Module - 6

SERIAL, ADC AND DAC INTERFACING

Module-6	
SERIAL, ADC AND DAC INTERFACING: General introduction to serial interfacing, RS232, MAX 232, UART, UART programming, data acquisition system, Analog to Digital Converter (ADC), ADC registers, Digital to Analog converter (DAC), DAC registers, ADC and DAC programming, PWM timer and architecture, PWM programming. **Text 2: Refer Ch 8, Ch 10** **Ref: UM10360, LPC 176x/5x User Manual**	**09 Hours**

Ref:

- **UM10360 LPC 176x/5x User Manual Chapter 24**

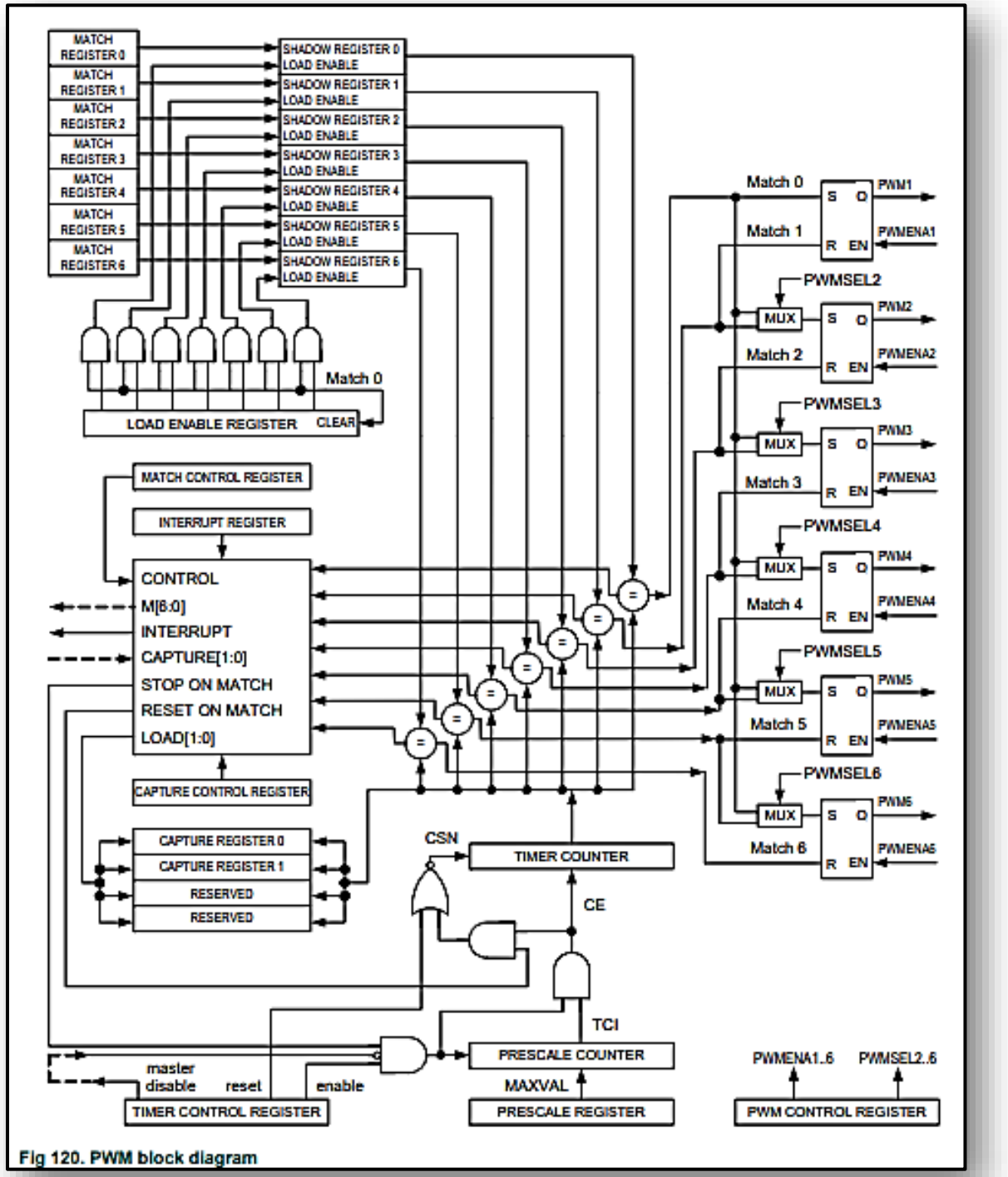


Fig 120. PWM block diagram

PWM Programming

Pulse Width Modulation Basics

- Pulse Width Modulation or PWM is a way to encode data such that it corresponds to the width of the pulse given a fixed frequency.

- It is also a way to control motors , power circuits , etc., using the ‘width’ of the pulse. PWM has numerous uses like Motion Control, Dimming, Encoding Analog Signal into its Digital form, in Power Regulation , etc.

- The PWM is based on the standard Timer block and inherits all of its features, although only the PWM function is pinned out on the LPC1768.
- The Timer is designed to count cycles of the peripheral clock (PCLK) and optionally generate interrupts or perform other actions when specified timer values occur, based on seven match registers. The PWM function is in addition to these features, and is based on match register events.
- PWM can be used for more applications. For instance, multi-phase motor control typically requires three non-overlapping PWM outputs with individual control of all three pulse widths and positions.

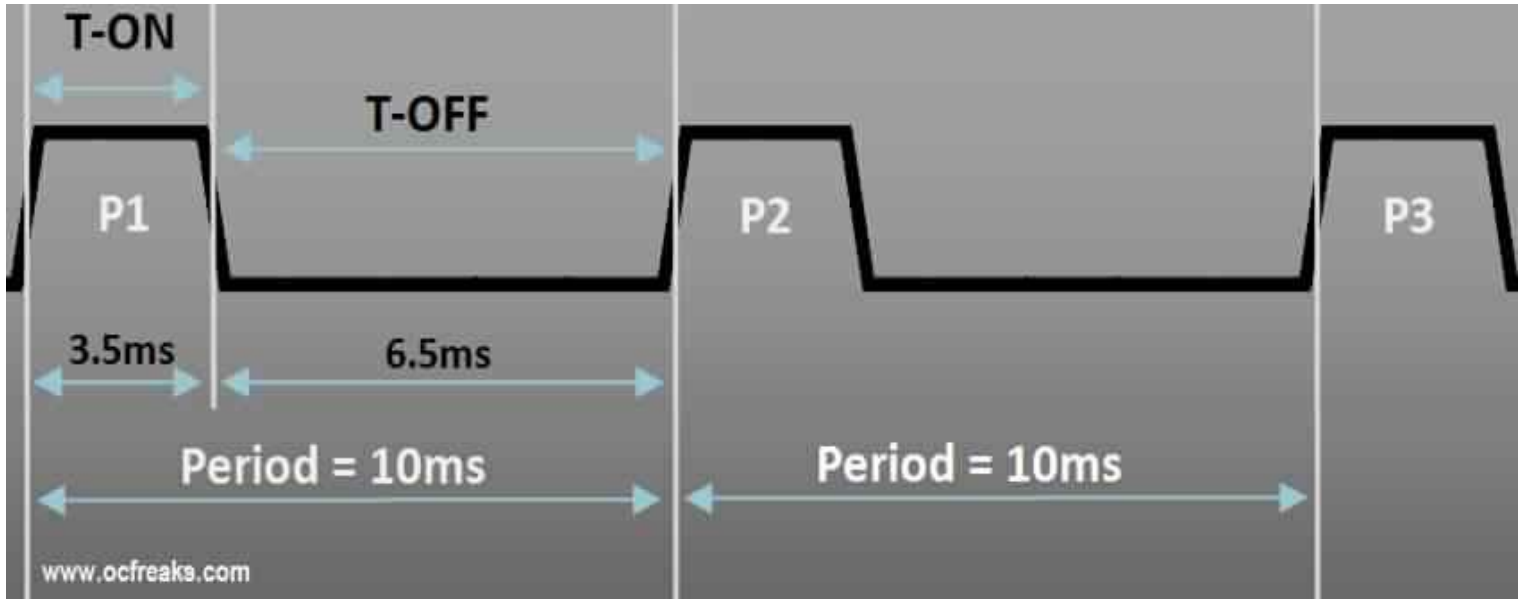
PWM Edges

A PWM signal contains 2 types of Edges which are called Leading Edge and Trailing Edge. The Diagram shown below explains this:

T-ON , T-OFF & Duty Cycle :

$$\text{Duty Cycle} = \frac{\text{T-ON}}{\text{T-ON + T-OFF}}$$

$$\text{Duty Cycle \%} = \frac{\text{T-ON}}{\text{T-ON + T-OFF}} \text{ X100}$$



Period i.e T = 10ms
Frequency = 100hz
T-ON = 3.5ms
T-OFF = 6.5ms
DutyCyle = 35%

Types of PWM

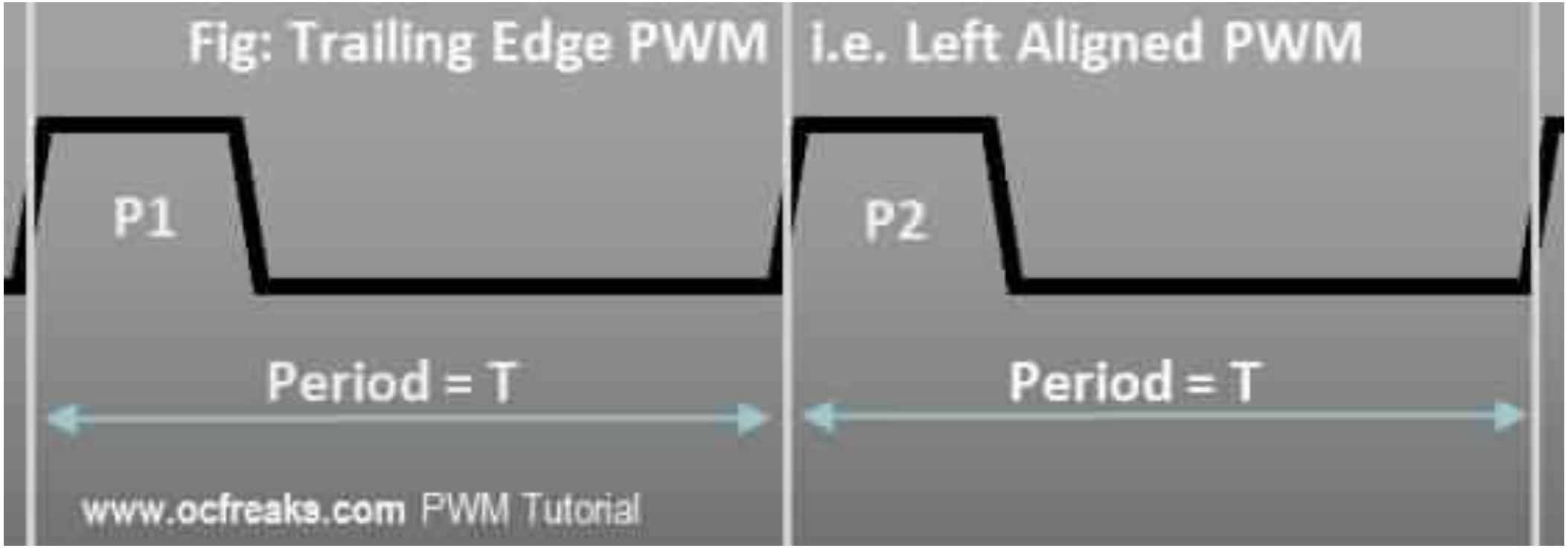
PWM Signal can be Classified in Different ways.

1) **Single Edge PWM**
2) **Double Edge PWM**

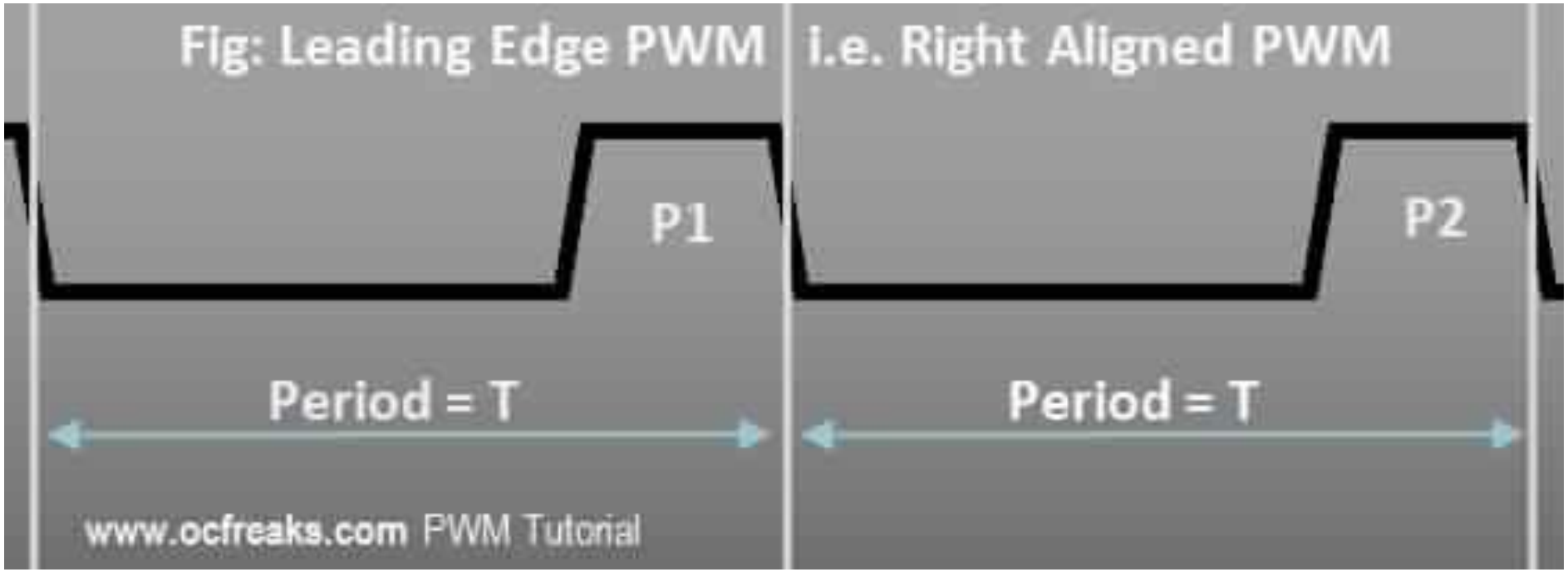
Single Edge PWM

In Single Edge PWM, the Pulse can either at the Beginning or the End of the Period and hence can be further Classified into **Leading Edge(Right Aligned) PWM and Trailing Edge(Left Aligned) PWM**.

In Trailing Edge PWM the Leading Edge is fixed at the Beginning of a Period and the **Trailing Edge is Modulated i.e. Varied**.

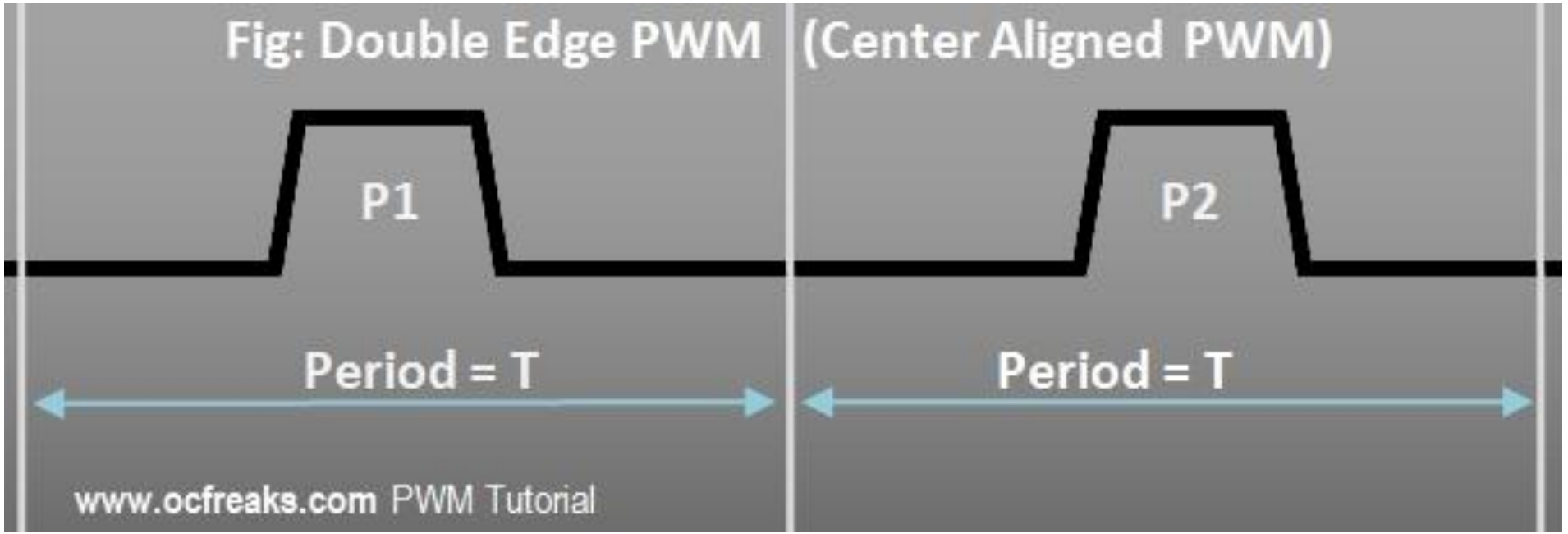


In Leading Edge PWM the Trailing Edge is fixed at the End of a Period and the **Leading Edge is Modulated i.e. Varied**.



Double Edge PWM

In Double Edge PWM the Pulse can be Positioned anywhere within the Period. Its called “Double Edge” because both the Edges are Modulated or Varied. In applications like Multi-Phase Motor control Double Edge PWM is used where the Pulse is Center Aligned to reduce Harmonics.



PWM Voltage

The **Average Voltage** of a PWM Output depends on its Duty Cycle. Lower the Duty Cycle lower will be the Voltage and Vice-Versa.

$$\mathbf{V_{average} = DutyCycle \times V_H}$$

In case the Low State Represents a Negative Voltage then the above equation can be generalized as follows:

$$\mathbf{V_{average} = (DutyCycle \times V_H) + ((1-D) \times V_L)}$$

Where , V_H = Voltage for High State & V_L = Voltage for Low State

ARM Cortex-M3 LPC1768 PWM Module

The LPC1768 Contains a single PWM Module called PWM1 and supports 2 types of PWM:

1. **Single Edge PWM** – Pulse starts with new Period i.e., Pulse is always at the beginning
2. **Double Edge PWM** – Pulse can be present anywhere within the Period

- A PWM block, similar to a Timer block, has a Timer Counter and an associated Prescale Register along with Match Registers. These work exactly the same way as in the case of Timers.

- Match Registers 1 to 6 (except 0) are pinned on LPC1768 i.e. the corresponding outputs are given to actual Pins on LPC1768 MCU. The PWM function must be selected for these Pins, using PINSELx Register, to get the PWM output. These pins are:

Output	PWM1.1	PWM1.2	PWM1.3	PWM1.4	PWM1.5	PWM1.6
Pin Name	P1.18 P2.0	P1.20 P2.1 P3.25	P1.21 P2.2 P3.26	P1.23 P2.3	P1.24 P2.4	P1.26 P2.5

PWM1.1 output corresponds to PWM Match Register 1 i.e., PWM1MR1 , PWM1.2 output corresponds to PWM1MR2 , and so on.

Also Match Register 0 i.e., PWM1MR0 is NOT pinned because it is used to generate the PWM Period. In LPC1768, PWM1.3 is not pinned out.

There are 7 match registers inside the PWM1 block.

- The first Match register PWM1MR0 is used to generate PWM period and hence we are left with 6 Match Registers PWM1MR1 to PWM1MR6 to generate 6 Single Edge PWM signals or 3 Double Edge PWM signals.

- Double edge PWM uses 2 match registers hence we can get only 3 double edge outputs.

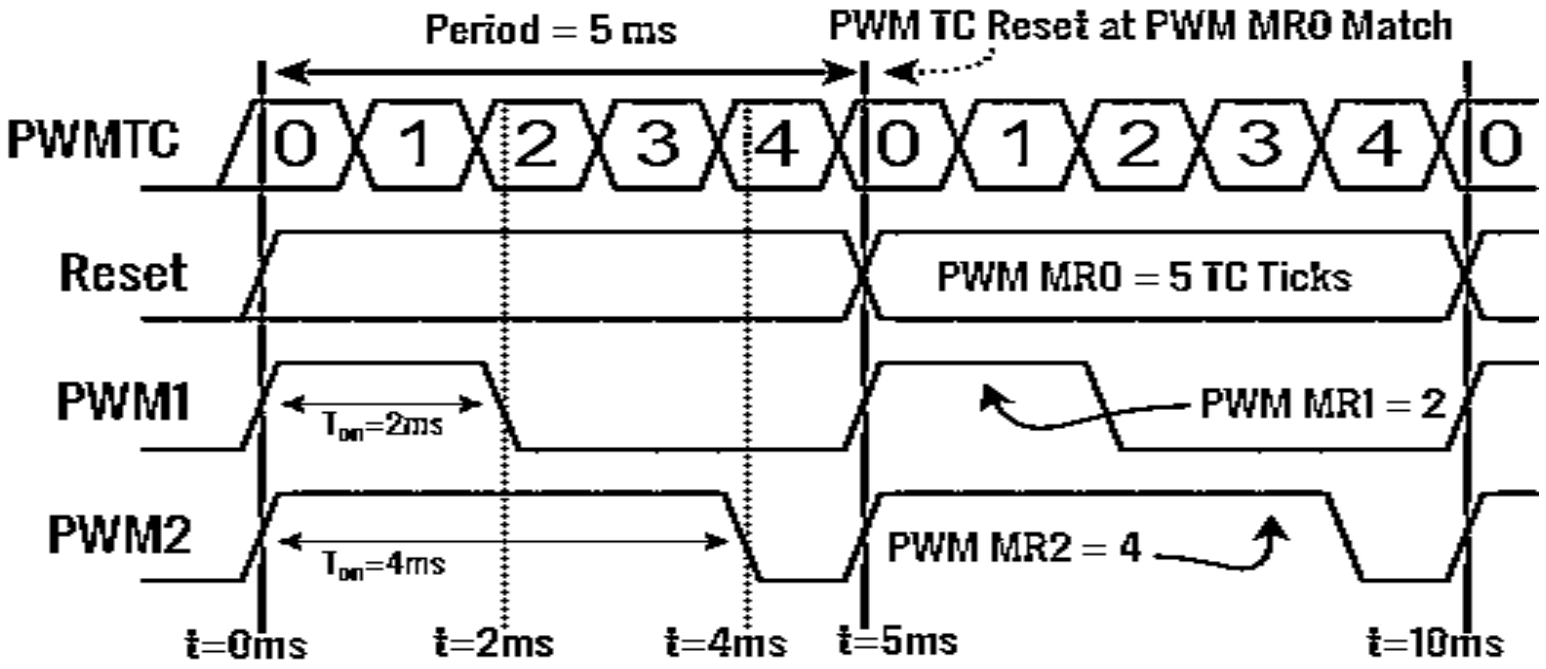
PWM Rules –

The Rules for Single Edged PWM are:

1. All single edged PWM outputs will go high at the beginning of a PWM cycle unless their match value is 0.
2. Each PWM output will go low when its match value is reached.

Note: If no match occurs i.e Match value is greater than Period then the output will remain high!

Consider that our PWM Period duration is 5 milliseconds and TC increments every 1 millisecond using appropriate prescale value. We set PWM1MR0 to 5 i.e. **5 ticks of PWM TC**. We have PWM1MR1 = 2 and PWM1MR2 = 4. Whenever the value in TC matches the value in any of the match register, its corresponding output is set to low until the start of next PWM cycle as shown in PWM Timing Diagram below:



Registers used in LPC176x PWM Programming

PWM1TCR – PWM Timer Control Register: This register is used to control the Timer Counter inside the PWM block. Only Bits: 0, 1 & 3 are used rest are reserved.

- **Bit 0 :** When 1 both PWM Timer counter and PWM Prescale counter are enabled. When 0 both are disabled.
- **Bit 1 :** When set to 1 it will reset both Timer and Prescale counter of PWM block (at next edge of PCLK).
- **Bit 3 :** Enables PWM mode i.e., the PWM outputs.
- Other Bits : Reserved.

PWM1PR – PWM Prescale Register: PWMPR is used to control the resolution of the PWM outputs. The Timer Counter(TC) will increment every PWMPR+1 Peripheral Clock Cycles (PCLK).

PWM1MR0 – PWM1MR6 (Match Registers):

The seven Match registers contain **Pulse Width Values i.e the Number of PWM1TC Ticks**.

PWM1MR0 holds the value of 1 complete cycle or the period.

PWM1MCR – PWM Match Control Registers: The PWM Match Control Register is used to specify what operations can be done when the value in a particular Match register equals the value in TC.
For each Match Register we have 3 options :

- Either generate an Interrupt ,
- Reset the TC
- Stop, which stops the counters and disables PWM.

Hence this register is divided into group of 3 bits. The first 3 bits are for Match Register 0 i.e PWMMR0 , next 3 for PWMMR1 , and so on.

1. **Bit 0 :** Interrupt on PWM1MR0 Match – If set to 1 then it will generate an Interrupt else disable if set to 0.
2. **Bit 1 :** Reset on PWM1MR0 Match – If set to 1 it will reset the PWM Timer Counter i.e. PWM1TC else disabled if set to 0.
3. **Bit 2 :** Stop on PWM1MR0 Match – If this bit is set 1 then both PWM1TC and PWM1PC will be stopped & disable the Counters.
4. Similarly {Bits 3,4,5} for PWM1MR1 , {Bits 6,7,8} for PWM1MR2 , {Bits 9,10,11} for PWM1MR3 ,{Bits 12,13,14} for PWM1MR4 ,{Bits 15,16,17} for PWM1MR5 , {Bits 18,19,20} for PWM1MR6.

PWM1IR – PWM Interrupt Register: If an interrupt is generated by any of the Match Register then the corresponding bit in PWM1IR will be set high. Writing a 1 to the corresponding location will clear that interrupt.

1. **Bits 0,1,2,3** are for PWM1MR0, PWM1MR1, PWM1MR2, PWM1MR3 respectively and
2. **Bits 8,9,10** are for PWM1MR4 , PWM1MR5 , PWM1MR6 respectively. Other bits are reserved.

PWMLER – Load Enable Register:

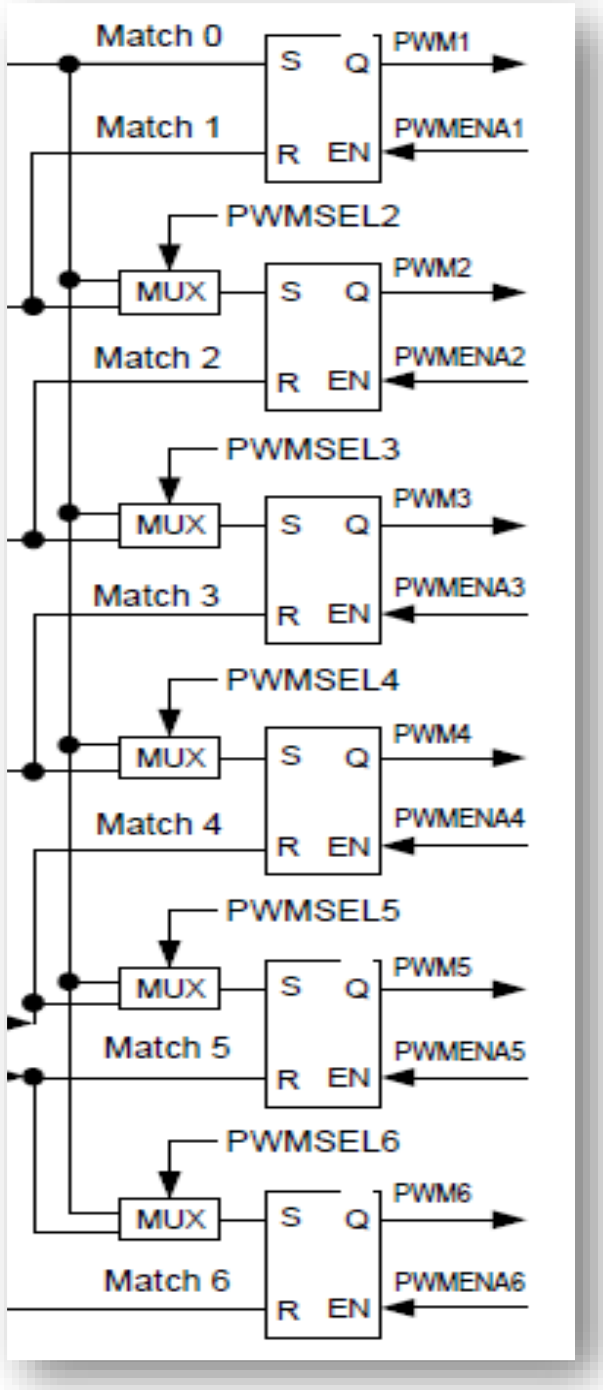
- The PWM Load Enable Register is used to control the way Match Registers are updated when PWM generation is active.
- When PWM mode is active and we apply new values to the Match Registers the new values won't get applied immediately. Instead what happens is that the value is written to a "Shadow Register". It can be thought of as a duplicate Match Register. Each Match Register has a corresponding Shadow Register.
- The value in this Shadow Register is transferred to the actual Match Register when: PWM1TC is reset (i.e., at the beginning of the next period) and the corresponding Bit in PWM1LER is 1.
- Bit 'x' in PWM1LER corresponds to match Register 'x'. i.e., Bit 0 is for PWM1MR0 , Bit 1 for PWM1MR1 , .. and so on.

PWM1PCR – PWM Control Register: This register is used for Selecting between Single Edged & Double Edged outputs and also to Enable/Disable the 6 PWM outputs which go to their corresponding Pins.

1. **Bits 2 to 6** are used to select between Single or Double Edge mode for PWM 2,3,4,5,6 outputs. If **Bit 2** is set to 1 then PWM1.2 (i.e., the one corresponding to PWM1MR2) output is double edged else if set 0 then its Single Edged. Similarly **Bit x** for PWM1.x correspondingly.
2. **Bits 9 to 14** are used to Enable/Disable PWM outputs. If **Bit 9**is set to 1 then PWM1.1 output is enabled , else disabled if set to 0. Similarly remaining bits for PWM1.x correspondingly.

How to generate PWM?

- When the value for the cycle time or the period is loaded into MR0, The selected PWM output goes high.
- TC starts counting. When it matches with the value in the corresponding match register, the PWM output goes low and remains low until TC matches with the MR0. Then the next cycle starts.
- For the next cycle if the match register value corresponding to the PWM output is changed, the pulse width is changed.



Configuring and Initializing PWM Block

1. Select the PWM function for the PIN on which you need the PWM output using applicable LPC_PINCON->PINSELx register.
2. Select Single Edge or Double Edge Mode using LPC_PWM1->PCR. By default its Single Edge Mode.
3. Load the value to LPC_PWM1->PR.
4. Set the Value for PWM Period in LPC_PWM1->MR0.
5. Set the Values for other Match Registers i.e the Pulse Widths.
6. Set appropriate bit values in LPC_PWM1->MCR, like for e.g. resetting PWM1TC for PWMMR0 match and optionally generate interrupts if required
7. Set Load Enable Bits for the Match Registers that you've used. This is important!
8. Then Enable PWM outputs using LPC_PWM1->PCR.
9. Now Reset PWM Timer using LPC_PWM1->TCR.
10. Finally, Enable Timer Counter and PWM Mode using LPC_PWM1->TCR.

Power Control for Peripherals register (PCONP)

The PCONP register allows turning off selected peripheral functions for the purpose of saving power. This is accomplished by gating off the clock source to the specified peripheral blocks

Table 46. Power Control for Peripherals register (PCONP - address 0x400F C0C4) bit 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 I^2C0 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. **Note:** Clear the PDN bit in the AD0CR before clearing this bit, and set this bit before setting PDN.	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 I^2C1 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

```
#include <LPC17xx.H>
void pwm_init(void);
void PWM1_IRQHandler(void);
unsigned long int i;
unsigned char flag,flag1;
int main(void)
{
        pwm_init();
        while(1);
}


        //end of main
```

```
void pwm_init(void)
{
        LPC_SC->PCONP = (1<<6);                                  //PWM1 is powered
        LPC_PINCON->PINSEL7 = 0x000C0000;  //pwm1.2 is selected for the pin P3.25

        LPC_PWM1->PR  = 0x00000000;        //Count frequency : Fpclk
        LPC_PWM1->PCR = 0x00000400;        //select PWM2 single edge
        LPC_PWM1->MCR = 0x00000003;        //Reset and interrupt on PWMMR0
        LPC_PWM1->MR0 = 30000;             //setup match register 0 count
        LPC_PWM1->MR2 = 0x00000100;        //setup match register MR1
        LPC_PWM1->LER = 0x000000FF;        //enable shadow copy register
        LPC_PWM1->TCR = 0x00000002;        //RESET COUNTER AND PRESCALER
        LPC_PWM1->TCR = 0x00000009;        //enable PWM and counter

        NVIC_EnableIRQ(PWM1_IRQn);
}
```

```
void PWM1_IRQHandler(void)
{
		LPC_PWM1->IR = 0xff;                                                    //clear the interrupts

		if(flag == 0x00)
  {
				LPC_PWM1->MR2 += 100;
				LPC_PWM1->LER = 0x000000FF;

				if(LPC_PWM1->MR2 >= 27000)                              //Is Duty Cycle 90% ??
				{
		flag1 = 0xff;
		flag = 0xff;
		LPC_PWM1->LER = 0x000000fF;
				}
		}
  else if(flag1 == 0xff)
  {
				LPC_PWM1->MR2 -= 100;
				LPC_PWM1->LER = 0x000000fF;

				if(LPC_PWM1->MR2 <= 0x300)                              //Is Duty Cycle 1% ??
				{
						flag  = 0x00;
						flag1 = 0x00;
						LPC_PWM1->LER = 0X000000fF;
				}
		}
}
```

General introduction to serial interfacing

Serial interfacing is a communication protocol used to transfer data between two or more devices in a sequential manner.

In ARM processors, serial interfacing is commonly used to connect devices such as

- sensors,
- actuators, and
- other peripherals to the processor etc…

Serial interfaces are typically classified into two categories:

1. **Synchronous** and
2. **Asynchronous**

Synchronous interfaces require a **clock signal** to be transmitted along with the data, while asynchronous interfaces do not.

Asynchronous interfaces are more commonly used in ARM processors, as they are simpler to implement and require fewer hardware resources.

The most common types of serial interfaces used in ARM processors are

1. **UART** (Universal Asynchronous Receiver-Transmitter) and
2. **SPI** (Serial Peripheral Interface).

- **UART** is a simple serial interface that uses two wires (RX and TX) for communication.
- **SPI**, on the other hand, is a more complex interface that can support multiple devices using a single bus.

To interface with a UART or SPI device, the ARM processor typically uses one of its built-in hardware modules known as a **USART** (Universal Synchronous/Asynchronous Receiver-Transmitter) or **SPI controller**. These modules provide the necessary hardware support to generate and receive the serial data, as well as handle other aspects of the communication protocol such as framing and error detection.

Programming the USART or SPI controller involves configuring various registers in the ARM processor to set up the communication parameters such as

a. Baud rate,
b. Data format, and
c. Flow control.

Once the controller is configured, the ARM processor can send and receive data by reading and writing to specific registers associated with the controller.

9.1.2 LCD Interface

- Now-a-days, many LCD modules are available which have built-in drivers for LCD and interfacing circuitry to interface them to microprocessor/microcontroller systems. These LCD modules allow display of characters as well as numbers. They are available in 16 X 2, 20 X 1, 20 X 2, 20 X 4 and 40 X 2 sizes.
- The main advantage of using LCD is modules is that they require very less power. The first figure represents number of character in each line and second figure represents number of lines the display has.
- In this section, we see the interfacing of LCD display unit to STM32 board in 4-bit mode, and display a fixed message on the LCD screen. For this we have to connect
  - Four data lines to D8 to D11 on the Arduino connector.
  - Register select (RS) input to D13
  - Enable (E) input to D12

Fig. 9.1.6 shows the connection between 16 X 2 LCD display unit and STM32 board.

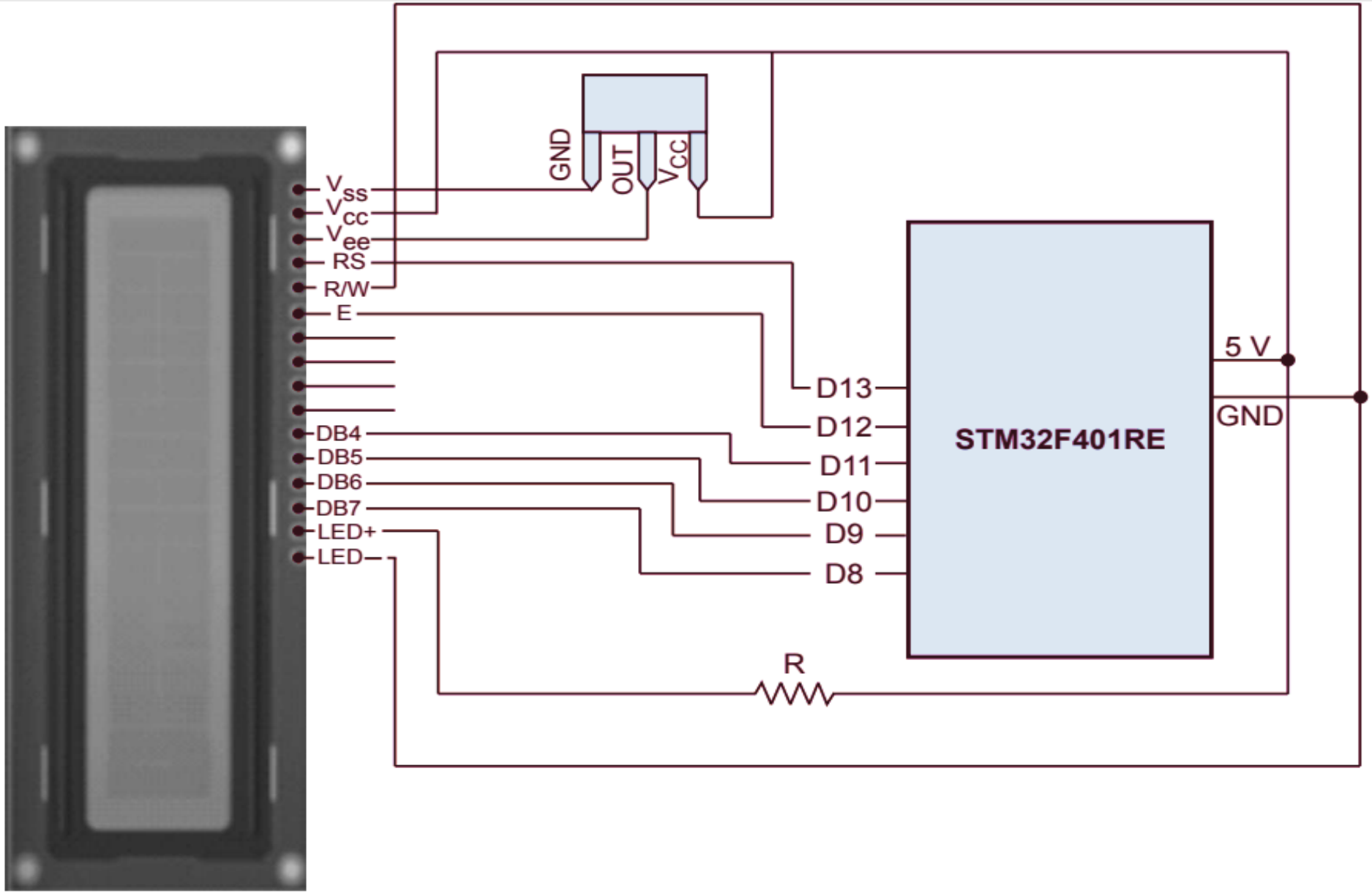


Fig. 9.1.6 Connection between LCD display unit and STM32 board

Example 9.1.5 *Write a program to display message "LCD interfacing" on the first line and display message "with STM32 Board" on the second line of LCD display.*

Solution :

```
#include "mbed.h"
#include "TextLCD.h"
#include "TextLCDScroll.h"
TextLCDscroll lcd (D13, D12, D11, D10, D9, D8, TextLCD : : LCD16×2);
int main()
{
        while(1)
        {
            lcd.setLine (0, "LCD Interfacing");
            lcd.setLine (1, "with STM32 Board");
            wait (20);
        }
}
```

- Here, we need to include two more header files TextLCD.h and TextLCDScrooll.h along with the mbed.h.

Example 9.1.6 *Write a program to display message "LCD interfacing with STM32 Board" on the first line and display message "Scrolling display" on the second line of LCD display.*

Solution :

```
#include "mbed.h"
#include "TextLCD.h"
#include "TextLCDScroll.h"
TextLCDscroll lcd (D13, D12, D11, D10, D9, D8, TextLCD : : LCD16×2);
int main()
{
        lcd.cls();
        lcd.setSpeed (2);
        while(1)
        {
            lcd.setLine (0, "LCD Interfacing with STM32 Board");
            lcd.setLine (1, "Scrolling Display");
            wait (100);
        }
}
```

- The function setSpeed sets the scolling speed at 2 characters/second.

9.2 Serial Port Terminal Application

- We know that, we can establish the serial communication link between PC and microcontroller by connecting USB connector. With this serial communication link data can be transferred from microcontroller to PC.
- Software tools such as CoolTerm, TeraTerm etc., are used to read data using serial communication link and to print the data on the PC screen. Such tools are known as **HyperTerminal Tools** and it is required to install such toll on the computer.
- Once HyperTerminal Tools is installed, we need to configure serial port.

Example 9.2.1 *Write a program to enable serial communication link and print the values read from the analog input.*

Solution :

```
#include "mbed.h"                  // include header file
Serial  pc (USBTX, USBRX);         // Make the serial communication
AnalogIn analog_value (A1);
int main()
{
        float val;
        while (1)
        {
          val = analog_value.read();     // read value from analog input
          pc.printf ("The value read from analog pin = %f\n", val);
        }
return 0;
}
```

Review Question

1. *Write a short note on Serial Port terminal application.*

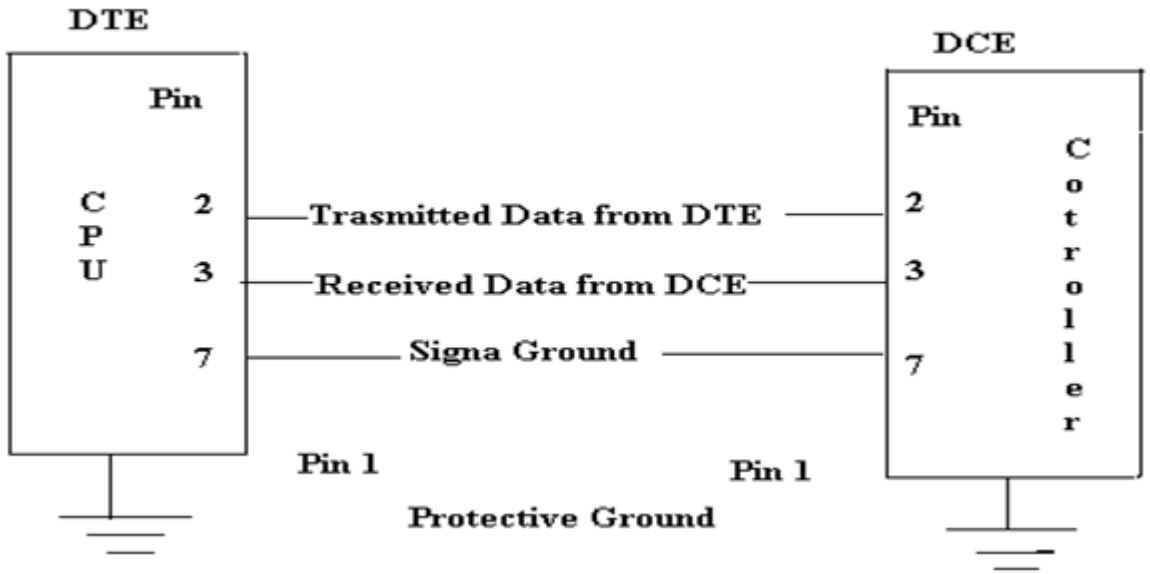
RS232 (Recommended Standard 232)

- RS232 (Recommended Standard 232) is a popular serial communication standard that has been used for decades in many electronic devices.
- The standard defines the electrical and physical characteristics of the communication protocol, including voltage levels, signal timing, and connector pinouts.
- To implement RS232 in an ARM processor, a UART module is typically used.
- The UART module is responsible for generating and receiving the serial data signals and for handling other aspects of the communication protocol such as framing and error detection.

RS232 (Recommended Standard 232)

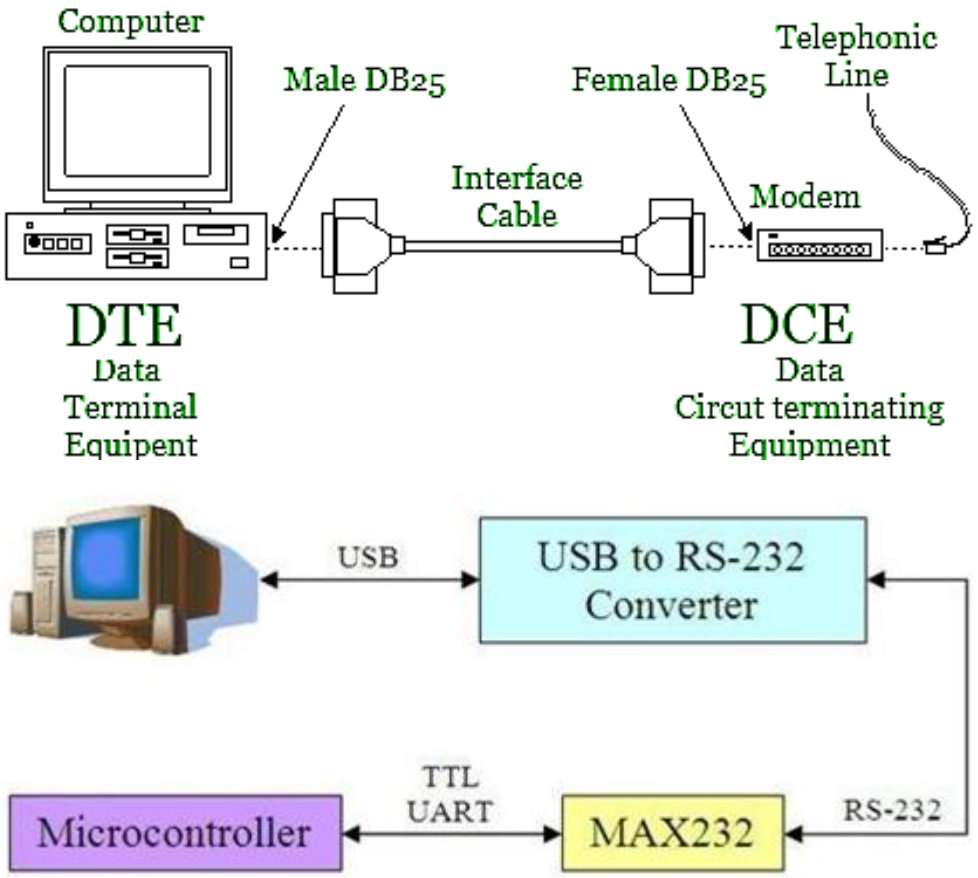
- The ARM processor provides hardware support for implementing RS232 through one of its built-in USART (Universal Synchronous/Asynchronous Receiver-Transmitter) modules.
- The USART module provides a flexible interface that can be configured to operate in either synchronous or asynchronous mode and supports a range of data formats, including 8-bit and 9-bit data, as well as parity and stop bits.
- To use the USART module for RS232 communication, the ARM processor must be configured with the appropriate baud rate, data format, and flow control settings.
- Once configured, the processor can transmit and receive data over the RS232 interface by reading and writing to the appropriate USART registers.

- Rs-232 cable is used to identify the difference between two signal levels between logic 1 and logic 0.
- The logic 1 is represented by the -12V and logic 0 is represented the +12V.
- The RS-232 cable works at different baud rates like 9600 bits/s, 2400bits/s, 4800bits/s etc.
- The RS-232 cable has two-terminal devices namely Data Terminal Equipment and Data communication Equipment. Both devices will send and receives signals.
- The data terminal equipment is a computer terminal and data communication Equipment is modems, or controllers, etc.

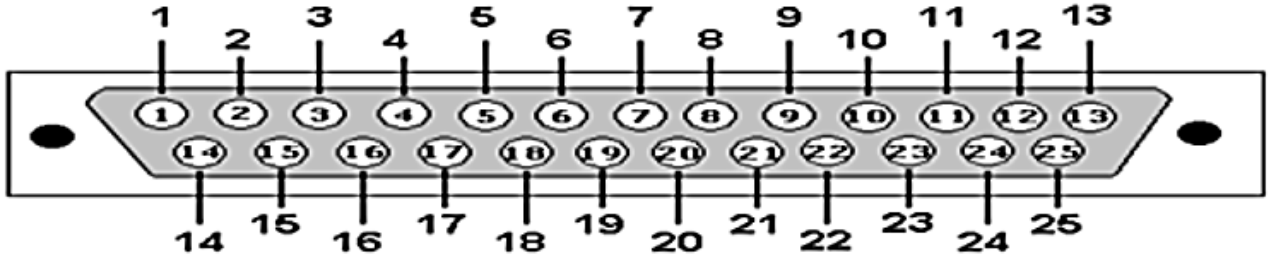


- **Data Terminal Equipment (DTE) :** It includes any unit that functions either as a source of or as a destination for binary digital data.
- **Data Circuit Terminating Equipment (DCE) :** It includes any functional unit that transmit or receives data in form of an analog or digital signal through a network.

- Now a day's most of the personal computers have two serial ports and one parallel port (RS232).
- The parallel port sends and receives the 8-bit data at a time over eight separate wires and this transfers the data very quickly, the parallel ports are typically used to connect a printer to a PC.
- A serial port sends and receives one-bit data at a time over one wire and it transfers data very slowly. The RS-232 stands for recommended slandered and 232 is a number X indicates the latest version like RS-232c, RS232s.



RS232 Pin Description



It is a 25-pin connector, each pin has its function is as follows.

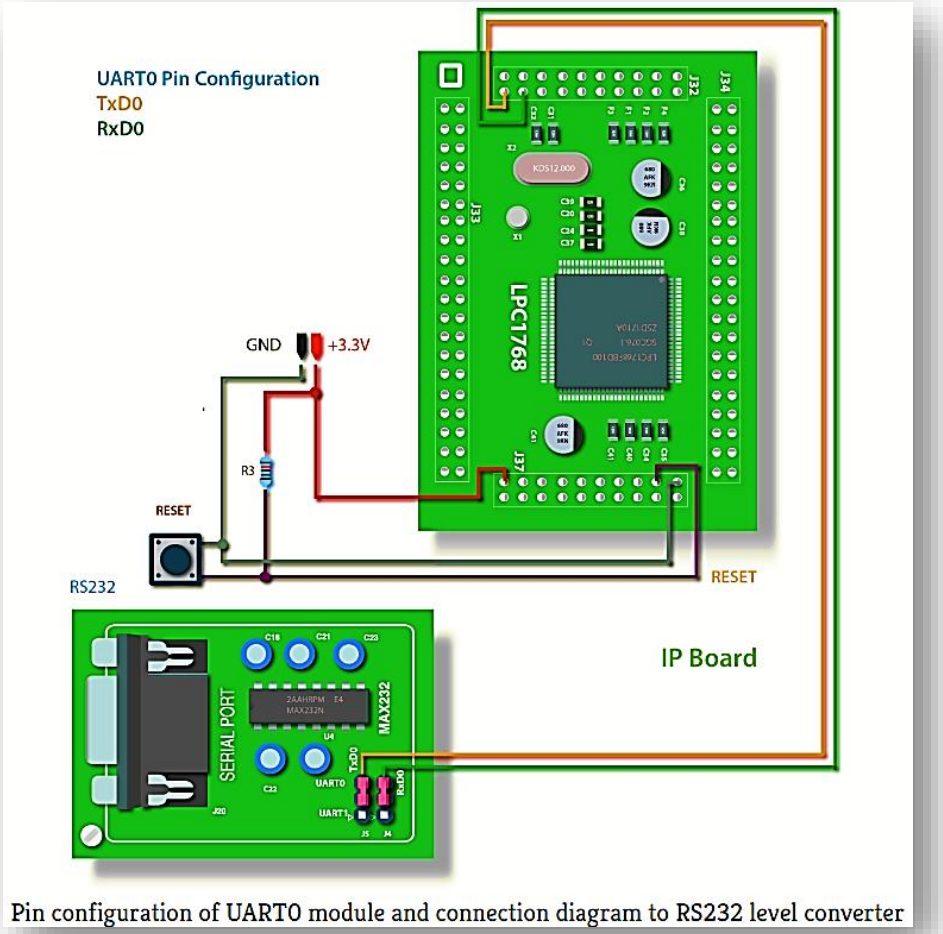
PIN 1*: (Protective Ground);* It is a ground Pin.
PIN 2: Transmit Data.
PIN 3: Receive Data.
PIN 2 & PIN 3: These pins are the most important pins for data transmitting and receiving. The 1 & 2-pins are used to data transmission and pin-3 used to data receiving purpose.
PIN 4: Request to send.
Pin 5*:* Clear to send.
PIN 6*:* Data Set Ready.
PIN *20:* Data terminal Ready.
PIN 4, PIN 5, PIN 6, PIN 20: These pins are the handshaking pins(flow of control). Normally terminals cannot transmit the data until clear to send transmission is received from the DCE.
PIN 7: This pin is the common reference for all signals, including data, timing, and control signals. The DCE and DTE work properly across the serial interface and the pin-7 must be connected both ends without interface would not work.
PIN 8: This pin is also known as received line signal detector carrier detect. This signal is activated when a suitable carrier is established between the local and remote DCE devices.

PIN9: This pin is a DTE serial connector, this signal follows the incoming ring to an extent. Normally this signal is used by DCE auto-answer mode.
PIN 10: Test Pin.
PIN 11: standby select.
PIN 12: Data Carrier Detect.
PIN 13: Clear to send.
PIN 14: Transmit data.
PIN 15: Transmit clock.
PIN 17: Receive clock.
PIN 24: External Clock.
PIN 15, 17, 24; Synchronous modems use the signals on these pins. These pins are controlled bit timing.
PIN 16: Receive data.
PIN 18: Test Pin.
PIN 19: Request to send.
PIN 21: (Signal Quality Detector); This pin Indicates the quality of the received carrier signal because the transmitting modem must be sent 0 or either 1 at each bit time, the modem controls the timing of the bits from the DTE.
PIN 22: (Ring Indicator): The ringing indicator means the DCE informs the DTE that the phone is ringing. All the modems designed for directly connected to the phone network equipped with the auto-answer.
PIN 23: Data Signal Rate Detector

RS232 (Recommended Standard 232)

The LPC 1768 micro-controller consists of 4 UART peripherals. (UART0, UART1, UART2, and UART3). Few of the striking features of these peripherals are:

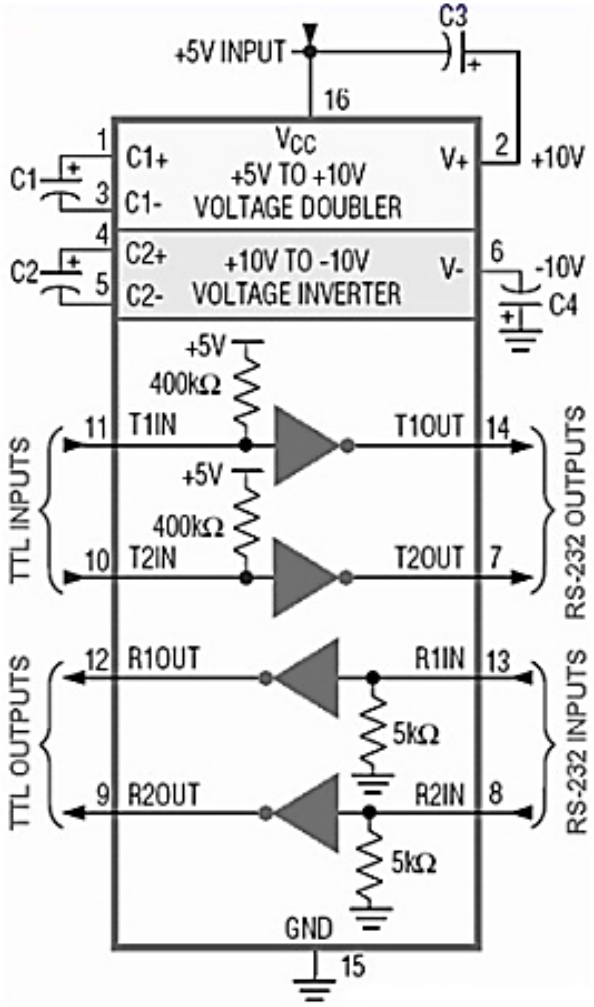
- Like any other UART peripheral, they can handle data sizes of 5 to 8 bits.
- They support 16 bytes receive and transmit FIFOs. Which means that they can store 16-bytes of data in a first in first out fashion without overwriting existing data in the FIFO buffer before it gets filled.
- It has a built-in baud rate generator.
- It supports built-in DMA (Direct Memory Access) for transmission and reception which is ideal when data of byte size has to be transmitted and the controller has to be relieved from basic data communication to perform other tasks.
- It has multi-processor addressing modes and has an IrDA mode to support infrared communication as well.



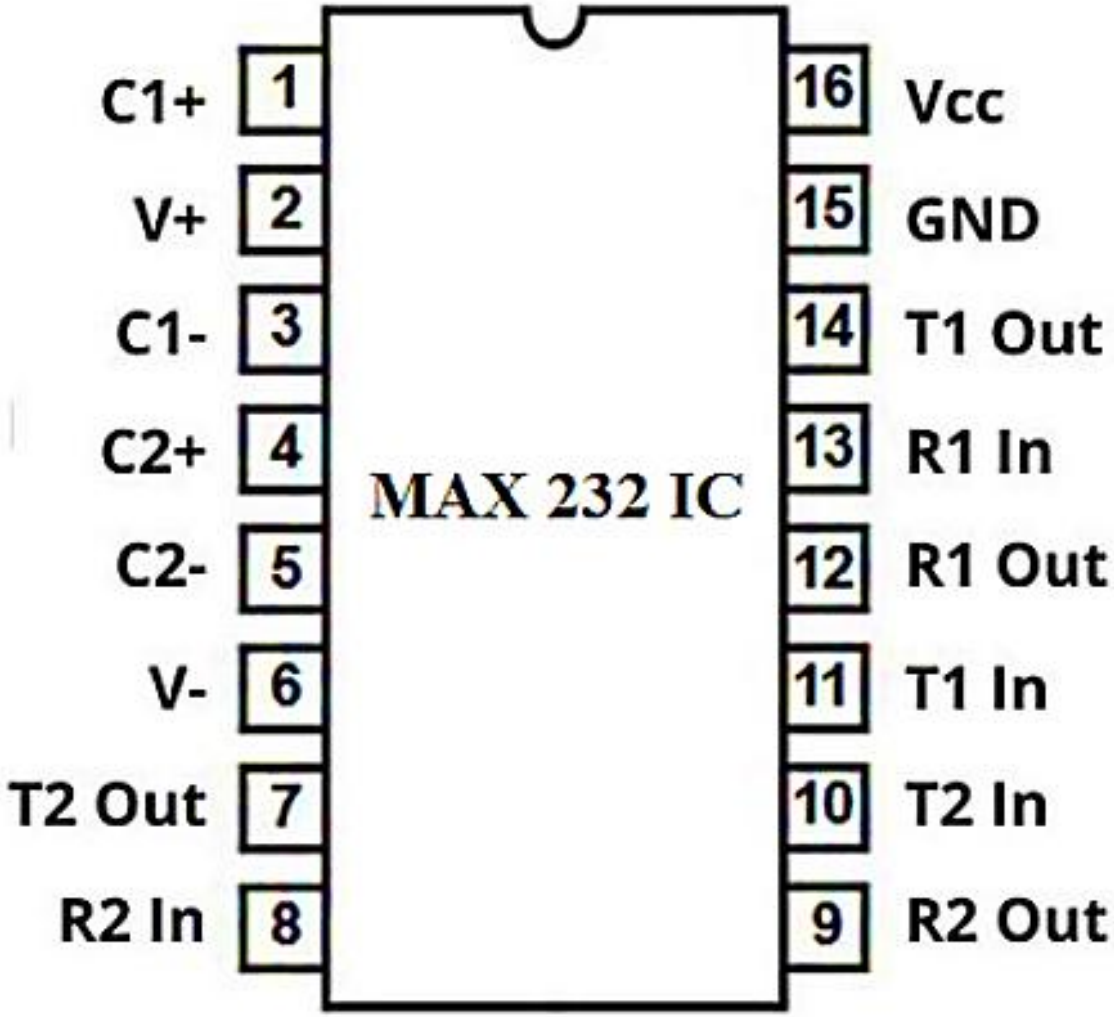
Pin configuration of UART0 module and connection diagram to RS232 level converter

MAX232

- MAX232 is a popular RS232 driver and receiver IC that is often used in embedded systems to interface with RS232 devices.
- The **MAX232 IC** is designed to convert the TTL-level (Transistor-Transistor Logic) signals from an ARM processor's UART module to the RS232 voltage levels needed to communicate with a serial device.
- The MAX232 IC typically operates from a single 5V power supply and has four voltage level converter channels, which can convert the TTL-level signals from the ARM processor's UART module to the higher voltage levels used by RS232 devices.
- The IC also includes capacitors to generate the required charge pumps for the voltage conversion.
- To interface an ARM processor with an RS232 device using the MAX232 IC, the UART module of the processor is typically connected to the input pins of the MAX232 IC. The MAX232 IC's output pins are then connected to the RS232 device's input and output pins, respectively.



MAX232 IC Structure



MAX232 IC Pin Configuration

- Pin-1(C1+): The positive terminal of a capacitor is connected to this pin
- Pin-2(Vs+): The capacitor’s positive leg is connected to it by grounding the negative leg.
- Pin-3(C1-): The negative pin of the capacitor is connected to this pin and the positive pin is connected to pin1
- Pin-4(C2+): The positive terminal of a capacitor is connected to this pin
- Pin-5(C2-): The negative terminal of a capacitor is connected where the positive terminal is connected to Pin4.
- Pin-6(Vs-): The negative terminal of a capacitor is connected to this pin & 5 volts is provided to the positive terminal of the capacitor.
- Pin-7(T2OUT): It provides the converted TTL signal in the form of RS-232. Here TTL signal can be obtained by T2IN Pin from the microcontroller and this pin is connected to Pin2 of DB-9 port of your computer like Rxd.
- Pin-8(R2IN): This Pin gets the signal of RS-232 like an input & provides the changed signal in the form of TTL on the R2OUT pin. This pin is connected to DB9 Port’s Txd pin, which is Pin3.
- Pin-9(R2OUT): It provides the signal changed within TTL form. The signal is received from Pc at R1In Pin. Connect this pin to your module (TTL) Rxd pin which receives the signal.
- Pin-10(T2IN): This pin gets transmitted signal from microcontroller & gives the changed RS-232 signal over T2OUT pin. Here, the signal can be transmitted from the microcontroller serial port’s txd pin. This pin can be connected to your Txd pin of the module.
- Pin-11(T1IN): This pin works like a T2IN.
- Pin-12(R1OUT): This pin works like an R2OUT.
- Pin-13(R1IN): This pin works like an R2IN.
- Pin-14(T1OUT): This pin works like aT2OUT.
- Pin-15(GND): This pin is a GND pin.
- Pin-16(VCC): This pin is a voltage supply pin where 5V is provided to this pin.

Applications of MAX232

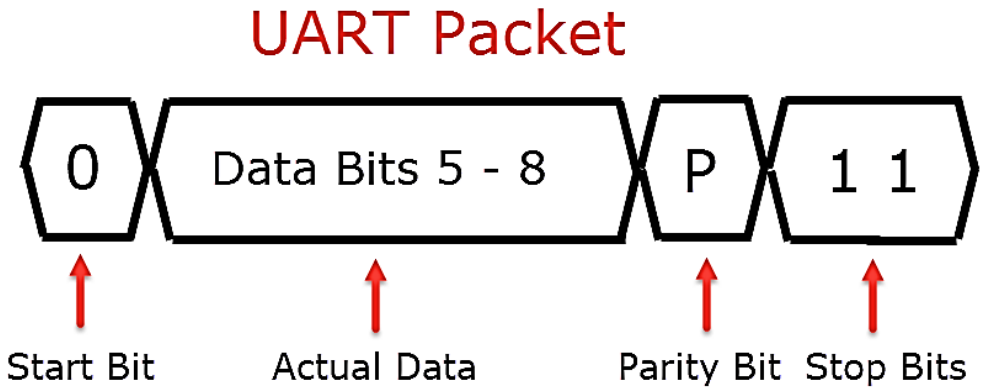
- It is used to interface microcontroller through PC
- Computers
- Used to convert from TTL or CMOS logic to RS232
- Modem.
- Terminals
- Utilized within RS232 cables
- Portable Computing
- Transactions of Interface
- RS232 System Powered with a Battery
- Fewer Power Modes

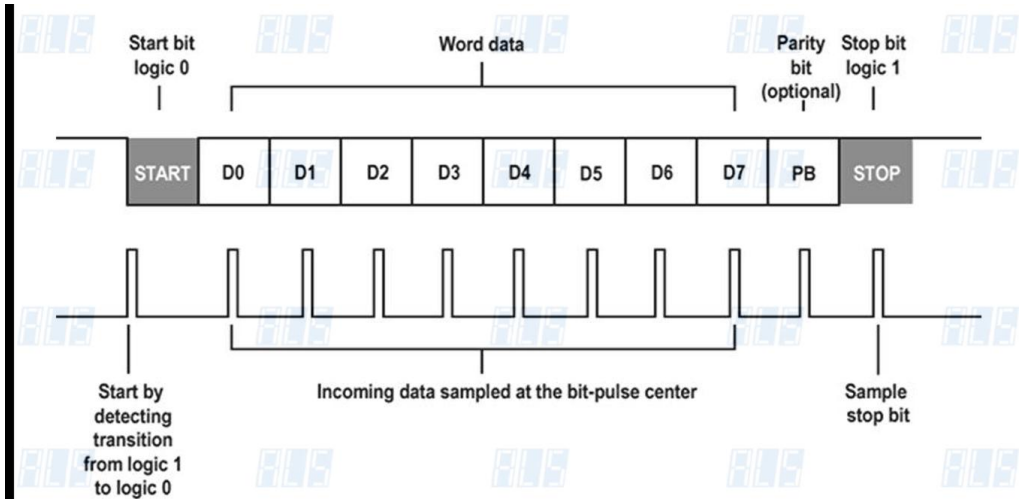
UART

- UART is used for Serial Communication and is one of the commonly used peripherals in LPC1768 MCU, mostly for communicating a computer for transmitting debug messages, receiving commands etc.
- UART is short for Universal Asynchronous Receiver / Transmitter. It is a type of serial communication usually between a Microcontroller and a Computer. UART is a full-duplex communication and hence it needs two wires/lines between the communicating devices. They are called as TX and RX.

The ‘A’ in UART stands for Asynchronous. What this means is that there is no clock signal between the communicating devices (in contrast to I2C and SPI, which are also Serial Communications but use a clock signal to synchronize data).

In UART, the data is transmitted in the form of ‘packets’ or ‘frames’. The structure of a typical UART data packet is shown below.





- At the beginning of a frame, there is START Bit, which is a ‘0’ and it indicates the receiver about the data about to be transmitted.
- The START Bit is followed by the actual data to be transmitted. Its length can be anywhere between 5-bits and 8-bits. After the data, there is a parity bit, which can be used for error checking. This bit is optional.
- Finally, to signify the end of current data transfer, there are STOP Bits, which is usually ‘1’ of length 1 or 2 bits wide. To transmit the next set of data, repeat the process.

UART in LPC1768

Coming to UART in LPC1768 MCU, it consists of four UART peripherals viz.

- UART0
- UART1
- UART2
- UART3

UART0, UART2 and UART3 are identical with basic UART functionality, while UART1 adds full modem control handshaking support.

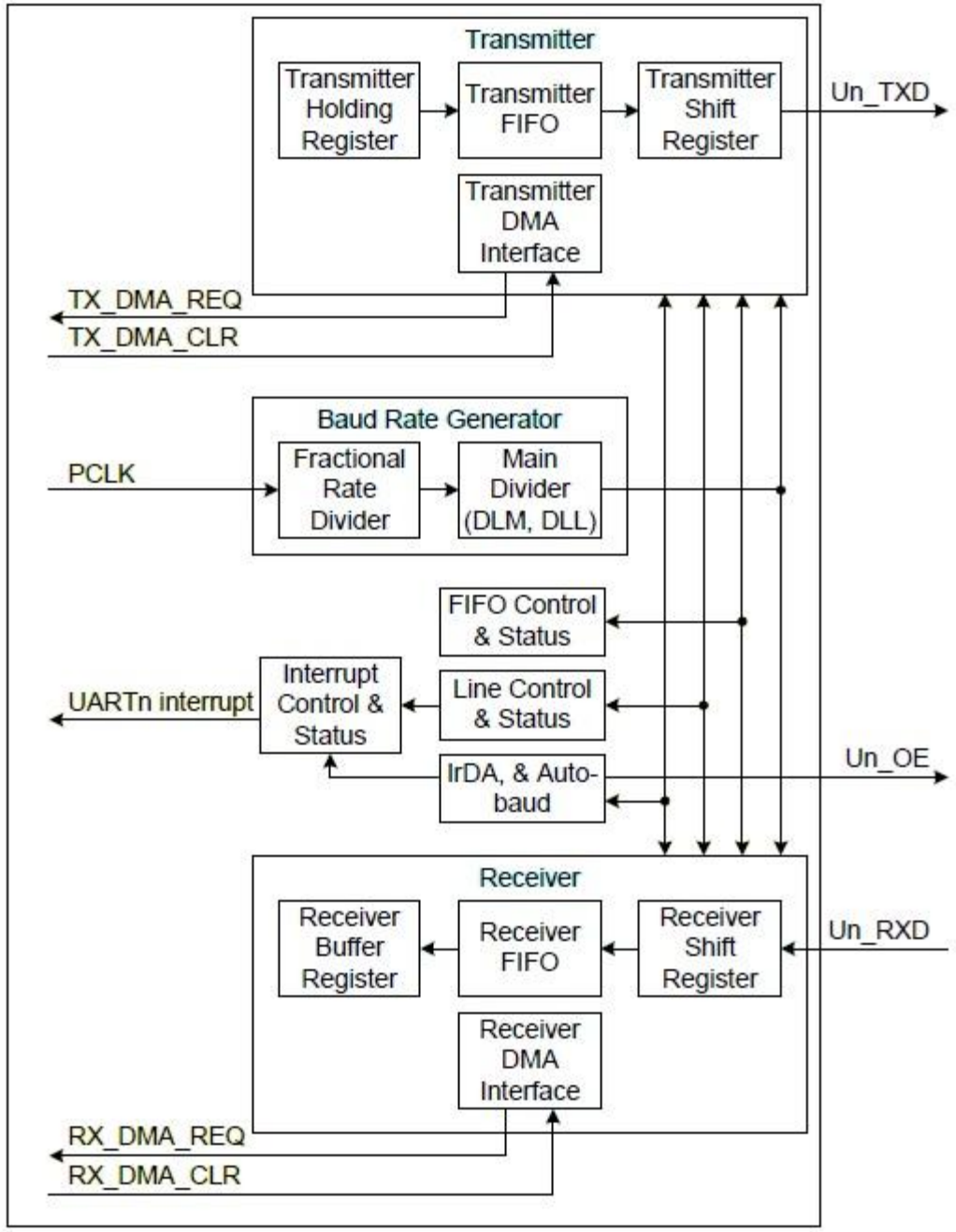
UART1 also supports RS-485/RS-232. The UART0 is used for in-system programming. Both UART0 and UART1 are enabled after reset.

Pins Associated with UART

UART	*Pins Mapped to TXD*	*Pins Mapped to RXD*
UART0	P0.2	P0.3
UART1	P0.15 / P2.0	P0.16 / P2.1
UART2	P0.10 / P2.8	P0.11 / P2.9
UART3	P0.0 / P0.25 / P4.28	P0.1 / P0.26 / P4.29

Architecture of UART in LPC1768

- In case of data transmission, the data to be transmitted is written into the THR Register. This data is then buffered into the TX FIFO.
- A shift register then reads the data from the buffer and transmits is via the corresponding TXD Pin.
- Coming to receiving data, a valid data is captured by the shift register in the receiver block and is transmitted to the RX FIFO.
- Finally, the data is held in the RBR register for the processor to access it.



LPC1768 UART Registers

- UART0, UART2 and UART3 have 14 registers each for UART Data, Interrupt, Status and Control.
- UART1 has a total of 18 registers. Some of the important registers of UART0/2/3.

NOTE: Use the nomenclature from the reference manual for UART Registers. For example, when I say UxRBR, it represents the Receiver Buffer Register of either UART0, UART2 or UART3 i.e. x = 0 or 2 or 3.

- **UxFCR – FIFO Control Register:** Controls the UARTx RX and UARTx TX FIFO operations.
- **UxLCR – Line Control Register:** Determines the data format used in transmission.
- **UxLSR – Line Status Register:** Provides the status of RX and TX blocks.
- Bits 1 to 4 get cleared after reading UxLSR.
- **UxTER – Transmit Enable Register:** Used to enable UART transmission. When TXEN (Bit 7) is set to 1, the TX block will keep on transmitting data as long as it is available. If TXEN is set to 0, then transmission will stop.
- **UxRBR – Receiver Buffer Register:** This register contains the top byte of the RX FIFO i.e the oldest character received data in FIFO. Before reading from UxRBR, the DLAB (Divisor Latch Access) bit in UxLCR register must be 0.
- **UxTHR – Transmit Holding Register:** UxTHR contains the top byte in TX FIFO. It is the newest character in the TX FIFO. The DLAB in UxLCR must be 0 in order to access UxTHR.
- **UxDLL and UxDLM – Divisor Latch Registers (LSB and MSB):** These registers are a part of baud rate generator. Together, they contain the 16-bit divisor value for the APB Clock to generate the desired baud rate. UxDLL contains the lower 8-bits of the divisor while UxDLM contains the higher 8-bits. Before accessing these registers, the DLAB bit in UxLCR must be 1.
- **UxFDR – Fractional Divider Register:** It controls the clock pre-scalar value for baud rate generation. The APB Clock is divided into a desired clock based of the divider and multiplier values in this register.

Register	Description
RBR	Contains the recently received Data
THR	Contains the data to be transmitted
FCR	FIFO Control Register
LCR	Controls the UART frame formatting(Number of Data Bits, Stop bits)
DLL	Least Significant Byte of the UART baud rate generator value.
DLM	Most Significant Byte of the UART baud rate generator value.

- **UxFCR – FIFO Control Register:** Controls the UARTx RX and UARTx TX FIFO operations. LPC1768 has inbuilt 16byte FIFO for Receiver/Transmitter. Thus it can store 16-bytes of data received on UART without overwriting. If the data is not read before the Queue(FIFO) is filled then the new data will be lost and the OVERRUN error bit will be set.

FCR						
31:8	7:6	5:4	3	2	1	0
RESERVED	RX TRIGGER	RESERVED	DMA MODE	TX FIFO RESET	RX FIFO RESET	FIFO ENABLE

Bit 0 – FIFO:
This bit is used to enable/disable the FIFO for the data received/transmitted.
0--FIFO is Disabled.
1--FIFO is Enabled for both Rx and Tx.

Bit 1 – RX_FIFO:
This is used to clear the 16-byte Rx FIFO.
0--No impact.
1--CLears the 16-byte Rx FIFO and the resets the FIFO pointer.

Bit 2 – Tx_FIFO:
This is used to clear the 16-byte Tx FIFO.
0--No impact.
1--Clears the 16-byte Tx FIFO and the resets the FIFO pointer.

Bit 3 – DMA_MODE:
This is used for Enabling/Disabling DMA mode.
0--Disables the DMA.
1--Enables DMA only when the FIFO(bit-0) bit is SET.

Bit 7:6 – Rx_TRIGGER:
This bit is used to select the number of bytes of the receiver data to be written so as to enable the interrupt/DMA.
00-- Trigger level 0 (1 character or 0x01)
01-- Trigger level 1 (4 characters or 0x04)
10-- Trigger level 2 (8 characters or 0x08)
11-- Trigger level 3 (14 characters or 0x0E)

UxLCR – Line Control Register: Determines the data format used in transmission.

LCR						
31:8	7	6	5:4	3	2	1:0
Reserved	DLAB	Break COntrol	Parity Select	Parity Enable	Stop Bit Select	Word Length Select

Bit 1:0 – WLS : WordLenghtSelect
These two bits are used to select the character length
00-- 5-bit character length
01-- 6-bit character length
10-- 7-bit character length
11-- 8-bit character length

Bit 2 – Stop Bit Selection:
This bit is used to select the number(1/2) of stop bits
0-- 1 Stop bit
1-- 2 Stop Bits

Bit 3 – Parity Enable:
This bit is used to Enable or Disable the Parity generation and checking.
0-- Disable parity generation and checking.
1-- Enable parity generation and checking.

Bit 5:4 – Parity Selection:
These two bits will be used to select the type of parity.
00-- Odd parity. Number of 1s in the transmitted character and the attached parity bit will be odd.
01-- Even Parity. Number of 1s in the transmitted character and the attached parity bit will be even.
10-- Forced "1" stick parity.
11-- Forced "0" stick parity

Bit 6 – Break Control
0-- Disable break transmission.
1-- Enable break transmission. Output pin UARTn TXD is forced to logic 0

Bit 8 – DLAB: Divisor Latch Access Bit
This bit is used to enable the access to divisor latch.
0-- Disable access to divisor latch
0-- Enable access to divisor latch

LSR (Line Status Register): The is a read-only register that provides status information of the UART TX and RX blocks.

LSR								
31:8	7	6	5	4	3	2	1	0
Reserved	RXFE	TEMT	THRE	BI	FE	PE	OE	RDR

Bit 0 – RDR: Receive Data Ready
This bit will be set when there is a received data in RBR register. This bit will be automatically cleared when RBR is empty.
0-- The UARTn receiver FIFO is empty.
1-- The UARTn receiver FIFO is not empty.

Bit 1 – OE: Overrun Error
The overrun error condition is set when the UART Rx FIFO is full and a new character is received. In this case, the UARTn RBR FIFO will not be overwritten and the character in the UARTn RSR will be lost.
0-- No overrun
1-- Buffer over run

Bit 2 – PE: Parity Error
This bit is set when the receiver detects a error in the Parity.
0-- No Parity Error
1-- Parity Error

Bit 3 – FE: Framing Error
This bit is set when there is error in the STOP bit(LOGIC 0)
0-- No Framing Error
1-- Framing Error

Bit 4 – BI: Break Interrupt

This bit is set when the RXDn is held in the spacing state (all zeroes) for one full character transmission

0-- No Break interrupt

1-- Break Interrupt detected.

Bit 5 – THRE: Transmitter Holding Register Empty

THRE is set immediately upon detection of an empty THR. It is automatically cleared when the THR is written.

0-- THR register is Empty

1-- THR has valid data to be transmitted

Bit 6 – TEMT: Transmitter Empty

TEMT is set when both UnTHR and UnTSR are empty; TEMT is cleared when any of them contain valid data.

0-- THR and/or the TSR contains valid data.

1-- THR and the TSR are empty.

Bit 7 – RXFE: Error in Rx FIFO

This bit is set when the received data is affected by Framing Error/Parity Error/Break Error.

0-- RBR contains no UARTn RX errors.

1-- RBR contains at least one RX error.

TER (Transmitter Enable register): This register is used to Enable/Disable the transmission

TER		
31:8	7	6-0
Reserved	TXEN	Reserved

Bit 7 – TXEN: Trsnamitter Enable
When this bit is 1, the data written to the THR is output on the TXD pin.
If this bit is cleared to 0 while a character is being sent, the transmission of that character is completed, but no further characters are sent until this bit is set again.
In other words, a 0 in this bit blocks the transfer of characters.

- Note: By default this bit will be set after Reset.

UxTER – Transmit Enable Register: Used to enable UART transmission. When TXEN (Bit 7) is set to 1, the TX block will keep on transmitting data as long as it is available. If TXEN is set to 0, then transmission will stop.

UxRBR – Receiver Buffer Register: This register contains the top byte of the RX FIFO i.e the oldest character received data in FIFO. Before reading from UxRBR, the DLAB (Divisor Latch Access) bit in UxLCR register must be 0.

UxTHR – Transmit Holding Register: UxTHR contains the top byte in TX FIFO. It is the newest character in the TX FIFO. The DLAB in UxLCR must be 0 in order to access UxTHR.

UxDLL and UxDLM – Divisor Latch Registers (LSB and MSB): These registers are a part of baud rate generator. Together, they contain the 16-bit divisor value for the APB Clock to generate the desired baud rate. UxDLL contains the lower 8-bits of the divisor while UxDLM contains the higher 8-bits. Before accessing these registers, the DLAB bit in UxLCR must be 1.

UxFDR – Fractional Divider Register: It controls the clock pre-scalar value for baud rate generation. The APB Clock is divided into a desired clock based of the divider and multiplier values in this register.

Bits [3:0]	DIVADDVAL	Pre-scalar divisor value.
Bits [7:4]	MULVAL	Pre-scalar multiplier value. Minimum value is 1.

Baud Rate Calculation

As per the reference manual of LPC1768 MCU, the following formula can be used to calculate the baud rate for UART0/2/3 peripherals.

$$UARTx_{BAUD} = \frac{PCLK}{16 \times (256 \times UxDLM + UxDLL) + (1 + \frac{DIVADDVAL}{MULVAL})}$$

This equation can be rearranged as follows:

$$UARTx_{BAUD} = \frac{MULVAL}{DIVADDVAL + MULVAL} \times \frac{PCLK}{16 \times (256 \times UxDLM + UxDLL)}$$

Here,

- PCLK is the Peripheral Clock in Hz
- MULVAL and DIVADDVAL are part of UxFDR register to set the clock pre-scalar
- UxDLL and UxDLM are baud rate divider values.

When choosing MULVAL and DIVADDVAL value, you must comply to the following rules:

- $1 \leq MULVAL \leq 15$
- $0 \leq DIVADDVAL \leq 14$
- $DIVADDVAL < MULVAL$

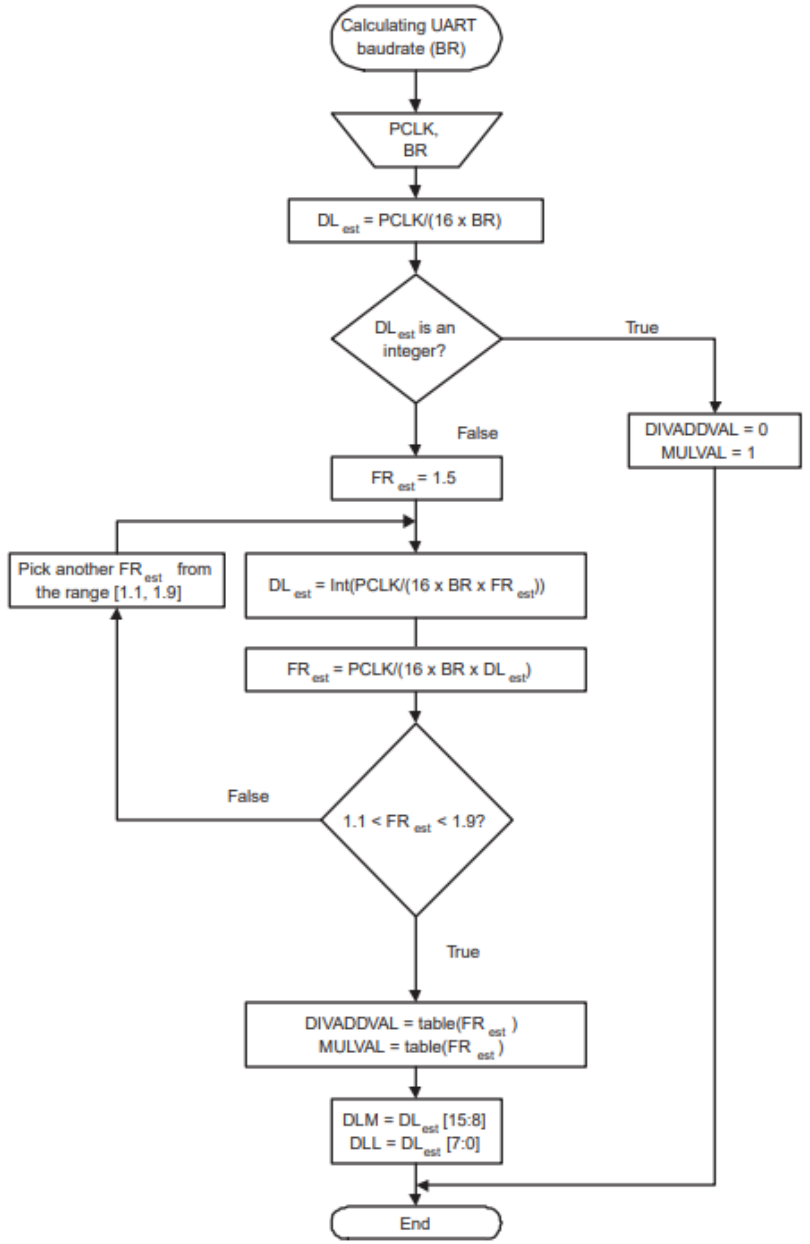


Fig 46. Algorithm for setting UART dividers

Table 286. Fractional Divider setting look-up table

FR	DivAddVal/ MulVal	FR	DivAddVal/ MulVal	FR	DivAddVal/ MulVal	FR	DivAddVal/ MulVal
1.000	0/1	1.250	1/4	1.500	1/2	1.750	3/4
1.067	1/15	1.267	4/15	1.533	8/15	1.769	10/13
1.071	1/14	1.273	3/11	1.538	7/13	1.778	7/9
1.077	1/13	1.286	2/7	1.545	6/11	1.786	11/14
1.083	1/12	1.300	3/10	1.556	5/9	1.800	4/5
1.091	1/11	1.308	4/13	1.571	4/7	1.818	9/11
1.100	1/10	1.333	1/3	1.583	7/12	1.833	5/6
1.111	1/9	1.357	5/14	1.600	3/5	1.846	11/13
1.125	1/8	1.364	4/11	1.615	8/13	1.857	6/7
1.133	2/15	1.375	3/8	1.625	5/8	1.867	13/15
1.143	1/7	1.385	5/13	1.636	7/11	1.875	7/8
1.154	2/13	1.400	2/5	1.643	9/14	1.889	8/9
1.167	1/6	1.417	5/12	1.667	2/3	1.900	9/10
1.182	2/11	1.429	3/7	1.692	9/13	1.909	10/11
1.200	1/5	1.444	4/9	1.700	7/10	1.917	11/12
1.214	3/14	1.455	5/11	1.714	5/7	1.923	12/13
1.222	2/9	1.462	6/13	1.727	8/11	1.929	13/14
1.231	3/13	1.467	7/15	1.733	11/15	1.933	14/15

14.4.12.1.1 Example 1: PCLK = 14.7456 MHz, BR = 9600

According to the provided algorithm DL_{est} = PCLK/(16 x BR) = 14.7456 MHz / (16 x 9600) = 96. Since this DL_{est} is an integer number, DIVADDVAL = 0, MULVAL = 1, DLM = 0, and DLL = 96.

14.4.12.1.2 Example 2: PCLK = 12 MHz, BR = 115200

According to the provided algorithm DL_{est} = PCLK/(16 x BR) = 12 MHz / (16 x 115200) = 6.51. This DL_{est} is not an integer number and the next step is to estimate the FR parameter. Using an initial estimate of FR_{est} = 1.5 a new DL_{est} = 4 is calculated and FR_{est} is recalculated as FR_{est} = 1.628. Since FRest = 1.628 is within the specified range of 1.1 and 1.9, DIVADDVAL and MULVAL values can be obtained from the attached look-up table.

The closest value for FRest = 1.628 in the look-up Table 286 is FR = 1.625. It is equivalent to DIVADDVAL = 5 and MULVAL = 8.

Based on these findings, the suggested UART setup would be: DLM = 0, DLL = 4, DIVADDVAL = 5, and MULVAL = 8. According to Equation 4 the UART rate is 115384. This rate has a relative error of 0.16% from the originally specified 115200.

Steps for Configuring UART0

- Below are the steps for configuring the UART0.

Step1: Configure the GPIO pin for UART0 function using PINSEL register.

Step2: Configure the FCR for enabling the FIXO and Reste both the Rx/Tx FIFO.

Step3: Configure LCR for 8-data bits, 1 Stop bit, Disable Parity and Enable DLAB.

Step4: Get the PCLK from PCLKSELx register 7-6 bits.

Step5: Calculate the DLM,DLL vaues for required baudrate from PCLK.

Step6: Updtae the DLM,DLL with the calculated values.

Step6: Finally clear DLAB to disable the access to DLM,DLL.

Example for UART initialization

```
void UART0_Init(void)
{
        LPC_SC->PCONP |= 0x00000008;          //UART0 peripheral enable
        LPC_PINCON->PINSEL0 &= ~0x000000F0;
        LPC_PINCON->PINSEL0 |= 0x00000050;
        LPC_UART0->LCR = 0x00000083;          //enable divisor latch, parity disable, 1 stop
bit, 8bit word length
        LPC_UART0->DLM = 0X00;
        LPC_UART0->DLL = 0x1A;       //select baud rate 9600 bps
        LPC_UART0->LCR = 0X00000003;
        LPC_UART0->FCR = 0x07;
        LPC_UART0->IER = 0X03;                          //select Transmit and receive
interrupt
        NVIC_EnableIRQ(UART0_IRQn);                     //Assigning channel
}
```

```
#include<LPC17xx.h>
void delay(unsigned int r1);
void UART0_Init(void);

unsigned int  i ;
unsigned char *ptr, arr[] = "Hello world\r\n";
#define THR_EMPTY 0x20

                                        //U0THR is Empty
int main(void)
{
        UART0_Init();
        while(1)
        {
                ptr = arr;
                while ( *ptr != '\0')
                {
                        while ((LPC_UART0->LSR &
THR_EMPTY) != THR_EMPTY) ; //Is U0THR is EMPTY??
                        LPC_UART0->THR = *ptr++;

                }
                for(i=0;i<=80000;i++);
        }
}
```

```
void UART0_Init(void)
{
        LPC_SC->PCONP |= 0x00000008;
        //UART0 peripheral enable
        LPC_PINCON->PINSEL0 = 0x00000050;
        LPC_UART0->LCR = 0x00000083;
        //enable divisor latch, parity disable, 1 stop bit, 8bit
word length
        LPC_UART0->DLM = 0X00;
        LPC_UART0->DLL = 0x1A;
        //select baud rate 9600 bps for 4Mhz
        LPC_UART0->LCR = 0X00000003;
        //Disable divisor latch
        LPC_UART0->FCR = 0x07;
                        //FIFO enable,RX FIFO reset,TX
FIFO reset
}
```

Data Acquisition System

- **Data acquisition** is the process of sampling signals that measure real-world physical conditions and converting the resulting samples into digital numeric values that can be manipulated by a computer.
- Data acquisition systems, abbreviated by the acronyms *DAS*, *DAQ*, or *DAU*, typically convert analog waveforms into digital values for processing.

The components of data acquisition systems include:

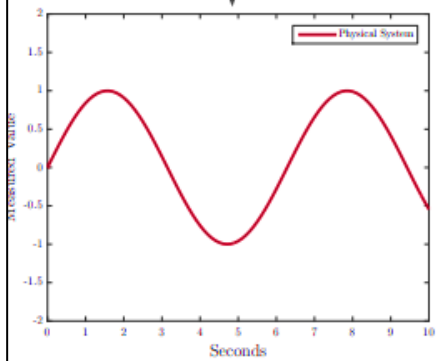
- Sensors, to convert physical parameters to electrical signals.
- Signal conditioning circuitry, to convert sensor signals into a form that can be converted to digital values.
- Analog-to-digital converters, to convert conditioned sensor signals to digital values.

Data Acquisition System (Contd.)

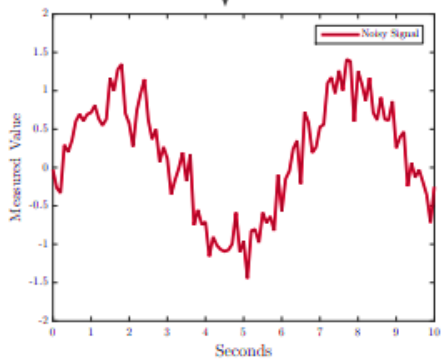
- Data acquisition applications are usually controlled by software programs developed using various general purpose programming languages such as Assembly, BASIC, C, C++, C#, Fortran, Java, LabVIEW, Lisp, Pascal, etc.
- Stand-alone data acquisition systems are often called data loggers.
- There are also open-source software packages providing all the necessary tools to acquire data from different, typically specific, hardware equipment.
- These tools come from the scientific community where complex experiment requires fast, flexible, and adaptable software.
- Those packages are usually custom-fit but more general DAQ packages like the Maximum Integrated Data Acquisition System can be easily tailored and are used in several physics experiments.

Digital Data Acquisition System

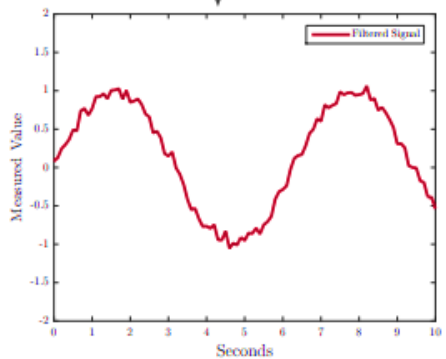
Physical System → **Transducer Sensor** → **Signal Conditioning** → **Analog - Digital Converter** → **Computer**



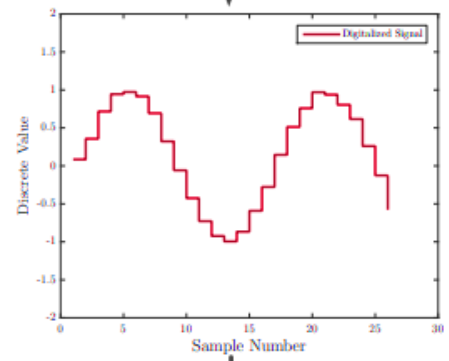
Physical Signal

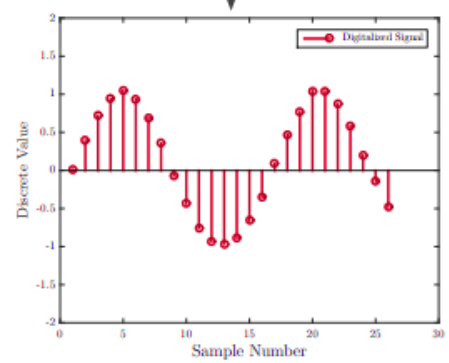


Noisy Electrical Signal



Conditioned Signal





Digitalized Signal

8 Bit Code

0	1	1	1
0	0	0	1
0	1	1	1
1	0	1	1
1	1	0	0
1	1	1	1
0	1	1	0
1	0	1	0
1	1	1	0
1	1	1	1
0	1	1	1
1	1	1	0
1	1	1	1
0	0	1	1
1	1	0	0
0	1	1	1
0	1	1	1
0	1	1	0
0	1	1	0
1	0	1	1
1	1	1	0
0	0	0	1
1	1	1	1
1	0	1	1
0	0	1	0
0	0	1	1
1	1	1	1
1	0	0	1
1	1	0	1
1	1	1	0

Sources and systems- Digital Data Acquisition Systems

- Data acquisition begins with the physical phenomenon or physical property to be measured. Examples of this include temperature, vibration, light intensity, gas pressure, fluid flow, and force. Regardless of the type of physical property to be measured, the physical state that is to be measured must first be transformed into a unified form that can be sampled by a data acquisition system. The task of performing such transformations falls on devices called sensors.
- A data acquisition system is a collection of software and hardware that allows one to measure or control the physical characteristics of something in the real world. A complete data acquisition system consists of DAQ hardware, sensors and actuators, signal conditioning hardware, and a computer running DAQ software. If timing is necessary (such as for event mode DAQ systems), a separate compensated distributed timing system is required.
- A sensor, which is a type of transducer, is a device that converts a physical property into a corresponding electrical signal (e.g., strain gauge, thermistor). An acquisition system to measure different properties depends on the sensors that are suited to detect those properties. Signal conditioning may be necessary if the signal from the transducer is not suitable for the DAQ hardware being used.
- The signal may need to be filtered, shaped, or amplified in most cases. Various other examples of signal conditioning might be bridge completion, providing current or voltage excitation to the sensor, isolation, and linearization. For transmission purposes, single ended analog signals, which are more susceptible to noise can be converted to differential signals. Once digitized, the signal can be encoded to reduce and correct transmission errors.

DAQ hardware

- DAQ hardware is what usually interfaces between the signal and a PC. It could be in the form of modules that can be connected to the computer's ports (parallel, serial, USB, etc.) or cards connected to slots (S-100 bus, AppleBus, ISA, MCA, PCI, PCI-E, etc.) in a PC motherboard or in a modular crate (CAMAC, NIM, VME). Sometimes adapters are needed, in which case an external breakout box can be used.
- DAQ cards often contain multiple components (multiplexer, ADC, DAC, TTL-IO, high-speed timers, RAM). These are accessible via a bus by a microcontroller, which can run small programs. A controller is more flexible than a hard-wired logic, yet cheaper than a CPU so it is permissible to block it with simple polling loops. For example: Waiting for a trigger, starting the ADC, looking up the time, waiting for the ADC to finish, move value to RAM, switch multiplexer, get TTL input, let DAC proceed with voltage ramp.
- Today, signals from some sensors and Data Acquisition Systems can be streamed via Bluetooth.

DAQ device drivers

DAQ device drivers are needed for the DAQ hardware to work with a PC. The device driver performs low-level register writes and reads on the hardware while exposing API for developing user applications in a variety of programs.

Input devices

- 3D scanner
- Analog-to-digital converter
- Time-to-digital converter

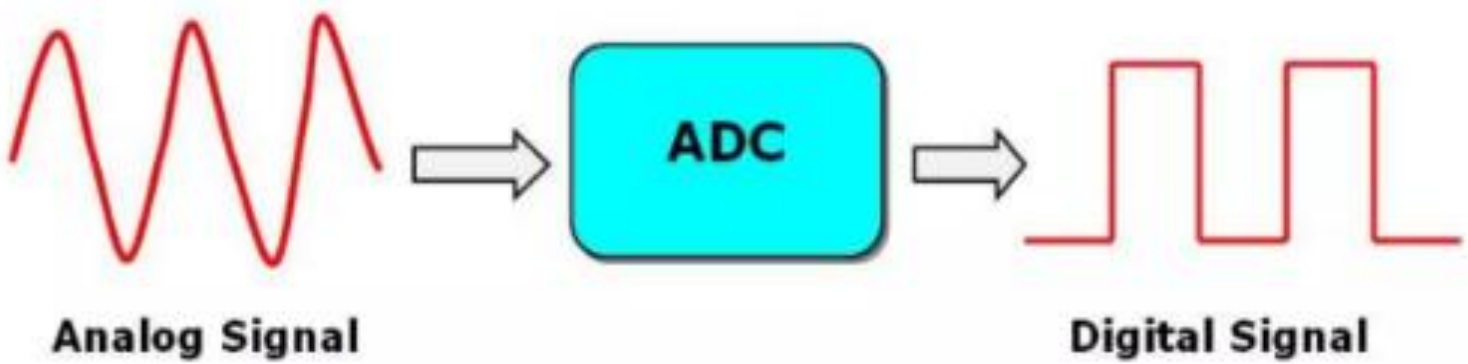
- **Hardware**
  - Computer Automated Measurement and Control (CAMAC)
  - Industrial Ethernet
  - Industrial USB
  - LAN eXtensions for Instrumentation
  - Network interface controller
  - PCI eXtensions for Instrumentation
  - VMEbus
  - VXI

DAQ software

Specialized DAQ software may be delivered with the DAQ hardware. Software tools used for building large-scale data acquisition systems include EPICS. Other programming environments that are used to build DAQ applications include ladder logic, Visual C++, Visual Basic, LabVIEW, and MATLAB.

What is ADC ?

- Analog to Digital Conversion(i.e. ADC) , as the name suggests , is all about converting a given analog signal into its digital form or say a digital value.
- This conversion or measurement happens in presence of a fixed and accurate reference voltage.
- The analog signal is compared to this reference voltage and then estimations are made to get the final measured value.



Resolution

- ADC is used to convert analog signal/voltage into its equivalent digital number so that microcontroller can process that numbers and make it human readable.
- The ADC characterized by resolution. The resolution of ADC indicates the number of digital values.
- In LPC2148 microcontroller we have in-built 10-bit ADC. So for 10-bit ADC resolution is 10-bit and maximum value will be 2^{10}=1024. This means our digital value or discrete level lies between 0 to 1023.
- There is one more term important to understand while dealing with ADC and it is step size. Step size is the minimum change in input voltage which can be resolved by ADC. The concept of step size is closely associated with the resolution of ADC.

$$\text{Step Size} = \frac{Input\ Range}{Resolution} = \frac{3.3V}{2^{10}}$$

$$= \frac{3.3}{1023} = 0.0032258064\,V$$

$$= 3.23\,\text{mV (Approx.)}$$

- The digital output of an ADC is given by

$$D = \frac{V_i}{V_r} \times 2^n$$

D: digital output value
V_i: input voltage
V_r: reference voltage
n: the size of the ADC (in bit)

For example, if our ADC is 8-bit size, the reference voltage is given by 3.3V, the digital value for an analog voltage of 2V is b10011011.

While if our ADC is 10-bit size, the digital output would be b1001101100

ADC in ARM7 Microcontroller

- The ADC in ARM7 Microcontroller is 10-bit successive approximation analog to digital converter.
- ARM7 has two inbuilt ADC Modules, named as ADC0 & ADC1.
- ADC0 has 6-Channels (AD0.1-AD0.6).
- ADC1 has 8-Channels (AD1.0-AD1.7).
- ADC operating frequency is 4.5 MHz (max.), operating frequency decides the conversion time.
- Supports power down mode.
- Burst conversion mode for single or multiple inputs.
- There are several registers associated with ADC feature but we will mainly discussing about ADC Control Register **(ADCR)** & ADC Global Data Register **(ADGDR)**.

LPC1768 ADC Block

- LPC1768 has an inbuilt 12 bit Successive Approximation ADC which is multiplexed among 8 input pins.
  The ADC reference voltage is measured across VREFN to VREFP, meaning it can do the conversion within this range.
- Usually the VREFP is connected to VDD and VREFN is connected to GND.
- As LPC1768 works on 3.3 volts, this will be the ADC reference voltage.
- Now the resolution of ADC = 3.3/(2^{12}) = 3.3/4096 =0.000805 = 0.8mV

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.

The below block diagram 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

LPC1768 ADC Registers

Register	*Function*	*Description*
ADCR	A/D Control Register.	Used to Start ADC, Select Operating Mode, Channel Selection, Set Clock Division.
ADGDR	A/D Global Data Register.	Contains the Data of the most recent A/D Conversion
ADINTEN	A/D Interrupt Enable Register.	Used to enable interrupt when a channel completes conversion.
ADDR0 – ADDR7	A/D Channel Data Register.	These registers hold the result of the recent conversion in that respective channel.
ADSTAT	A/D Status Register.	Holds the status of all ADC Channels.
ADTRM	A/D Trim Register.	Contains Trim values for ADC an DAC.

ADCR								
31:28	27	26:24	23:22	21	20:17	16	15:8	7:0
Reserved	EDGE	START	Reserved	PDN	Reserved	BURST	CLCKDIV	SEL

Bit 7:0 – SEL : Channel Select
These bits are used to select a particular channel for ADC conversion. One bit is allotted for each channel. Setting the Bit-0 will make the ADC to sample AD0[0] for conversion. Similary setting bit-7 will do the conversion for AD0[7].

Bit 15:8 – CLCKDIV : Clock Divisor
The APB clock (PCLK_ADC0) is divided by (this value plus one) to produce the clock for the A/D converter, which should be less than or equal to 13 MHz.

Bit 16 – BURST
This bit is used for BURST conversion. If this bit is set the ADC module will do the conversion for all the channels that are selected(SET) in SEL bits.
CLearing this bit will disable the BURST conversion.

Bit 21 – PDN : Power Down Mode
Setting this bit brings ADC out of power down mode and makes it operational.
Clearing this bit will power down the ADC.

Bit 24:26 – START
When the BURST bit is 0, these bits control whether and when an A/D conversion is started:
000 - Conversion Stopped
001- Start Conversion Now

The remaining cases (010 to 111) are about starting conversion on occurrence of edge on a particular CAP or MAT pin.

Bit 27 - EDGE
This bit is significant only when the START field contains 010-111. It starts conversion on selected CAP or MAT input.
0 - On Falling Edge
1 - On Rising Edge

ADGDR					
31	27	26:24	23:16	15:4	3:0
DONE	OVERRUN	CHN	Reserved	RESULT	Reserved

- **Bit 15:4 - RESULT**
  This field contains the 12bit A/D conversion value for the selected channel in **ADCR.SEL**
  The vale for this register should be read oncve the conversion is completed ie DONE bit is set.
- **Bit 26:24 - CHN : Channel**
  These bits contain the channel number for which the A/D conversion is done and the converted value is available in RESULT bits(e.g. 000 identifies channel 0, 011 channel 3…).
- **Bit 27 - OVERRUN**
  This bit is set during the BURST mode where the previous conversion data is overwritten by the new A/D conversion value.
- **Bit 31 - DONE**
  This bit is set to 1 when an A/D conversion completes. It is cleared when this register is read and when the ADCR is written. If the ADCR is written while a conversion is still in progress, this bit is set and a new conversion is started.

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

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.

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.

Setting up and configuring ADC Module for software controlled mode :

```
#define CLKDIV 3 // 4Mhz ADC clock (ADC_CLOCK=PCLK/CLKDIV) where "CLKDIV-1" is actually used , in our case PCLK=12mhz
#define BURST_MODE_OFF (0<<16) // 1 for on and 0 for off
#define PowerUP (1<<21) //setting it to 0 will power it down
#define START_NOW ((0<<26)|(0<<25)|(1<<24)) //001 for starting the conversion immediately
#define ADC_DONE (1<<31)
```

Initial step:

```
unsigned long AD0CR_setup = (CLKDIV<<8) | BURST_MODE_OFF | PowerUP;
AD0CR = AD0CR_setup | SEL_AD06; AD0CR |= START_NOW;
```

Fetching the conversion result:

```
while( (AD0DR6 & ADC_DONE) == 0 ); //this loop will terminate when bit 31 of AD0DR6 changes to 1
result = (AD0DR6>>6) & 0x3FF;
```

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 AD0CR. 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.
- #define CLKDIV 3 *// 4Mhz ADC clock (ADC_CLOCK=PCLK/CLKDIV) where "CLKDIV-1" is actually used , in our case PCLK=12mhz*
- #define BURST_MODE_ON (1<<16) *// 1 for on and 0 for off*
- #define CLKS_10bit ((0<<19)|(0<<18)|(0<<17)) *//10 bit resolution*
- #define PowerUP (1<<21) *//setting it to 0 will power it down*

Initial step:

```
unsigned int AD0CR_setup = (CLKDIV<<8) | BURST_MODE_ON | CLKS_10bit |
PowerUP; AD0CR = AD0CR_setup | SEL_AD06 | SEL_AD07;
```

Fetching the conversion result:

In Burst mode we use an ISR which triggers at the completion of a conversion in any one of the channel.

Analog to Digital Convertor Program

```
#include<LPC17xx.h>
#include "lcd.h"
#include<stdio.h>
#defineRef_Vtg                3.300
#defineFull_Scale      0xFFF                          //12 bit ADC

int main(void)
{
        unsigned int adc_temp;
        unsigned int i;
        float in_vtg;
        unsigned char vtg[7],dval[7], blank[]="   ";
        unsigned char Msg3[11] = {"MIT:"};
        unsigned char Msg4[12] = {"Dept of IT:"};

  lcd_init();
  LPC_PINCON->PINSEL3 |= 0xC0000000;                                        //P1.31 as
AD0.5
         LPC_SC->PCONP |= (1<<12);

                //enable the peripheral ADC

        temp1 = 0x80;
        lcd_com();
        delay_lcd(800);
        lcd_puts(&Msg3[0]);

        temp1 = 0xC0;
        lcd_com();
        delay_lcd(800);
        lcd_puts(&Msg4[0]);

        while(1)
        {
                LPC_ADC->ADCR = (1<<5)|(1<<21)|(1<<24);

        //0x01200001;//ADC0.5, start conversion and operational
                for(i=0;i<2000;i++);

                                //delay for conversion
                while((adc_temp = LPC_ADC->ADGDR) == 0x80000000);
                                                        //wait till 'done' bit is 1, indicates
conversion complete
                adc_temp = LPC_ADC->ADGDR;
                adc_temp >>= 4;
                adc_temp &= 0x00000FFF;

                                //12 bit ADC
                in_vtg = (((float)adc_temp * (float)Ref_Vtg))/((float)Full_Scale);
        //calculating input analog voltage
                sprintf(vtg,"%3.2fV",in_vtg);

        //convert the readings into string to display on LCD
                sprintf(dval,"%x",adc_temp);
                for(i=0;i<2000;i++);

                temp1 = 0x8A;
                lcd_com();
                delay_lcd(800);
                lcd_puts(&vtg[0]);

                temp1 = 0xCB;
                lcd_com();
                lcd_puts(&blank[0]);

                temp1 = 0xCB;
                lcd_com();
                delay_lcd(800);
                lcd_puts(&dval[0]);

                for(i=0;i<200000;i++);
                for(i=0;i<7;i++)
                vtg[i] = dval[i] = 0x00;
                adc_temp = 0;
                in_vtg = 0;
        }
}
```

Simple program

```
unsigned int  adc_data()
{
  unsigned int adcdata;
  while(!(AD0GDR & 0x80000000)); // Check end of conversion (Done bit) and
     read result
     adcdata=AD0GDR;
  return ((adcdata >> 6) & 0x3ff) ; // Return 10 bit result
}
int main(void)
{
  unsigned int  adc;
  IODIR1=0XFF<<16; // for LCD
  PINSEL1 = 1<<27|1<<28|1<<29 ; // enable P0.27 for AD0.0, P0.28 for AD0.1,
     P0.29 for AD0.2,
     lcd_init(); //initialize LCD
  while(1)
  {
```

```
AD0CR = 0x01200301 ; // Select AD0.0, Select clock for ADC, Start of
    conversion
  adc = adc_data();
     sprintf(adcreading,"%d",adc);  // read data in decimal format
  string(adcreading); // display result on LCD

 AD0CR = 0x01200302 ; // Select AD0.1, Select clock for ADC, Start of
    conversion
      adc = ADC_GetAdcReading();
         sprintf(adcreading,"%d",adc);  // read data in decimal format
      string(adcreading); // display result on LCD

            AD0CR = 0x01200304 ; // Select AD0.2, Select clock for ADC, Start of
    conversion
         adc = ADC_GetAdcReading();
            sprintf(adcreading,"%d",adc);  // read data in decimal format
         string(adcreading); // display result on LCD
}}
```

DAC Programming

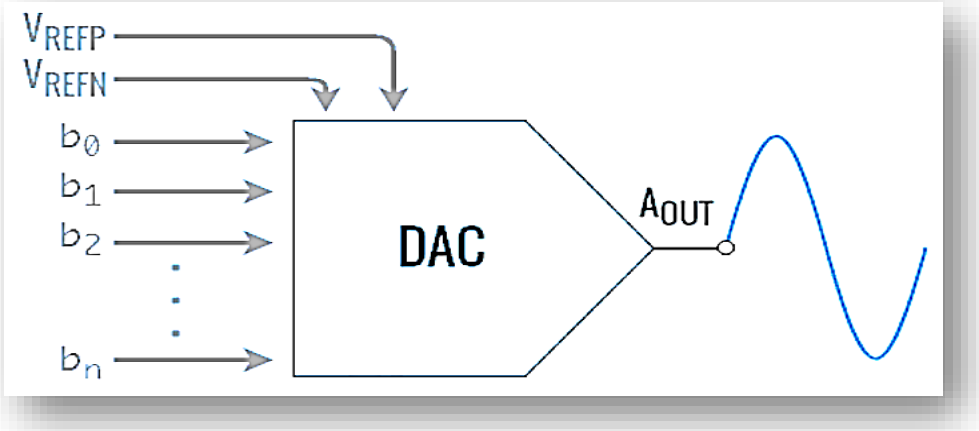
ARM Cortex-M3 LPC1768 incorporate a 10-bit Digital to Analog Converter that provides buffered analog output. LPC1768 DAC has only 1 output pin, referred to as **AOUT**. The analog voltage at the output of this pin is given as:

$$V_{AOUT} = (VALUE * (V_{REFP}-V_{REFN})/1024) + V_{REFN}$$

When we have $V_{REFN} = 0$:

$$V_{AOUT} = VALUE * V_{REF}/1024$$

where VALUE is the 10-bit digital value which is to be converted into its analog counterpart and V_{REF} is the input reference voltage.



Pins relating to LPC1768 DAC block:

Pin	Description
AOUT (P0.26)	Analog Output pin. Provides the converted Analog signal which is referenced to V_{SSA} i.e. the Analog GND. Set Bits[21:20] in PINSEL1 register to [10] to enable this function.
V_{REFP}, V_{REFN}	These are reference voltage input pins used for both ADC and DAC. V_{REFP} is positive reference voltage and V_{REFN} is negative reference voltage pin. In example shown below we will use V_{REFN}=0V(GND).
V_{DDA}, V_{SSA}	V_{DDA} is Analog Power pin and V_{SSA} is Ground pin used to power the ADC module. These are generally same as V_{CC} and GND but with additional filtering to reduce noise.

DAC Registers in ARM Cortex-M3 LPC1768:

The DAC module in ARM LPC1768 has 3 registers viz. DACR, DACCTRL, DACCNTVAL. Out of these DACCTRL, DACCNTVAL are related with DMA operation, and only DACR is used in the DAC programming. Hence DACCTRL, DACCNTVAL are not discussed here which are not included in the syllabus.

Also note that the DAC doesn't have a power control bit in PCONP register. Simply select the AOUT alternate function for pin P0.26 using PINSEL1 register to enable DAC output.

The DACR register in LPC1768

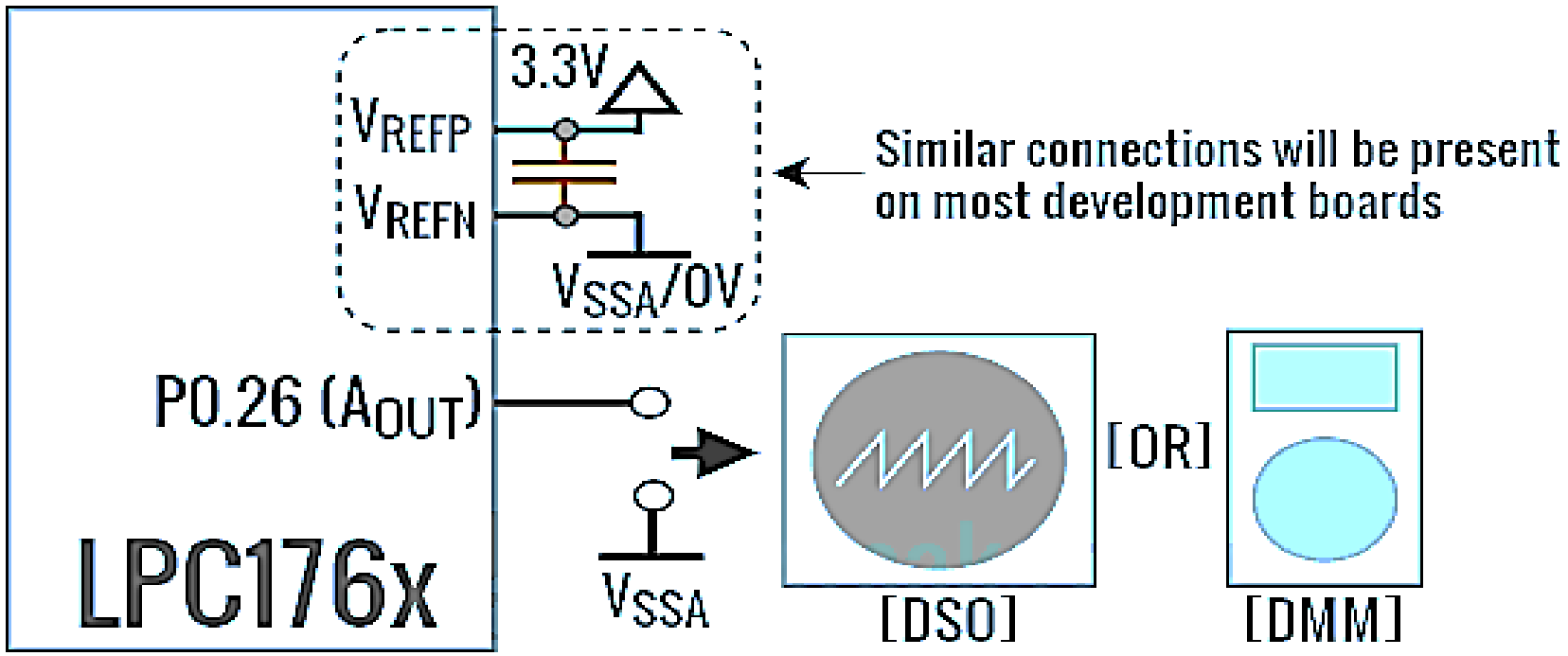
The field containing bits [15:6] is used to feed a digital value which needs to be converted and bit 16 is used to select settling time. The bit significance is as shown below:

1. **Bit[5:0]:** Reserved.
2. **Bit[15:6] – VALUE:** After a new VALUE is written to this field, given settling time selected using BIAS has elapsed, we get the converted analog voltage at the output. The formula for analog voltage at AOUT pin is as shown above.
3. **Bit[16] – BIAS:** Setting this bit to 0 selects settling time of 1us max with max current consumption of 700uA at max 1Mhz update rate. Setting it to 1 will select settling time of 2.5us but with reduced max current consumption of 300uA at max 400Khz update rate.
4. **Bits[31:17]**: Reserved

ARM Cortex-M3 LPC1768 DAC example

Let us see a basic LPC1768 DAC example. In this DAC example, we will be changing the output from 0V to V_{REFP} (V_{REFN}=0V) and then falling back to 0V in steps of 10ms. Vrefp = 3.3 V. This DAC program will basically output a sawtooth waveform. We will be using BIAS = 0 i.e. settling time of 1us. You can connect an Oscilloscope or a Multimeter between P0.26 and GND to check the changing analog output. Since the output is buffered you can drive an LED from A_{OUT} but it won't glow until it reaches its forward bias voltage of around 1.7 Volts. So, keep this in mind when checking for analog output using an LED.

The digital values from 0 to 1023 are sent to DACR with 10 millisecond delay between each value. So the converted analog voltage will be increasing in steps up to the maximum value of 3.3 V corresponding to 1023. These analog voltages are observed on a CRO as sawtooth. This process repeats continuously to get a sawtooth waveform. If there is no delay between the values, then we get ramp waveform.


3.3V
VREFP
VREFN
VSSA/0V
Similar connections will be present
on most development boards
P0.26 (AOUT)
LPC176x
VSSA
[OR]
[DSO]
[DMM]

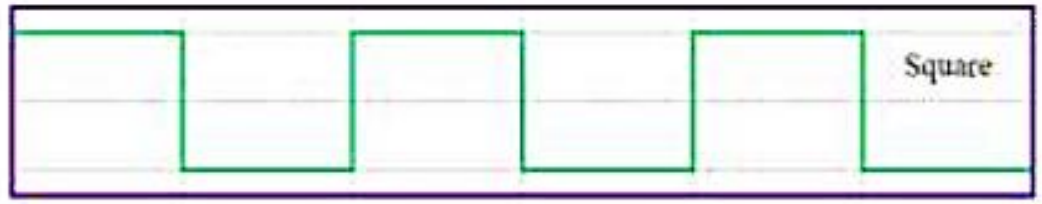
```c
#include <lpc17xx.h>

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

int main(void)
{
  //SystemInit(); //gets called by Startup code before main(), sets CCLK = 100Mhz
  initTimer0();
  LPC_PINCON->PINSEL1 |= (1<<21); //Select AOUT function for P0.26 , bits[21:20] = [10]
  unsigned int value=0; //Binary value used for Digital To Analog Conversion
  while(1)
  {
    if(value > 1023) value=0; //For 10-bit DAC max-value is 2^10 - 1 = 1023
    LPC_DAC->DACR = (value<<6);
    delayMS(10);
    value++;
  }
}
void initTimer0(void)
{
  //Assuming that PLL0 has been setup with CCLK = 60Mhz
  //using default PCLK = 25 which is configure by startup code
  LPC_TIM0->CTCR = 0x0;
  LPC_TIM0->PR = 25000-1; //25000 clock cycles @25Mhz PCLK = 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
}
```

//To generate triangular wave:


```
#include <lpc17xx.h>

int main(void)

{

      unsigned int temp, i; //Binary value used for Digital to Analog Conversion

      LPC_PINCON->PINSEL1 |= (1<<21); //Select AOUT function for P0.26 , bits[21:20] = [10]

      while(1)

         {

            for(i=0;i!=1023;i++)    //keep on incrementing value from 00 till 1023 in step of 1
         {
             temp=i;
             LPC_DAC->DACR = (temp<<6);

         }
           for(i=1023; i!=0;i--) //keep on decrementing value from 1023 till 00 in step of 1
        {
         temp=i;
         LPC_DAC->DACR = (temp<<6);

        }

   }

}
```

```
#include <LPC17xx.H>

int main ()
{
        unsigned long int temp=0x00000000;
        unsigned int i=0;

 LPC_PINCON->PINSEL0 &= 0xFF0000FF ;                    // Configure P0.4 to P0.11 as GPIO
        LPC_GPIO0->FIODIR  |= 0x00000FF0 ;
        LPC_GPIO0->FIOMASK = 0XFFFFF00F;

  while(1)
  {
        //output 0 to FE
    for(i=0;i!=0xFF;i++)
    {
        temp=i;
        temp = temp << 4;
        LPC_GPIO0->FIOPIN = temp;
    }
        // output FF to 1
    for(i=0xFF; i!=0;i--)
    {
        temp=i;
        temp = temp << 4;
        LPC_GPIO0->FIOPIN = temp;
    }
        }//End of while(1)
}//End of main()
```

```
//To generate square wave
#include <lpc17xx.h>
void delayMS(unsigned int milliseconds);
int main(void)
{
    LPC_PINCON->PINSEL1 |= (1<<21); //Select AOUT function for P0.26 , bits[21:20] = [10]
     while(1)
     {
            LPC_DAC->DACR = (1023<<6);
            delayMS(9999); //for 10 millisecond delay
            LPC_DAC->DACR = (0<<6);
            delayMS(9999); //for 10 millisecond delay
     }
}
void delayMS(unsigned int count) //Using Timer0
{
        LPC_TIM0->CTCR = 0x0; //Timer mode
        LPC_TIM0->PR = 2; //Increment TC at every 3 pclk
        LPC_TIM0->TCR = 0x02; //Reset Timer
        LPC_TIM0->TCR = 0x01; //Enable timer
        while(LPC_TIM0->TC < count); //wait until timer counter reaches the desired delay
        LPC_TIM0->TCR = 0x00; //Disable timer
}
```

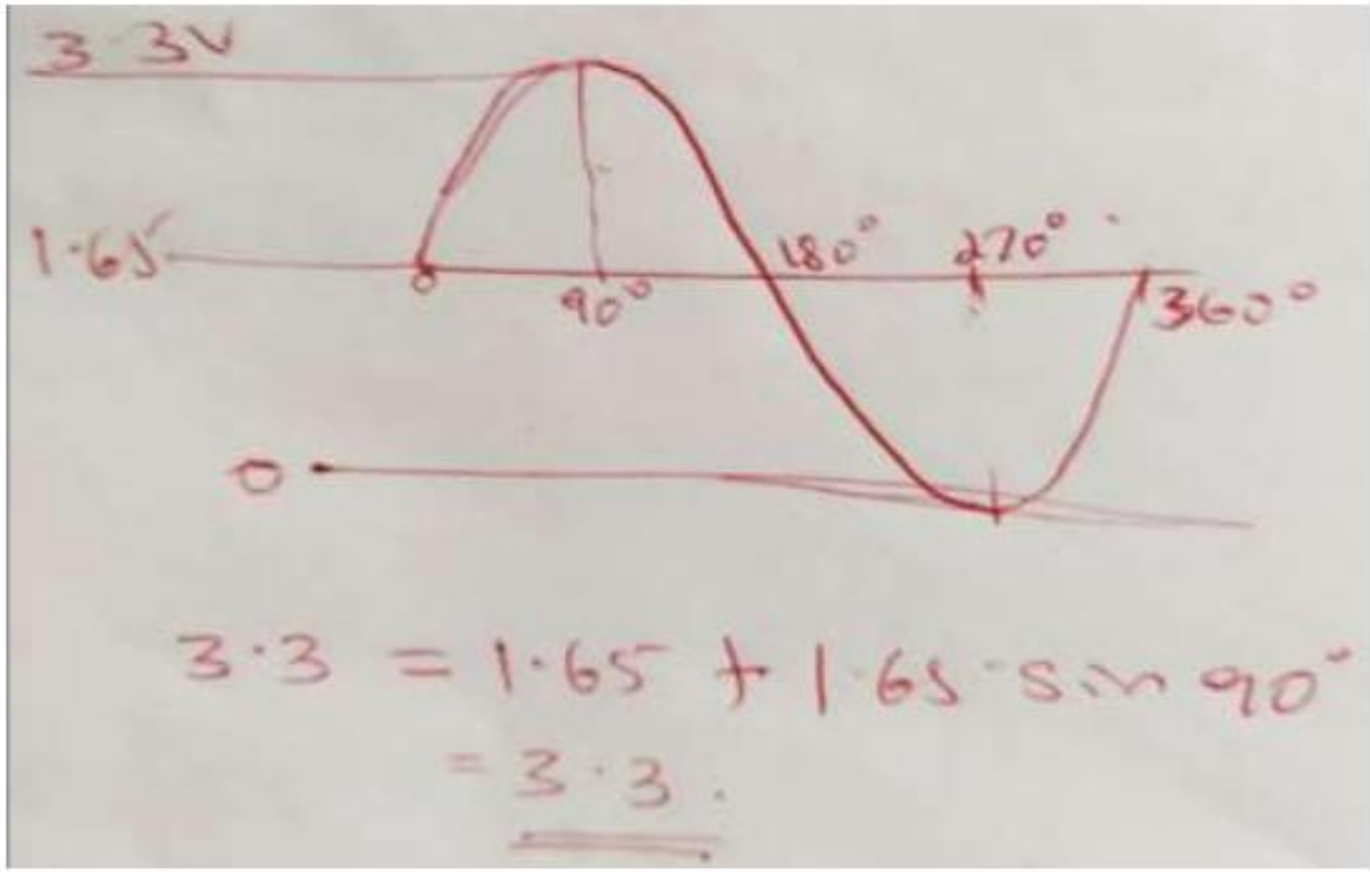


SQUARE WAVE

```
#include <LPC17xx.H>

void delay(void);

int main ()
{
                                LPC_PINCON->PINSEL0 &= 0xFF0000FF ;
                                // Configure P0.4 to P0.11 as GPIO
   LPC_GPIO0->FIODIR  |= 0x00000FF0 ;
                                LPC_GPIO0->FIOMASK = 0XFFFFF00F;
                while(1)
   {
                LPC_GPIO0->FIOPIN  = 0x00000FF0 ;
      delay();
      LPC_GPIO0->FIOCLR  = 0x00000FF0 ;
      delay();
   }
}

void delay(void)
{
                unsigned int i=0;
                for(i=0;i<=9500;i++);
}
```

//To generate sinewave:

Find values for sine function which is given by Y = A+Asin(θ) where, A=1.65V because we are generating unipolar sinewave. Let the swing be from 0V to 3.3V and hence 3.3/2= 1.65V is the mid value.



We can assign 0v with 0, mid value 1.65V with 1023/2 = 511 and 3.3V with 1023. Considering 10 samples to construct one full cycle of sine wave each sample differ with its adjacent sample by an angle of 360/10 = 36 degree/sample.

```
#include <LPC17xx.H>

int count=0,sinevalue,value;
unsigned char sine_tab[49]=
        { 0x80,0x90,0xA1,0xB1,0xC0,0xCD,0xDA,0xE5,0xEE,0xF6,0xFB,0xFE,0xFF,0xFE,0xFB,0xF6,0xEE,0xE5,0xDA,
0xCD,0xC0,0xB1,0xA1,0x90, 0x80,0x70,0x5F,0x4F,0x40,0x33,0x26,0x1B,0x12,0x0A,0x05,0x02,
0x00,0x02,0x05,0x0A,0x12,0x1B,0x26,0x33,0x40,0x4F,0x5F,0x70,0x80};

int main(void)
{
        LPC_PINCON->PINSEL0 &= 0xFF0000FF ;          // Configure P0.0 to P0.15 as GPIO
  LPC_GPIO0->FIODIR  |= 0x00000FF0 ;
        LPC_GPIO0->FIOMASK = 0XFFFFF00F;

        count = 0;
        while(1)
        {
                for(count=0;count<48;count++)
                {
                        sinevalue = sine_tab[count];//+0X10 ;
                        value= 0x00000FF0 & (sinevalue << 4);
                        LPC_GPIO0->FIOPIN = value;
                }
        }
}
```