Durham University

Real Time Computing Course

Reference Notes

Table of Contents

Where is the software?....2
Access to the Lab....2
Starting an Exercise....2
- Programming....2
- Failure to Connect....3

Parallel I/O, LEDs and Joystick....4
- Programming Parallel Ports....4
- Joystick....5

Binary Manipulation....6
- AND, OR, XOR, INVERT Examples....6
- Left and Right Shift Examples....6

Analogue Output (DAC)....7
Analogue Input (ADC)....8
- Analogue to Digital Converter....8
- Register Use....8
- Initialisation....8
- Using the ADC....8

Pulse Width Modulation (PWM)....10
- PWM Pin configuration....11
- PWM Frequency....11
- PWM Pulse Width....11
- Start the PWM....12
- Changing Pulse Width....12

Timers....13
- Timer Configuration....13
- Using a timer as a counter....14

Interrupts....16
- Set interrupt type....16
- Set the address of the interrupt service routine....16
- Enable the Vectored Interrupt....16
- Write your interrupt service routine....17

Example: Getting Interrupts from a Timer....17
Printing to the LCD Screen....18
- Printf and simplePrintf....18
- Scrolling Text Display....18
- Graphics Control....19

R1

Where is the software?

All the software can be found on the NPCS machines under Academic Software, Engineering and Computing Sciences. It is all free and can be installed on your own PC, but will be of little use to you without the hardware **which must on no account be removed from the lab**.

Access to the Lab

There is a limited number of ARM processor boards available and at times there is great demand for the use of the computing lab. The exercises you are required to do in this course have been designed so that you have enough time to complete them to a reasonable standard in the 2 hour lab slot to which you have been allocated. Access to the lab is not currently possible during evenings and weekends.

Starting an Exercise

Each new program you write should be saved in its own project folder with its own "makefile".

1. Create a folder to work in for your new program. (It is recommended to use your CIS account for this – it is automatically mounted as the J: drive.)
2. Copy the standard **makefile** from Duo or the T: drive to your folder.
3. Open Notepad++, create a new file and use ***File/Save as...*** to save it as a C file, eg ex1.c, in your folder. It must have the .c file extension.
4. Type in your C program and save it.
5. Open your copy of the makefile in Notepad++ and set ***EXERCISE*** to the name of your program (without the .c file extension, eg EXERCISE = ex1).
6. Decide if you want to use the microcontroller's external memory. If you do, uncomment the ***EXTMEM*** line, otherwise comment it out (by means of a "#" character at the start of the line). You need EXTMEM if you want to use the LCD or need more than about 40K of RAM memory. You cannot use the LEDs with EXTMEM.
7. Save the makefile (Note: it **MUST** be called just makefile, without a file extension)
8. To compile, select ***make*** from the ***macro*** menu. To remove everything except the c file and makefile, choose ***make clean*** from the macro menu.

Programming

1. To download your program into the flash memory of the microcontroller, open "***Flash Magic***".
2. Check the following settings:

   COM Port: COM 3 (but see below, "Failure to Connect")

   Baud Rate: 115200

   Device: LPC2478

Interface: None(ISP)

Oscillator(MHz): 12.0

3. Tick the "***Erase blocks used by Hex File***" box.
4. Browse to the hex file produced by the compiler (it will be in your project folder and have the same name as your project with a .hex extension) and select (open) it.
5. You can tick "***Verify after programming***" if you like, but make sure the other options here are **NOT** ticked. Verification checks that the code has been correctly programmed. In practice verification is not normally necessary.
6. Click the "***Start***" button to start programming and observe the messages to make sure it is successful. If the programmer fails to connect to the board, first check that the board is connected to the PC via the usb cable. If it is, refer to the note below.
7. After programming you may need to press the reset button on the ARM interface board to start your program. The program will remain in the flash memory even after the power is removed.

Note

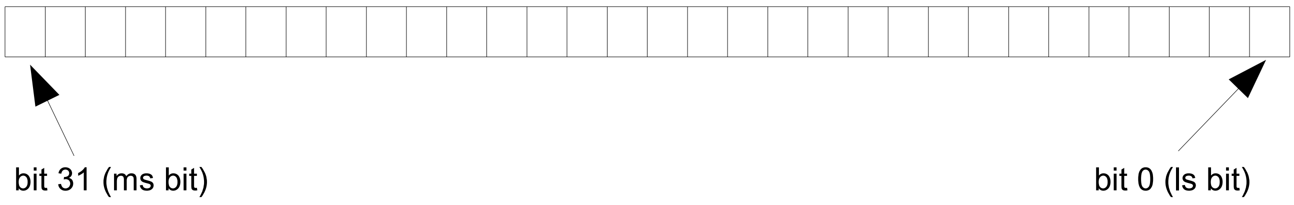
The settings you enter above will be remembered by flash magic as long as you stay logged in, but each time you log in you will need to check them again.

Failure to Connect

If Flash Magic cannot connect to the board, the COM port may not be correct. On the desktop, right click "My Computer" and select "properties". Click on "hardware" , then select "Device Manager" (ignore the warning message – you only want to view the settings). Expand the "Ports (COM & LPT)" list and you should see "USB Serial Port" with the correct COM number. Use this number in the dialogue for Flash Magic instead of COM3.

Parallel I/O, LEDs and Joystick

The microcontroller has 5 parallel i/o ports (general purpose i/o) named P0 to P4. Each port controls 32 pins, each pin may be an input or an output. We use part of P0 for the joystick (up, down, left, right and centre push buttons) and all of P3 for the LEDs.



Programming Parallel Ports

Each port has a set of 32 bit registers. Each bit in each register controls the corresponding pin. Thus, for example bit 12 for the P3 registers controls pin P3.12 (port 3 bit 12). A set of of FIOx registers as shown below is used to control each port x (eg FIO0DIR for P0).

Register	Function
FIOxDIR	1 in a bit position makes that i/o bit to be an output, 0 makes it an input
FIOxPIN	The current state of digital port pins can be read from this register, regardless of pin direction.
FIOxSET	This register controls the state of output pins. Writing 1s produces highs at the corresponding port pins. Writing 0s has no effect.
FIOxCLR	This register controls the state of output pins. Writing 1s produces lows at the corresponding port pins. Writing 0s has no effect.

eg

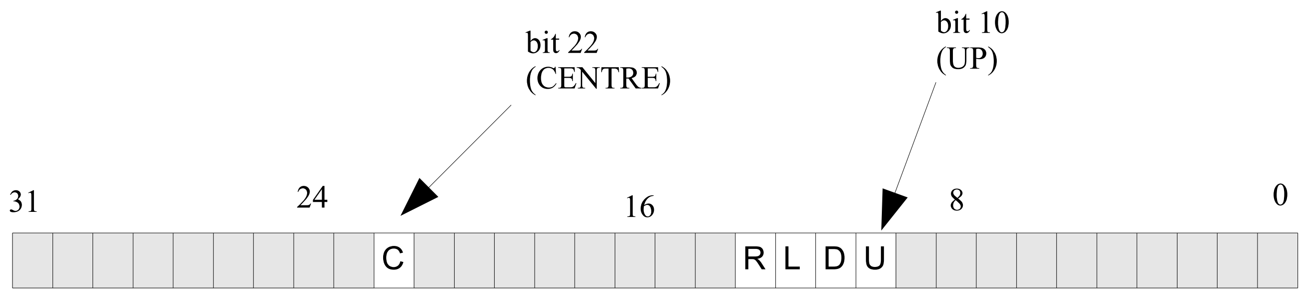
```
FIO3DIR = 0xFFFFFFFF;  // sets all port 3 bits to be outputs
FIO3SET = 0xFFFFFFFF;  // sets all outputs high (all LEDs on)
FIO3CLR = 1; // sets bit 0 low (LED0 off), others unaffected
FIO3CLR = 0xF0000000;  // turns off top 4 LEDs, others unaffected
FIO3PIN = 0x0000000F;  // affects all LEDs: bottom 4 on,
                       // all others off
```

NOTE: **To be able to use these register names (eg FIO3SET) you must #include <lpc24xx.h> at the top of your C program.**

Joystick

The joystick consists of five switches, one each for UP, DOWN, RIGHT, LEFT, CENTRE:

Function	Port Pin
UP	P0.10
DOWN	P0.11
LEFT	P0.12
RIGHT	P0.13
CENTRE	P0.22



Switch activated, port pin = 0, switch not pressed, pin = 1.

eg

UP switch is bit 10:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0				0				0				0				0				4				0				0			

To test if the UP switch is pressed:

```
    if( (FIO0PIN & 0x00000400) == 0) {
        // UP switch is pressed
    }
```

Knowing that the UP switch is reflected in bit 10, a more readable way of writing the same code is:

```
    #define UP (1<<10)
    if( (FIO0PIN & UP) == 0) {
        // UP switch is pressed
    }
```

The expression (1<<10) will be evaluated by the compiler for you, and will end up as 0x00000400.

NOTE: other bits in P0 are used for other functions and should not be changed

Binary Manipulation

The bits in the binary representation of numbers in the computer can be manipulated using the following commands:

bitwise **complement**	`~`
bitwise **and**	`&`
bitwise **exclusive or**	`^`
bitwise **or**	`|`
left shift	`<<`
right shift	`>>`

AND, OR, XOR, INVERT Examples

The effect of the operations on two integers j and k taking values in binary as shown is given in this table:

j	k	~j	~k	j & k	j ^ k	j | k
0	0	1	1	0	0	0
0	1	1	0	0	1	1
1	0	0	1	0	1	1
1	1	0	0	1	0	1
011	110	100	001	010	101	111

Left and Right Shift Examples

The shift commands operate as follows:

Eg if `long m = 0xf0;`

Then (where m is a 32 bit number):

`m`	00000000 00000000 00000000 11110000	**bit pattern defined**
`m << 4`	00000000 00000000 00001111 00000000	**left shifted 4 bits**
`m >> 3`	00000000 00000000 00000000 00011110	**right shifted 3 bits**

R4

Analogue Output (DAC)

The microcontroller contains a 10 bit digital to analogue converter (DAC). To use the DAC you need to first set the pin being used for the DAC output to have the analogue out function (you only need to do this once in your program before you start using the DAC).

To select the analogue output function set bits 21 and 20 of register PINSEL1 to 10.

PINSEL1:

XXXX	XXXX	XX10	XXXX	XXXX	XXXX	XXXX	XXXX

You only want to change these 2 bits. You can do this, for example, by clearing these 2 bits, then setting them as required.

To set an analogue output, write a value V into the DAC control register DACR in the bit positions shown. The DAC output has a range 0 to 3.3V corresponding to digital values of V in the range 0 to 0x3FF.

DACR:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	V	V	V	V	V	V	V	V	V	V	0	0	0	0	0	0

Analogue Input (ADC)

These notes give a simplified view of the analogue to digital converter (ADC) contained in the microcontroller. The full details are given in the NXP User Manual for the NXP2478 microcontroller.

Analogue to Digital Converter

The ADC has 8 inputs (channels), you can tell it to convert the voltage on any one of these inputs. We are using channel 1. The converter gives a 10 bit result, where 0 = 0 volts (or less). 0x3FF (all bits set to 1) = 3.3 volts (or more).

Register Use

You will need to access the following registers in your program:

PINSEL1 – to set the pin to the analogue input function

PCONP (power control for peripherals) – to turn on the ADC

AD0CR – ADC control register

AD0DR1 – ADC data register 1, for reading the ADC conversion result

Initialisation

To use the ADC you first need to set it up:

1. Set the pin we are using to be an analogue input.
2. Turn on the ADC.

You only need to do these once in the program, but you have to do them before you can start to use the ADC.

To select the pin we are using to be an analogue input we need to set bits 17 and 16 of register PINSEL1 to 01.

PINSEL1:

XXXX	XXXX	XXXX	XX01	XXXX	XXXX	XXXX	XXXX

You must not change any other bits, just these 2 bits.

To turn on the ADC, set bit 12 in the Peripheral Control Register (PCONP) **without changing any other bits in that register.**

Using the ADC

In its simplest form you only need to use two ADC registers, AD0CR (control register) and AD0DR1 (which holds the result for channel 1).

AD0CR:

0	0	0	0	0	START			0	0	P	0	0	0	0	0	CLKDIV								SEL							

Bits	**Function**
START	001 = start a single conversion now
P	1 = enable ADC, 0 = power down ADC
CLKDIV	divider for main clock which sets ADC clock frequency (see below)
SEL	channel select (0 to 7) bits, use 00000010 for channel 1

The clock frequency entering the ADC is 12 MHz. The CLKDIV value must be chosen to divide this down to get a frequency less than or equal to 4.5 MHz (the fastest the ADC will work). eg Setting CLKDIV to 00000010 (ie 2) would result in an ADC clock frequency of 12/2 = 6 MHz, which is too high.

AD0DR1:

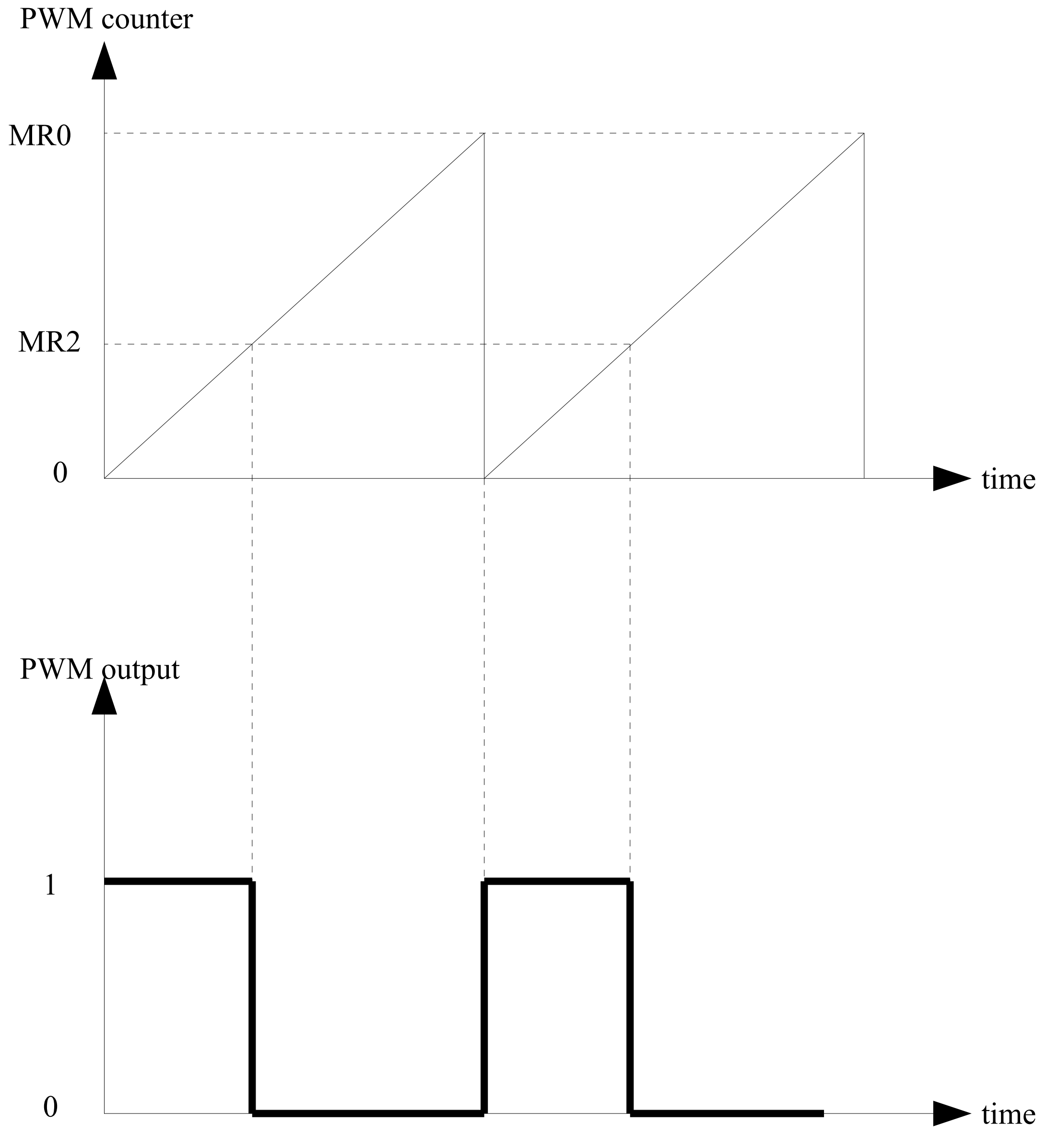
D	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**V**	**V**	**V**	**V**	**V**	**V**	**V**	**V**	**V**	**V**	0	0	0	0	0	0

Bit	**Function**
D	This bit becomes set to 1 when an A/D conversion completes. It is set to 0 automatically when this register is read.
V	When D is 1, this field contains a binary value representing the input voltage on the ADC channel 1 pin.

To take an ADC reading you must write the correct value into AD0CR to start the conversion, then keep reading AD0DR1, waiting until D becomes 1, at which point the value in the bits marked V is the result. This process will take just one reading: the ADC will not take a new reading until you write to AD0CR to tell it to start again.

Pulse Width Modulation (PWM)

The microcontroller has 2 PWM units, PWM0 and PWM1. We will be using PWM0. The PWM unit counts clock pulses up to a value stored in a “match” register (MR0) and then resets to zero and starts counting again. In its simplest form, each time the counter resets to zero it sets an output (you can select which one) to 1. You can set another match register (eg MR2) so that when the counter reaches this value the output is set back to zero.



So the value in MR0 determines the period of the pulse output, the value in MR2 determines the pulse width.

PWM Pin configuration

We will use the PWM output which can be produced on P1.3 (port 1 bit 3). You have to set P1.3 to have the PWM function, which you can do by setting bits 6 and 7 in PINSEL2 to logic 1:

PINSEL2:

XXXX	XXXX	XXXX	XXXX	XXXX	XXXX	11XX	XXXX

You must only change these two bits.

The P1.3 pin will then have the PWM0[2] function: channel 2 output of PWM number 0.

You next need to tell the PWM unit to control this PWM0[2] output by setting bit 10 in PWM0PCR to logic 1 :

PWM0PCR:

XXXX	XXXX	XXXX	XXXX	XXXX	X1XX	XXXX	XXXX

Set bit 10 in PWM0PCR to 1 without changing any other bits.

PWM Frequency

Next you need to decide what frequency of pulses you want on your PWM output. The input clock to the PWM unit is 12 MHz. The PWM unit will count these clock pulses up to the value stored in the match register PMW0MR0, and then restart at zero. Eg if you put a million in this register your PWM frequency would be 12 Hz (this would be far too low for our purposes!).

```
PWM0MR0 = your value;    // set your required count value
```

We now need to tell the PWM to reset when the counter gets to this value in MR0. We do this by setting bit 1 in the match control register PWM0MCR:

XXXX	XXXX	XXXX	XXXX	XXXX	XXXX	XXXX	XX1X

Make sure you set this bit without changing any others.

PWM Pulse Width

Our output will be set high when the counter resets. Because we are using PWM0[2], ie channel 2, the output will go low when count equals the value in match register 2, PWM0MR2. Put the initial value you want to use in that register:

```
PWM0MR2 = your value;    // set your required value
```

Start the PWM

You can now start the counter and PWM unit. You write a value into the PWM timer control register (PWM0TCR) with bit 0 set to enable counting, and bit 3 set to enable PWM, and the other bits all 0:

PWM0TCR:

0000	0000	0000	0000	0000	0000	0000	1001

Changing Pulse Width

Once the PWM is running, you can change the pulse width by changing the value in PWM0MR2. The PWM unit has some logic to ensure that the value is only updated on the next cycle. First you write the new value you want into PWM0MR2, then you allow MR2 to be updated on the next cycle by setting bit 2 in the “latch enable” register (PWM0LER):

```
PWM0MR2 = new value;     // set your required value
PWM0LER = (1<<2);   // MR2 will be updated on next cycle
```

Bit 1 in PWM0LER allows MR0 to be updated too, so if you wanted to, you could change both frequency (MR0) and pulse width (MR2) at the same time in this way, eg

```
PWM0MR0 = new value;     // set your required frequency
PWM0MR2 = new value;     // set your required pulse width
PWM0LER = (1<<2) | 1;    // update MR0 and MR2 on next cycle
```

NOTE: The motor needs a separate power supply (mains block). The motor will not run until this is connected (although you can still see the pwm outputs on an oscilloscope).

Timers

The microcontroller has 4 timers which can count the microcontroller clock and indicate when a particular count value has been reached. By default the timer will count the 12MHz peripheral clock pulses.

As with the ADC, the timer you are using should be enabled in the power control register PCONP. The bits for turning on each timer are:

timer	**bit in PCONP register**
T0	1
T1	2
T2	22
T3	23

One way to use a timer to wait for an accurate length of time:

- stop and reset the timer
- clear the timer interrupt flag
- load a value into a “match” register corresponding to the required time interval
- program the match to generate an interrupt
- start the timer
- wait until the time has elapsed

NOTE: Timer T1 is used by the LCD initialisation function. You are free to use it yourself, but remember to call lcdInit() or textInit() BEFORE you configure or use the T1 timer.

Timer Configuration

(a) stop and reset the timer

The Timer Control Register (TxTCR, where x = 0,1,2 or 3 for the required timer) is 8 bits:

0	0	0	0	0	0	RESET	ENABLE

If the RESET bit is one the count value in the timer is set to 0. If the ENABLE bit is 1 the counter is allowed to start counting.

(b) clear the timer interrupt flag

There are 4 “match” registers, MR0 to MR4. Each has a bit (interrupt flag) in TxIR which becomes set to 1 to indicate the counter has reached (or passed) the value stored in that register:

X	X	X	X	MR3	MR2	MR1	MR0

To clear one of the bits in this register before starting a count, write to this register with a 1 in the bit position you want clearing. (Remember: a 1 **clears** the bit to 0.)

Note that the counter will carry on counting past the MR2 value. After it reaches FFFFFFFF it will roll over to 00000000 and continue counting.

(c) load a value into a "match" register

eg for timer 0:

```
T0MR0 = 1000000;    // required interval, in 12 MHz clocks
```

(d) enable interrupt from match

TxMCR:

0	0	0	0	MR3S	MR3R	MR3I	MR2S	MR2R	MR2I	MR0S	MR1R	MR1I	MR0S	MR0R	MR0I

The "I" bits (MR0I, MR1I etc) allow an interrupt on match when set to 1.

The "R" bits (MR0R, MR1R etc) cause the count value to be reset on match when set to 1.

The "S" bits (MR0S, MR1S etc) cause the counter to stop on match when set to 1.

eg for timer 0:

```
T0MCR |= 3;    // reset counter and set interrupt when MR0
               // matches counter, but continue counting
```

(e) start timer

(Write to TxTCR to enable counting.)

(f) wait for elapsed time

Wait until the bit in TxIR becomes set, indicating a match.

Using a timer as a counter

Instead of counting pulses from a clock, the timer can count pulses on an input pin, for example the rev counter attached to the motor is connected to pin P1.19.

To use this feature,

set the input pin (P1.19) to counter mode

```
PINSEL3 |= (3 << 6);
```

and stop and reset the timer.
Tell it to count pulses on P1.19:

```
T1CTCR = 5;
```

and start it.

You can read the value of the counter at any time from the timer count register T1TC, eg

```
countval = T1TC;
```

NOTE: You **HAVE** to use T1 to count pulses on P1.19.

Details of these register settings can be found in the LPC24xx User Manual.

Interrupts

Interrupts use the Vectored Interrupt Controller (VIC) built into the microcontroller.

Each possible source of interrupt has an input to the VIC. Some that may be of interest to you are listed below. There are others as well, see LPC24xx User Manual for details.

source	VIC input
Timer 0	4
Timer 1	5
Timer 2	26
Timer 3	27
PWM0,1	8
ADC	18

To use interrupts you have to:

- set the interrupt type to be IRQ
- set the address of the interrupt service routine
- enable the vectored interrupt

You also have to write your “interrupt service routine” - the C function that will be called automatically when the interrupt occurs.

Set interrupt type

Clear the bit corresponding to the VIC input in register VICIntSelect.

Eg for Timer1:

```
VICIntSelect &= ~(1<<5);
```

Set the address of the interrupt service routine

Set the function address in the vector address register corresponding to the VIC input you are using.

Eg for Timer 1:

```
VICVectAddr5 = (unsigned int)yourISR;
```

Enable the Vectored Interrupt

Set the bit in the VIC interrupt enable register (VICIntEnable) corresponding to the VIC input you are using.

Eg for Timer1:

```
    VICIntEnable |= (1<<5);
```

Write your interrupt service routine

When the interrupt occurs, the running program will be suspended and your interrupt service routine will be called. When your interrupt service routine finishes, the program that was interrupted will resume where it left off.

Your interrupt service routine cannot return anything and cannot accept any parameters. It **must be declared in a special way** to tell the compiler that it is an interrupt service routine.

eg.

```
void yourISR(void) __attribute__ ((interrupt));
```

This says the function yourISR (call it what you like) has type void and is passed no parameters. It has a special attribute of type interrupt. Note that there are TWO underscore characters before and after the word “attribute”.

Example: Getting Interrupts from a Timer

Eg Timer 0.

First stop and reset the timer.

Set up VIC

Timer 0 is VIC input 4.

Set the Timer 0 to IRQ (not FIRQ): `VICIntSelect &= ~(1<<4);`
Set service routine: `VICVectAddr4 = (unsigned int)yourISR;`
Enable it: `VICIntEnable |= (1<<4);`
(priority left at 15 unless set VICVectCntl4 to some other value)

Set up timer

Set up the timer the way you want it and then enable it. Set it up so that your chosen event (eg reaching the value in a match register) sets the interrupt flag.

On interrupt

- Clear the timer interrupt flag in T0IR.
- Clear the VIC : `VICVectAddr = 0;`
- Do anything else you need in the service routine.

Cautionary Notes

It is very unwise to do anything complicated in your interrupt service routine, especially if it involves using library functions (eg LCD functions or anything involving floating point numbers). You might get away with a simple write of text to the display, but just use that for debugging purposes.

Printing to the LCD Screen

The LCD is a colour graphics screen with a resolution of 320 x 240 pixels. There is a library of functions to help you display things on the screen. There are two ways of using this:

- Simple scrolling text display.
- Full graphics control

You can use either of these, or even both within the same program. To use them you must include the appropriate header file at the start of your program:

```
#include <textDisplay.h>
```

Also, before using the LCD you must call the LCD initialisation function:

```
textInit();
```

Printf and simplePrintf

Printf produces scrolling text output on the LCD screen. The resulting program will be very large.

Alternatively, simplePrintf can be used in exactly the same way as printf but results in a much smaller program. You can only print strings (%s) characters (%c) signed integers (%d), unsigned integers (%u) or hexadecimal integers (%x) using simplePrintf.

Scrolling Text Display

This is the easiest way to show text on the screen. It automatically scrolls the display when necessary. **The x and y coordinates used below are character positions** in the x and y directions, starting at 0,0 in the top left corner (they are NOT pixel coordinates).

```
void textSetCursor(int x, int y)
```

Sets cursor to character column x, row y.

eg `textSetCursor(5,10); //next character will be at row 10, column 5`

```
void textClear(void)
```

Clears screen.

eg `textClear();`

```
void textPutc(char c)
```

Writes the character at the current cursor position and moves cursor to next character position.

eg `textPutc('X');  // puts the character X on the screen` `void textPuts(char *str)`

Writes the string of characters to the display starting at the current cursor position and moves the cursor to the next character position.

eg `textPuts(“hello there!”);`

`textHighlightLine(int line)`

Highlights the specified row of characters.

`textRedrawLine(int line)`

Redraws the current line (without highlighting).

`int textGetx(void)`

Returns the current x (column) position of the cursor.

eg `x = textGetx();`

`int textGety(void)`

Returns the current y (row) position of the cursor.

eg `y = textGety();`

`int textGetMaxx(void)`

Returns the maximum column number in the display.

`int textGetMaxy(void)`

Returns the maximum row number in the display.

Graphics Control

For low level graphics control you need to include ***<lcd_grph.h>*** at the top of your program. **The x,y coordinates used in these functions are pixel positions.**

The following lists the available functions, where x,y are the pixel position:

lcd_init(); (Initialise screen)
lcd_fillScreen(colour);
void lcd_point(x, y, colour);
void lcd_drawRect(x0, y0, x1, y1, colour);
void lcd_fillRect(x0, y0, x1, y1, colour);
void lcd_line(x0, y0, x1, y1, colour);
void lcd_circle(x0, y0, r, colour);
void lcd_fillcircle(x0, y0, r, colour);
tBool lcd_putChar(x, y, ch);
void lcd_putString(x, y, pStr);
void lcd_fontColor(foreground, background);

where x,y,x0,y0 are unsigned short, ch is an unsigned char, pStr is a pointer to a string of char (or the name of a char array) and colour, foreground and background are one of the colours defined below.

Colours (which are defined in the include file):
 BLACK
 NAVY
 DARK_GREEN
 DARK_CYAN
 MAROON
 PURPLE
 OLIVE
 LIGHT_GRAY
 DARK_GRAY
 BLUE
 GREEN
 CYAN
 RED
 MAGENTA
 YELLOW
 WHITE

The display dimensions in pixels are defined in the include file as:

```
      #define DISPLAY_WIDTH  240
      #define DISPLAY_HEIGHT 320
```

The top LH corner pixel is (0,0) and the bottom RH corner pixel is (239,319).

When printing text using lcd_putChar or lcd_putString bear in mind that the characters used are 5 pixels in the X direction by 7 pixels in the Y direction. With lcd_putString a blank pixel is left between characters.

eg

```
#include <lcd_grph.h>

main()
{
...
lcd_init();                     // initialise display
lcd_line(0,0,239,319,RED);     //draw red line from top LH to bottom RH corner
lcd_putString(8,160,"Hello!");      // draw some text starting at x=8, y=160
...
}
```

PRB