFFF, and the highest *word* address is 0x0007FFFC. You should remember that the ARM is a 32-bit processor that is byte-addressable; this means that words adjacent to each other in memory differ by 4 in their address. In addition, all word accesses must be *aligned* on a word boundary: the lowest two bits of the address must be zero for such accesses.

Question: How does the ARM processor respond if you make word accesses that are not aligned on a word boundary? You may want to refer to your lecture notes to find out, or see page A2-26 of the *ARM Architecture Reference Manual* (page 58 of the PDF document); you can find this document on your CD-ROM in the *reference* directory.

Figure 1 gives a diagrammatical representation of the address space on the DSLMU Microcontroller Board. Please note that this figure is not to scale! You can consult Table 1 in the *Hardware Reference Manual* if you would like more information.

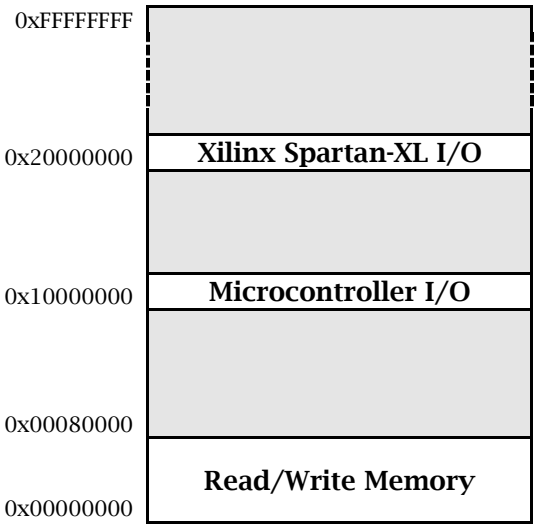

Figure 1: Memory Map on the DSLMU Microcontroller Board

As you can see from Figure 1, the microcontroller's own internal input/output space starts at address 0x10000000. This internal I/O space ends at address 0x1FFFFFFF. However, most of this space is "reserved for future use". In addition, each *port* (location) in this I/O space is only 1 byte (8 bits) wide and only appears in the least significant byte of a word. In other words, only addresses 0x10000000, 0x10000004, 0x10000008, etc, are valid, and these can only be accessed in byte-sized quantities.

The Xilinx Spartan-XL FPGA has its own input/output address space starting at address 0x20000000 and ending at address 0x2000001F. Each byte in this space can be accessed individually.

The following table shows the ports that are defined in the microcontroller's internal I/O space; each port is listed as an *offset* from address 0x10000000. You should consult the *Hardware Reference Manual* for an in-depth discussion of each port.

Offset	Mode	Port Name	Function
0x00	R/W	Port A	Bidirectional data port to LEDs, LCD, etc.
0x04	R/W	Port B	Control port (some bits are read only)
0x08	R/W	Timer	8-bit free-running 1 kHz timer
0x0C	R/W	Timer Compare	Allows timer interrupts to be generated
0x10	RO	Serial RxD	Read a byte from the serial port
0x10	WO	Serial TxD	Write a byte to the serial port
0x14	WO	Serial Status	Serial port status port
0x18	R/W	IRQ Status	Bitmap of currently-active interrupts
0x1C	R/W	IRQ Enable	Controls which interrupts are enabled
0x20	WO	Debug Stop	Stops program execution when written to

Table 1: Microcontroller I/O Space

This experiment only uses three ports in the Microcontroller I/O space: **Port A**, the **Timer** port and the **Timer Compare** port. Port A is used to access the eight LEDs on the MU Board

as well as the LCD module on the Expansion Board; this port is *bidirectional:* it can be used for input and/or output. The Timer and Timer Compare ports access the free-running timer; this timer is discussed later in this experiment.

Using Komodo and the Komodo ARM Environment

To start using the DSLMU Microcontroller Board, you need to do three things:

1. Make sure that the Boot Select switch (in the bottom right-hand corner of the MU Board) is in the correct position, as shown in Figure 2,
2. Turn the power on using the On/Off Switch, and
3. Start the Komodo debugger on the Host PC.

Setting the Boot Select switch to the position shown in Figure 2 makes sure that the Komodo ARM Environment is started every time the power is turned on to the board; you should actually see the words "Komodo ARM Environment" on the LCD module. This Environment consists of software that runs on the board itself; it is sometimes called the "back-end". It communicates with the Komodo debugger running on the Host PC through the serial cable.

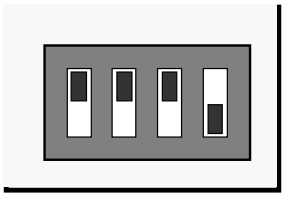

Figure 2: Boot Select Switch

Now that you have read *An Introduction to Komodo*, you should know that **`kmd`** is the command to start the Komodo debugger. This debugger (also called the "front-end") synchronises itself with the Komodo ARM Environment running on the actual board.

When you start Komodo for the first time, you will notice that the system is in a state known as "Reset". You can also reach this state by pressing the **Reset** button in Komodo at any time. When the ARM processor is reset, the PC register is set to 0x00000000 and the Current Program Status Register CPSR is set to 0x000000D3. In an ordinary system, the ARM processor would immediately begin executing instructions at address 0x00000000. On this board, however, the Komodo ARM Environment halts the processor so that you can observe its state.

> **Note:** Almost every number that you type into Komodo must be entered in *hexadecimal*, not decimal. You can type in these numbers with or without the standard "`0x`" prefix, but you must remember that, "`0x`" or no "`0x`", the number is always treated as if it is in hexadecimal.
>
> In addition, almost every number is *shown* in hexadecimal as well, usually without the "`0x`" prefix.

You should already know, from previous experiments, that debuggers allow you to execute your programs in a number of ways. You can, of course, just press the **Run** button to begin normal execution. This is suitable for running programs that have no problems, but gives very little help if there are bugs present. You can also use the **Single-Step** button to execute a single instruction before halting the processor once more. This makes it possible to observe exactly what each instruction does. You will want to display a disassembled listing of your code when doing this.

Apart from these, you can also use the **Multi-Step** button to execute more than one instruction at one time. In addition, you can use the **Walk** button to execute your instructions one-at-a-time in slow motion. You should read *An Introduction to Komodo* for further details.

Input/Output Ports

To be at all useful, a computer system must have some capability for input and/or output to the "real world". The simplest form of I/O is provided by a *parallel port*, which essentially maps a memory location to some external hardware. In the case of an output port, this means that the bits "stored" in this memory location are used to control that hardware, such setting Light Emitting Diodes (LEDs) on or off. In the case of an input port, "reading" the memory location actually checks the state of some real-world device. For example, the state of one of the bits so read might actually come from a push button switch.

Real-world devices that are attached to the processor in this way, and which inhabit the processor's I/O address space, are known as *peripherals*. The address of such peripherals in the I/O address space is often known as a *port*.

Some processors, such as the Intel Pentium, have a special I/O address space that is outside of the "normal" memory map. Such processors have special instructions to access the peripherals attached to this I/O address space.

However, most RISC processors (including the ARM) only have a single address space. Thus, some memory locations must be sacrificed for input/output. This is (hopefully!) not a significant problem with 4 GB of address space available.

Often, peripheral devices do not use the full 32-bit data bus: the most common devices have only an 8-bit interface. This is reflected in the DSLMU Microcontroller Board, where almost all of the peripherals will be accessed eight bits (or fewer) at a time. You should therefore remember to use the *byte-sized* load and store instructions (`ldrb` and `strb`, respectively) when communicating with the I/O ports in this experiment.

Bit Manipulation

Although some I/O devices are byte-wide (or, occasionally, larger), many inputs and outputs are smaller—often a single bit in size. For example, the position of a switch or a button can be represented with a single bit. It is usual to cluster several (functionally connected) I/O bits together into partial or full bytes to simplify hardware requirements. The LEDs used in this experiment illustrate this concept quite nicely.

On most processors, the smallest quantity that can normally be addressed is a byte. This means that accessing individual bits (ie, *bit addressing*) must be done in software. Bits are normally addressed such that bit 0 is the least significant bit.

The ARM processor can alter a single *byte* in memory using the `strb` instruction. This instruction does too much if all you need to change is a single bit: the other seven bits would also be written (and thus possibly changed)! The solution is to use what is known as a *read-modify-write cycle*.

A read-modify-write cycle consists of three steps:

1. Read the current value of the port,
2. Modify just the bits that need to be changed, and
3. Write the new value to the port.

You can often read the current (ie, original) value of a port by using the `ldrb` instruction; you need to check the documentation for that port to see whether this is actually possible or not (since some ports are write-only).

Once the current value of the port is in a register, you can change the bits that need to be modified. You can do this using the ARM *logical instructions* `and`, `orr`, `bic` and `eor`. These instructions perform the operations shown in Table 2:

Instruction	Action	Comments
`and r1, r2, m1`	*r1* = *r2* AND *m1*	Can be used to clear bits
`orr r1, r2, m1`	*r1* = *r2* OR *m1*	Can be used to set bits
`bic r1, r2, m1`	*r1* = *r2* AND (NOT *m1*)	Can be used to clear bits
`eor r1, r2, m1`	*r1* = *r2* XOR *m1*	Can be used to invert bits

Table 2: ARM Logical Instructions

In the case of a read-modify-write cycle, the registers *r1* and *r2* in Table 2 would be the same: it would be the register containing the current value of the port. Value *m1* would be an appropriate *bit mask*: a value that has some bits set to 1 and other bits set to 0. This bit mask *m1* would appear either in some register or as an immediate operand.

It is a simple and worthwhile exercise to learn the basic bit masks by heart. Table 3 lists the bit masks necessary to set, clear or invert a single bit:

Bit	Mask for `orr`, `bic`, `eor`		Mask for `and`	
	Hex	Binary	Hex	Binary
7	0x80	0b10000000	0x7F	0b01111111
6	0x40	0b01000000	0xBF	0b10111111
5	0x20	0b00100000	0xDF	0b11011111
4	0x10	0b00010000	0xEF	0b11101111
3	0x08	0b00001000	0xF7	0b11110111
2	0x04	0b00000100	0xFB	0b11111011
1	0x02	0b00000010	0xFD	0b11111101
0	0x01	0b00000001	0xFE	0b11111110

Table 3: Bit masks for changing single bits

A few examples should illustrate how the logical instructions can be used to implement a read-modify-write cycle:

```
ldrb   r0, [ port_address ]   ; Read the current value of the port
orr    r0, r0, #0x20          ; Set bit 5 (ie, set the bit to 1)
strb   r0, [ port_address ]   ; Write the new value to the port

ldrb   r0, [ port_address ]
and    r0, r0, #0xFB          ; Clear bit 2 (ie, set the bit to 0)
strb   r0, [ port_address ]

ldrb   r0, [ port_address ]
bic    r0, r0, #0x04          ; Clear bit 2 (ie, set the bit to 0)
strb   r0, [ port_address ]   ; (Same as previous example; uses "bic" vs. "and")

ldrb   r0, [ port_address ]
eor    r0, r0, #0x80          ; Invert (toggle) bit 7
strb   r0, [ port_address ]
```

The last example is particularly useful: the `eor` instruction allows you to invert (toggle) bits without knowing their current state. In this example, the instruction inverts bit 7: if that bit was a 0, it makes it a 1; if it was a 1, it makes it a 0.

Note that the `bic` instruction is very similar in purpose to `and`. However, the `and` instruction uses an *inverted* parameter when compared to `bic`.

> **Note:** You need to remember that all memory addressing on the ARM processor *must* be done relative to a register. In other words, the argument *`port_address`* for the `ldrb` and `strb` instructions must be a register or a register + offset.

Task 1: Flashing Lights

Examine the program *flash-v1.s*. You should open this file using the **kate** editor; the listing in Figure 3 has had most of its comments removed to save space:

```
        .text                       ; Executable code follows

_start: .global _start              ; "_start" is required by the linker
        .global main                ; "main" is our main program

        b       main

        .set    portA,  0x10000000  ; Address of Port A in the I/O space
        .set    value1, 0b11111111  ; Value to turn the LEDs on
        .set    value2, 0b00000000  ; Value to turn the LEDs off

main:                               ; Entry to the function "main"
        ldr     r1, =portA          ; Load address of Port A into register R1

main_loop:                          ; Infinite loop...
        mov     r0, #value1         ; Load value1 into R0 (to turn the LEDs on)
        strb    r0, [r1]            ; Write the byte to Port A
        mov     r0, #value2         ; Load value2 into R0 (to turn the LEDs off)
        strb    r0, [r1]            ; Write the byte to Port A
        b       main_loop           ; Do this forever (or until stopped)

        .end
```

Figure 3: Program *flash-v1.s*

Assemble and link this program in the usual way. Run the Komodo debugger and download the program to the DSLMU Microcontroller Board. Start the program running; the necessary instructions to do this are in *An Introduction to Komodo*.

The following commands assemble and link *flash-v1.s* into *flash-v1.elf:*

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

If you prefer, you can use the **make** command instead. The appropriate make-file is called *flash-v1.make*, and can be used by typing:

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

Once the program is running (you should see the eight LEDs in the top left-hand corner of the MU Board turn on), connect the oscilloscope probe to test point TP0 (in the top right-hand corner of the board). Remember to connect the probe's ground wire to the horizontal bar (TP8–TP9)! Adjust the oscilloscope so that you can see the square wave pattern produced on the test point (note that TP0 corresponds to bit 0 of Port A, TP1 to bit 1, and so on).

Once you can see the square wave pattern, count the number of instructions in the program loop and calculate the number of instructions executed per second in units of MIPS (Millions of Instructions Per Second). Report your findings to the Laboratory assessor.

What would be the loop frequency (the flash rate of the LEDs) if one additional instruction were to be added to the loop? Would it make any difference where that one extra instruction was placed?

Checkpoint 1: ..... Signature: .....

Task 2: Flashing Lights with Delay

The program *flash-v2.s* in Figure 4 is similar to *flash-v1.s* above. The major difference is that two delay loops (calls to the function `delay`) have been inserted into the main loop.

These delay loops slow the rate of turning the LEDs on and off so that you can see this happening! If possible, **please read the version of this file on your CD-ROM**: it has *many* more comments, and explains why setting up a stack is necessary. You should also look at the file *flash-v3.s* on your CD-ROM for another version of the same program:

```
        .text                       ; Executable code follows

        .set    portA,   0x10000000 ; Address of Port A in the I/O space
        .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

_start: .global _start              ; "_start" is required by the linker
        .global main                ; "main" is our main program

        ldr     sp, =stack_top      ; Initialise the stack pointer
        b       main                ; and jump to the main program

; ------------------------------------------------------------------------
; Function: void main (void)

main:                               ; Entry to the function "main"

        ldr     r1, =portA          ; Load address of Port A into register R1

main_loop:                          ; Infinite loop...
        mov     r0, #value1         ; Load value1 into R0 (to turn the LEDs on)
        strb    r0, [r1]            ; Write R0 to Port A

        str     r1, [sp, #-4]!      ; Save R1 to the stack to conform to ATPCS
        ldr     r0, =waitval        ; R0 = number of loops to wait
        bl      delay               ; Delay the program by doing nothing useful
        ldr     r1, [sp], #4        ; Restore R1 from the stack

        mov     r0, #value2         ; Load value2 into R0 (to turn the LEDs off)
        strb    r0, [r1]            ; Write R0 to Port A

        str     r1, [sp, #-4]!      ; Save R1 to the stack (no need to save R0)
        ldr     r0, =waitval        ; R0 = number of loops to wait
        bl      delay               ; Delay the program by doing nothing useful
        ldr     r1, [sp], #4        ; Restore R1 from the stack

        b       main_loop           ; Do this forever (or until stopped)

; ------------------------------------------------------------------------
; Function: void delay (int delaycount)

delay:                              ; Function: delay by wasting time in a loop
        subs    r0, r0, #1          ; Decrement the number of cycles to wait
        bne     delay               ; Repeat the loop if not finished
        mov     pc, lr              ; Finished: return to the caller

; ------------------------------------------------------------------------
; Stack space

        .bss                        ; Allocate uninitialised memory
        .align                      ; Make sure it is word-aligned

        .skip   2048                ; Allow 2KB for the stack
stack_top:                          ; Top of stack pointer

        .end
```

Figure 4: Program *flash-v2.s*

Assemble and link the program *flash-v2.s* in the usual way. Start the program running using Komodo, then use the oscilloscope to measure and calculate the frequency of the flashing LEDs (ie, the rate at which the LEDs flash).

Using Komodo, change the value of `waitval` until you get a frequency of 1 Hz for the flashing LEDs, as measured on the oscilloscope. What value of `waitval` gives you this frequency? Report your findings to the Laboratory assessor.

Checkpoint 2: ..... Signature: .....

Timers and Counters

In the previous tasks, you observed that delays can be generated by calculating the time it takes the ARM processor to execute a section of code. However, this is a very poor way of producing a delay for the following reasons:

- A small change in your code can drastically alter its timing. For example, how would you change the value of `waitval` to maintain the same flash frequency if a `nop` ("do nothing") instruction was added just after "`subs r0, r0, #1`" in Figure 4?
- The execution time of a block of code may vary in an unpredictable way due to cache hits or misses. Why this is so is beyond the scope of this course.
- Hardware interrupts, the topic of Experiment 5, could insert additional "invisible" delays into the code, also in an unpredictable way.
- If the program is running in a multi-tasking environment, it could be suspended for an indefinite (and arbitrary) time by the operating system.
- The code could simply be ported to a system with a different clock frequency—this is quite common in real life, by the way!

The only way for a program to accurately measure time and provide delays is by using an external fixed-frequency time reference that is not affected by any activity in the system. Such a time reference is usually provided by a *timer counter*. This particular hardware peripheral provides three closely-related functions:

- a *counter*, a hardware circuit that increments or decrements a value based on an external stimulus,
- a *timer*, a counter circuit that counts the pulses of a regular clock signal, and
- a *prescaler*, a divider (also called a modulo-*n* counter) that is used to reduce the frequency of pulses coming in to the timer counter. This has the effect of reducing the number of counts a timer counter needs to make.

Figure 5 shows a typical timer counter; in fact, the peripheral shown contains *both* a counter *and* a timer. You can think of a timer as a peripheral that counts a known (but usually programmable) number of clock pulses generated by an oscillator; counting these pulses allows you to know when a particular interval of time has elapsed. Since oscillators (clocks) in computer systems are usually much faster than what is required, you have the option of using a prescaler to slow the clock down to an acceptable rate.

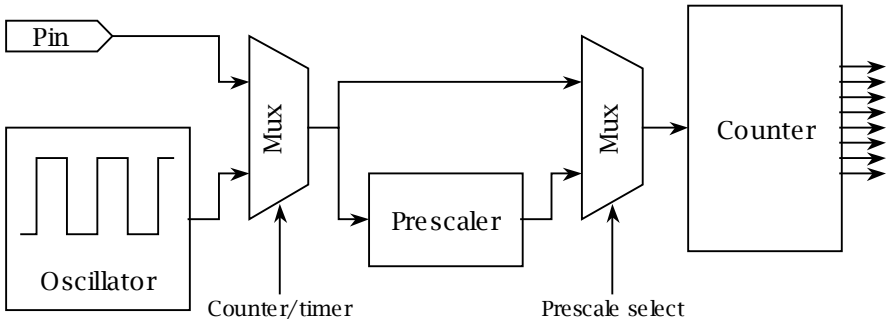

Figure 5: A typical timer counter circuit

Since the hardware required to implement a timer counter is relatively simple, most microcontrollers have one or more of them as part of the physical integrated circuit.

There are actually a number of possible implementations of timer counters. These include free-running timers, one-shot timers and reloadable timers.

Free-running Timers

Free-running timers have an integer value that is incremented or decremented on each clock pulse. This value has a fixed word length, usually 8 or 16 bits in size, and will cycle modulo its word length raised to the power of two. (In other words, the value will roll around from its highest possible value to its lowest, or vice versa). The value is software-readable, and thus can be used to calculate a time interval by comparing the current reading with a previous one.

Note that there will always be some uncertainty when reading values from free-running timers due to clock resolution: the value may just be about to change when it is read, meaning that the value so read may be up to one clock tick out. Such errors are *not* cumulative as the timer continues to run.

Free-running timers sometimes allow the counter to be written to by the processor; doing so may introduce cumulative timing errors. In addition, many free-running timers have one or more *comparison registers*. Such registers can be set up so that when the timer reaches the value so programmed, it generates an output requesting attention (usually in the form of an *interrupt*).

The DSLMU Microcontroller Board provides a free-running timer peripheral that you can access using ports 0x10000008 and 0x1000000C. These ports are explained in greater detail on pages 8 and 9 of the *Hardware Reference Manual*.

One-shot Timers

A *one-shot timer* is a somewhat more sophisticated timer that counts to a predetermined number of clock pulses, then signals the fact and stops. The predetermined number of clock pulses (the count) is user-definable. Typically, this type of timer will count down and stop at zero, and is usually able to provide an interrupt signal. This sort of timer is useful when you know the length of time needed, but do not know exactly when that length of time should start.

Figure 6 compares a free-running timer with a one-shot timer used continuously. Every time the one-shot timer reaches zero, it requests attention from the processor and must wait until the processor responds (this may take more time than the timing interval *m*, or it may take less). Note that while the free-running timer is able to maintain precise timing, the one-shot timer looses it during the time it waits for the processor.

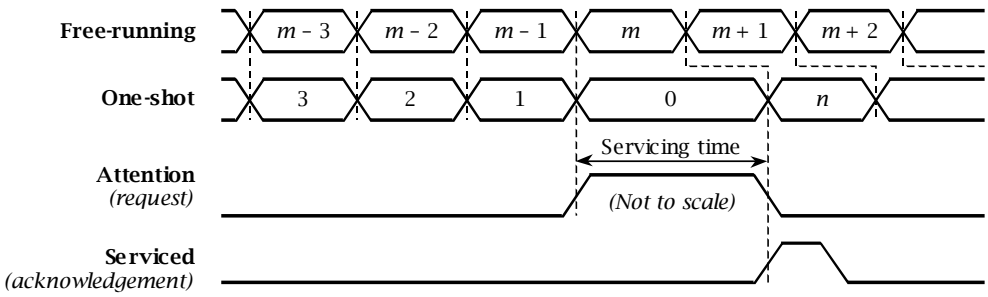

Figure 6: Free-running and one-shot timers compared

Reloadable Timers

Reloadable timers are similar in function to one-shot timers, except that they do not stop at zero. Instead, these timers reload their counters with a value stored in a separate register and start counting again. This means that reloadable timers can be programmed as modulo-*n* counters (where *n* is the "value stored in the separate register" already mentioned).

The main advantage that reloadable timers have is that they allow continuous time intervals to be generated without needing a processor to intervene at the end of every such interval.

As well as providing a regular source of processor interrupts, reloadable counters are used to provide programmable clocks within a microcontroller-based system. For example, one reloadable timer on the system may be dedicated to providing a *baud clock* for the serial communications port: it would take the system clock and divide it down into a steady stream of pulses. These pulses would then determine just when a bit of data would be sent to the serial port transmitter or read from the serial port receiver.

Task 3: The Free-Running Timer

Copy the file *flash-v2.s* in Figure 4 and call it *timer-flash.s*. Modify this program to read the free-running timer at address 0x10000008, so that the LEDs are alternatively turned on and turned off every time the timer value reaches 0x00. Assemble and link your program as usual, and use Komodo to download it to the DSLMU Microcontroller Board and run it. Using the oscilloscope, measure the frequency of the flashing lights. Show your working program to the Laboratory assessor.

Hint: You may want to read the first paragraph of the Timer port description in the *Hardware Reference Manual*. You can find the appropriate paragraph on page 8 of that document; the Manual can be found on your CD-ROM or as an Appendix.

Warning: It is very tempting to solve this task by writing a value to the Timer port. Do *not* do this, as it will cause the timer to loose time—up to 1 ms every time this is done (see Figure 6 for an illustration of this)! Instead, think through the possibility of using *two* loops to read the Timer port…

Checkpoint 3: ..... Signature: .....

Task 4: Slower Delays using the Timer

Copy the program *timer-flash.s* that you modified for Task 3 and call it *slower-flash.s*. Reduce the frequency of the flashing LEDs to 0.5 Hz (ie, the LEDs should be on for one second, then off for one second). Make sure your program is written in a modular fashion (using functions). Your delay function should ideally take a parameter that tells it how many milliseconds to wait. Make sure you conform to the ATPCS; see the file *flash-v2.s* for an example of how to set up a stack.

Assemble and link this file as usual, then run it on the DSLMU Microcontroller Board using the Komodo debugger. Use the oscilloscope to measure the frequency of the LEDs. Show your working program to the Laboratory assessor.

Hint: Remember that one second is 1000 ms, *not* 1024 ms! Once again, do *not* write to the Timer port; instead, consider reading the Timer port, then reading it again to see if the value has changed…

Warning: This task is not as easy as it sounds. You should *definitely* spend time thinking about this problem *before* you come into the Laboratory!

Checkpoint 4: ..... Signature: .....

Task 5: Traffic Lights Controller

Up to now, you have treated the eight LEDs on the DSLMU Microcontroller Board as a single entity: either all LEDs were turned on or all were turned off. These LEDs can be controlled individually, however: each LED is connected to a single bit of Port A in the Microcontroller I/O space, at address 0x10000000. Writing a 1 to a bit in this port will turn the corresponding LED on; writing a 0 will turn it off. Reading this port will return the last value written to it. The actual correspondence of LEDs to bits is shown in Figure 7:

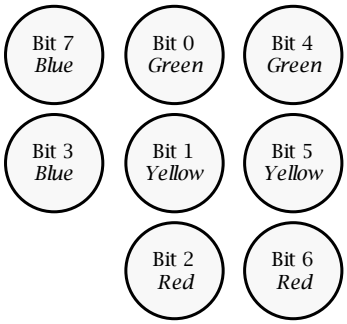

Figure 7: Correspondence of LEDs to bits in Port A

Write a program in ARM assembly language that treats the LEDs as two sets of traffic lights guarding a typical intersection. These traffic lights should follow a continuous non-stop cycle of allowing "traffic" on one side first, then on the other. Table 4 shows the LEDs that should be on for one cycle of such a sequence.

State	Left LEDs	Right LEDs	Wait for
1st	Red	Red	1 sec
2nd	Green	Red	2½ sec
3rd	Yellow	Red	1 sec
4th	Red	Red	1 sec
5th	Red	Green	2½ sec
6th	Red	Yellow	1 sec

Table 4: Traffic lights sequence

Please note that you should write your program in a modular fashion. In particular, you should have just *one* function that causes the program to be delayed. Your code must be well-commented and conform to the ATPCS.

Use the Komodo debugger to download your program to the DSLMU Microcontroller Board. Show your working program to the Laboratory assessor. You must also be prepared to display any of the port signals on the oscilloscope.

Checkpoint 5: ..... Signature: .....

Credits: Jim Garside at the University of Manchester contributed some of the sections in this experiment.