General Purpose Input/output programming

In this tutorial we will go through LPC1768 GPIO Programming. LPC1768 is an **ARM Cortex-M3** based MCU by Phillips/NXP and has plenty of General Purpose Input Output pins to play with. The Name of Registers, Data structures that have been used in this guide are defined in `LPC17xx.h` header file. `LPC17xx.h` header is based on CMSIS(Cortex Microcontroller System Interface Standard) developed by ARM. System startup, core CPU access and peripheral definitions are given by **CMSIS-CORE** component of CMSIS. We will be using the definitions given by CMSIS-CORE. The **register definitions** for Cortex-M3 LPC17xx MCUs are organized into groups depending on their functionality using "**C Structure**" definitions. From a programming point of view, this makes interfacing peripherals simple. For example all registers for Port 0 are grouped into structure defined as `LPC_GPIO0`.

Prerequisite : Before getting into this you need to have basic understanding of Binary and Hexadecimal system and Bitwise operations in C, here are two tutorials which can go through (or if you are already acquainted with these you can skip them and continue below) :

- Hexadecimal and Binary Number System basics for Embedded Programming
- Tutorial : Embedded programming basics in C – bitwise operations

Most of the function oriented pins on LPC176x Microcontrollers are grouped into **Ports**. LPC1768 has **5 ports** viz. Port **0 to 4**. The associated registers for each port are grouped into a structure with the following naming convention: `LPC_GPIOx` , where x is the port number. From the programming point of view, these ports are 32-bit wide i.e. a **maximum 32 pins** can be mapped, but each port may have a few or many pins which cannot be used i.e. they are 'reserved'. For this tutorial will be using LPC1768 in LQFP100 package as reference. If you are using LQFP80 package please refer the manual on which pins are available.

- In Port 0 Pins 12, 13, 14 & 31 are not available.
- In Port 1 Pins 2, 3, 7, 6, 5, 11, 12, & 13 are not available.
- In Port 2 only pins 0 to 13 are available and rest are reserved.
- In Port 3 only pins 25,26 are available and rest are reserved.
- Finally in Port 4 only 28,29 are available and rest are reserved.

The naming convention for Pins on MCU is '**Px.y**' where '**x**' is the port number (0,1,2,3 or 4 in our case since we have only 5 ports to play with in lpc1768) and '**y**' is simply the pin number in port '**x**'. For example: **P0.7** refers to Pin number **7** of Port **0** , **P2.11** refers to Pin number **11** in Port **2**.

GPIO Registers in LPC1768

Registers on LPC1768 are present on **Peripheral AHB** bus(Advanced High performance Bus) for fast read/write timing. So, these are basically Fast I/O or Enhanced I/O and hence the naming convention in datasheet uses a prefix of "**FIO**" instead of something like "GIO" for all the registers related to GPIO. Lets go through these as given below.

1) FIODIR : This is the GPIO direction control register. Setting a bit to 0 in this register will configure the corresponding pin to be used as an Input while setting it to 1 will configure it as Output.

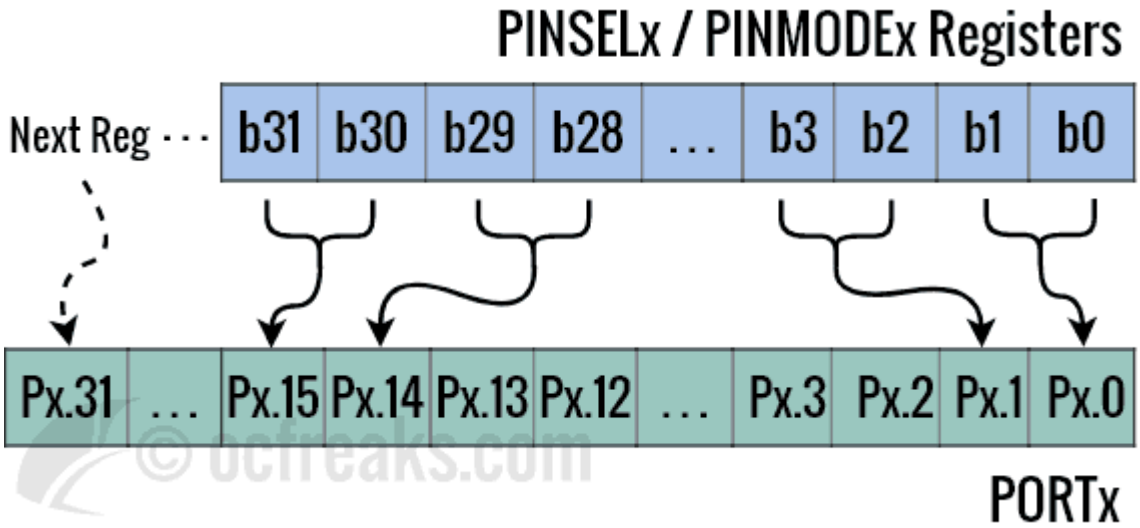
2) FIOMASK : This gives masking mechanism for any pin i.e. it is used for Pin access control. Setting a bit to 0 means that the corresponding pin will be affected by changes to other registers like FIOPIN, FIOSET, FIOCLR. Writing a 1 means that the corresponding pin won't be affected by other registers.

3) FIOPIN : This register can be used to Read or Write values directly to the pins. Regardless of the direction set for the particular pins it gives the current start of the GPIO pin when read.

4) FIOSET : It is used to drive an 'output' configured pin to Logic 1 i.e HIGH. Writing Zero does NOT have any effect and hence it cannot be used to drive a pin to Logic 0 i.e LOW. For driving pins LOW FIOCLR is used which is explained below.

5) FIOCLR : It is used to drive an 'output' configured pin to Logic 0 i.e LOW. Writing Zero does NOT have any effect and hence it cannot be used to drive a pin to Logic 1.

Most of the PINS of LPC176x MCU are **Multiplexed** i.e. these pins can be configured to provide up to 4 different **functions**. By default, after Power-On or Reset : all pins of all ports are set as GPIO so we can directly use them when learning GPIO usage. The different functions that any particular pin provides can be selected by setting appropriate value in the PINSEL register for the corresponding pin. Each pin on any port has 2 corresponding bits in PINSEL register. The first 16 pins (0-15) on a given port will have a corresponding 32 bit PINSEL register and the rest 16 bits will have another register. For example bits 0 & 1 in PINSEL0 are used to select function for Pin 1 of Port 0, bits 2 & 3 in PINSEL0 are used to select function for PIN 2 of port 0 and so on. The same is applicable for PINMODE register which will go through in last section of this article. Have a look at the diagram given below.



Here 'bx' refers to x^{th} Bit and 'Px.y' refers to y^{th} Pin on port 'x'. Just remember that assigning '0' to the corresponding PINSEL registers forces the corresponding pins to be used as GPIO. Since by default all pins are configured as GPIOs we don't need to explicitly assign a '0' value to PINSEL register in our programming examples.

As per the CMSIS convention, the registers that we saw are grouped into structures. `LPC_GPIOx` is defined as a pointer to this structure in `LPC17xx.h` header. These registers are defined as members of this structure. Hence to use any register, for e.g. FIODIR, we must use the arrow "->" operator to de-reference members of structure (since the structure itself is a pointer) to access the register as follows : `LPC_GPIO0->FIODIR` = some value.

Note : LPC17xx MCUs have Peripheral Power Control feature which allows individual peripherals to be turned off for power saving using PCONP Register which is a member of **`LPC_SC`** (system control) structure. We don't need to explicitly Power-On the GPIO block because its Power is always enabled.

Pins on LPC176x are **5V tolerant** when used for GPIO but when used as inputs for ADC block they are **not**. However, I would recommend that wherever possible, use a buffer for level translation between **5V and 3.3V**.

GPIO Programming & Examples

Now lets see how we can assign values to registers. We can use Hexadecimal notation & decimal notation for assigning values. If your compiler supports other notations like binary notation use can use that too.

Lets say we want to set **PIN 3** on **Port 0 as output**. It can be done in following ways:

```
CASE 1. LPC_GPIO0->FIODIR = (1<<3); //(binary using left shift - direct assign: other pins set to 0)

CASE 2. LPC_GPIO0->FIODIR |= 0x0000008; // or 0x8; (hexadecimal - OR and assign: other pins not affected)

CASE 3. LPC_GPIO0->FIODIR |= (1<<3); //(binary using left shift - OR and assign: other pins not affected)
```

- In many scenarios, Case 1 must be avoided since we are directly assigning a value to the register. So while we are making P0.2 '1' others are forced to be assigned a '0' which can be avoided by **ORing** and then assigning Value.
- Case 2 can be used when bits need to be changed in bulk and
- Case 3 when some or single bit needs to be changed.

First thing to note here is that preceding Zeros in Hexadecimal Notation can be ignored because they have no meaning since we are working with unsigned values here (positive only) which are assigned to Registers. For eg. `0x2F` and `0x02F` and `0x002F` all mean the same.

Note that bit **31** is **MSB on extreme left** and bit **0** is the **LSB on extreme right** i.e. we are using Big Endian Format. Hence bit 0 is the 1st bit from right , bit 1 is the 2nd bit from right and so on. BIT and PIN Numbers are **Zero**(0) indexed which is quite evident since Bit '**x**' refers to **(x-1)th** location in the corresponding register.

Finally, **All GPIO pins are configured as Input with pullup after Reset by default!**

Now, Lets go through some examples :

Example #1)

Consider that we want to configure Pin 4 of Port 0 i.e P0.4 as Ouput and want to drive it High(Logic 1). This can be done as :

```
LPC_GPIO0->FIODIR |= (1<<4); // Config P0.4 as Ouput
LPC_GPIO0->FIOSET |= (1<<4); // Make ouput High for P0.4
```

Example #2)

Making output configured Pin 17 High of Port 0 i.e P0.17 and then Low can be does as follows:

```
LPC_GPIO0->FIODIR |= (1<<17); // P0.17 is Output pin
LPC_GPIO0->FIOSET |= (1<<17); // Output for P0.17 becomes High
LPC_GPIO0->FIOCLR |= (1<<17); // Output for P0.17 becomes Low
```

Example #3)

Configuring P0.5 and P0.11 as Ouput and Setting them High:

```
LPC_GPIO0->FIODIR |= (1<<5) | (1<<11); // Config P0.5 and P0.11 as Ouput
LPC_GPIO0->FIOSET |= (1<<5) | (1<<11); // Make ouput High for P0.5 and P0.11
```

Example #4)

Configuring 1st 8 Pins of Port 0 (P0.0 to P0.7) as Ouput and Setting them High:

```
LPC_GPIO0->FIODIR |= 0xFF; // Config P0.0 to P0.7 as Ouput
LPC_GPIO0->FIOSET |= 0xFF; // Make output High for P0.0 to P0.7
```

Now lets play with some real world examples.

The below examples are given, assuming 100Mhz CCLK which is configured & initialized by system startup code generated by Keil UV5/UV4.

Example #5)

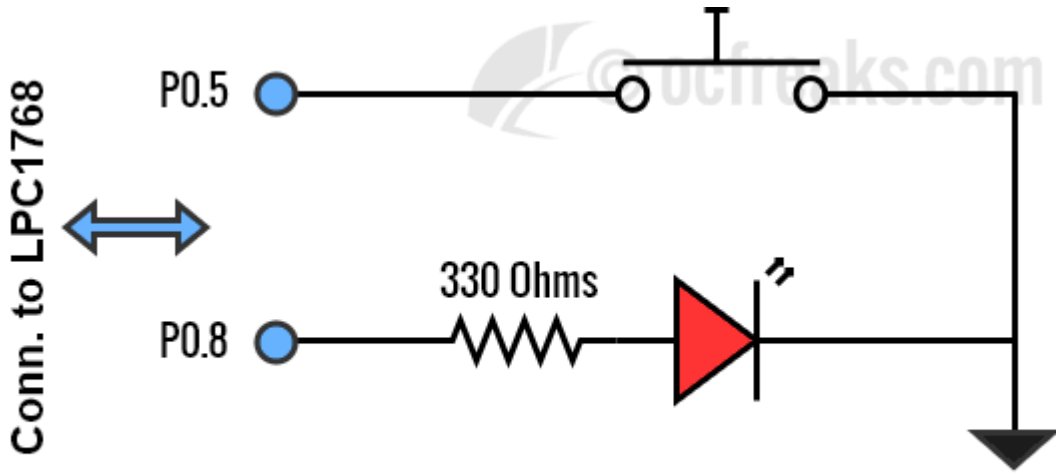
Simple Blinky Example - Here we repeatedly make pins 0 to 3 in port 0 (P0.0 to P0.3) High then Low then High and so on. You can connect LED to any 1 pin using 330Ohm series resistor or connect LEDs to all Pins using a buffer ICs like 74HC245 to drive the LEDs. The LED(s) are to be connected between output and ground. Here we will introduce some delay between making all pins High and Low so it can be noticed.

```
#include <lpc17xx.h>

void delay(void);

int main(void)
{
        LPC_GPIO0->FIODIR = 0xF; // Configure pins 0 to 3 on Port 0 as Output

        while(1)
        {
                LPC_GPIO0->FIOSET = 0xF; // Output HIGH
                delay();
                LPC_GPIO0->FIOCLR = 0xF; // Output LOW
                delay();
        }
        return 0; // normally this wont execute
}

void delay(void) //Hardcoded delay function
{
        int count,i=0;
        for(count=0; count < 6000000; count++) // You can edit this as per your
needs
        {
                i++; // something needs to be here else compiler will remove
the for loop!
        }
}
```

Example #6)

Configuring P0.5 as Input and monitoring it for a external event like connecting it to LOW or GND. P0.8 is configured as output and connected to LED. If Input for P0.5 is a 'Low' (GND) then output for P0.8 is made High which will activate the LED and make it glow (Since the other END of LED is connected to LOW i.e GND). Since by default, internal Pull-ups are enabled the 'default' state of the pins configured as Input will be always 'High' unless it is explicitly pulled 'Low' by connecting it

to Ground. Consider one end of a tactile switch connected to P0.5 and other to ground. When the switch is pressed a 'LOW' will be applied to P0.5. The setup is shown in the figure below:



OCFreaks.com LPC1768 GPIO tutorial
Schematic for Example 6

```
#include <lpc17xx.h>

int main(void)
{
        LPC_GPIO0->FIODIR &= ~((1<<5)) ; // explicitly making P0.5 as Input - even though by default its already Input
        LPC_GPIO0->FIODIR |= (1<<8); // Configuring P0.8 as Output

        while(1)
        {
                if( !(LPC_GPIO0->FIOPIN & (1<<5)) ) // Evaluates to True for a 'LOW' on P0.5
                {
                        LPC_GPIO0->FIOSET |= (1<<8); // drive P0.8 High
                }
        }
        return 0; // this wont execute normally
}
```

Example #7)

Now lets extended example 6 so that when the button is pressed, the LED will glow and when released or not pressed the LED won't glow. Capturing inputs in this way, using switches, leads to a phenomenon called 'bouncing' which needs to be resolved using 'debouncing' which I have explained in a previous GPIO tutorial for LPC2100 MCUs - please refer it for explanation. The algorithm used below is very basic for the sake of brevity, you can find a proper debouncing algorithms online.

```
#include <lpc17xx.h>

void tinyDelay(void);

int main(void)
```

```
{
	int flag=0, pinSamplePrev, pinSampleCurr;
	LPC_GPIO0->FIODIR &= ~((1<<5));
	LPC_GPIO0->FIODIR |= (1<<8);

	pinSamplePrev = LPC_GPIO0->FIOPIN & (1<<5); //Initial Sample

	while(1)
	{
		pinSampleCurr = LPC_GPIO0->FIOPIN & (1<<5); //New Sample

		if( pinSampleCurr != pinSamplePrev )
		{
			//P0.5 might get low or high momentarily due to noise
depending the external conditions or some other reason
			//hence we again take a sample to insure its not due to
noise

			tinyDelay(); // momentary delay

			// now we again read current status of P0.5 from IO0PIN

			pinSampleCurr = LPC_GPIO0->FIOPIN & (1<<5);

			if( pinSampleCurr != pinSamplePrev )
			{
				//State of P0.5 has indeed changed
				if(flag) //First time Flag will be = 0 hence
else part will execute
				{
					LPC_GPIO0->FIOSET |= (1<<8); // drive
P0.8 High, so LED turns on
					flag=0; //next time 'else' part will
excute
				}
				else
				{
					LPC_GPIO0->FIOCLR |= (1<<8); // drive
P0.8 Low, so LED turns off
					flag=1; //next time 'if' part will
excute
				}

				//set current value as previous since it has
been processed
				pinSamplePrev = pinSampleCurr;
			}
		}
	}
	return 0; // this wont execute ever, you can comment it to shut up the
compiler complaining.
}

void tinyDelay(void) //Hardcoded delay - use can use timer to get precise delays
{
	int z,c;
```

```
        c=0;
        for(z=0; z<3000; z++) //Higher value for higher clock speed
        {
                c++; //just so compiler doesn't remove the 'for' loop
        }
}
```

Easing play with Bits

Using the left shift operation is not confusing but when too many are used together I find it a little bit messy and affects code readability to some extent. For this I define a Macro `BIT(x)` as:

```
#define BIT(x) (1<<x)
```

After that is defined we can directly use BIT(x). Using this, Example 3 can be re-written as:

```
LPC_GPIO0->FIODIR |= BIT(5) | BIT(11); // Config P0.4 and P0.21 as Ouput
LPC_GPIO0->FIOSET |= BIT(5) | BIT(11); // Make ouput High for P0.4 and P0.21
```

Example 6 can be re-written as:

```
#include <lpc17xx.h>

#define BIT(x) (1<<x)

int main(void)
{
        LPC_GPIO0->FIODIR &= ~(BIT(5));
        LPC_GPIO0->FIODIR |= BIT(8);

        while(1)
        {
                if( !( LPC_GPIO0->FIOPIN & BIT(5) ) )
                {
                        LPC_GPIO0->FIOSET |= BIT(8);
                }
        }
        return 0;
}
```

Similarly, we can define `BITN(x)` as follow:

```
#define BITN(x) (!(1<<x))
```

This gives you 1's complement or Negation for (1<<x). Here x^{th} bit will be 0 and all other bits will be 1s.

Pin modes of Port pins in Cortex-M3 LPC176x MCUs

LPC1768 MCU supports 4 pin modes. These include the internal (on-chip) pull-up and pull-down resistor modes and a special operating mode. Pull-up and Pull-down resistors prevent the inputs from 'floating' by either pulling them to logic HIGH or LOW. The state of the inputs is therefore always defined.

The PINMODE and PINMODE_OD registers together are used to control the pin mode. A total of 3 bits are used to control the mode of an any pin, 2 bits from PINMODE and 1 bit from PINMODE_OD register. Hence we have a total of 10 PINMODE registers and a total of 5 PINMODE_OD. For the pins which are not available the corresponding bits are reserved. Bits 0 & 1 of PINMODE0 corresponds to Pin 0 of Port 0, bits 2 & 3 of PINMODE0 corresponds to Pin 1 of Port 0 and so on. PINMODE_ODx has 1:1 correspondence to all the pins in PORTx.

Note : The on-chip pull-up/pull-down resistor can be selected for every pin regardless of the function selected for that pin with the exception of the I2C pins for the I2C0 interface and the USB pins.

Pin mode select register bits for PINMODE0-9:

00	Internal pull-up resistor enabled.
01	Repeater mode - Retains its last state if it is configured as an input and is not driven externally.
10	Both pull-up and pull-down resistors disabled.
11	Internal pull-down resistor enabled.

The PINMODE_OD register controls the open drain mode for port pins. Setting any bit to 1 will enable open-drain mode for the corresponding pin. Setting any bit to 0 will enable the normal(default) mode for the pin.

As a beginner you will be mostly using either Pull-up or Pull-down mode with non open-drain mode (default). You can read more about the Pin Modes on Page 103 of the LPC176x User Manual(Rev. 01).

References and Further reading:

- LPC176x User Manual
- (CMSIS) Cortex Microcontroller System Interface Standard
- CMSIS-Core Documentation for Cortex-M MCUs
- A Guide to Debouncing

7 Segment Display in lpc1768

Introduction

Seven segment LED displays are often found in clock radios, VCRs, microwave ovens, toys and many other household items. They are primarily used to display decimal numbers but they can also display a few alphabets and other characters. This experiment describes interfacing a seven segment LED display to a PIC16F688 microcontroller. You will make a hexadecimal counter that counts from 0 (00h) to 15 (0Fh) and display the value on the seven segment LED display.

Required Theory

A seven segment LED display is an special arrangement of 7 LED elements to form a rectangular shape using two vertical segments on each side with one horizontal segment on the top, middle, and bottom. By individually turning the segments on or off, numbers from 0 to 9 and some letters can be displayed. Seven segment displays sometime also have an eighth segment to display the decimal point. Therefore, a seven-segment display will require seven outputs from the microcontroller to display a number, and one more output if the decimal point is to be displayed too.

The segments are marked with non-capital letters: a, b, c, d, e, f, g and dp, where dp is the decimal point. The 8 LEDs inside the display can be arranged with a common cathode or common anode configuration. With a common cathode display, the cathodes of all the segment LEDs are tied together and this common point must be connected to the ground. A required LED segment is then turned on by applying a logic 1 to its anode. In common anode displays, all the anodes are tied together and the common anode is connected to the supply voltage Vcc. Individual segments are turned on by applying logic 0 to their cathodes.

When more than one seven segment display is used, a multiplexing technique is used to minimize the required number of microcontroller pins. We will discuss about that technique later.

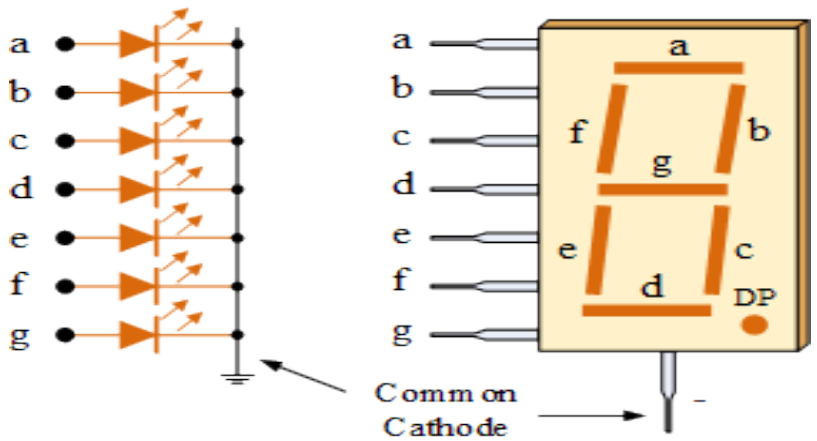


fig :common cathode display(to glow a particular segment pass logic 1)

	h	g	f	e	d	c	b	a	hex value
0	0	0	1	1	1	1	1	1	3F
1	0	0	0	0	0	1	1	0	06
2	0	1	0	1	1	0	1	1	5B
3	0	1	0	0	1	1	1	1	4F
4	0	1	1	0	0	1	1	0	66
5	0	1	1	0	1	1	0	1	6D
6	0	1	1	1	1	1	0	1	7D
7	0	0	0	0	0	1	1	1	07
8	0	1	1	1	1	1	1	1	7F
9	0	1	1	0	1	1	1	1	6F
A	0	1	1	1	0	1	1	1	77
b	0	1	1	1	1	1	0	0	7C
C	0	0	1	1	1	0	0	1	39
d	0	1	0	1	1	1	1	0	5E
E	0	1	1	1	1	0	0	1	79
F	0	1	1	1	0	0	0	1	71

Table: hexadecimal equivalent values for each digit to be displayed

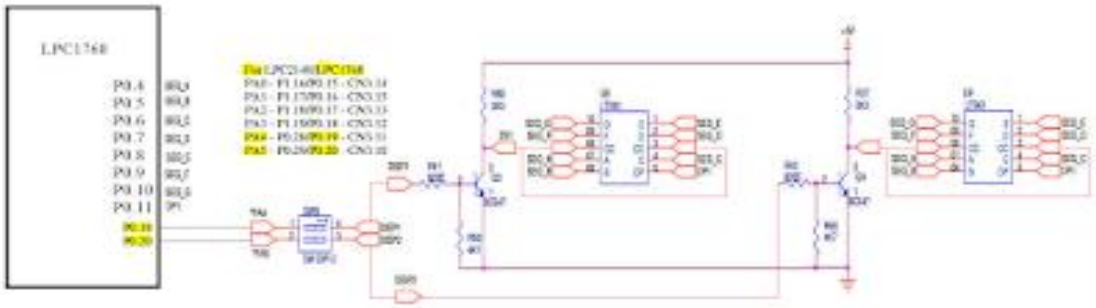


fig:interfacing diagram of LPC1768 to seven segment dislay

Programs:

1. Display 4321 over display units

```
#include <LPC17xx.h>

unsigned
seven_seg[10]={0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F
};
unsigned int dig_count=0,j,i;
unsigned int dig_value[4]={4,3,2,1};
unsigned int dig_select[] = {0<<23,1<<23,2<<23,3<<23};

void display()
{
        LPC_GPIO1->FIOPIN=dig_select[dig_count];
        LPC_GPIO0->FIOPIN=seven_seg[dig_value[dig_count]]<<4;
```

```
}
void delay()
{
	for(j=0;j<=60000;j++);
}

int main()
{
	//LPC_PINCON -> PINSEL0 = 0x0;
	//LPC_PINCON -> PINSEL3 = 0x0;
	LPC_GPIO0->FIODIR = 0xFF<<4;
	LPC_GPIO1->FIODIR = 0xF<<23;
	while(1)
	{
		delay();
		dig_count=(dig_count+1)%4;
		display();
	}

}
```

---

2. Up Counter

```
//upcountsevenseg
#include<LPC17XX.h>
#define FIRSTSEG  0<<23;
#define SECONDSEG  1<<23;
#define THIRDSEG  2<<23;
#define FOURTHSEG  3<<23;

unsigned int dig_1 = 0x00;
unsigned int dig_2 = 0x00;
unsigned int dig_3 = 0x00;
unsigned int dig_4 = 0x00;

unsigned int i,dig_count = 0, temp1 = 0x00, one_sec_flag = 0x00;
unsigned int array_dec[10] = {0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D,
0x7D, 0x07, 0x7F, 0x6F};
unsigned long int temp2 = 0x0;

void display(void){
	if(dig_count == 0x01){
		temp1 = dig_1;
		LPC_GPIO1->FIOPIN = FIRSTSEG;
	}
	if(dig_count == 0x02){
		temp1 = dig_2;
```

```
			LPC_GPIO1->FIOPIN = SECONDSEG;
		}
		if(dig_count == 0x03){
			temp1 = dig_3;
			LPC_GPIO1->FIOPIN = THIRDSEG;
		}
		if(dig_count == 0x04){
			temp1 = dig_4;
			LPC_GPIO1->FIOPIN = FOURTHSEG;
		}
		temp1 &= 0x0F;
		temp2 = array_dec[temp1];
		temp2<<=4;
		LPC_GPIO0->FIOPIN = temp2;
		for(i=0; i<1000; i++);
		LPC_GPIO0->FIOCLR = 0x00000FF0;
}

void delay(){
		for(i=0; i<50; i++);
}

int main(){
		LPC_GPIO0->FIODIR |=0xFF<<4;
		LPC_GPIO1->FIODIR |=15<<23;
			while(1){
				delay();
				dig_count +=1;
				if(dig_count==0x05){
					delay();
					dig_count = 0x01;
					one_sec_flag = 0xFF;
				}
				if(one_sec_flag==0xFF){
					one_sec_flag = 0x00;
					dig_1+=1;
					if(dig_1==0xA){
						dig_1 = 0;
						dig_2+=1;
						if(dig_2==0xA){
							dig_2=0;
							dig_3+=1;
							if(dig_3==0xA){
								dig_3 = 0;
								dig_4+=1;
								if(dig_4==0xA){
									dig_4 = 0;
								}
```

```
                                        }
                                    }
                                }
                            }
                            display();
                        }
    }
```

3. Down Counter

```
////downcountsevenseg
#include<LPC17XX.h>
#define FIRSTSEG  0<<23;
#define SECONDSEG  1<<23;
#define THIRDSEG  2<<23;
#define FOURTHSEG  3<<23;

unsigned int dig_1 = 0x09;
unsigned int dig_2 = 0x09;
unsigned int dig_3 = 0x09;
unsigned int dig_4 = 0x09;

unsigned int i,dig_count = 0, temp1 = 0x00, one_sec_flag = 0x00;
unsigned int array_dec[10] = {0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D,
0x7D, 0x07, 0x7F, 0x6F};
unsigned long int temp2 = 0x0;

void display(void){
        if(dig_count == 0x01){
                temp1 = dig_1;
                LPC_GPIO1->FIOPIN = FIRSTSEG;
        }
        if(dig_count == 0x02){
                temp1 = dig_2;
                LPC_GPIO1->FIOPIN = SECONDSEG;
        }
        if(dig_count == 0x03){
                temp1 = dig_3;
                LPC_GPIO1->FIOPIN = THIRDSEG;
        }
        if(dig_count == 0x04){
                temp1 = dig_4;
                LPC_GPIO1->FIOPIN = FOURTHSEG;
        }
        temp1 &= 0x0F;
        temp2 = array_dec[temp1];
        temp2<<=4;
        LPC_GPIO0->FIOPIN = temp2;
```

```
        for(i=0; i<1000; i++);
        LPC_GPIO0->FIOCLR = 0x00000FF0;
}

void delay(){
        for(i=0; i<50; i++);
}

int main(){
        LPC_GPIO0->FIODIR |=0xFFFF<<4;
        LPC_GPIO1->FIODIR |=15<<23;
                while(1){
                        delay();
                        dig_count +=1;
                        if(dig_count==0x05){
                                delay();
                                dig_count = 0x01;
                                one_sec_flag = 0xFF;
                        }
                        if(one_sec_flag==0xFF){
                                one_sec_flag = 0x00;
                                dig_1-=1;
                                if(dig_1==0x0){
                                        dig_1=9;
                                        dig_2-=1;
                                        if(dig_2==0x0){
                                                dig_2=9;
                                                dig_3-=1;
                                                if(dig_3==0x0){
                                                        dig_3 =9;
                                                        dig_4-=1;
                                                        if(dig_4==0x0){
                                                                dig_4 = 9;
                                                        }
                                                }
                                        }
                                }
                        }
                        display();
                }
}
```

4. UP/Down Counter

```
#include<LPC17XX.h>
#define FIRSTSEG  0<<23;
#define SECONDSEG  1<<23;
#define THIRDSEG  2<<23;
#define FOURTHSEG  3<<23;
```

```
unsigned int dig_1 = 0x00;
unsigned int dig_2 = 0x00;
unsigned int dig_3 = 0x00;
unsigned int dig_4 = 0x00;

unsigned int i,dig_count = 0, temp1 = 0x00, one_sec_flag = 0x00;
unsigned int array_dec[10] = {0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D,
0x7D, 0x07, 0x7F, 0x6F};
unsigned long int temp2 = 0x0;


void display(void){
	if(dig_count == 0x01){
		temp1 = dig_1;
		LPC_GPIO1->FIOPIN = FIRSTSEG;
	}
	if(dig_count == 0x02){
		temp1 = dig_2;
		LPC_GPIO1->FIOPIN = SECONDSEG;
	}
	if(dig_count == 0x03){
		temp1 = dig_3;
		LPC_GPIO1->FIOPIN = THIRDSEG;
	}
	if(dig_count == 0x04){
		temp1 = dig_4;
		LPC_GPIO1->FIOPIN = FOURTHSEG;
	}
	temp1 &= 0x0F;
	temp2 = array_dec[temp1];
	temp2<<=4;
	LPC_GPIO0->FIOPIN = temp2;
	for(i=0; i<1000; i++);
	LPC_GPIO0->FIOCLR = 0x00000FF0;
}

void delay(){
	for(i=0; i<50; i++);
}

int main(){
	LPC_GPIO0->FIODIR |=0xFF<<4;
	LPC_GPIO1->FIODIR |=15<<23;
while(1)
{
			delay();
			dig_count +=1;
```

```
		if(dig_count==0x05){
			delay();
			dig_count = 0x01;
			one_sec_flag = 0xFF;
		}
if((LPC_GPIO2->FIOPIN & 1))
{

		if(one_sec_flag==0xFF){
			one_sec_flag = 0x00;
			dig_1+=1;
			if(dig_1==0xA){
				dig_1 = 0;
				dig_2+=1;
				if(dig_2==0xA){
					dig_2=0;
					dig_3+=1;
					if(dig_3==0xA){
						dig_3 = 0;
						dig_4+=1;
						if(dig_4==0xA){
							dig_4 = 0;
						}
					}
				}
			}
		}
	}
else if(!(LPC_GPIO2->FIOPIN & 1))
{
		if(one_sec_flag==0xFF){
			one_sec_flag = 0x00;
			dig_1-=1;
			if(dig_1==0xffffffff){
				dig_1=9;
				dig_2-=1;
				if(dig_2==0xffffffff){
					dig_2=9;
					dig_3-=1;
					if(dig_3==0xffffffff){
						dig_3 =9;
						dig_4-=1;
						if(dig_4==0xffffffff){
							dig_4 = 9;
						}
					}
				}
			}
```

```
                }
        }


                display();
}
}
```

5. Hex UP/Down Counter

```
#include<LPC17XX.h>
#define FIRSTSEG  0<<23;
#define SECONDSEG  1<<23;
#define THIRDSEG  2<<23;
#define FOURTHSEG  3<<23;

unsigned int dig_1 = 0x00;
unsigned int dig_2 = 0x00;
unsigned int dig_3 = 0x00;
unsigned int dig_4 = 0x00;

unsigned int i,dig_count = 0, temp1 = 0x00, one_sec_flag = 0x00;
unsigned int array_dec[16] = {0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D,
0x7D, 0x07, 0x7F, 0x6F,0x77,0x7C,0x39,0x5E,0x79,0x71};
unsigned long int temp2 = 0x0;


void display(void){
        if(dig_count == 0x01){
                temp1 = dig_1;
                LPC_GPIO1->FIOPIN = FIRSTSEG;
        }
        if(dig_count == 0x02){
                temp1 = dig_2;
                LPC_GPIO1->FIOPIN = SECONDSEG;
        }
        if(dig_count == 0x03){
                temp1 = dig_3;
                LPC_GPIO1->FIOPIN = THIRDSEG;
        }
        if(dig_count == 0x04){
                temp1 = dig_4;
                LPC_GPIO1->FIOPIN = FOURTHSEG;
        }
        temp1 &= 0x0F;
        temp2 = array_dec[temp1];
        temp2<<=4;
        LPC_GPIO0->FIOPIN = temp2;
        for(i=0; i<1000; i++);
```

```
	LPC_GPIO0->FIOCLR = 0x00000FF0;
}

void delay(){
	for(i=0; i<50; i++);
}

int main(){
	LPC_GPIO0->FIODIR |=0xFF<<4;
	LPC_GPIO1->FIODIR |=15<<23;
while(1)
{
			delay();
			dig_count +=1;
			if(dig_count==0x05){
				delay();
				dig_count = 0x01;
				one_sec_flag = 0xFF;
			}
	if((LPC_GPIO2->FIOPIN & 1))
	{

			if(one_sec_flag==0xFF){
				one_sec_flag = 0x00;
				dig_1+=1;
				if(dig_1>0xF){
					dig_1 = 0;
					dig_2+=1;
					if(dig_2>0xF){
						dig_2=0;
						dig_3+=1;
						if(dig_3>0xF){
							dig_3 = 0;
							dig_4+=1;
							if(dig_4>0xF){
								dig_4 = 0;
							}
						}
					}
				}
			}
		}
	else if(!(LPC_GPIO2->FIOPIN & 1))
	{
			if(one_sec_flag==0xFF){
				one_sec_flag = 0x00;
				dig_1-=1;
				if(dig_1==0x0){
```

```
                                        dig_1=0xF;
                                        dig_2-=1;
                                        if(dig_2==0x0){
                                                dig_2=0xF;
                                                dig_3-=1;
                                                if(dig_3==0x0){
                                                        dig_3 =0xF;
                                                        dig_4-=1;
                                                        if(dig_4==0x0){
                                                                dig_4 = 0xF;
                                                        }
                                                }
                                        }
                                    }
                                }
                        }


                                display();
        }
        }
```

LPC21768 Timers

In this discussion will go through ARM Cortex-M3 LPC1768 Timer Tutorial. In a previous LPC1768 programming tutorial we saw a blinky example using GPIO and harcoded delays, now it's time to improvise and use precise delay using timers! LPC1768/LPC1769 has four 32-bit Timer blocks. Each Timer block can be used as a 'Timer' (like for e.g. triggering an interrupt every't' microseconds) or as a 'Counter' and can be also used to demodulate PWM signals given as input.

Each Timer module has its own **Timer Counter (TC)** and **Prescale Register (PR)** associated with it. When a Timer is Reset and Enabled, the TC is set to 0 and incremented by 1 every **'PR+1'** clock cycles – where PR is the value stored in Prescale Register. When it reaches its maximum value it gets reset to 0 and hence restarts counting. Prescale Register is used to define the **resolution** of the timer. If **PR is 0** then TC is incremented every **1 clock cycle of the peripheral clock**. If **PR=1** then TC is incremented every **2 clock cycles of peripheral clock and so on**. By setting an appropriate value in PR we can make timer increment or count : every peripheral clock cycle or 1 microsecond or 1 millisecond or 1 second and so on.

Each Timer has **four 32-bit Match Registers** and **four 32-bit Capture Registers**. Timer 0,1,3 have two Match outputs while Timer 2 has four.

Match Register

A Match Register is a Register which contains a specific value set by the user. When the Timer starts – every time after TC is incremented the value in TC is compared with match register. If it matches then it can Reset the Timer or can generate an interrupt as defined by the user. We are only concerned with match registers in this tutorial.

Match Registers can be used to:

- Stop Timer on Match(i.e when the value in count register is same as than in Match register) and trigger an optional interrupt.
- Reset Timer on Match and trigger an optional interrupt.
- To count continuously and trigger an interrupt on match.

External Match Output

When a corresponding Match register(MRx) equals the Timer Counter(TC) the match output can be controlled using External Match Register(EMR) to : either toggle, go HIGH, go LOW or do nothing.

Capture Register

As the name suggests it is used to Capture Input signal. When a transition event occurs on a Capture pin , it can be used to copy the value of TC into any of the 4 Capture Register or to generate an Interrupt. Hence these can be also used to demodulate PWM signals. We are not going to use them in this tutorial since we are only concerned with using Timer block as a 'Timer'. We'll see them in a later tutorial.

Registers used for LPC1768 Timer Programming

We will be using CMSIS based `lpc17xx.h` header file for programming. In CMSIS, all the Registers used to program and use timers are defined as members of structure(pointer) `LPC_TIMx` where x is the Timer module from 0 to 3. So for Timer1 we will use `LPC_TIM0` and so on. Registers can be accessed by de-referecing the pointer using "->" operator. For, example we can access TCR of Timer0 block as `LPC_TIM0->TCR`.

Now lets see some of the main registers concerned mainly with timer operation.

1) PR: Prescale Register (32 bit) – Stores the maximum value of Prescale counter after which it is reset.

2) PC: Prescale Counter Register (32 bit) – This register increments on every PCLK(Peripheral clock). This register controls the resolution of the timer. When PC reaches the value in PR , PC is reset back to 0 and Timer Counter is incremented by 1. Hence if PR=0 then Timer Counter Increments on every 1 PCLK. If PR=9 then Timer Counter Increments on every 10th cycle of PCLK. Hence by selecting an appropriate prescale value we can control the resolution of the timer.

3) TC: Timer Counter Register (32 bit) – This is the main counting register. Timer Counter increments when PC reaches its maximum value as specified by PR. If timer is not reset explicitly(directly) or by using an interrupt then it will act as a free running counter which resets back to zero when it reaches its maximum value which is 0xFFFFFFFF.

4) TCR: Timer Control Register – This register is used to enable , disable and reset TC. When bit0 is 1 timer is enabled and when 0 it is disabled. When bit1 is set to 1 TC and PC are set to zero together in sync on the next positive edge of PCLK. Rest of the bits of TCR are reserved.

5) CTCR: Count Control register – Used to select Timer/Counter Mode. For our purpose we are always gonna use this in Timer Mode. When the value of the CTCR is set to 0x0 Timer Mode is selected.

6) MCR: Match Control register – This register is used to control which all operations can be done when the value in MR matches the value in TC. Bits 0,1,2 are for MR0 , Bits 3,4,5 for MR1 and so on.. Heres a quick table which shows the usage:

For MR0:

- Bit 0: Interrupt on MR0 i.e. trigger an interrupt when MR0 matches TC. Interrupts are enabled when set to 1 and disabled when set to 0.
- Bit 1: Reset on MR0. When set to 1 , TC will be reset when it matched MR0. Disabled when set to 0.
- Bit 2: Stop on MR0. When set to 1 , TC & PC will stop when MR0 matches TC.

Similarly bits 3-5 , 6-8 , 9-11 are for MR1 , MR2 , MR3 respectively.

7) IR: Interrupt Register – It contains the interrupt flags for 4 match and 4 capture interrupts. Bit0 to bit3 are for MR0 to MR3 interrupts respectively. And similarly the next 4 for CR0-3 interrupts. when an interrupt is raised the corresponding bit in IR will be set to 1 and 0 otherwise. Writing a 1 to the corresponding bit location will reset the interrupt – which is used to acknowledge the completion of the corresponding ISR execution.

8) EMR: External Match Register – It provides both status and control of External Match Output Pins. First four bits are for EM0 to EM3. Next 8 bits are for EMC0 to EMC3 in pairs of 2.

- Bit 0 – EM0: External Match 0. When a match occurs between TC and MR0, depending on bits[5:4] i.e. EMC0 of this register, this bit can either toggle, go LOW, go HIGH, or do nothing. This bit is driven to MATx.0 where x=Timer number.
- Similarly for Bits 1, 2 & 3.
- Bits[5:4] – EMC0: External Match 0. The values in these bits select the functionality of EM0 as follows:
    - 0x0 – Do nothing
    - 0x1 – Clear the corresponding External Match output to 0 (MATx.m pin is LOW).
    - 0x2 – Set the corresponding External Match output to 1 (MATx.m pin is HIGH).
    - 0x3 – Toggle the corresponding External Match output.
- Similarly for Bits[7:6] – EMC1, Bits[9,8] – EMC2, Bits[11:10] – EMC3.

Note: Timer0/1/3 have only two Match outputs Pinned while Timer2 has four Match output Pinned. Hence EM2,EM3 and EMC2,EMC3 are not applicable for Timer0/1/3.

Now lets actually use a Timer and see it in action.

Setting up & configuring Timers in LPC176x

To use timers we need to first configure them. We need to set appropriate values in CTCR, IR, PR registers and reset PC, TC registers. Finally we assign TCR = 0x01 which enables the timer.

I would recommend to use the following sequence for Setting up Timers:

1. Set appropriate value in `LPC_TIMx->CTCR`
2. Define the Prescale value in `LPC_TIMx->PR`
3. Set Value(s) in Match Register(s) if required
4. Set appropriate value in `LPC_TIMx->MCR` if using Match registers / Interrupts
5. Reset Timer – Which resets PR and TC
6. Set `LPC_TIMx->TCR` to `0x01` to Enable the Timer when required
7. Reset `LPC_TIMx->TCR` to `0x00` to Disable the Timer when required

Setting the Peripheral Clock for Timer: The input clock for timers can be set using peripheral clock selection registers **PCLKSEL0 & PCLKSEL1**.

- For Timer0 bits[3:2] in PCLKSEL0 are used.
- For Timer1 bits[5:4] in PCLKSEL0 are used.
- For Timer2 bits[13:12] in PCLKSEL1 are used.
- For Timer3 bits[15:14] in PCLKSEL1 are used.

These bits allows us to choose 4 different CCLK(SystemCoreClock in startup code) dividers to get the final PCLK as follows :

- [00] – PCLK = CCLK/4 (Default after reset)
- [01] – PCLK = CCLK
- [10] – PCLK = CCLK/2
- [11] – PCLK = CCLK/8

LPC1768 Timer Prescaler Calculations:

The delay or time required for 1 clock cycle when PCLK = 'X' Mhz is given by :

$$T_{PCLK} = \frac{1}{PCLK_{Hz}}$$

$$= \frac{1}{X * 10^6}$$

Seconds

It is also the maximum resolution Timer block can proved at a given PCLK frequency of X Mhz. The general formula for Timer resolution at X Mhz PCLK and a given value for prescale (PR) is as given below:

$$T_{RES} = \frac{PR+1}{PCLK_{Hz}}$$

=

PR+1 $X * 10^6$

Seconds

Hence, we get the **Formula for Prescaler (PR) for required Timer resolution (T_{RES} in Secs)** at given **PCLK(in Hz)** frequency as:

$PR = (PCLK_{Hz} * T_{RES}) - 1$

$PR = ((X * 10^6) * T_{RES}) - 1$

Note that here, the resolution is also the time delay required to increment TC by 1.

Hence, Prescaler value for 1 micro-second resolution/ 1us time delay at 25 Mhz PCLK is,

$PR_{1us} = (25Mhz * 1uS) -1 = (25*10^6 * 10^{-6}) -1 = 24$

Prescale for 1 mS (milli-second) resolution at 25Mhz PCLK is,

$PR_{1ms} = (25Mhz * 1ms) -1 = (25*10^6 * 10^{-3}) -1 = 24999$

The maximum resolution of all the timers is 10 nano-seconds when using PCLK = CCLK = 100Mhz and PR=0 which is as follows,

$T_{MAXRES} = [1 / (100Mhz)] = 10ns$

Now lets implement to basic function required for Timer Operation:

1. `void initTimer0(void);`
2. `void delayMS(unsigned int milliseconds);`

#1) initTimer0(void); [Used in Example #1]

Attention Plz! : This function is used to setup and initialize the Timer block. Timer blocks use peripheral clock as their input and hence peripheral clock must be initialized before Timer is initialized. In our case it is assumed that LPC1768 CPU Clock (CCLK) is set at 100Mhz and Peripheral Clock (PCLK) is set to CCLK/4 i.e. 25Mhz. Note that these clocks are default which are configured by the startup code.

```
#define PRESCALE (25000-1)

void initTimer0(void)
{
        /*Assuming that PLL0 has been setup with CCLK = 100Mhz and PCLK = 25Mhz.*/
        LPC_SC->PCONP |= (1<<1); //Power up TIM0. By default TIM0 and TIM1 are enabled.
        LPC_SC->PCLKSEL0 &= ~(0x3<<3); //Set PCLK for timer = CCLK/4 = 100/4 (default)

        LPC_TIM0->CTCR = 0x0;
        LPC_TIM0->PR = PRESCALE; //Increment TC at every 24999+1 clock cycles
        //25000 clock cycles @25Mhz = 1 mS

        LPC_TIM0->TCR = 0x02; //Reset Timer
}
```

#2) delayMS(unsigned int milliseconds); – LPC1768 Timer Delay Function

```
void delayMS(unsigned int milliseconds) //Using Timer0
{
        LPC_TIM0->TCR = 0x02; //Reset Timer

        LPC_TIM0->TCR = 0x01; //Enable timer

        while(LPC_TIM0->TC < milliseconds); //wait until timer counter reaches the desired delay

        LPC_TIM0->TCR = 0x00; //Disable timer
}
```

Real World LPC1768/LPC1769 Timer Examples with sample code

Example 1) – Simple Blinky Example using Timer & GPIO

Now lets write a C/C++ blinky program which flashes a LED every half a second. Since 0.5 second = 500 millisecond we will invoke timer delay function 'delayMS' as `delayMS(500)`. The LED is connected to Pin P0.22.

```
#include <lpc17xx.h>

#define PRESCALE (25000-1) //25000 PCLK clock cycles to increment TC by 1

void delayMS(unsigned int milliseconds);
void initTimer0(void);

int main(void)
{
        //SystemInit(); //called by Startup Code before main(), hence no need to call again.
        initTimer0(); //Initialize Timer0
        LPC_GPIO0->FIODIR = (1<<22); //Configure P0.22 as output

        while(1)
        {
                LPC_GPIO0->FIOSET = (1<<22); //Turn ON LED
                delayMS(500); //0.5 Second(s) Delay
                LPC_GPIO0->FIOCLR = (1<<22); //Turn LED OFF
                delayMS(500);
        }

        //return 0; //normally this wont execute ever
}

void initTimer0(void)
{
        /*Assuming that PLL0 has been setup with CCLK = 100Mhz and PCLK = 25Mhz.*/
        LPC_SC->PCONP |= (1<<1); //Power up TIM0. By default TIM0 and TIM1 are enabled.
```

```
        LPC_SC->PCLKSEL0 &= ~(0x3<<3); //Set PCLK for timer = CCLK/4 = 100/4
(default)

        LPC_TIM0->CTCR = 0x0;
        LPC_TIM0->PR = PRESCALE; //Increment LPC_TIM0->TC at every 24999+1 clock
cycles
        //25000 clock cycles @25Mhz = 1 mS

        LPC_TIM0->TCR = 0x02; //Reset Timer
        }

void delayMS(unsigned int milliseconds) //Using Timer0
{
        LPC_TIM0->TCR = 0x02; //Reset Timer
        LPC_TIM0->TCR = 0x01; //Enable timer

        while(LPC_TIM0->TC < milliseconds); //wait until timer counter reaches
the desired delay

        LPC_TIM0->TCR = 0x00; //Disable timer
}
```

Example 2) – Blinky example code using Timer & Match Outputs

In this C/C++ Example we will use Match outputs 0 and 1 of Timer 0 i.e. MAT0.0 and MAT0.1. MAT0.0 is pinned to P1.28 and MAT0.1 pinned to P1.29. You can connect LED to any of the Match output pin (P1.28 or P1.29) and see the code in action.

```
#include <lpc17xx.h>

#define PRESCALE (25000-1) //25000 PCLK clock cycles to increment TC by 1

void initTimer0(void);

int main (void)
{
        //SystemInit(); //called by Startup Code before main(), hence no need
to call again.
        initTimer0(); //Initialize Timer0
        LPC_PINCON->PINSEL3 |= (1<<24) | (1<<25) | (1<<27) | (1<<26); //config
MAT0.0(P1.28) and MAT0.0(P1.29) outputs

        initTimer0();

        while(1)
        {
                //Idle loop
        }
        //return 0; //normally this won't execute
}

void initTimer0(void)
{
```

```
        /*Assuming that PLL0 has been setup with CCLK = 100Mhz and PCLK =
25Mhz.*/
        LPC_SC->PCONP |= (1<<1); //Power up TIM0. By default TIM0 and TIM1 are
enabled.
        LPC_SC->PCLKSEL0 &= ~(0x3<<3); //Set PCLK for timer = CCLK/4 = 100/4
(default)

        LPC_TIM0->CTCR = 0x0;
        LPC_TIM0->PR = PRESCALE; //Increment LPC_TIM0->TC at every 24999+1 clock
cycles
        //25000 clock cycles @25Mhz = 1 mS

        LPC_TIM0->MR0 = 500; //toggle time in mS
        LPC_TIM0->MCR = (1<<1); //Reset on MR0 Match
        LPC_TIM0->EMR |= (1<<7) | (1<<6) | (1<<5) | (1<<4); //Toggle Match
output for MAT0.0(P1.28), MAT0.1(P1.29)

        LPC_TIM0->TCR = 0x02; //Reset Timer
        LPC_TIM0->TCR = 0x01; //Enable timer
}
```

Example 3) – LPC1768 Timer Interrupt Example Code

Here we will use a timer interrupt function which will be called periodically to blink an LED. The Timer Interrupt Service Routine (ISR) will toggle P1.29 every time it is called. If you are using C++ file then you will need to add `extern "C"` before the ISR definition or C++ linker won't be able to link it properly to generate finally binary/hex file. For C code this is not required.

Attention Plz! : Please take your time to go through "initTimer0()" function. Here I've setup LPC176x Timer Interrupt handler which gets triggered when value in TC equals the value in MR0. I will discuss Interrupts in an upcoming tutorial.

```
#include <lpc17xx.h>

#define PRESCALE (25000-1) //25000 PCLK clock cycles to increment TC by 1

void initTimer0();

int main(void)
{
        //SystemInit(); //called by Startup Code before main(), hence no need
to call again.
        LPC_GPIO1->FIODIR |= (1<<29); //set P1.29 as output
        initTimer0();

        while(1)
        {
                //Idle loop
        }
        //return 0; //normally this won't execute
}


void initTimer0(void)
{
```

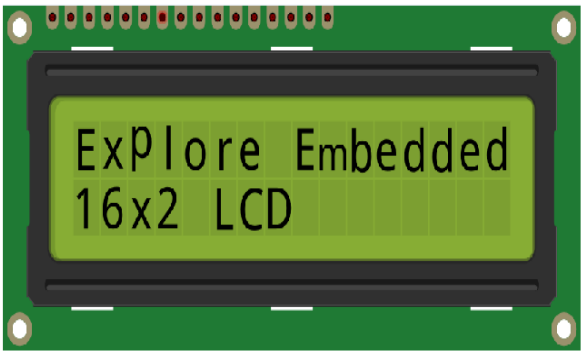
```
        /*Assuming that PLL0 has been setup with CCLK = 100Mhz and PCLK =
25Mhz.*/
        LPC_SC->PCONP |= (1<<1); //Power up TIM0. By default TIM0 and TIM1 are
enabled.
        LPC_SC->PCLKSEL0 &= ~(0x3<<3); //Set PCLK for timer = CCLK/4 = 100/4
(default)

        LPC_TIM0->CTCR = 0x0;
        LPC_TIM0->PR = PRESCALE; //Increment LPC_TIM0->TC at every 24999+1 clock
cycles
        //25000 clock cycles @25Mhz = 1 mS

        LPC_TIM0->MR0 = 500; //Toggle Time in mS
        LPC_TIM0->MCR |= (1<<0) | (1<<1); // Interrupt & Reset on MR0 match
        LPC_TIM0->TCR |= (1<<1); //Reset Timer0

        NVIC_EnableIRQ(TIMER0_IRQn); //Enable timer interrupt

        LPC_TIM0->TCR = 0x01; //Enable timer
}

extern "C" void TIMER0_IRQHandler(void) //Use extern "C" so C++ can link it
properly, for C it is not required
{
        LPC_TIM0->IR |= (1<<0); //Clear MR0 Interrupt flag
        LPC_GPIO1->FIOPIN ^= (1<<29); //Toggle LED
}
```

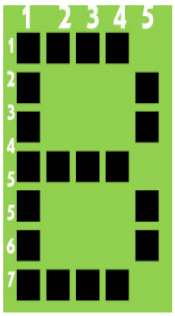
LPC1768: Lcd 4bit

In this tutorial we are going to see how to interface a 2x16 LCD with LPC1768 in 4-bit mode. As per the name the 2x16 has 2 lines with 16 chars on each lines. It supports all the ascii chars and is basically used for displaying the alpha numeric characters. Here each character is displayed in a matrix of 5x7 pixels. Apart from alpha numeric chars it also provides the provision to display the custom characters by creating the pattern. Scope of this tutorial is to show how to display the alpha numeric chars on LCD, Generating and displaying the custom chars will be discussed in subsequent tutorials.

LCD UNIT

Let us look at a pin diagram of a commercially available LCD like **JHD162** which uses a **HD44780** controller and then describe its operation.





Pin Number	Symbol	Pin Function
1	VSS	Ground
2	VCC	+5v
3	VEE	Contrast adjustment (VO)
4	RS	Register Select. 0:Command, 1: Data
5	R/W	Read/Write, R/W=0: Write & R/W=1: Read

6	EN	Enable. Falling edge triggered
7	D0	Data Bit 0 (Not used in 4-bit operation)
8	D1	Data Bit 1 (Not used in 4-bit operation)
9	D2	Data Bit 2 (Not used in 4-bit operation)
10	D3	Data Bit 3 (Not used in 4-bit operation)
11	D4	Data Bit 4
12	D5	Data Bit 5
13	D6	Data Bit 6
14	D7	Data Bit 7/Busy Flag
15	A/LED+	Back-light Anode(+)
16	K/LED-	Back-Light Cathode(-)

Apart from the voltage supply connections the important pins from the programming perspective are the data lines(8-bit Data bus), Register select, Read/Write and Enable pin.

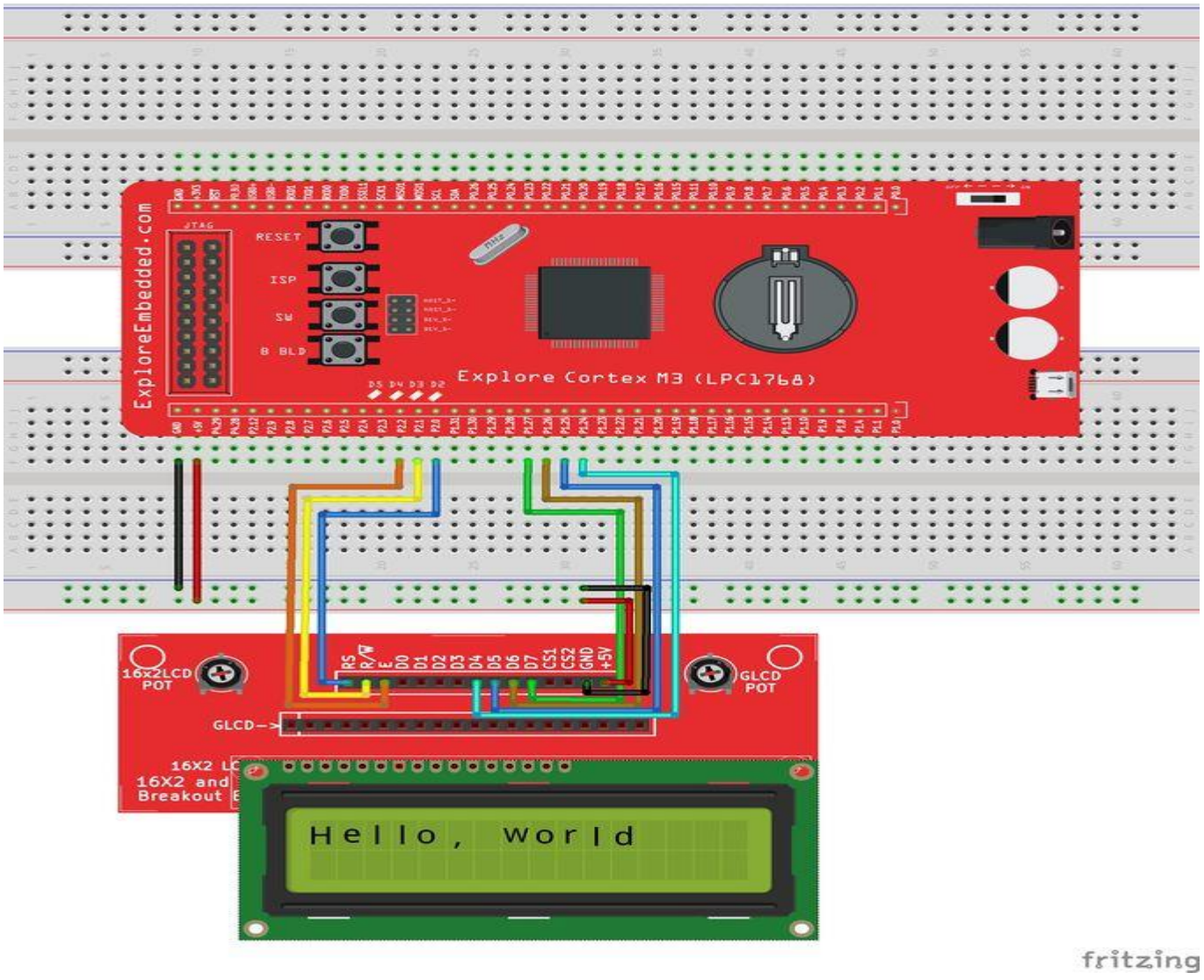
Data Bus: As shown in the above figure and table, an alpha numeric lcd has a 8-bit data bus referenced as D0-D7. As it is a 8-bit data bus, we can send the data/cmd to LCD in bytes. It also provides the provision to send the the data/cmd in chunks of 4-bit, which is used when there are limited number of GPIO lines on the microcontroller.

Register Select(RS): The LCD has two register namely a Data register and Command register. Any data that needs to be displayed on the LCD has to be written to the data register of LCD. Command can be issued to LCD by writing it to Command register of LCD. This signal is used to differentiate the data/cmd received by the LCD.
If the RS signal is **LOW** then the LCD interprets the 8-bit info as **Command** and writes it **Command register** and performs the action as per the command.
If the RS signal is **HIGH** then the LCD interprets the 8-bit info as **data** and copies it to **data register**. After that the LCD decodes the data for generating the 5x7 pattern and finally displays on the LCD.

Read/Write(RW): This signal is used to write the data/cmd to LCD and reads the busy flag of LCD. For write operation the RW should be **LOW** and for read operation the R/W should be **HIGH**.

Enable(EN): This pin is used to send the enable trigger to LCD. After sending the data/cmd, Selecting the data/cmd register, Selecting the Write operation. A HIGH-to-LOW pulse has to be send on this enable pin which will latch the info into the LCD register and triggers the LCD to act accordingly.

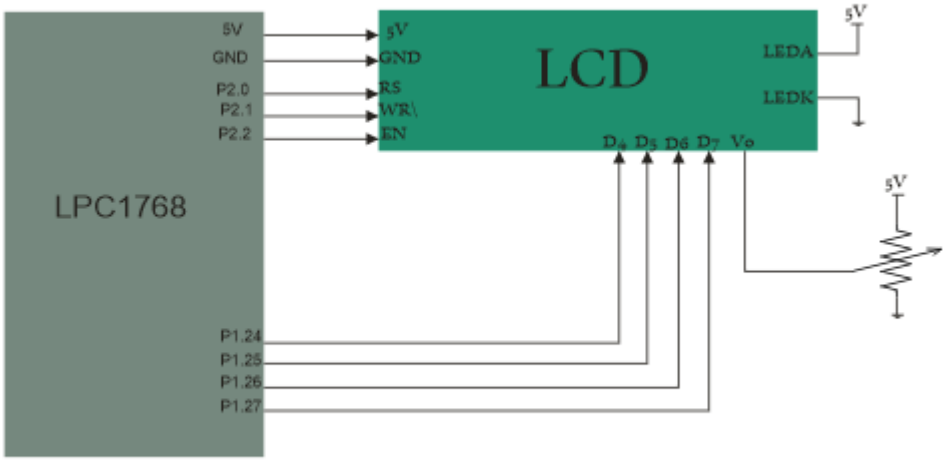
Hardware Connections



Schematic

Below schematic shows the minimum connection required for interfacing the LCD with the microcontroller.

As we are interfacing the LCD in 4-bit mode, only the higher 4 data lines are used as data bus.



Port Connection

This section shows how to configure the GPIO for interfacing the LCD.
The below configuration is as per the above schematic. You can connect the LCD to any of the PORT pins available on your boards and update this section accordingly

```
/* Configure the data bus and Control bus as per the hardware connection */



#define LcdDataBusPort        LPC_GPIO1->FIOPIN
#define LcdControlBusPort    LPC_GPIO2->FIOPIN



#define LcdDataBusDirnReg     LPC_GPIO1->FIODIR
#define LcdCtrlBusDirnReg       LPC_GPIO2->FIODIR



#define LCD_D4      24
#define LCD_D5      25
#define LCD_D6      26
#define LCD_D7      27



#define LCD_RS      0
#define LCD_RW      1
```

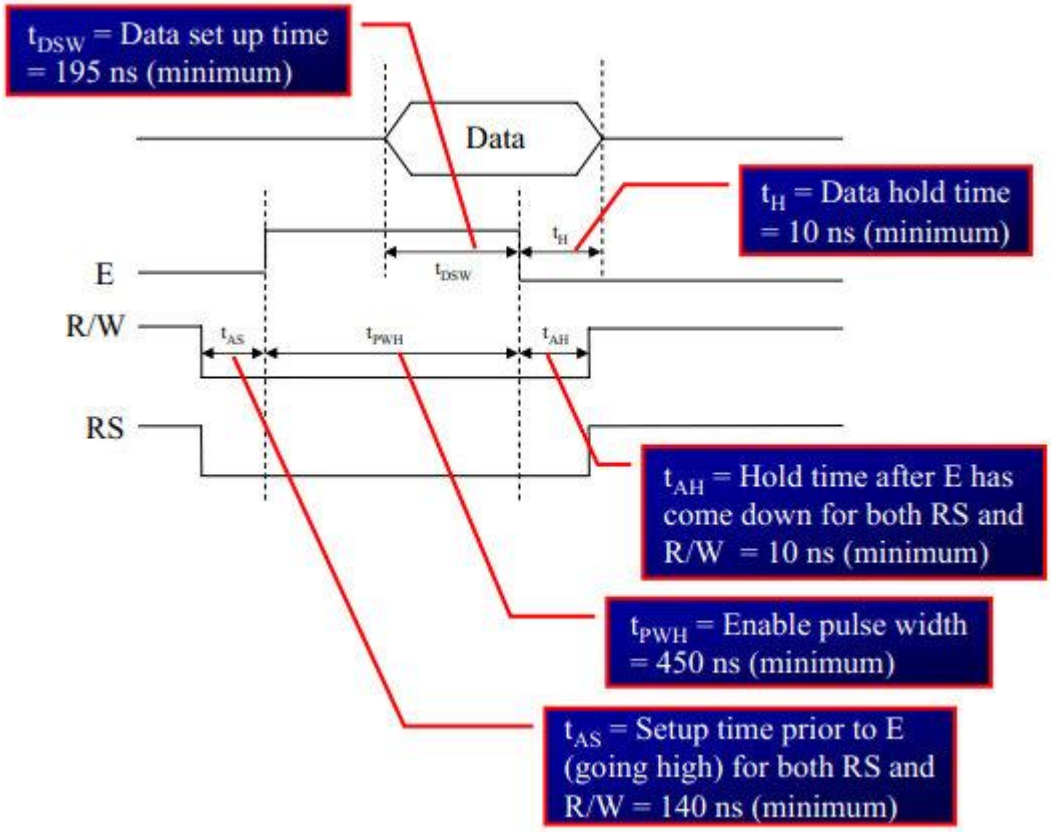
```
#define LCD_EN      2
```

LCD Operation

In this section we are going to see how to send the data/cmd to the LCD along with the timing diagrams. First lets see the timing diagram for sending the data and the command signals(RS,RW,EN), accordingly we write the algorithm and finally the code.

Timing Diagram

The below image shows the timing diagram for sending the data to the LCD.
As shown in the timing diagram the data is written after sending the RS and RW signals. It is still ok to send the data before these signals.
The only important thing is the data should be available on the databus before generating the High-to-Low pulse on EN pin.



Steps for Sending Command:

1. Send the I/P command to LCD.
2. Select the Control Register by making RS low.
3. Select Write operation making RW low.
4. Send a High-to-Low pulse on Enable PIN with some delay_us.

```
/* Function to send the command to LCD. As it is 4bit mode, a byte of data is sent in two 4-bit nibbles */
void Lcd_CmdWrite(char cmd)
{
    sendNibble((cmd >> 0x04) & 0x0F);   //Send higher nibble
    LcdControlBusPort &= ~(1<<LCD_RS); // Send LOW pulse on RS pin for selecting Command register
    LcdControlBusPort &= ~(1<<LCD_RW); // Send LOW pulse on RW pin for Write operation
    LcdControlBusPort |= (1<<LCD_EN);  // Generate a High-to-low pulse on EN pin
    delay(1000);
    LcdControlBusPort &= ~(1<<LCD_EN);



    delay(10000);



    sendNibble(cmd & 0x0F);            //Send Lower nibble
    LcdControlBusPort &= ~(1<<LCD_RS); // Send LOW pulse on RS pin for selecting Command register
    LcdControlBusPort &= ~(1<<LCD_RW); // Send LOW pulse on RW pin for Write operation
    LcdControlBusPort |= (1<<LCD_EN);  // Generate a High-to-low pulse on EN pin
    delay(1000);
    LcdControlBusPort &= ~(1<<LCD_EN);



    delay(10000);
}
```

Steps for Sending Data:

1. Send the character to LCD.
2. Select the Data Register by making RS high.
3. Select Write operation making RW low.
4. Send a High-to-Low pulse on Enable PIN with some delay in_us.

The timings are similar as above only change is that **RS** is made high for selecting Data register.

```
/* Function to send the data to LCD. As it is 4bit mode, a byte of data is sent in two 4-bit nibbles
void Lcd_DataWrite(char dat)
{
    sendNibble((dat >> 0x04) & 0x0F);   //Send higher nibble
    LcdControlBusPort |= (1<<LCD_RS);  // Send HIGH pulse on RS pin for selecting data register
    LcdControlBusPort &= ~(1<<LCD_RW); // Send LOW pulse on RW pin for Write operation
    LcdControlBusPort |= (1<<LCD_EN);  // Generate a High-to-low pulse on EN pin
    delay(1000);
    LcdControlBusPort &= ~(1<<LCD_EN);
```

```
     delay(10000);



    sendNibble(dat & 0x0F);            //Send higher nibble
    LcdControlBusPort |= (1<<LCD_RS);  // Send HIGH pulse on RS pin for selecting data register
    LcdControlBusPort &= ~(1<<LCD_RW); // Send LOW pulse on RW pin for Write operation
    LcdControlBusPort |= (1<<LCD_EN);  // Generate a High-to-low pulse on EN pin
    delay(1000);
    LcdControlBusPort &= ~(1<<LCD_EN);



    delay(10000);
}
```

Code Examples

Example 1

Here is the complete code for displaying the data on 2x16 LCD in 4-bit mode.

```
#include<lpc17xx.h>



/* Configure the data bus and Control bus as per the hardware connection */
#define LcdDataBusPort      LPC_GPIO1->FIOPIN
#define LcdControlBusPort   LPC_GPIO2->FIOPIN



#define LcdDataBusDirnReg   LPC_GPIO1->FIODIR
#define LcdCtrlBusDirnReg   LPC_GPIO2->FIODIR



#define LCD_D4     24
#define LCD_D5     25
#define LCD_D6     26
#define LCD_D7     27



#define LCD_RS     0
#define LCD_RW     1
#define LCD_EN     2
```

```
/* Masks for configuring the DataBus and Control Bus direction */
#define LCD_ctrlBusMask    ((1<<LCD_RS)|(1<<LCD_RW)|(1<<LCD_EN))
#define LCD_dataBusMask    ((1<<LCD_D4)|(1<<LCD_D5)|(1<<LCD_D6)|(1<<LCD_D7))


/* local function to generate some delay */
void delay(int cnt)
{
    int i;
    for(i=0;i<cnt;i++);
}









/* Function send the a nibble on the Data bus (LCD_D4 to LCD_D7) */
void sendNibble(char nibble)
{
    LcdDataBusPort&=~(LCD_dataBusMask);                    // Clear previous data
    LcdDataBusPort|= (((nibble >>0x00) & 0x01) << LCD_D4);
    LcdDataBusPort|= (((nibble >>0x01) & 0x01) << LCD_D5);
    LcdDataBusPort|= (((nibble >>0x02) & 0x01) << LCD_D6);
    LcdDataBusPort|= (((nibble >>0x03) & 0x01) << LCD_D7);
}




/* Function to send the command to LCD.
   As it is 4bit mode, a byte of data is sent in two 4-bit nibbles */
void Lcd_CmdWrite(char cmd)
{
    sendNibble((cmd >> 0x04) & 0x0F);  //Send higher nibble
    LcdControlBusPort &= ~(1<<LCD_RS); // Send LOW pulse on RS pin for selecting Command register
```

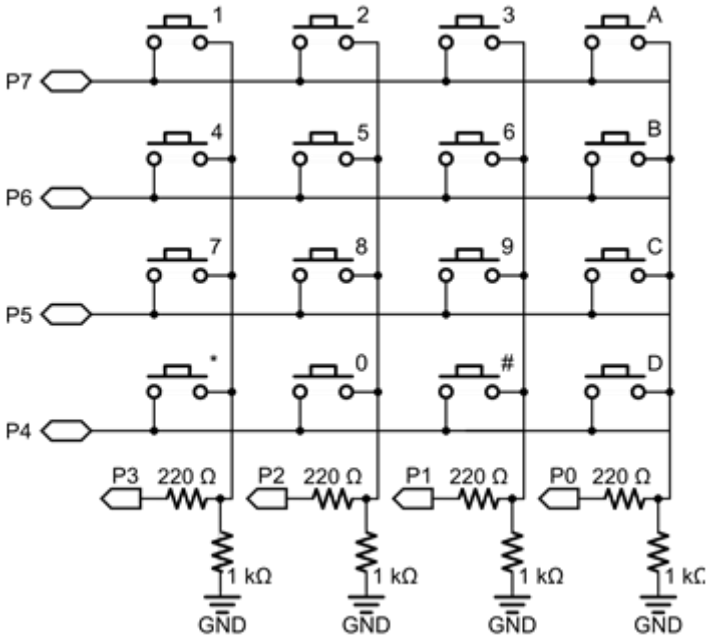
```
    LcdControlBusPort &= ~(1<<LCD_RW); // Send LOW pulse on RW pin for Write operation
    LcdControlBusPort |= (1<<LCD_EN);  // Generate a High-to-low pulse on EN pin
    delay(1000);
    LcdControlBusPort &= ~(1<<LCD_EN);



    delay(10000);



    sendNibble(cmd & 0x0F);            //Send Lower nibble
    LcdControlBusPort &= ~(1<<LCD_RS); // Send LOW pulse on RS pin for selecting Command register
    LcdControlBusPort &= ~(1<<LCD_RW); // Send LOW pulse on RW pin for Write operation
    LcdControlBusPort |= (1<<LCD_EN);  // Generate a High-to-low pulse on EN pin
    delay(1000);
    LcdControlBusPort &= ~(1<<LCD_EN);



    delay(10000);
}









void Lcd_DataWrite(char dat)
{
    sendNibble((dat >> 0x04) & 0x0F);  //Send higher nibble
    LcdControlBusPort |= (1<<LCD_RS);  // Send HIGH pulse on RS pin for selecting data register
    LcdControlBusPort &= ~(1<<LCD_RW); // Send LOW pulse on RW pin for Write operation
    LcdControlBusPort |= (1<<LCD_EN);  // Generate a High-to-low pulse on EN pin
    delay(1000);
    LcdControlBusPort &= ~(1<<LCD_EN);



    delay(10000);



    sendNibble(dat & 0x0F);            //Send Lower nibble
    LcdControlBusPort |= (1<<LCD_RS);  // Send HIGH pulse on RS pin for selecting data register
    LcdControlBusPort &= ~(1<<LCD_RW); // Send LOW pulse on RW pin for Write operation
    LcdControlBusPort |= (1<<LCD_EN);  // Generate a High-to-low pulse on EN pin
    delay(1000);
    LcdControlBusPort &= ~(1<<LCD_EN);
```

```
    delay(10000);
}




int main()
{
    char i,a[]={"Good morning!"};
    SystemInit();                          //Clock and PLL configuration


    LcdDataBusDirnReg |= LCD_dataBusMask;  // Configure all the LCD pins as output
    LcdCtrlBusDirnReg |= LCD_ctrlBusMask;




    Lcd_CmdWrite(0x02);                    // Initialize Lcd in 4-bit mode
    Lcd_CmdWrite(0x28);                    // enable 5x7 mode for chars
    Lcd_CmdWrite(0x0E);                    // Display OFF, Cursor ON
    Lcd_CmdWrite(0x01);                    // Clear Display
    Lcd_CmdWrite(0x80);                    // Move the cursor to beginning of first line



    Lcd_DataWrite('H');
    Lcd_DataWrite('e');
    Lcd_DataWrite('l');
    Lcd_DataWrite('l');
    Lcd_DataWrite('o');
    Lcd_DataWrite(' ');
    Lcd_DataWrite('w');
    Lcd_DataWrite('o');
    Lcd_DataWrite('r');
    Lcd_DataWrite('l');
    Lcd_DataWrite('d');
```

```
    Lcd_CmdWrite(0xc0);
    for(i=0;a[i]!=0;i++)
    {
        Lcd_DataWrite(a[i]);
    }


    while(1);
}
```

4x4 Matrix Keypad

A matrix keypad is the kind of keypad you see on microwave ovens, gas pumps, and calculators. A matrix keypad you can connect to a breadboard is also great for prototypes and inventions where things like codes, times, or other values have to be entered.



How it Works

This 4x4 matrix keypad has 16 built-in pushbutton contacts connected to row and column lines. A microcontroller can scan these lines for a button-pressed state. All the column lines are connected to controller output pins, and all the row lines to input pins for scanning. Then, it picks a column and sets it high. After that, it checks the row lines one at a time. If the row connection stays low, the button on the row has not been pressed. If it goes high, the microcontroller knows which column (the one it set high), and which row, (the one that was detected high when checked).

Program to scan Keypad and display on LCD

```
#include <LPC17xx.h>
# include "lcd-disp.c"     // use all the functions of lcd program
 void scan(void);
 unsigned char Msg1[13] = "KEY PRESSED=";
 unsigned char row, var,flag,key;
 unsigned long int i,var1,temp, temp1, temp2, temp3;
 unsigned char SCAN_CODE[16] = {0x11,0x21,0x41,0x81,
                                0x12,0x22,0x42,0x82,
                                0x14,0x24,0x44,0x84,
                                0x18,0x28,0x48,0x88};

 unsigned char ASCII_CODE[16] = {'0','1','2','3',
                                '4','5','6','7',
                                '8','9','A','B',
                                'C','D','E','F'};

 int main(void)
 {
       LPC_GPIO2->FIODIR |= 0x00003C00; //made output P2.10 to P2.13
(rows)
       //LPC_GPIO1->FIODIR &= 0xF87FFFFF; //made input P1.23 to P1.26
(cols)  //not required since it is by default

       lcd_init();
       temp1 = 0x80;    //point to first line of LCD
       lcd_com();
       delay_lcd(800);
       lcd_puts(&Msg1[0]);     //display the messsage
```

```
while(1)
{
	while(1)
	{
		for(row=1;row<5;row++)
		{
			if(row == 1)
			var1 = 0x00000400;
			else if(row == 2)
			var1 = 0x00000800;
			else if(row == 3)
			var1 = 0x00001000;
			else if(row == 4)
			var1 = 0x00002000;

			temp = var1;

			LPC_GPIO2->FIOCLR = 0x00003C00;  //first clear the port and send appropriate value for
			LPC_GPIO2->FIOSET = var1;		//enabling the row
			flag = 0;
			scan();		//scan if any key pressed in the enabled row
			if(flag == 1)
			break;
		} //end for(row=1;row<5;row++)
		if(flag == 1)
		break;
	} //2nd while(1)
```

```
            for(i=0;i<16;i++)                    //get the ascii code for
display
            {
                  if(key == SCAN_CODE[i])
                  {
                        key = ASCII_CODE[i];
                        break;
                  } //end if(key == SCAN_CODE[i])
            }//end for(i=0;i<16;i++)
            temp1 = 0xc0;     //display in the second line
            lcd_com();
            delay_lcd(800);
            lcd_puts(&key);
      }//end while 1
 }//end main


 void scan(void)
 {
      temp3 = LPC_GPIO1->FIOPIN;
      temp3 &= 0x07800000;   //check if any key pressed in the enabled
row
      if(temp3 != 0x00000000)
      {
                  flag = 1;
                  temp3 >>= 19;   //Shifted to come at HN of byte
                  temp >>= 10;    //shifted to come at LN of byte
                  key = temp3|temp;      //get SCAN_CODE
      }//if(temp3 != 0x00000000)
 }//end scan
```

LPC1768 ADC Programming Tutorial

in this tutorial we will go through ARM Cortex-M3 LPC1768 ADC programming tutorial. Basically we convert an Analog signal to its Digital version. We will also see LPC176x ADC Interfacing Example. This also applies to LPC1769 Microcontroller.

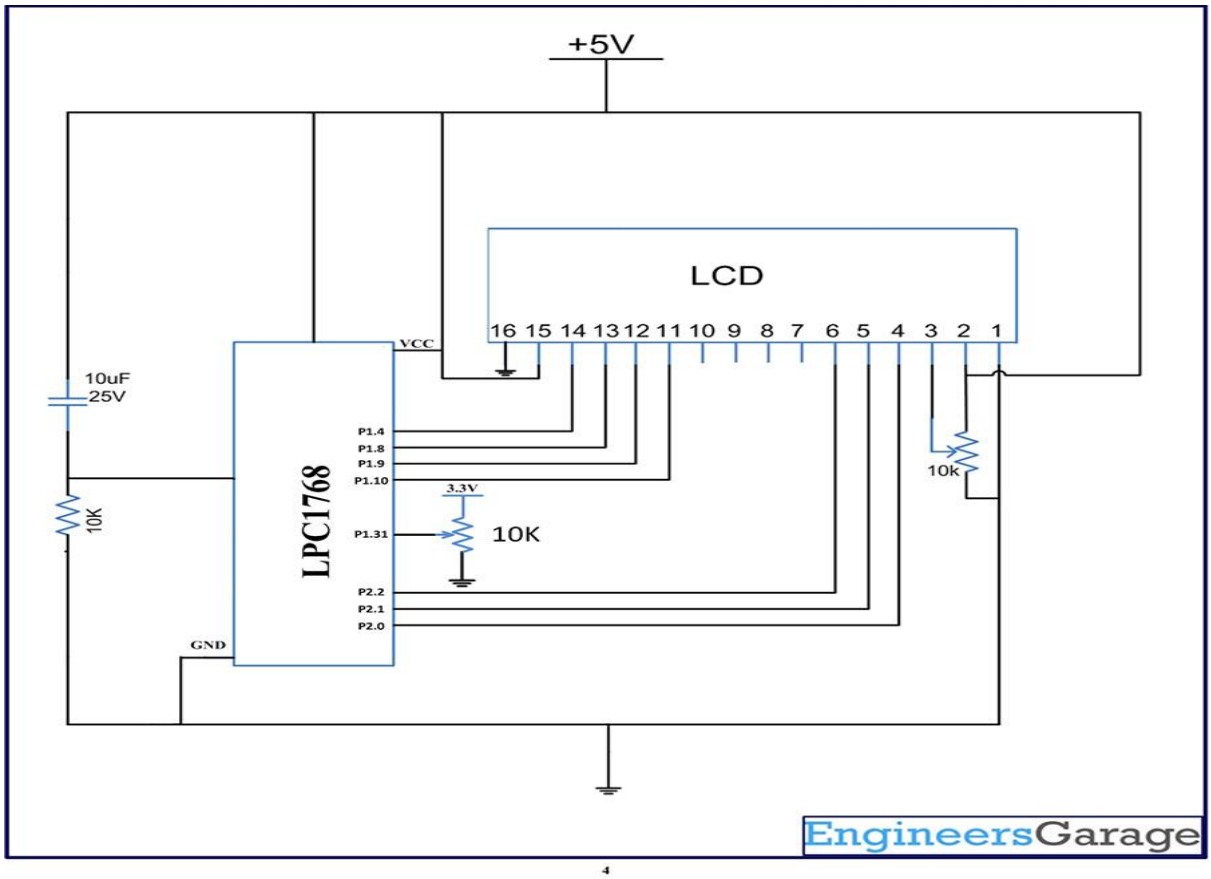


Table of Contents

ADC in ARM LPC176x

Analog to Digital Conversion is used when want to interface an external analog signal or when interfacing analog sensors, like for example a temperature sensor. The ADC block in ARM Cortex-M3 LPC1768 Microcontroller is based on Successive Approximation(SAR) conversion method. LPC1768 ADC Module uses 12-bit SAR having a conversion rate of 200 kHz. The Measurement range if from V_{REFN} to V_{REFP}, or commonly from 0V to ~3V. Maximum of 8 multiplexed inputs can be used for ADC.

Pins relating to ADC Module of LPC1768/LPC1769 :

Pin	Description
AD0.0 to AD0.7 (P0.23/24/25/26, P1.30/31, P0.3/2)	Analog input pins. **Note from Datasheet:** "If ADC is used, signal levels on analog input pins must not be above the level of V_{DDA} at any time. Otherwise, A/D converter readings will be invalid. If the A/D converter is not used in an application then the pins associated with A/D inputs can be used as 5V tolerant digital IO pins."
V_{REFP}, V_{REFN}	These reference voltage pins used for ADC and DAC.
V_{DDA}, V_{SSA}	V_{DDA} is Analog Power pin and V_{SSA} is Ground pin used to power the ADC module.

Analog Power Pin i.e V_{DDA} must be properly isolated/decoupled from V_{CC}, at bare minimum using a ferrite bead and a decoupling capacitor, to suppress noise present on V_{CC} rail and also to suppress MCU's switching noise which can be induced on the V_{CC} rail.

LPC176x ADC Registers

Lets see registers which are used for ADC programming.

1) ADCR – A/D Control Register : This is the main control register for AD0.

1. **Bits[7:0] – SEL:** Bit 'x'(in this group) is used to select pin A0.x in case of AD0.
2. **Bits[15:8] – CLKDIV:** ADC Peripheral clock i.e. PCLK_ADC0 is divided by CLKDIV+1 to get the ADC clock. Note that ADC clock speed must be <= 13Mhz! As per datasheet user must program the smallest value in this field which yields a clock speed of 4.5 MHz or a bit less.
3. **Bit[16] – BURST:** Set to 1 for doing repeated conversions, else 0 for software controlled conversions. Note: START bits must be set to 000 when BURST=1 or conversions will not start. Refer user manual for detailed info.
4. **Bit[21] – PDN :** Set it to 1 for powering up the ADC and making it operational. Set it to 0 for bringing it in powerdown mode.
5. **Bits[26:24] – START:** These bits are used to control the start of ADC conversion when BURST (bit 16) is set to 0. 000 = No start , 001 = Start the conversion now, for other values refer LPC17xx User Manual.
6. **Bit[27] – EDGE:** Set this bit to 1 to start the conversion on falling edge of the selected CAP/MAT signal and set this bit to 0 to start the conversion on rising edge of the selected signal. (Note: This bit is of used only in the case when the START contains a value between 010 to 111 as shown above.)
7. Other bits are reserved.

2) ADGDR – A/D Global Data Register : Contains the ADC's flags and the result of the most recent A/D conversion.

1. **Bits[15:4] – RESULT:** When DONE bit = 1, these bits give a binary fraction which represents the voltage on the pin AD0.x, in range V_{REFP} & V_{REFN}. Value of 0x0 indicates that voltage on the given pin was less than, equal to or greater than V_{REFN}. And a value of 0xFFF means that the input voltage was close to, equal to or greater than the V_{REFP}.
2. **Bits[26:24] – CHN:** Represents the channel from which RESULT bits were converted. 000= channel 0, 001= channel 1 and so on.
3. **Bit[30] – OVERRUN:** In burst mode this bit is 1 in case of an Overrun i.e. the result of previous conversion being lost(overwritten). This bit will be cleared after reading ADGDR.
4. **Bit[31] – DONE:** When ADC conversion completes this bit is 1. When this register(ADGDR) is read and ADCR is written, this bit gets cleared i.e. set to 0. If ADCR

is written while a conversion is in progress then this bit is set and a new conversion is started.
5. Other bits are reserved.

4) ADDR0 to ADDR7 – A/D Data registers : This register contains the result of the most recent conversion completed on the corresponding channel [0 to 7]. Its structure is same as ADGDR except Bits[26:24] are not used/reserved.

5) ADSTAT – A/D Status register : This register contains DONE and OVERRUN flags for all of the A/D channels along with A/D interrupt flag.

1. **Bits[7:0] – DONE[7 to 0]:** Here xth bit mirrors DONEx status flag from the result register for A/D channel x.
2. **Bits[15:8] – OVERRUN[7 to 0]:** Even here the xth bit mirrors OVERRUNx status flag from the result register for A/D channel x
3. **Bit 16 – ADINT:** This bit represents the A/D interrupt flag. It is 1 when any of the individual A/D channel DONE flags is asserted and enabled to contribute to the A/D interrupt via the ADINTEN(given below) register.
4. Other bits are reserved.

LPC1768 ADC Modes, Setup and Programming

ADC modes in LPC1768 & LPC1769:

1. **Software controlled mode :** In Software mode only one conversion will be done at a time. To perform another conversion you will need to re-initiate the process. In software mode, only 1 bit in the SEL field of ADCR can be 1 i.e. only 1 Channel(i.e. Pin) can be selected for conversion at a time. Hence conversions can be done only any channel but one at a time.
2. **Hardware or Burst mode :** In Hardware or Burst mode, conversions are performed continuously on the selected channels in round-robin fashion. Since the conversions cannot be controlled by software, Overrun may occur in this mode. Overrun is the case when a previous conversion result is replaced by new conversion result without previous result being read i.e. the conversion is lost. Usually an interrupt is used in Burst mode to get the latest conversion results. This interrupt is triggered when conversion in one of the selected channel ends.

POWERING THE ADC:

PCON register bits description:

Bit	Symbol	Description	Reset value
0	–	Reserved.	NA

1	PCTIM0	Timer/Counter 0 power/clock control bit.	1
2	PCTIM1	Timer/Counter 1 power/clock control bit.	1
3	PCUART0	UART0 power/clock control bit.	1
4	PCUART1	UART1 power/clock control bit.	1
5	–	Reserved.	NA
6	PCPWM1	PWM1 power/clock control bit.	1
7	PCI2C0	The I2C0 interface power/clock control bit.	1
8	PCSPI	The SPI interface power/clock control bit.	1
9	PCRTC	The RTC power/clock control bit.	1
10	PCSSP1	The SSP 1 interface power/clock control bit.	1
11	–	Reserved.	NA
12	PCADC	A/D converter (ADC) power/clock control bit.	0
13	PCCAN1	CAN Controller 1 power/clock control bit.	0
14	PCCAN2	CAN Controller 2 power/clock control bit.	0
15	PCGPIO	Power/clock control bit for IOCON, GPIO, and GPIO interrupts.	1
16	PCRIT	Repetitive Interrupt Timer power/clock control bit.	0
17	PCMCPWM	Motor Control PWM	0
18	PCQEI	Quadrature Encoder Interface power/clock control bit.	0

19	PCI2C1	The I2C1 interface power/clock control bit.	1
20	–	Reserved.	NA
21	PCSSP0	The SSP0 interface power/clock control bit.	1
22	PCTIM2	Timer 2 power/clock control bit.	0
23	PCTIM3	Timer 3 power/clock control bit.	0
24	PCUART2	UART 2 power/clock control bit.	0
25	PCUART3	UART 3 power/clock control bit.	0
26	PCI2C2	I2C interface 2 power/clock control bit.	1
27	PCI2S	I2S interface power/clock control bit.	0
28	–	Reserved.	NA
29	PCGPDMA	GPDMA function power/clock control bit.	0
30	PCENET	Ethernet block power/clock control bit.	0
31	PCUSB	USB interface power/clock control bit.	0

Fig. 2: Bit Description Of PCON Register In LPC1768 For Powering ADC

Following reset the PCADC (Power Clock Control Bit to ADC) bit is cleared and the ADC is disabled. The first step is to set PCADC followed by PDN in ADC0CR.

Code snipet:

```
LPC_SC->PCONP |= (1 << 12); /* Enable CLOCK for internal ADC controller */
```

SELECTING CLOCK FOR ADC:

By assuming that the main clock for the LPC1768 has been programmed. Each LPC1768 peripheral including the ADC has a clock derived from the main clock as illustrated.

Peripheral Clock Divider

As shown in the below block the frequency of the peripheral clock is determined by two bits in the PCLKSEL registers. ADC is included in PCLKSEL0. With the NXP LPC1768 the peripheral clocks are always active. Following RESET PCLKSEL registers are cleared setting the peripheral clock frequency to CCLK/4 to all the peripherals. The user will have a choice of frequency which can be determined by two bits for each peripherals among the peripheral clock selection registers PCLKSEL0 and PCLKSEL1. By default at reset all values are 00 ie CCLK/4.

00	PCLK_peripheral = CCLK/4
01	PCLK_peripheral = CCLK
10	PCLK_peripheral = CCLK/2
11	PCLK_peripheral = CCLK/8, except for CAN1, CAN2, and CAN filtering when “11” selects = CCLK/6.

Fig. 3: Bit Value Of PCLK_Peripheral For Clock Frequency

Bit	**Symbol**	**Description**
1:0	PCLK_WDT	Peripheral clock selection for WDT.
3:2	PCLK_TIMER0	Peripheral clock selection for TIMER0.
5:4	PCLK_TIMER1	Peripheral clock selection for TIMER1.
7:6	PCLK_UART0	Peripheral clock selection for UART0.
9:8	PCLK_UART1	Peripheral clock selection for UART1.

11:10	–	Reserved.
13:12	PCLK_PWM1	Peripheral clock selection for PWM1.
15:14	PCLK_I2C0	Peripheral clock selection for I2C0.
17:16	PCLK_SPI	Peripheral clock selection for SPI.
19:18	–	Reserved.
21:20	PCLK_SSP1	Peripheral clock selection for SSP1.
23:22	PCLK_DAC	Peripheral clock selection for DAC.
25:24	PCLK_ADC	Peripheral clock selection for ADC.
27:26	PCLK_CAN1	Peripheral clock selection for CAN1.[1]
29:28	PCLK_CAN2	Peripheral clock selection for CAN2.[1]
31:30	PCLK_ACF	Peripheral clock selection for CAN acceptance filtering.[1]

Fig. 4: Bit Value and description of PCLK_peripheral in LPC1768

SELECTING THE ADC FUNCTION TO GPIO:

The block diagram below shows the ADC input pins multiplexed with other GPIO pins. The ADC pin can be enabled by configuring the corresponding PINSEL register to select ADC function. When the ADC function is selected for that pin in the Pin Select register, other Digital signals are disconnected from the ADC input pins.

Adc Channel	**Port Pin**	**Pin Functions**	**Associated PINSEL Register**
AD0	P0.23	0-GPIO, 1-**AD0[0]**, 2-I2SRX_CLK, 3-CAP3[0]	14,15 bits of PINSEL1

AD1	P0.24	0-GPIO, 1-**AD0[1]**, 2-I2SRX_WS, 3-CAP3[1]	16,17 bits of PINSEL1
AD2	P0.25	0-GPIO, 1-**AD0[2]**, 2-I2SRX_SDA, 3-TXD3	18,19 bits of PINSEL1
AD3	P0.26	0-GPIO, 1-**AD0[3]**, 2-AOUT, 3-RXD3	20,21 bits of PINSEL1
AD4	P1.30	0-GPIO, 1-VBUS, 2- , 3-**AD0[4]**	28,29 bits of PINSEL3
AD5	P1.31	0-GPIO, 1-SCK1, 2- , 3-**AD0[5]**	30,31 bits of PINSEL3
AD6	P0.3	0-GPIO, 1-RXD0, 2-**AD0[6]**, 3-	6,7 bits of PINSEL0
AD7	P0.2	0-GPIO, 1-TXD0, 2-**AD0[7]**, 3-	4,5 bits of PINSEL0

Fig. 5: PINSEL register to select ADC function in LPC1768

Code snipet:

LPC_PINCON->PINSEL3|= 0x01<<30;

LPC_PINCON->PINSEL3|= 0x01<<31; / Select the P1_31 AD0[5] for ADC function */***1A. Setting up and configuring ADC Module for software controlled mode**

First, lets define some values which will help us in setting up ADCR register to configure & Initialize ADC block.

```
#define ADC_CLK_EN (1<<12)

#define SEL_AD0_0  (1<<0) //Select Channel AD0.0

#define CLKDIV     1 // ADC clock-divider (ADC_CLOCK=PCLK/CLKDIV+1)

#define PWRUP      (1<<21) //setting it to 0 will power it down

#define START_CNV  (1<<24) //001 for starting the conversion immediately

#define ADC_DONE   (1U<<31) //define it as unsigned value or compiler will
throw #61-D warning

#define ADCR_SETUP_SCM ((CLKDIV<<8) | PowerUP)
//SCM = Software Controlled Mode
```

Now we assign ADCR_SETUP to ADCR along with channel selection information to select channels as required. Finally we assign(by ORing) START_NOW to ADCR to start the conversion process as shown:

```
LPC_SC->PCONP |= ADC_CLK_EN; //Enable ADC clock
LPC_ADC->ADCR =  ADCR_SETUP_SCM | SEL_AD0_0;
LPC_ADC->ADCR |= START_CNV;

//==OR==
LPC_SC->PCONP |= ADC_CLK_EN; //Enable ADC clock
LPC_ADC->ADCR =  ADCR_SETUP | SEL_AD0_0 | START_CNV;
```

1B. Fetching the conversion result in software controlled mode :

In software controlled mode we continuously monitor bit 31 in the corresponding channel data register ADDR. If bit 31 changes to 1 from 0, it means that current conversion has been completed and the result is ready. For example, if we are using channel 0 of AD0 then we monitor for changes in bit 31 as follows :

```
while((LPC_ADC->ADDR0 & ADC_DONE) == 0); //this loop will end when bit 31 of
AD0DR6 changes to 1.
```

After this we extract the result which is stored in ADDR bits 4 to 15. Here we right shift ADDR by 6 places and force all other unrelated bits to 0 by using a 12-bit mask value of 0xFFF. 0xFFF is a mask containing 1's in bit locations 0 to 11 and rest 0's. In our case with ADDR0 being used, it can be done as follows :

```
result = (LPC_ADC->ADDR0>>4) & 0xFFF;
```

2A. Setting up and configuring ADC Module for Burst mode

Configuring ADC Module is similar to what was done in software controlled mode except here we use the CLKS bits and don't use the START bits in ADCR. ADC_DONE is also not applicable since we are using an ISR which gets triggered when a conversion completes on any of the enabled channels. Additionally, we define the following constants:

```
#define SEL_AD0_1 (0x2) //Select Channel AD0.1

#define BURST_ON  (1<<16) // 1 for on and 0 for off

#define ADCR_SETUP_BURST ((CLKDIV<<8) | BURST_ON | PWRUP)
```

We configure and setup the ADC module in a similar manner(as shown above) as follows :

```
LPC_SC->PCONP |= ADC_CLK_EN; //Enable ADC clock
LPC_ADC->ADCR =  ADCR_SETUP_BURST | SEL_AD0_0 | SEL_AD0_1;
```

Note that in this case we can select multiple channels for conversion when setting up AD0CR. START bits are not applicable here since the conversions starts as soon we setup AD0CR register.

2B. Fetching the conversion result in Burst mode :

In Burst mode we use an ISR which triggers at the completion of a conversion in any one of the channel. Now, we just need to find the Channel for which the conversion was done. For this we fetch the channel number from ADGDR which also stores the conversion result. Bits [26:24] in ADGDR contain the channel number. Hence, we shift it 24 places and use a 3-bit mask value of 0x7 as shown below:

```
unsigned long ADGDR_Read = LPC_ADC->ADGDR;
int channel = (ADGDR_Read>>24) & 0x7; //Extract Channel Number
```

After knowing the Channel number, we have 2 options to fetch the conversion result from. Either we can fetch it from ADGDR or from ADDRx of the corresponding channel. Lets use ADGDR for extracting the conversion result as follows:

```
int currentResult = (ADGDR_Read>>4) & 0xFFF; //Extract Conversion Result
```

ARM Cortex-M3 LPC1768/LPC1769 ADC Example

Note that in this case no input protection nor filtering was required. But, when Interfacing external analog signals it is recommended to use some form of input protection.

Interfacing Potentiometer using ADC on LPC176x

This example performs Analog to Digital conversion in Software Controlled Mode. Here we use P0.23 as analog input for measuring the voltage. P0.23 corresponds to Channel 0 of AD0 i.e. AD0.0. For testing I had used a 10K potentiometer for ADC Interfacing and connected the middle leg to P0.23 of my LPC1768 development board. You can also use a 5K potentiometer for this ADC Example. Make sure your board V_{REFP}, V_{REFN}, V_{DDA} & V_{SSA} connections. These connections will be generally present on most LPC176x development boards. If not you can hook it up to V_{CC} and GND respectively using decoupling capacitors and ferrite beads to reduce noise.

C/C++ Source Code :

```
LPC1768/LPC1769 ADC Interfacing Example 1 Source Code using KEIL ARM

#include <lpc17xx.h>
#include <stdio.h>
#include "ocf_lpc176x_lib.h" //contains uart, timer & fputc to retarget printf

#define VREF       3.3 //Reference Voltage at VREFP pin, given VREFN = 0V(GND)
#define ADC_CLK_EN (1<<12)
#define SEL_AD0_0  (1<<0) //Select Channel AD0.0
#define CLKDIV     1 //ADC clock-divider (ADC_CLOCK=PCLK/CLKDIV+1) = 12.5Mhz @
25Mhz PCLK
#define PWRUP      (1<<21) //setting it to 0 will power it down
#define START_CNV  (1<<24) //001 for starting the conversion immediately
#define ADC_DONE   (1U<<31) //define it as unsigned value or compiler will
throw #61-D warning
#define ADCR_SETUP_SCM ((CLKDIV<<8) | PWRUP)

int main(void)
{
        //SystemInit(); //Gets called by Startup code, sets CCLK=100Mhz,
PCLK=25Mhz
```

```
        initUART0(); //Initialize UART0 for uart_printf() - both defined in
tmr_uart_printf.cpp
        initTimer0(); //For delayMS() - both defined in tmr_uart_printf.cpp

        LPC_SC->PCONP |= ADC_CLK_EN; //Enable ADC clock
        LPC_ADC->ADCR =  ADCR_SETUP_SCM | SEL_AD0_0;
        LPC_PINCON->PINSEL1 |= (1<<14) ; //select AD0.0 for P0.23
        int result = 0;
        float volts = 0;

        printf("OCFreaks.com LPC176x ADC Tutorial Example 1.\nSoftware
Controlled ADC Mode on AD0.0 Channel.\n");

        while(1)
        {
                LPC_ADC->ADCR |= START_CNV; //Start new Conversion

                while((LPC_ADC->ADDR0 & ADC_DONE) == 0); //Wait untill
conversion is finished

                result = (LPC_ADC->ADDR0>>4) & 0xFFF; //12 bit Mask to extract
result

                volts = (result*VREF)/4096.0; //Convert result to Voltage

                printf("AD0.0 = %dmV\n" , (int)(volts*1000)); //Display milli-
volts


                delayMS(500); //Slowing down Updates to 2 Updates per second
        }

        //return 0; //This won't execute
}
```

References:

1. http://www.ocfreaks.com/
2. https://www.exploreembedded.com/
3. https://www.engineersgarage.com/
4. http://researchdesignlab.blogspot.com/
5. http://itsurarm.blogspot.com/
6. https://openlabpro.com/