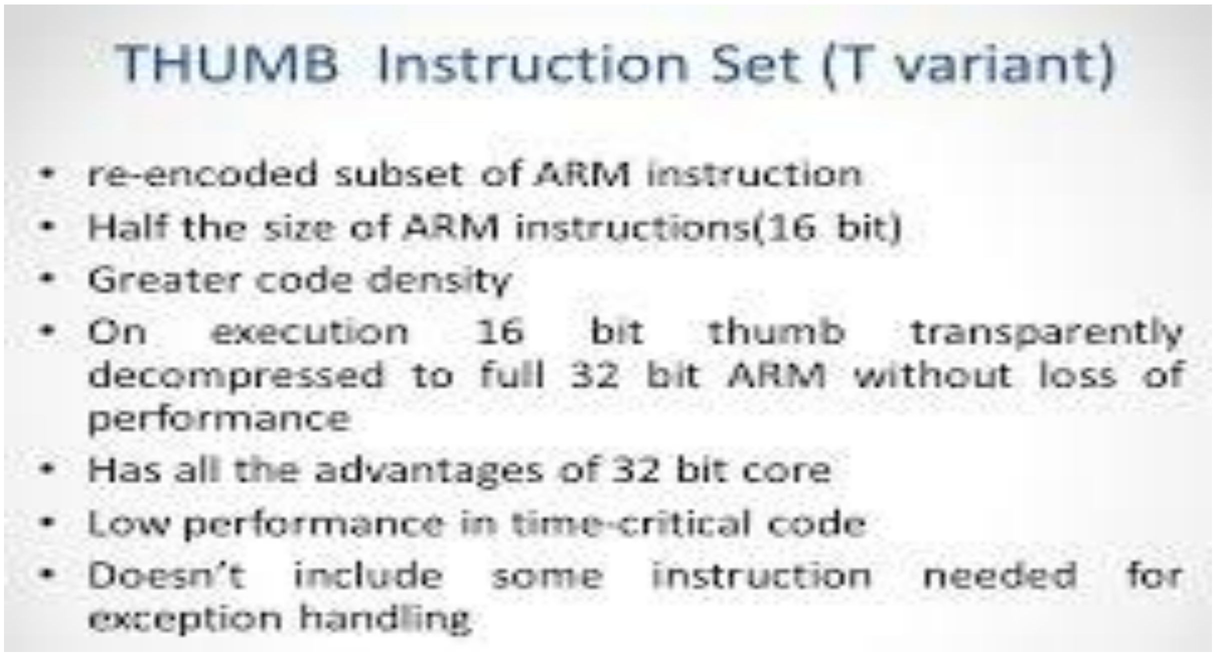
Architecture

Arm 7 TDMI Brief Overview

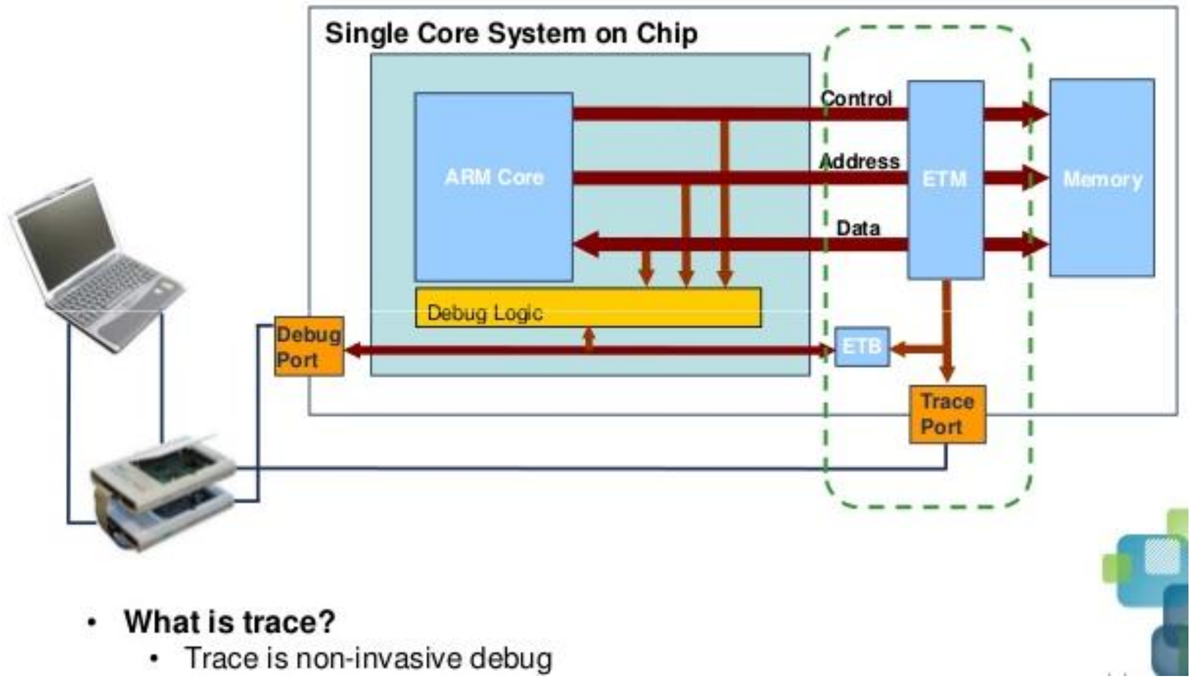
*T*humb Instruction Set



THUMB Instruction Set (T variant)

- re-encoded subset of ARM instruction
- Half the size of ARM instructions(16 bit)
- Greater code density
- On execution 16 bit thumb transparently decompressed to full 32 bit ARM without loss of performance
- Has all the advantages of 32 bit core
- Low performance in time-critical code
- Doesn't include some instruction needed for exception handling

On chip *D*ebug



- **What is trace?**
    - Trace is non-invasive debug

Noninvasive debug is defined as a debug process where you can observe the processor but not control it.

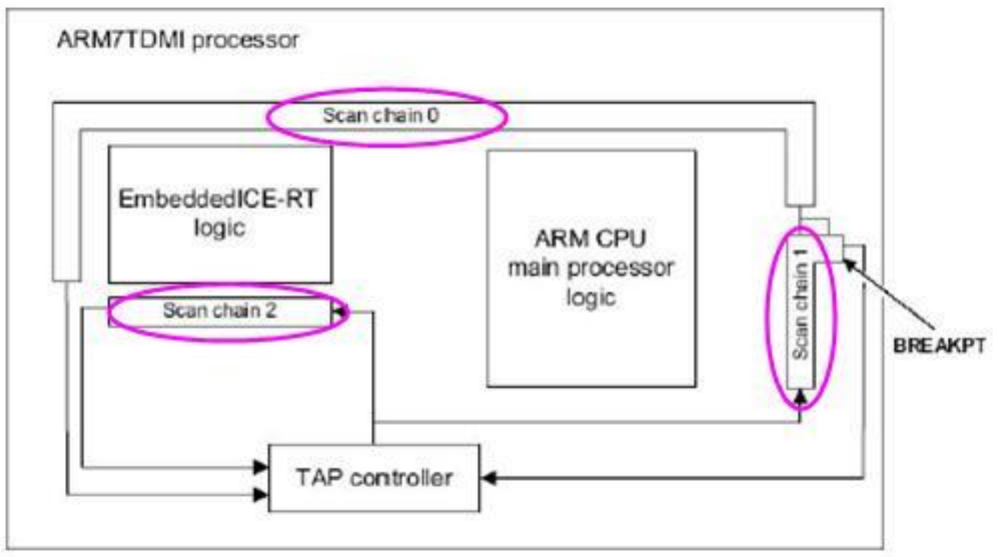
Embedded Ice Logic

Embedded ICE logic:

The EmbeddedICE logic provides integrated on-chip debug support for the ARM7TDMI core. You use the EmbeddedICE logic to program the conditions under which a breakpoint or watchpoint can occur.The EmbeddedICE logic contains a Debug Communications Channel (DCC), used to pass information between the target and the host debugger. The EmbeddedICE logic is controlled through the Joint Test Action Group (JTAG) test access port.

About EmbeddedICE Logic:

The ARM7TDMI processor EmbeddedICE Logic provides integrated on-chip debug support for the ARM7TDMI core.The EmbeddedICE Logic is programmed serially using the ARM7TDMI processor TAP controller illustrates the relationship between the core, EmbeddedICE Logic, and the TAP controller, showing only the pertinent signals.



Fig(EmbeddedICE Logic)

The EmbeddedICE Logic comprises:

1. Two real-time watchpoint units
2. Two independent registers:

Two Independent Register:-

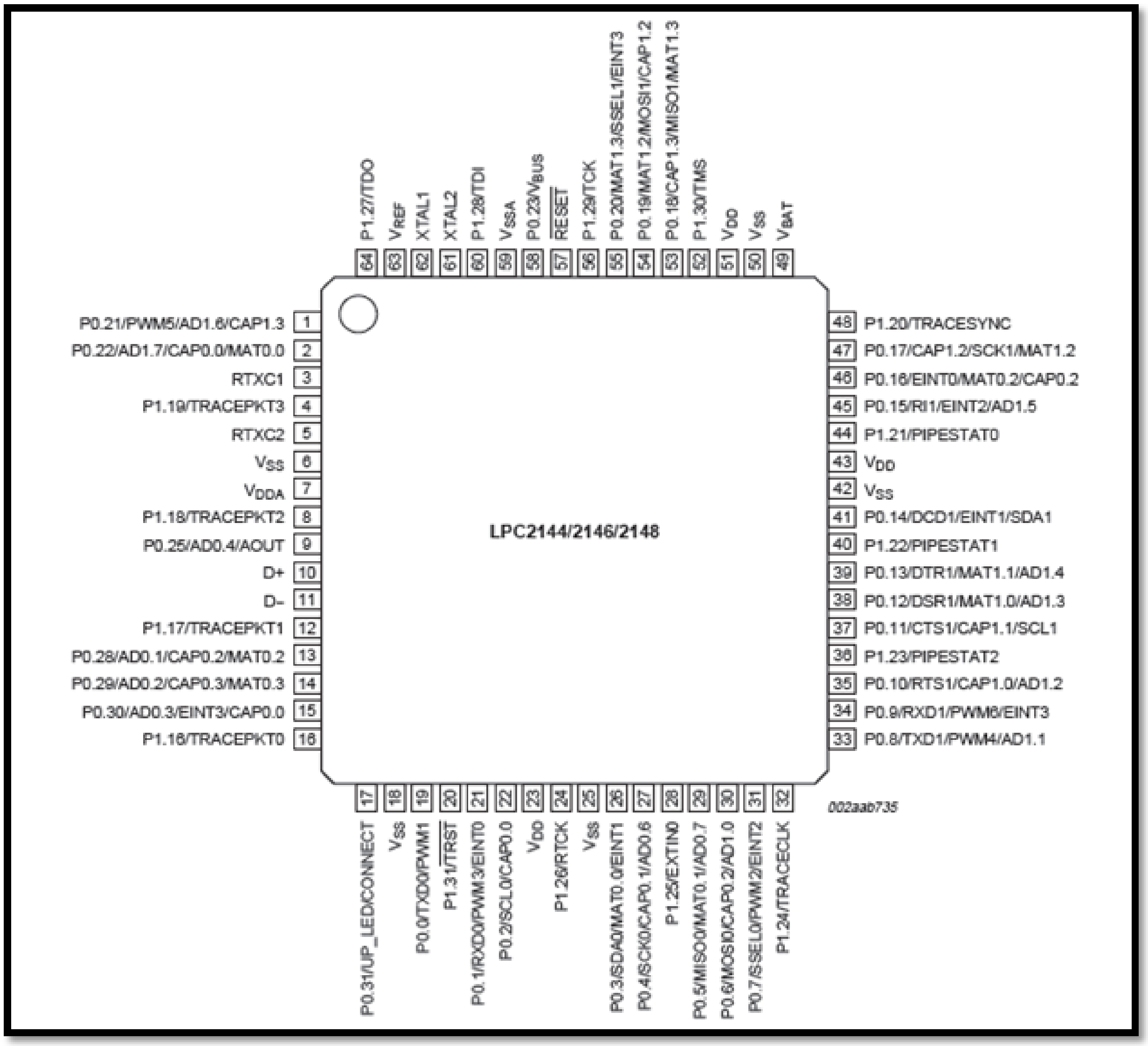
- The debug control register
- Debug status register.

Debug Communications Channel (DCC):-

The debug control register and the debug status register provide overall control of EmbeddedICE operation.You can program one or both watchpoint units to halt the execution of instructions by the core. Execution halts when the values programmed into EmbeddedICE match the values currently appearing on the address bus, data bus, and various control signals.

Basics of Programming for LPC2148

It is more important to have basic knowledge of pin configuration, memory, I/O ports and basic registers. Here is the **pin diagram of LPC 2148**.



MEMORY

LPC2148 has 32kB on chip SRAM and 512 kB on chip FLASH memory. It has inbuilt support up to 2kB end point USB RAM also. This huge amount of memory is well suited for almost all the applications.

We will explain the basic functions of above memory.

On chip FLASH memory system

The LPC2148 incorporates a 512 kB Flash memory system. This memory may be used for both code and data storage. The FLASH memory can be programmed by means of

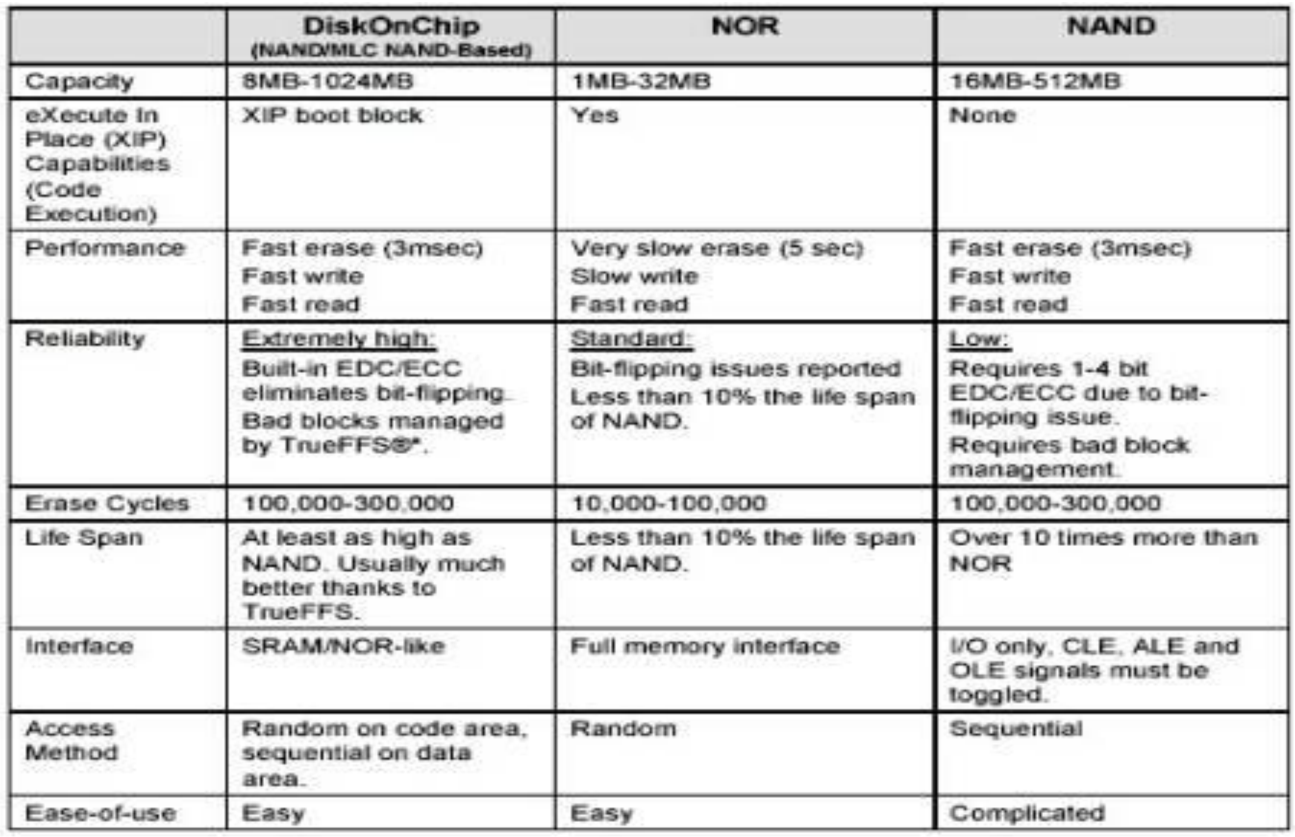
1. Serialbuilt-in JTAG interface
2. Using In System Programming (ISP) and UART0or
3. By means of InApplication Programming (IAP) capabilities.

The application program, using the IAP functions, may also erase and/or program the Flash while the application is running, allowing a great degree of flexibility for data storage field firmware upgrades, etc. When the LPC2148 on-chip boot loader is used, 500 kB of Flash memory is available for user code.

The LPC2148 Flash memory provides minimum of 100,000 erase/write cycles and 20 years of data-retention.

On chip SRAM

The LPC2148 provides 32 kB of static RAM which may be used for code and/or data storage. It may be accessed as 8-bits, 16-bits, and 32-bits.



	DiskOnChip (NAND/MLC NAND-Based)	NOR	NAND
Capacity	8MB-1024MB	1MB-32MB	16MB-512MB
eXecute In Place (XIP) Capabilities (Code Execution)	XIP boot block	Yes	None
Performance	Fast erase (3msec) Fast write Fast read	Very slow erase (5 sec) Slow write Fast read	Fast erase (3msec) Fast write Fast read
Reliability	Extremely high: Built-in EDC/ECC eliminates bit-flipping. Bad blocks managed by TrueFFS®*.	Standard: Bit-flipping issues reported Less than 10% the life span of NAND.	Low: Requires 1-4 bit EDC/ECC due to bit-flipping issue. Requires bad block management.
Erase Cycles	100,000-300,000	10,000-100,000	100,000-300,000
Life Span	At least as high as NAND. Usually much better thanks to TrueFFS.	Less than 10% the life span of NAND.	Over 10 times more than NOR
Interface	SRAM/NOR-like	Full memory interface	I/O only, CLE, ALE and OLE signals must be toggled.
Access Method	Random on code area, sequential on data area.	Random	Sequential
Ease-of-use	Easy	Easy	Complicated

I/O Ports

LPC 2148 has two I/O Ports each of 32 bit wide giving us total 64 I/O Pins. Ports are named as P0 and P1.

Pins of each port are labelled as PX.Y where X stands for port number, 0 or 1 where else Y stands for pin number 0 to 31.

Each pin can perform alternate functions also. For eg. P0.8 serves as GPIO as well as transmitter pin of UART1, PWM4 and AD1.1.

On RESET, each pin is configured as GPIO. For any of the other use, programmer must configure it properly.

Pin Configuration

Pin Description

Figure 1 Pin Description

The first step towards programming is HOW TO CONFIGURE GPIO Pins. Let's start with the associated concepts and registers.

PIN	TYPE	Description
P0.0-P0.31 & P1.16-P1.31	Input/Output	General purpose Input /Output. On RESET, all pins are configured as Input.

Total of 30 input/output and asingle output only pin out of 32 pins are available on PORT0. PORT1 has up to 16 pinsavailable for GPIO functions. PORT0 and PORT1 are controlled via two groups of registers explained below.

1. IOPIN

It is GPIO Port Pin value register. The current state of the GPIO configured port pins can always be read from this register, regardless of pin direction.

2.IODIR

GPIO Port Direction control register. This register individually controls the direction of each port pin.

3.IOCLR

GPIO Port Output Clear registers. This register controls the state of output pins. Writing ones produces lows at the corresponding port pins and clears the corresponding bits in the IOSET register. Writing zeroes has no effect.

4.IOSET

GPIO Port Output Set registers. This register controls the state of output pins in conjunction with the IOCLR register. Writing ones produces highs at the corresponding port pins. Writing zeroes has no effect.

This is the set of register used to configure I/O Pins. Now let’s move to individual registers in deep.

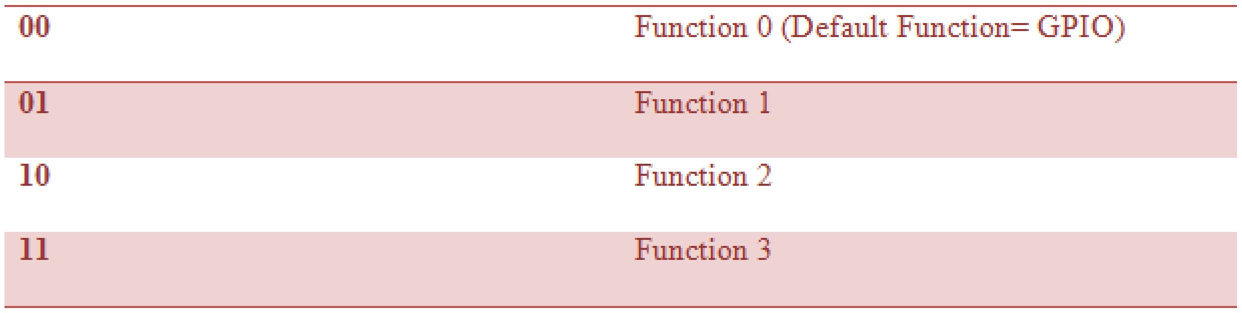
Pin Connect Block

REGISTERS FOR C PROGRAMMING

1. PINSEL0

Port 0 has 32 pins (P0.0 to P0.31). Each pin can have multiple functions. On RESET, all pins are configured as GPIO pins. However we can re-configure using the registers PINSEL0 and PINSEL1.

PINSEL0 is used to select function of P0.0 to P0.15. Each pin has up to 4 functions so 2 bits/pin in PINSEL0 is provided for selecting function.



00	Function 0 (Default Function= GPIO)
01	Function 1
10	Function 2
11	Function 3

2. PINSEL1

PINSEL1 is used to select function of pins P0.16 to P0.31

3. PINSEL2

PINSEL2 is used to select function of pins P1.16 to P1.31

4. IO0DIR

IO0DIR is used to configure pins of port 0-P0 as input or output pins.

1= output pin

0= input pin

Example: IO0DIR=0x0000ffff means P0.0 to P0.15 are configured as output pins and P0.16 to P0.31 are configured as input pins.

5. IO1DIR

IO1DIR is used to configure pins of port 1-P1 as input or output pins.

1= output pin

0= input pin

Example: IO0DIR=0xaaaaaaaa means even pins (P1.0, P1.2, P1.4 etc.) are configured as input pins and odd pins (P1.1, P1.3, P1.5 etc.) are configured as input pins.

6. IO0SET

It is used to set pins of Port0-P0 to logic 1.

Example: IO0SET=0x0000ffff will set pins P0.0 to P0.15 at logic 1. It will not affect other pins.

7. IO0CLR

It is used to set pins of Port0-P0 to logic 0.

Example: IO0SET=0x0000ffff will set pins P0.0 to P0.15 at logic 0. It will not affect other pins.

8. IO1SET

It is used to set pins of Port1-P1 to logic 1.

Example: IO1SET=0x0000ffff will set pins P1.0 to P1.15 at logic 1. It will not affect other pins.

9.IO1CLR

It is used to set pins of Port1-P1 to logic 0.

Example: IO1SET=0x0000ffff will set pins P1.0 to P1.15 at logic 0. It will not affect other pins.

Once the use of above all registers is perfectly understood, you are good to go with programming.

Example: Blink LEDs connected on pins P1.16-P1.23

Step-1: Specify the direction of pins as output using IO0DIR,IO1DIR.

Step-2a: Set pins P1.25 using IO1SET→Buzzer.

Step-2b: Set pins P0.31 using IO0SET→LED

Step-3a: Clear pins P1.25 using IO1CLR→Buzzer.

Step-3b: Clear pins P0.31 using IO0CLR→LED.

Step-4: Go to step-2.

UART

For UART0 the TxD pin is P0.0 and RxD pin is P0.1 and similarly for UART 1 the TxD pin is P0.8 and RxD pin is P0.9 as shown in the table below :

Pins:	TxD	RxD	Application
UART0	P0.0	P0.1	In System Prog.
UART1	P0.8	P0.9	Peripheral

Note 1: Both UART0 & UART1 blocks internally have a 16-byte FIFO (First In First Out) structure to hold the Rx and Tx data. Each byte in this FIFO represents a character which was sent or received **in order**. Both blocks also contain 2 registers each, for data access and assembly.

Tx has THR(Transmit Holding Register) and TSR(Transmit Shift Register) – When we write Data to be sent into THR it is then transferred to TSR which assembles the data to be transmitted via Tx Pin.

Similarly Rx has RSR(Receive Shift Register) and RBR(Receive Buffer Register) – When a valid data is Received at Rx Pin it is first assembled in RSR and then passed in to Rx FIFO which can be then accessed via RBR.

Note 2: The 4 mentioned UART Pins on LPC214x are 5V tolerant .. still make sure you are using 3.3v Voltage-level for UART communication. Hence , if you have RS232 to serial TTL module make sure it has MAX3232 =OR= if you are using FTDI's FT232R based USB to serial TTL module make sure it is configured for 3.3v I/O.

Clocks, PLL, VPBDIV

LPC 2148 needs 2 clocks - one for its peripherals and the other for its CPU. CPU works faster at higher frequencies and peripherals need not be fast. The Peripheral clock - PCLK and CPU CLock - CCLK can get clock input from a PLL or from an external source directly. Configuration of PLL and VPB Divider must be the first things in list after a reset.

VPB Divider

The VPB Divider Dialog controls the VLSI Peripheral Bus Divider control register. This register controls the VBP clock rate (PCLK) based on the MPU clock rate (CCLK).

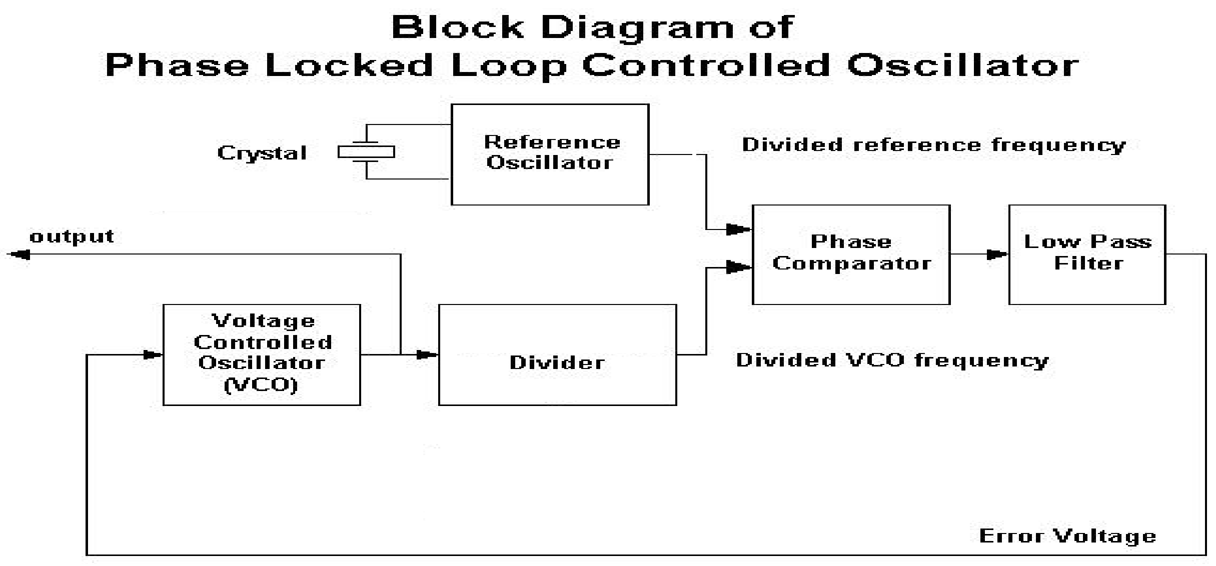
> **VPBDIV** (VPB Divider Control Register) contains the bit settings that determine the MPU clock (CCLK) divisor for calculating the peripheral clock (PCLK). Use the list box to select the clock divider.

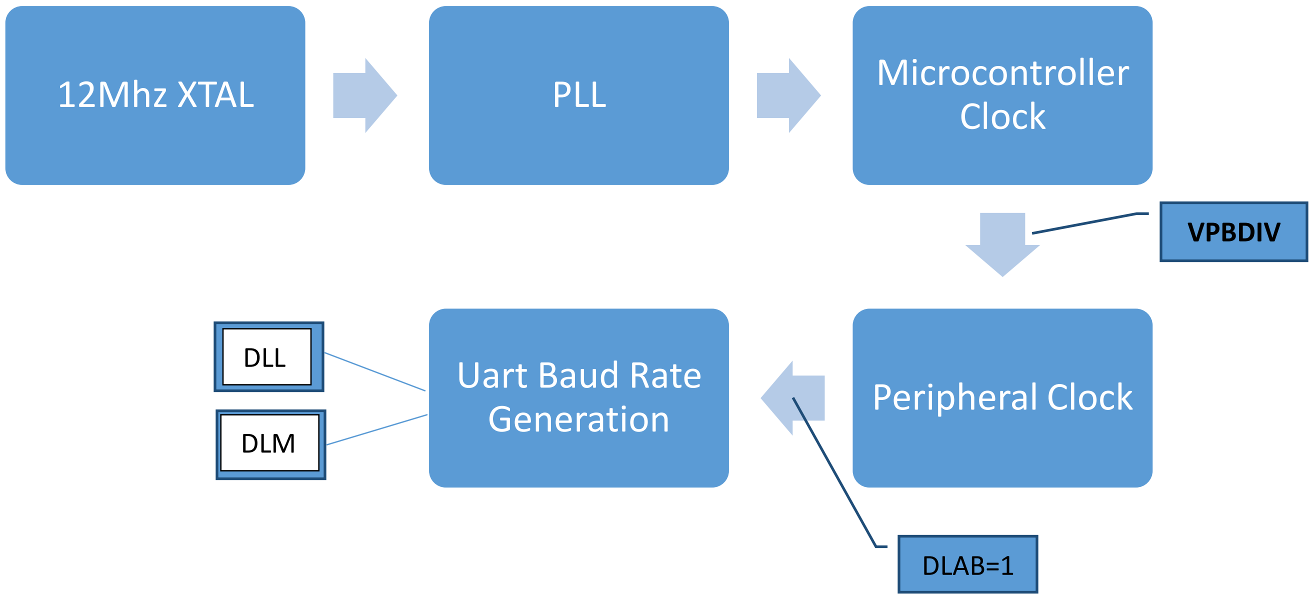
PLL = Phase Lock Loop

1. A circuit which synchronizes an adjustable oscillator with another oscillator by the comparison of phase between the two signals.
2. An electronic circuit that synchronizes itself to an external reference signal.

Uses of PLL

- ***To generate High Frequency Clock in Microprocessor***.
- In Mobile Communication to generate Carrier Frequency.





Baud rate: In register U0LCR or U1LCR set bit DLAB =1. This enables access to registers DLL and DLM for setting the baud rate. Also, if needed, set the fractional baud rate in the fractional divider register.

UART Baud Rate Generation:

Note : In real world there are very less chances that you will get the actual baudrate same as the desired baudrate. In most cases the actual baudrate will drift a little above or below the desired baud and also, as the desired baudrate increases this drift or error will also increase – this is because of the equation itself and the limitations on MULVAL , DIVADDVAL! For e.g. if the desired baud rate is

9600 and you get a baud like 9590 , 9610 , 9685 , 9615 , etc.. then in almost all cases it will work as required. In short , a small amount of error in actual baudrate is generally tolerable in most systems.

The master formula for calculating baud rate is given as :

$$\text{BaudRate} = \frac{\text{PCLK in Hertz}}{16 \text{ x } (256\text{xDLM} + \text{DLL}) \text{ x } (1 + \text{DIVADDVAL}/\text{MULVAL})}$$

which can be further simplified to :

$$\text{BaudRate} = \frac{\text{PCLK in Hertz}}{16 \text{ x } (256\text{xDLM} + \text{DLL})} \text{ x } \frac{\text{MULVAL}}{\text{MULVAL} + \text{DIVADDVAL}}$$

$$\text{BaudRate} = \text{Base Value} \text{ X } \text{Fractional Multiplier}$$

with following conditions strictly applied :

0 < MULVAL <= 15 0 <= DIVADDVAL <= 15 if DIVADDVAL > 0 & DLM = 0 then, DLL >= 2

As it can been seen this formula has 2 prominent parts which are : A **Base value** and a**Fractional Multiplier** i.e:

BaudRate = [Base] x [Fraction(i.e. Prescale)]

This Fractional Multiplier can be used to scale down or keep the base value as it is .. hence its very useful for fine-tuning and getting the baudrate as accurate as possible.

Where PCLK is the Peripheral Clock value in Hz , U0DLM and U0DLL are the divisor registers which we saw earlier and finally DIVADDVAL and MULVAL are part of the Fractional baudrate generator register.

Now we know the formula .. how we do actually start ?

1) The Dirty and Simplest Method:

The quickest and also the dirtiest(accuracy wise) method without using any algorithm or finetuning is to set DLM=0 and completely ignore the fraction by setting it to 1. In this case MULVAL=1 and DIVADDVAL=0 which makes the Fraction Multiplier = 1. Now we are left with only 1 unknown in the equation which is U0DLL and hence we simplify the equation and solve for U0DLL.

$$U0DLL = \frac{\text{PCLK in Hertz}}{16 \times \text{Desired\_BaudRate}}$$

But I'd recommend that stay away from above method, if possible, as it works only for particular bauds given a specific PCLK value and moreover U0DLL might get out of range i.e. > 255 in which case you have to start increasing DLM and recompute a new value for U0DLL.

2) The more involved method(Recommended):

Now lets see three examples for calculating Baud Rates Manually using some finetuning :

In these method we again start with DLM=0 , DIVADDVAL=0 and MULVAL=1 and get an initial value for DLM. If you are lucky you will get a very close baudrate to the desired one. If not then we perform some finetuning using DLM , MULVAL and DIVADDVAL and get a new value for DLM. There is on one single method to perform finetuning and one can also make an algorithm for computing the best match given the desired baudrate and PCLK. The finetuning method which I have given below is a basic one suitable for beginners(in my opinion :P).

Ex 1 : PCLK = 30 Mhz and Required Baud Rate is 9600 bauds.

Lets start with **U0DLM = 0 , DIVADDVAL = 0 and MULVAL = 1**
We have PCLK = 30 Mhz = 30 x 10^6 Hz

So the equation now gets simplified and we can find U0DLL.

We get U0DLL = 195.3125 , since it must be an integer we use 195. With U0DLL = 195 our actual baud rate will be = 9615.38 with an error of +15.28 from 9600. Now all we have to do is to get this error as low as possible. This can be done by multiplying it by a suitable fraction defined using MULVAL and DIVADDVAL – as given in equation. But since MULVAL & DIVADDVAL can be maximum 15 we don't have much control over the fraction's value. **Note: *As per my convention I calculate BaudRate Error as* = Actual BaudRate – Desired BaudRate.**

Lets see how :
First lets compute the required fraction value given by : **[Desired Baud / Actual Baud Rate]**. In our case its 0.9984. In short this is the value that needs to be multiplied by baud value we got above to bring back it to ~9600. The closest downscale value possible(<1) to 0.9948 is 0.9375 when MULVAL=15 & DIVADDVAL=1. But 0.9375 is too less because 9615.38x0.9375 = 9014.42 which is way too far. Now , one thing can be done is that we set MULVAL = 15 and now start decreasing U0DLL from 195. Luckily when U0DLL = 183 we get a baud rate of 9605.53 i.e ~9605! Yea! Hence the final settings in this case will be : **PCLK = 30 x 10^6 Hz**

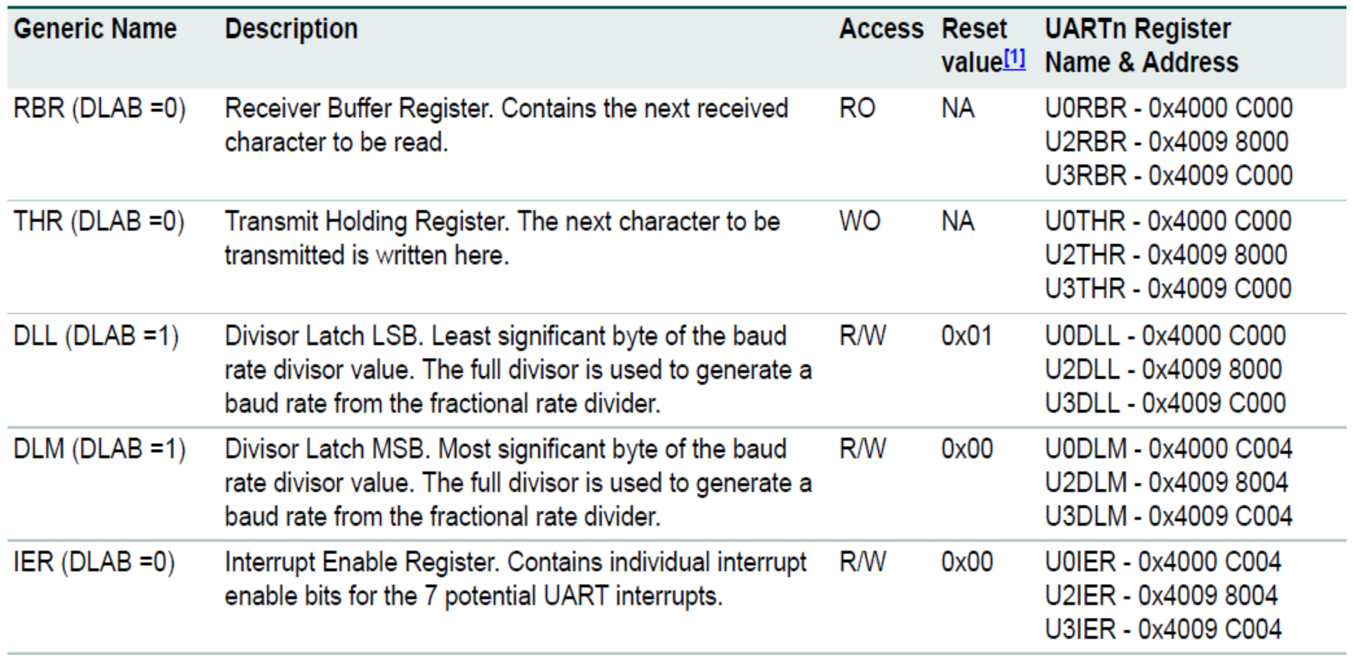
U1DLL = 0xB7;//183
U1DLM = 0
U1FDR =0xF1; //MULVAL = 15 DIVADDVAL = 1 → **@9605 ~ 9600**

Configuring and Initializing UART

Once you are aware of how UART communication works , configuring and initializing UART is pretty straight forward. In Source Code Examples for this tutorial we will use the following configuration :

- BaudRate = 9600 buads (with PCLK=30Mhz)
- Data Length = 8 bits
- No Parity Bit
- and 1 Stop Bit

Table 271. UART0/2/3 Register Map



Generic Name	Description	Access	Reset value[1]	UARTn Register Name & Address
RBR (DLAB =0)	Receiver Buffer Register. Contains the next received character to be read.	RO	NA	U0RBR - 0x4000 C000 U2RBR - 0x4009 8000 U3RBR - 0x4009 C000
THR (DLAB =0)	Transmit Holding Register. The next character to be transmitted is written here.	WO	NA	U0THR - 0x4000 C000 U2THR - 0x4009 8000 U3THR - 0x4009 C000
DLL (DLAB =1)	Divisor Latch LSB. Least significant byte of the baud rate divisor value. The full divisor is used to generate a baud rate from the fractional rate divider.	R/W	0x01	U0DLL - 0x4000 C000 U2DLL - 0x4009 8000 U3DLL - 0x4009 C000
DLM (DLAB =1)	Divisor Latch MSB. Most significant byte of the baud rate divisor value. The full divisor is used to generate a baud rate from the fractional rate divider.	R/W	0x00	U0DLM - 0x4000 C004 U2DLM - 0x4009 8004 U3DLM - 0x4009 C004
IER (DLAB =0)	Interrupt Enable Register. Contains individual interrupt enable bits for the 7 potential UART interrupts.	R/W	0x00	U0IER - 0x4000 C004 U2IER - 0x4009 8004 U3IER - 0x4009 C004

Therefore for init. The DLL and DLM use
U1LCR=0x80|0x03 ;// DLAB=1|Config=0x03, now for 8bit data,1Stop Bit, No Parity
after that
U1LCR=0x00|Config;//DLAB=0|Config=0x03.

Basic Uart Code

Enhance This code for UART1

```
#define CR     0x0D
#include <LPC21xx.H>

void init_serial (void);
int putchar (int ch);
int getchar (void);
unsigned char test;

int main(void)
{
  char *Ptr = "*** UART0 Demo ***\n\n\rType
                    Characters to be echoed!!\n\n\r";
  VPBDIV = 0x02;         // Divide Pclk by two
  init_serial();
  while(1)
  {

    while (*Ptr)
    {
      putchar(*Ptr++);
    }
    putchar(getchar());  // Echo terminal
  }
}


void init_serial (void)
{
 PINSEL0    = 0x00000005; // Enable RxD0 and TxD0
 U0LCR    = 0x00000083;   //8 bits, no Parity, 1 Stop bit
              U0DLL      = 0x000000C3; //9600 Baud Rate @ 30MHz VPB Clock
              U0LCR    = 0x00000003;
}


int putchar (int ch)
 {
 if (ch == '\n')
 {
   while (!(U0LSR & 0x20));
   U0THR = CR;
 }
 while (!(U0LSR & 0x20));
 return (U0THR = ch);
}

int getchar (void)
{
 while (!(U0LSR & 0x01));
 return (U0RBR);
}
```

15.4.7 UART1 Line Control Register (U1LCR - 0x4001 000C)

The U1LCR determines the format of the data character that is to be transmitted or received.

Table 299: UART1 Line Control Register (U1LCR - address 0x4001 000C) bit description

Bit	Symbol	Value	Description	Reset Value
1:0	Word Length Select	00	5-bit character length.	0
		01	6-bit character length.	
		10	7-bit character length.	
		11	8-bit character length.	
2	Stop Bit Select	0	1 stop bit.	0
		1	2 stop bits (1.5 if U1LCR[1:0]=00).	
3	Parity Enable	0	Disable parity generation and checking.	0
		1	Enable parity generation and checking.	
5:4	Parity Select	00	Odd parity. Number of 1s in the transmitted character and the attached parity bit will be odd.	0
		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.	
6	Break Control	0	Disable break transmission.	0
		1	Enable break transmission. Output pin UART1 TXD is forced to logic 0 when U1LCR[6] is active high.	
7	Divisor Latch Access Bit (DLAB)	0	Disable access to Divisor Latches.	0
		1	Enable access to Divisor Latches.	
31:8	-		Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

15.4.3 UART1 Divisor Latch LSB and MSB Registers (U1DLL - 0x4001 0000 and U1DLM - 0x4001 0004, when DLAB = 1)

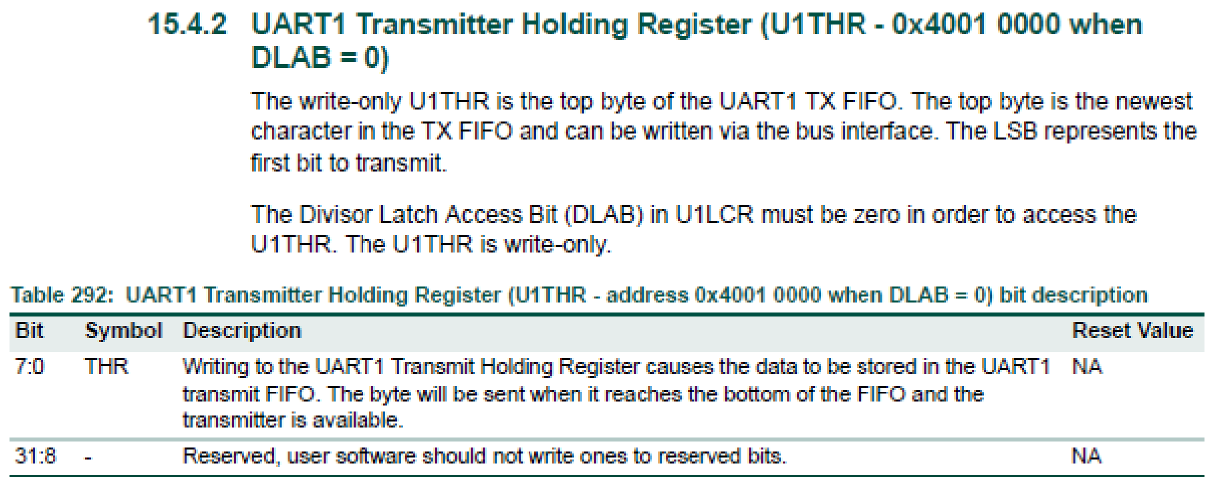
The UART1 Divisor Latch is part of the UART1 Baud Rate Generator and holds the value used, along with the Fractional Divider, to divide the APB clock (PCLK) in order to produce the baud rate clock, which must be 16x the desired baud rate. The U1DLL and U1DLM registers together form a 16-bit divisor where U1DLL contains the lower 8 bits of the divisor and U1DLM contains the higher 8 bits of the divisor. A 0x0000 value is treated like a 0x0001 value as division by zero is not allowed.The Divisor Latch Access Bit (DLAB) in U1LCR must be one in order to access the UART1 Divisor Latches. Details on how to select the right value for U1DLL and U1DLM can be found later in this chapter, see

Table 293: UART1 Divisor Latch LSB Register (U1DLL - address 0x4001 0000 when DLAB = 1) bit description

Bit	Symbol	Description	Reset Value
7:0	DLLSB	The UART1 Divisor Latch LSB Register, along with the U1DLM register, determines the baud rate of the UART1.	0x01
31:8	-	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

Table 294: UART1 Divisor Latch MSB Register (U1DLM - address 0x4001 0004 when DLAB = 1) bit description

Bit	Symbol	Description	Reset Value
7:0	DLMSB	The UART1 Divisor Latch MSB Register, along with the U1DLL register, determines the baud rate of the UART1.	0x00
31:8	-	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA



15.4.2 UART1 Transmitter Holding Register (U1THR - 0x4001 0000 when DLAB = 0)

The write-only U1THR is the top byte of the UART1 TX FIFO. The top byte is the newest character in the TX FIFO and can be written via the bus interface. The LSB represents the first bit to transmit.

The Divisor Latch Access Bit (DLAB) in U1LCR must be zero in order to access the U1THR. The U1THR is write-only.

Table 292: UART1 Transmitter Holding Register (U1THR - address 0x4001 0000 when DLAB = 0) bit description

Bit	Symbol	Description	Reset Value
7:0	THR	Writing to the UART1 Transmit Holding Register causes the data to be stored in the UART1 transmit FIFO. The byte will be sent when it reaches the bottom of the FIFO and the transmitter is available.	NA
31:8	-	Reserved, user software should not write ones to reserved bits.	NA

Interrupt Related Registers:

1) U0IER – Interrupt Enable Register: Set a bit to 0 to disable and 1 to enable the corresponding interrupt.

1. **Bit 0** – RBR Interrupt Enable
2. **Bit 1** – THRE Interrupt Enable

2) U0IIR – Interrupt Identification Register: Refer User Manual when in doubt. In some application the usage of this register might get a bit complicated.

This register is organized as follows:

1. **Bit 0 – Interrupt Pending :** 0 means atleast one interrupt is pending , 1 means no interrupts are pending. Note: This bit is ACTIVE LOW!
2. **Bits [3 to 1] – Interrupt Identification :**

    - [011] is for Receive Line Status(RLS) ,
    - [010] means Receive Data Available(RDA) ,
    - [110] is for Character Time-out Indicator(CTI) ,
    - [001] is for THRE Interrupt.

3. **Bits [7 to 6] – FIFO Enable.**
4. **Bit 8 – ABEOInt :** 1 means Auto Baud Interrupt has successfully ended and 0 otherwise.
5. **Bit 9 – ABTOInt :** 1 means Auto Baud Interrupt has Timed-out.
6. All others bits are reserved.

Hint_Interrupt

15.4.10 UART1 Line Status Register (U1LSR - 0x4001 0014)

The U1LSR is a read-only register that provides status information on the UART1 TX and RX blocks.

Table 281: UARTn Line Status Register (U0LSR - address 0x4000 C014, U2LSR - 0x4009 8014, U3LSR - 0x4009 C014) bit description

Bit	Symbol	Value	Description	Reset Value
0	Receiver Data Ready (RDR)		UnLSR0 is set when the UnRBR holds an unread character and is cleared when the UARTn RBR FIFO is empty.	0
		0	The UARTn receiver FIFO is empty.	
		1	The UARTn receiver FIFO is not empty.	
1	Overrun Error (OE)		The overrun error condition is set as soon as it occurs. An UnLSR read clears UnLSR1. UnLSR1 is set when UARTn RSR has a new character assembled and the UARTn RBR FIFO is full. In this case, the UARTn RBR FIFO will not be overwritten and the character in the UARTn RSR will be lost.	0
		0	Overrun error status is inactive.	
		1	Overrun error status is active.	
2	Parity Error (PE)		When the parity bit of a received character is in the wrong state, a parity error occurs. An UnLSR read clears UnLSR[2]. Time of parity error detection is dependent on UnFCR[0]. **Note:** A parity error is associated with the character at the top of the UARTn RBR FIFO.	0
		0	Parity error status is inactive.	
		1	Parity error status is active.	
3	Framing Error (FE)		When the stop bit of a received character is a logic 0, a framing error occurs. An UnLSR read clears UnLSR[3]. The time of the framing error detection is dependent on UnFCR0. Upon detection of a framing error, the Rx will attempt to resynchronize to the data and assume that the bad stop bit is actually an early start bit. However, it cannot be assumed that the next received byte will be correct even if there is no Framing Error. **Note:** A framing error is associated with the character at the top of the UARTn RBR FIFO.	0
		0	Framing error status is inactive.	
		1	Framing error status is active.	
4	Break Interrupt (BI)		When RXDn is held in the spacing state (all zeroes) for one full character transmission (start, data, parity, stop), a break interrupt occurs. Once the break condition has been detected, the receiver goes idle until RXDn goes to marking state (all ones). An UnLSR read clears this status bit. The time of break detection is dependent on UnFCR[0]. **Note:** The break interrupt is associated with the character at the top of the UARTn RBR FIFO.	0
		0	Break interrupt status is inactive.	
		1	Break interrupt status is active.	
5	Transmitter Holding Register Empty (THRE))		THRE is set immediately upon detection of an empty UARTn THR and is cleared on a UnTHR write.	1
		0	UnTHR contains valid data.	
		1	UnTHR is empty.	
6	Transmitter Empty (TEMT)		TEMT is set when both UnTHR and UnTSR are empty; TEMT is cleared when either the UnTSR or the UnTHR contain valid data.	1
		0	UnTHR and/or the UnTSR contains valid data.	
		1	UnTHR and the UnTSR are empty.	

Introduction to Interrupts

An interrupt is a signal sent to the CPU which indicates that a system event has a occurred which needs immediate attention".

An '**Interrupt ReQuest**' i.e. an '**IRQ**' can be thought of as a special request to the CPU to execute a ***function***(small piece of code) when an interrupt occurs. This ***function*** or 'small piece of code' is technically called an '**Interrupt Service Routine**' or '**ISR**'.

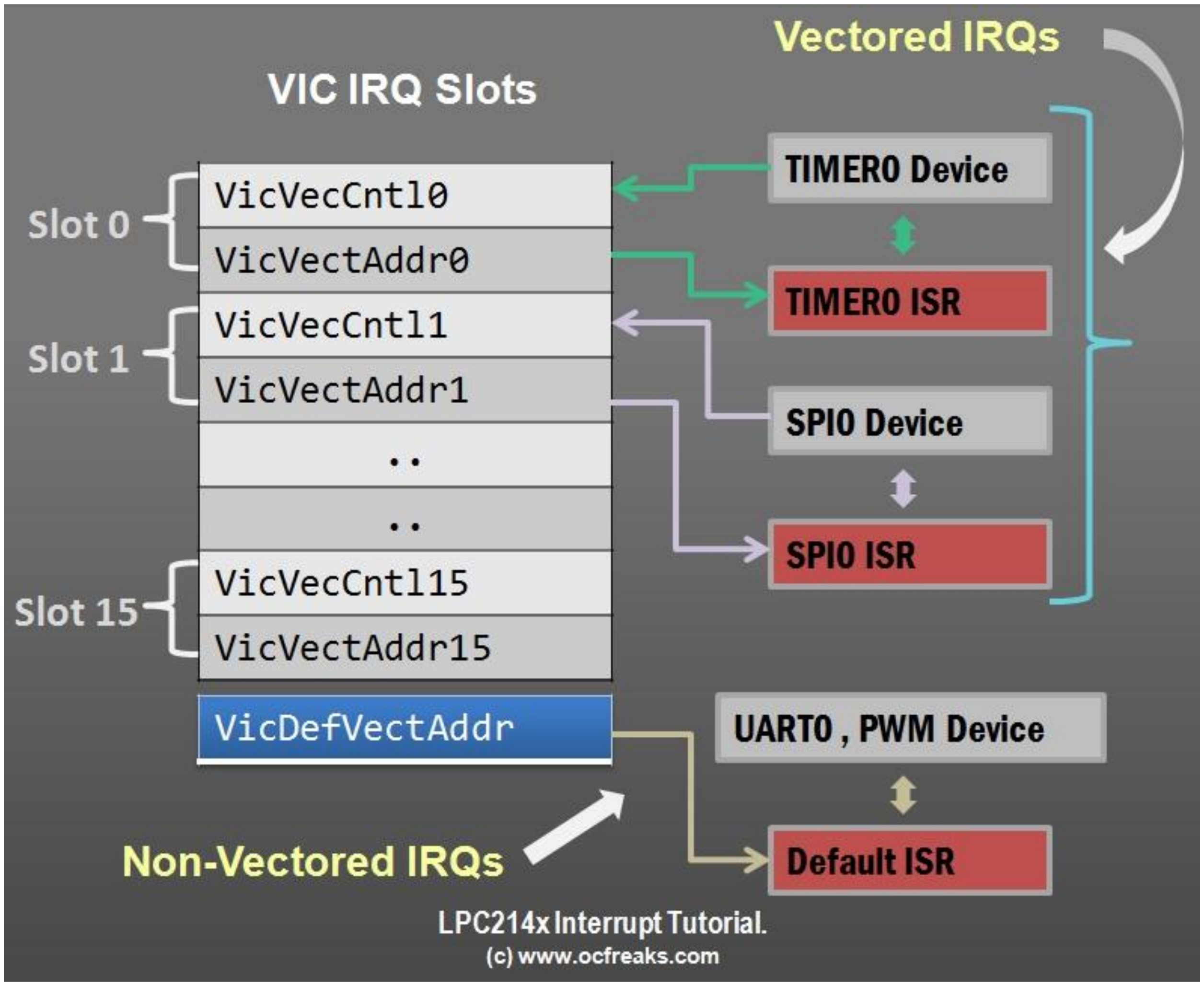
So when an IRQ arrives to the CPU, it stops executing the code current code and start executing the ISR. After the ISR execution has finished the CPU gets back to where it had stopped.

Interrupts in LPC214x are handled by **Vectored Interrupt Controller (VIC)** and are classified into 3 types based on the priority levels. (Though Interrupts can classified in many different ways as well)

1. **Fast Interrupt Request i.e FIQ** : which has highest priority
2. **Vectored Interrupt Request i.e Vectored IRQ**: which has 'middle' or priority between FIQ and Non-Vectored IRQ.
3. **Non-Vectored IRQ**: which has the lowest priority.

Better classify them as 2 types:

1. **Fast IRQs or FIQs**
2. **Normal IRQs or IRQs** which can be further classified as: ***Vectored IRQ and Non-Vectored IRQ.***



So what does Vectored mean?

In computing the term '**Vectored**' means that the CPU is aware of the address of the ISR when the interrupt occurs and

Non-Vectored means that CPU doesn't know the address of the ISR nor the source of the IRQ when the interrupt occurs and it needs to be supplied by the ISR address.

For the Vectored Stuff, the System internally maintains a table called **IVT or Interrupt Vector Table** which contains the information about Interrupts sources and their corresponding ISR address.

Difference between FIQ and IRQ

ARM calls `FIQ` the *fast interrupt*, with the implication that `IRQ` is *normal priority*. In any real system, there will be many more sources of interrupts than just two devices and there will therefore be some external hardware interrupt controller which allows masking, prioritization etc. of these multiple sources and which drives the interrupt request lines to the processor. To some extent, this makes the distinction between the two interrupt modes redundant and many systems do not use `nFIQ` at all, or use it in a way analogous to the non-maskable (`NMI`) interrupt found on other processors (although `FIQ` is software maskable on most ARM processors).

So why does ARM call FIQ "fast"?

1. FIQ mode has its own dedicated banked registers, `r8-r14`. R14 is the link register which holds the return address(+4) from the FIQ. But if your FIQ handler is able to be written such that it only uses `r8-r13`, it can take advantage of these banked registers in two ways:
   - One is that it does not incur the overhead of pushing and popping any registers that are used by the interrupt service routine (ISR). This can save a significant number of cycles on both entry and exit to the ISR.
   - Also, the handler can rely on values persisting in registers from one call to the next, so that for example `r8` may be used as a pointer to a hardware device and the handler can rely on the same value being in `r8` the next time it is called.
2. FIQ location at the end of the exception vector table (`0x1C`) means that if the FIQ handler code is placed directly at the end of the vector table, no branch is required - the code can execute directly from `0x1C`. This saves a few cycles on entry to the ISR.
3. FIQ has higher priority than IRQ. This means that when the core takes an FIQ exception, it automatically masks out IRQs. An IRQ cannot interrupt the FIQ handler.
   So why do many systems not use FIQ?

1. FIQ handler code typically cannot be written in C - it needs to be written directly in assembly language. If you care sufficiently about ISR performance to want to use FIQ, you probably wouldn't want to leave a few cycles on the table by coding in C in any case, but more importantly the C compiler will not produce code that follows the restriction on using only registers `r8-r13`. Code produced by a C compiler compliant with ARM's `ATPCS` procedure call standard will instead use registers `r0-r3` for scratch values and will not produce the correct `cpsr` restoring return code at the end of the function.
2. All of the interrupt controller hardware is typically on the IRQ pin. Using FIQ only makes sense if you have a single highest priority interrupt source connected to the nFIQ input and many systems do not have a single permanently highest priority source. There is no value connecting multiple sources to the FIQ and then having software prioritize between them as this removes nearly all the advantages the FIQ has over IRQ.

Table 41. VIC register map

Name	Description	Access	Reset value[1]	Address
VICIRQStatus	IRQ Status Register. This register reads out the state of those interrupt requests that are enabled and classified as IRQ.	RO	0	0xFFFF F000
VICFIQStatus	FIQ Status Requests. This register reads out the state of those interrupt requests that are enabled and classified as FIQ.	RO	0	0xFFFF F004
VICRawIntr	Raw Interrupt Status Register. This register reads out the state of the 32 interrupt requests / software interrupts, regardless of enabling or classification.	RO	0	0xFFFF F008
VICIntSelect	Interrupt Select Register. This register classifies each of the 32 interrupt requests as contributing to FIQ or IRQ.	R/W	0	0xFFFF F00C
VICIntEnable	Interrupt Enable Register. This register controls which of the 32 interrupt requests and software interrupts are enabled to contribute to FIQ or IRQ.	R/W	0	0xFFFF F010
VICIntEnClr	Interrupt Enable Clear Register. This register allows software to clear one or more bits in the Interrupt Enable register.	WO	0	0xFFFF F014
VICSoftInt	Software Interrupt Register. The contents of this register are ORed with the 32 interrupt requests from various peripheral functions.	R/W	0	0xFFFF F018
VICSoftIntClear	Software Interrupt Clear Register. This register allows software to clear one or more bits in the Software Interrupt register.	WO	0	0xFFFF F01C
VICProtection	Protection enable register. This register allows limiting access to the VIC registers by software running in privileged mode.	R/W	0	0xFFFF F020
VICVectAddr	Vector Address Register. When an IRQ interrupt occurs, the IRQ service routine can read this register and jump to the value read.	R/W	0	0xFFFF F030
VICDefVectAddr	Default Vector Address Register. This register holds the address of the Interrupt Service routine (ISR) for non-vectored IRQs.	R/W	0	0xFFFF F034
VICVectAddr0	Vector address 0 register. Vector Address Registers 0-15 hold the addresses of the Interrupt Service routines (ISRs) for the 16 vectored IRQ slots.	R/W	0	0xFFFF F100
VICVectAddr1	Vector address 1 register.	R/W	0	0xFFFF F104
VICVectAddr2	Vector address 2 register.	R/W	0	0xFFFF F108
VICVectAddr3	Vector address 3 register.	R/W	0	0xFFFF F10C
VICVectAddr4	Vector address 4 register.	R/W	0	0xFFFF F110
VICVectAddr5	Vector address 5 register.	R/W	0	0xFFFF F114
VICVectAddr6	Vector address 6 register.	R/W	0	0xFFFF F118
VICVectAddr7	Vector address 7 register.	R/W	0	0xFFFF F11C
VICVectAddr8	Vector address 8 register.	R/W	0	0xFFFF F120
VICVectAddr9	Vector address 9 register.	R/W	0	0xFFFF F124
VICVectAddr10	Vector address 10 register.	R/W	0	0xFFFF F128
VICVectAddr11	Vector address 11 register.	R/W	0	0xFFFF F12C

VICVectAddr12	Vector address 12 register.	R/W	0	0xFFFF F130
VICVectAddr13	Vector address 13 register.	R/W	0	0xFFFF F134
VICVectAddr14	Vector address 14 register.	R/W	0	0xFFFF F138
VICVectAddr15	Vector address 15 register.	R/W	0	0xFFFF F13C
VICVectCntl0	Vector control 0 register. Vector Control Registers 0-15 each control one of the 16 vectored IRQ slots. Slot 0 has the highest priority and slot 15 the lowest.	R/W	0	0xFFFF F200
VICVectCntl1	Vector control 1 register.	R/W	0	0xFFFF F204
VICVectCntl2	Vector control 2 register.	R/W	0	0xFFFF F208
VICVectCntl3	Vector control 3 register.	R/W	0	0xFFFF F20C
VICVectCntl4	Vector control 4 register.	R/W	0	0xFFFF F210
VICVectCntl5	Vector control 5 register.	R/W	0	0xFFFF F214
VICVectCntl6	Vector control 6 register.	R/W	0	0xFFFF F218
VICVectCntl7	Vector control 7 register.	R/W	0	0xFFFF F21C
VICVectCntl8	Vector control 8 register.	R/W	0	0xFFFF F220
VICVectCntl9	Vector control 9 register.	R/W	0	0xFFFF F224
VICVectCntl10	Vector control 10 register.	R/W	0	0xFFFF F228
VICVectCntl11	Vector control 11 register.	R/W	0	0xFFFF F22C
VICVectCntl12	Vector control 12 register.	R/W	0	0xFFFF F230
VICVectCntl13	Vector control 13 register.	R/W	0	0xFFFF F234
VICVectCntl14	Vector control 14 register.	R/W	0	0xFFFF F238
VICVectCntl15	Vector control 15 register.	R/W	0	0xFFFF F23C

7.4.7 IRQ Status register (VICIRQStatus - 0xFFFF F000)

This is a read only register. This register reads out the state of those interrupt requests that are enabled and classified as IRQ. It does not differentiate between vectored and non-vectored IRQs.

Table 54. IRQ Status register (VICIRQStatus - address 0xFFFF F000) bit allocation
Reset value: 0x0000 0000

Bit	31	30	29	28	27	26	25	24
Symbol	-	-	-	-	-	-	-	-
Access	RO	RO	RO	RO	RO	RO	RO	RO
Bit	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**
Symbol	-	USB	AD1	BOD	I2C1	AD0	EINT3	EINT2
Access	RO	RO	RO	RO	RO	RO	RO	RO
Bit	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**
Symbol	EINT1	EINT0	RTC	PLL	SPI1/SSP	SPI0	I2C0	PWM0
Access	RO	RO	RO	RO	RO	RO	RO	RO
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Symbol	UART1	UART0	TIMER1	TIMER0	ARMCore1	ARMCore0	-	WDT
Access	RO	RO	RO	RO	RO	RO	RO	RO

Table 55. IRQ Status register (VICIRQStatus - address 0xFFFF F000) bit description

Bit	Symbol	Description	Reset value
31:0	See VICIRQStatus bit allocation table.	A bit read as 1 indicates a corresponding interrupt request being enabled, classified as IRQ, and asserted	0

7.4.4 Interrupt Enable register (VICIntEnable - 0xFFFF F010)

This is a read/write accessible register. This register controls which of the 32 interrupt requests and software interrupts contribute to FIQ or IRQ.

Table 48. Interrupt Enable register (VICIntEnable - address 0xFFFF F010) bit allocation
Reset value: 0x0000 0000

Bit	31	30	29	28	27	26	25	24
Symbol	-	-	-	-	-	-	-	-
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Bit	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**
Symbol	-	USB	AD1	BOD	I2C1	AD0	EINT3	EINT2
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Bit	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**
Symbol	EINT1	EINT0	RTC	PLL	SPI1/SSP	SPI0	I2C0	PWM0
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Symbol	UART1	UART0	TIMER1	TIMER0	ARMCore1	ARMCore0	-	WDT
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

7.4.12 Vector Address register (VICVectAddr - 0xFFFF F030)

This is a read/write accessible register. When an IRQ interrupt occurs, the IRQ service routine can read this register and jump to the value read.

Table 61. Vector Address register (VICVectAddr - address 0xFFFF F030) bit description

Bit	Symbol	Description	Reset value
31:0	IRQ_vector	If any of the interrupt requests or software interrupts that are assigned to a vectored IRQ slot is (are) enabled, classified as IRQ, and asserted, reading from this register returns the address in the Vector Address Register for the highest-priority such slot (lowest-numbered) such slot. Otherwise it returns the address in the Default Vector Address Register. Writing to this register does not set the value for future reads from it. Rather, this register should be written near the end of an ISR, to update the priority hardware.	0x0000 0000

7.4.10 Vector Address registers 0-15 (VICVectAddr0-15 - 0xFFFF F100-13C)

These are a read/write accessible registers. These registers hold the addresses of the Interrupt Service routines (ISRs) for the 16 vectored IRQ slots.

Table 59. Vector Address registers (VICVectAddr0-15 - addresses 0xFFFF F100-13C) bit description

Bit	Symbol	Description	Reset value
31:0	IRQ_vector	When one or more interrupt request or software interrupt is (are) enabled, classified as IRQ, asserted, and assigned to an enabled vectored IRQ slot, the value from this register for the highest-priority such slot will be provided when the IRQ service routine reads the Vector Address register -VICVectAddr (Section 7.4.10).	0x0000 0000

7.4.9 Vector Control registers 0-15 (VICVectCntl0-15 - 0xFFFF F200-23C)

These are a read/write accessible registers. Each of these registers controls one of the 16 vectored IRQ slots. Slot 0 has the highest priority and slot 15 the lowest. Note that disabling a vectored IRQ slot in one of the VICVectCntl registers does not disable the interrupt itself, the interrupt is simply changed to the non-vectored form.

Table 58. Vector Control registers 0-15 (VICVectCntl0-15 - 0xFFFF F200-23C) bit description

Bit	Symbol	Description	Reset value
4:0	int_request/ sw_int_assig	The number of the interrupt request or software interrupt assigned to this vectored IRQ slot. As a matter of good programming practice, software should not assign the same interrupt number to more than one enabled vectored IRQ slot. But if this does occur, the lower numbered slot will be used when the interrupt request or software interrupt is enabled, classified as IRQ, and asserted.	0
5	IRQslot_en	When 1, this vectored IRQ slot is enabled, and can produce a unique ISR address when its assigned interrupt request or software interrupt is enabled, classified as IRQ, and asserted.	0
31:6	-	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

Vectored Interrupt Controller in LPC2148

VIC , as per its design , can take **32** interrupt request inputs but only **16** requests can be assigned to Vectored IRQ interrupts in its LCP214x Implementation. We are given a set of **16 vectored IRQ slots** to which we can assign any of the **22** requests that are available in LPC2148. The slot numbering goes from **0 to 15** with **slot no. 0 having highest priority and slot no. 15 having lowest priority**.

Example: For example if you working with 2 interrupt sources say.. UART0 and TIMER0. Now if you want to give TIMER0 a higher priority than UART0 .. then assign TIMER0 interrupt a lower number slot than UART0 Like .. hook up TIMER0 to slot 0 and UART0 to slot 1 or TIMER0 to slot 4 and UART to slot 9 or whatever slots you like. The number of the slot doesn't matter as long TIMER0 slot is lower than UART0 slot.

Code:

```
VICVectAddr0=(unsigned long) uart1_isr;

VICVectCntl0=0x20|7;

VICIntEnable|=(1<<7);


void uart1_isr(void) __irq

{

        if(U1IIR==0x04)

        {

                ch=U1RBR;

                flag=1;

        }

        VICVectAddr=0x00;

}
```

Nesting of Interrupts in LPC2148

Nested interrupts
LPC2148

SPI

SPI Overview

- SPI stands for Serial Peripheral Interface
- designed by Motorola
- four wire protocol
- Full Duplex protocol
- Low power than I^2C (no need of Pull ups)
- Supports Single master and multiple slaves
- No hardware slave acknowledgement

SCK: Serial Clock, provided by Master

MOSI: Master Out Slave In

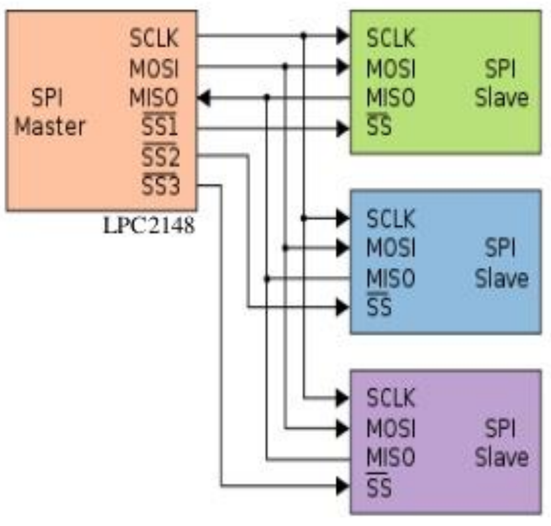
MISO: Master In Slave Out

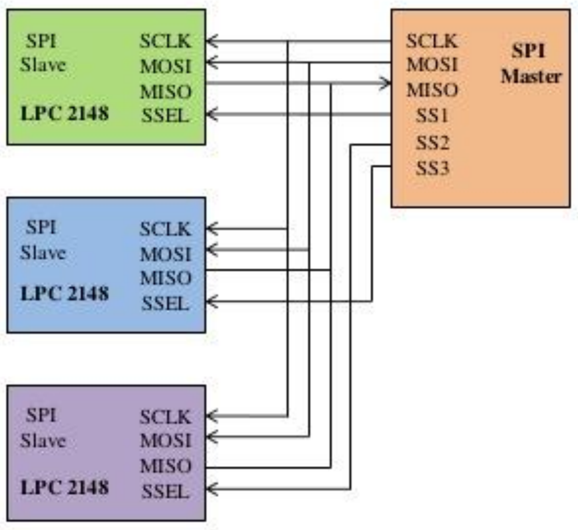
SS: Slave Select

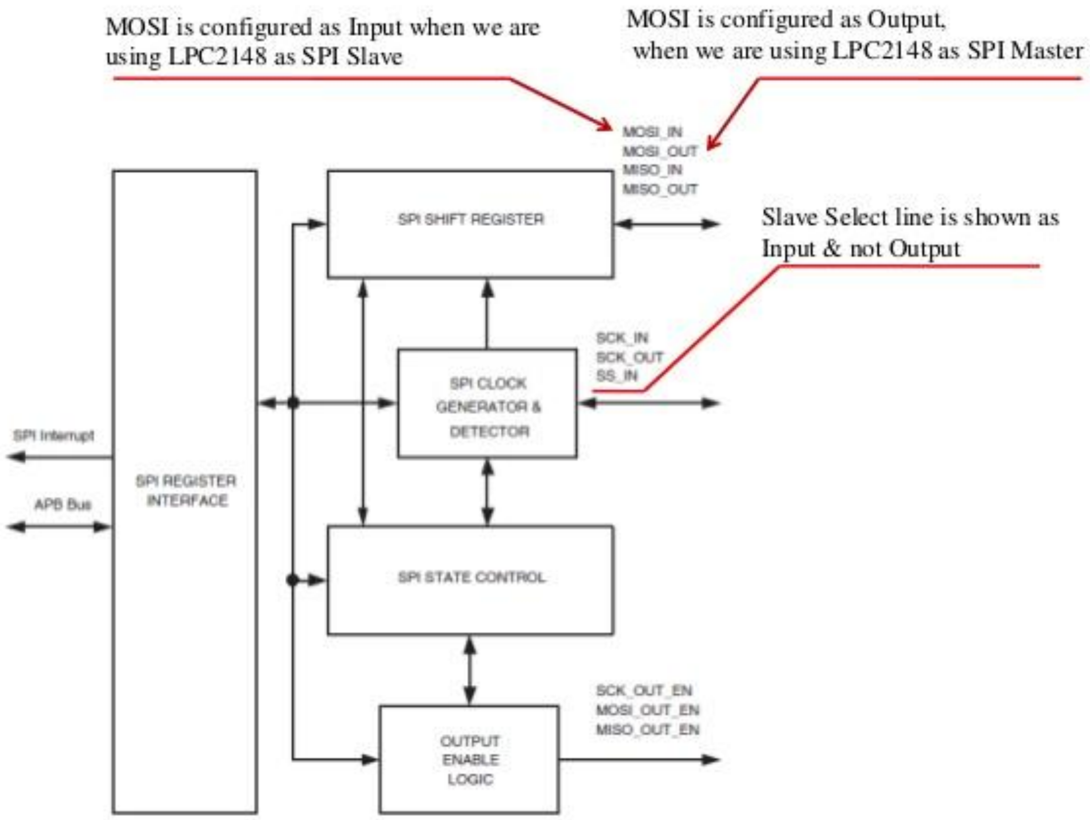
Data Transfer Modes of LPC2148

It can be broadly divided into two categories

- Master Mode
- Slave Mode





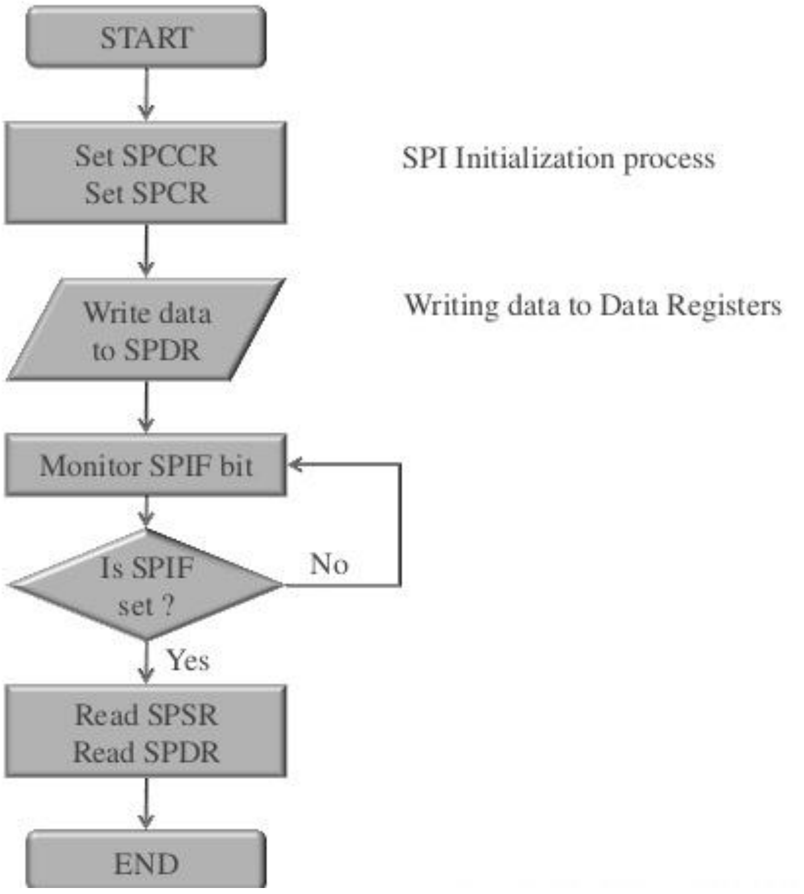

MOSI is configured as Input when we are using LPC2148 as SPI Slave
MOSI is configured as Output, when we are using LPC2148 as SPI Master
MOSI_IN
MOSI_OUT
MISO_IN
MISO_OUT
SPI SHIFT REGISTER
Slave Select line is shown as Input & not Output
SCK_IN
SCK_OUT
SS_IN
SPI CLOCK GENERATOR & DETECTOR
SPI Interrupt
APB Bus
SPI REGISTER INTERFACE
SPI STATE CONTROL
SCK_OUT_EN
MOSI_OUT_EN
MISO_OUT_EN
OUTPUT ENABLE LOGIC

Steps for Master Mode

- Set SPCCR (Clock Counter Register) with appropriate clock frequency
- Set SPCR (Control Register) to desired settings
- Write the data into SPDR (Data Register) of SPI
- Monitor SPIF bit of SPCR register, until it sets to 1(SPIF = 1 means data transfer is complete).
- Read SPSR (Status Register), this will clear SPIF bit
- Read SPDR (Data Register), reading data register will return data sent by the slave during last transfer.

Steps for Slave Mode

- Clock will be decided by Master
- Set SPCR (Control Register) to desired settings
- Write **dummy data** into SPDR (Data Register), so that SPI control block will generate Clock.
- Monitor SPIF bit of SPCR register, until it sets to 1
- Read SPSR (Status Register), this will clear SPIF bit
- Read SPDR (Data Register), reading data register will return data sent by the slave during last transfer.


START
Set SPCCR
Set SPCR
SPI Initialization process
Write data
to SPDR
Writing data to Data Registers
Monitor SPIF bit
Is SPIF
set ?
No
Yes
Read SPSR
Read SPDR
END

Clock Counter Register (SPCCR)

The SPCCR is used to set data transfer rate (SPI Frequency)
It is **8 bit register**, and the value entered in this register is used to calculate frequency

SPI data rate = (PCLK / SPCCR value)

The value entered must be ***even value***, thus LSB must be 0.
The value should be ***greater*** than **8**.
So maximum frequency is 1.875 MHz(PCLK = 15MHz).

Data Register (SPDR)

- It is 16 bit register used in data transfer, the data length is selectable (8 – 16bits).
- The data length can be configured by using bit 2 (BitEnable) and bits 11:8 of control register.
- There is no buffer between the data register and the internal shift register.
- A write to the data register goes directly into the internal shift register.
- Therefore, data should only be written to this register when a transmit is not currently in progress. Otherwise a Collision Error may occur.
- Read data is buffered. When a transfer is complete, the receive data is transferred to a single byte data buffer, where it is later read.

SPCR (Control Register)

Bit 2: BitEnable

This bit can be used to select the data length for SPI protocol
The minimum data length is 8bits and maximum 16bits
When BitEnable = 0 , data length is fixed & considered to be 8bits
When BitEnable = 1 , data length depends upon the combinations of 11:8 bits of SPCR

For example, we are using SPI compatible 12 bit ADC
In this case, the ADC result will be of 12 bits

BitEnable = 1
Bits 11: 8 = 1100 (Combination for 12 bits)

Bit 3: CPHA

CPHA stands for Clock Phase Control, and plays an important role in deciding the relation between sampling of data and clock pulse

CPHA = 0 ; data is sampled on first clock edge
CPHA = 1 ; data is sampled on the second rising edge

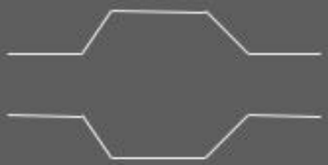
- *The important thing to be considered here is CPHA should be 0 when LPC2148 is used as a SPI Master.*
- *It is mentioned in the User Manual of LPC2148 that SSEL signal is inactive during the data transfer when CPHA is 0, but when CPHA is 1 the SSEL signal becomes active and immediately transforms itself into slave.*
- *This results into a Mode Fault and data transfer terminates.*
- *In this case MODF bit of Status Register will set to 1.*

Bit 4: CPOL

- *CPOL stands for* ***Clock Polarity Control***
- *The bit decides the polarity of SPI clock, when set to 1 clock is active low. In that case*
- *first clock will start with negative going pulse*
- *when set to 0 the clock is active high meaning that the clock will start with positive*
- *going edge*

CPOL = 0

CPOL = 1



Bit 5: MSTR

- *The bit is used to configure SPI block in Master/Slave Mode*
- *MSTR = 0   Slave Mode*
- *MSTR = 1   Master Mode*

Bit 6: LSBF

- *The bit decides the direction of bit transfer*
- *LSBF = 0 MSB is transferred first*
- *LSBF = 1 LSB is transferred first*

Bit 7: SPIE

- *It is an interrupt Enable bit,*
- *SPIE = 0   Interrupt are disabled*
- *SPIE = 1   Interrupts are enabled and will occur when SPI/ WCOL bit is set*

Bit 11:8 BITS

When bit 2 of this register is 1, this field controls the number of bits per transfer:

1000	8 bits per transfer
1001	9 bits per transfer
1010	10 bits per transfer
1011	11 bits per transfer
1100	12 bits per transfer
1101	13 bits per transfer
1110	14 bits per transfer
1111	15 bits per transfer
0000	16 bits per transfer

Example: The target device is 12bit ADC, which is slave and LPC2148 is Master

Control Register (SPCR)

Bit 0	1	2	3	4	5	6	7	8	9	10	11
-	-	1	0	0	1	0	0	1	1	0	0

Some Imp points…

a read or write of the SPI data register is required in order to clear the SPIF status bit.

The prime function of SPSR register is to indicate the completion of data transfer between two devices. The remaining bits of SPSR register

When CPHA = 0, the SSEL signal will always go inactive between data transfers. This is not guaranteed when CPHA = 1 (the signal can remain active). (pg 169)

PINSEL0

9:8	P0.4	00	GPIO Port 0.4	0
		01	SCK0 (SPI0)	
		10	Capture 0.1 (Timer 0)	
		11	AD0.6	
11:10	P0.5	00	GPIO Port 0.5	0
		01	MISO0 (SPI0)	
		10	Match 0.1 (Timer 0)	
		11	AD0.7	
13:12	P0.6	00	GPIO Port 0.6	0
		01	MOSI0 (SPI0)	
		10	Capture 0.2 (Timer 0)	
		11	Reserved[1][2] or AD1.0[3]	

6.4.4 Pin function select register values

The PINSEL registers control the functions of device pins as shown below. Pairs of bits in these registers correspond to specific device pins.

Table 40. Pin function select register bits

PINSEL0 and PINSEL1 Values	Function	Value after Reset
00	Primary (default) function, typically GPIO port	00
01	First alternate function	
10	Second alternate function	
11	Reserved	

The direction control bit in the IO0DIR/IO1DIR register is effective only when the GPIO function is selected for a pin. For other functions, direction is controlled automatically. Each derivative typically has a different pinout and therefore a different set of functions possible for each pin. Details for a specific derivative may be found in the appropriate data sheet.

8.4.1 GPIO port Direction register (IODIR, Port 0: IO0DIR - 0xE002 8008 and Port 1: IO1DIR - 0xE002 8018; FIODIR, Port 0: FIO0DIR - 0x3FFF C000 and Port 1:FIO1DIR - 0x3FFF C020)

This word accessible register is used to control the direction of the pins when they are configured as GPIO port pins. Direction bit for any pin must be set according to the pin functionality.

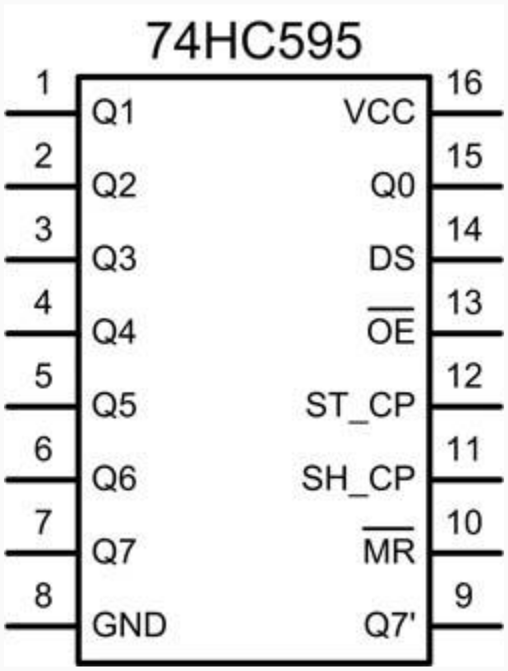
Legacy registers are the IO0DIR and IO1DIR, while the enhanced GPIO functions are supported via the FIO0DIR and FIO1DIR registers.

Table 67. GPIO port 0 Direction register (IO0DIR - address 0xE002 8008) bit description

Bit	Symbol	Value	Description	Reset value
31:0	P0xDIR		Slow GPIO Direction control bits. Bit 0 controls P0.0 ... bit 30 controls P0.30.	0x0000 0000
		0	Controlled pin is input.	
		1	Controlled pin is output.	

74HC595 Shift Register

The 74HC595 shift register has an 8 bit storage register and an 8 bit shift register. Data is written to the shift register serially, then latched onto the storage register. The storage register then controls 8 output lines. The figure below shows the 74HC595 pinout.

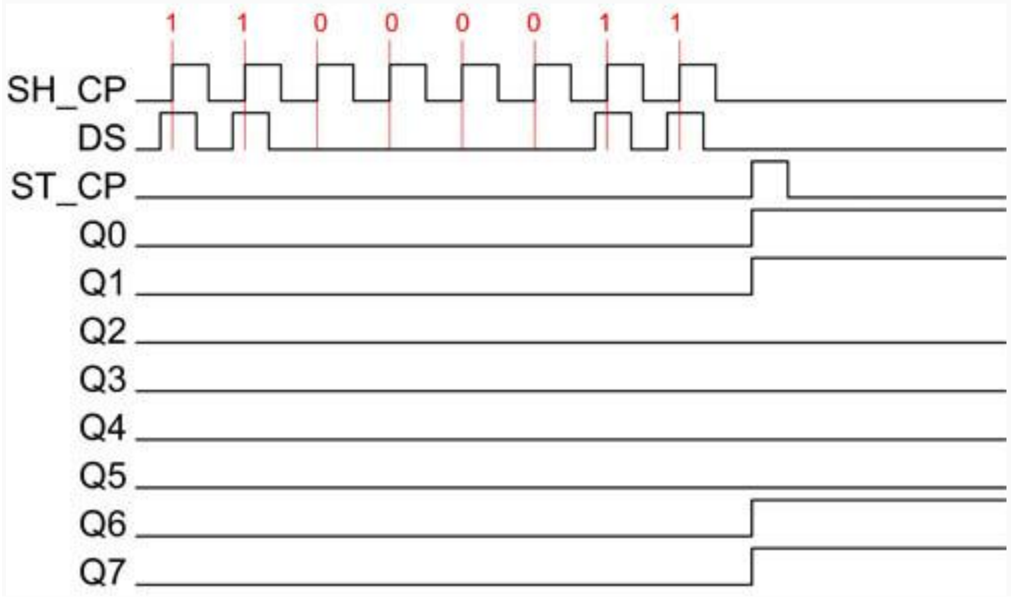


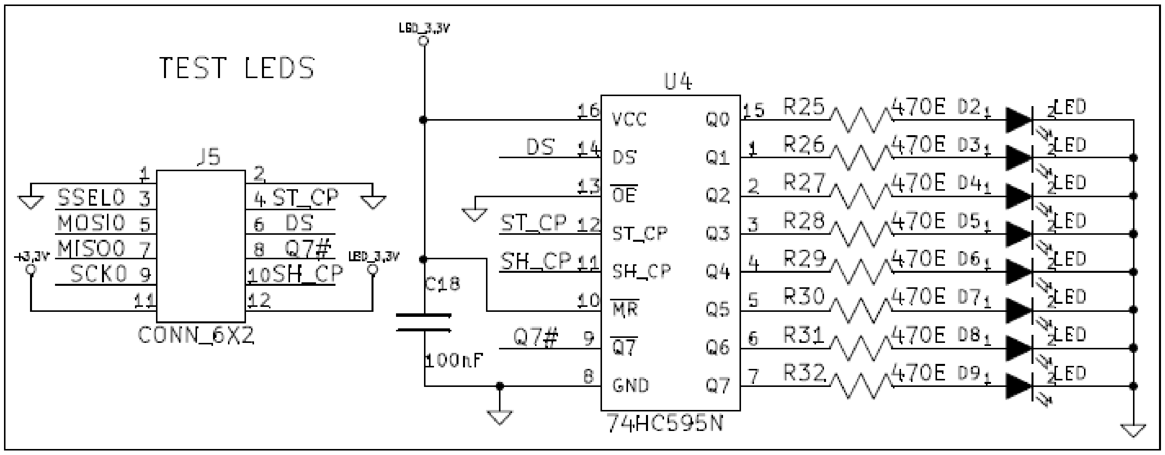
Pin 14 (DS) is the Data pin. On some datasheets it is referred to as "SER".

When pin 11 (SH_CP or SRCLK on some datasheets) goes from Low to High the value of DS is stored into the shift register and the existing values of the register are shifted to make room for the new bit.

Pin 12 (ST_CP or RCLK on some datasheets) is held low while data is being written to the shift register. When it goes High the values of the shift register are latched to the storage register which are then outputted to pins Q0-Q7.

The timing diagram below demonstrates how you would set the Q0-Q7 output pins to 11000011, assuming starting values of 00000000.





Programming SPI

```
void spi_init(void)
{
//Enable PINSEL0 Functionality
//Set Direction of pin0.7 to output
//Enable Master Mode
// Set baud rate
}
void SPI_data(unsigned char data)
{
    //Set bit 7
    S0SPDR=data;
    //polling condition
    //Clear bit 7
}
```

Spi Interrupt

spi_interrupt.c

ADC

Using potentiometer analog value is converted into digital by adc and sent to uart

adc.c

PLL

Lpc214x MCUs have 2 PLL blocks viz. PPL0 and PLL1. PLL0 is used to generate the System Clock which goes to CPU and on-chip peripherals while PPL1 is strictly for USB. PLL interrupts are only available for PLL0 and not for PLL1. Input clock to both the PLLs must be between 10 MHz to 25 MHz strictly. This input clock is multiplied with a suitable multiplier and scaled accordingly. But here we have a upper limit of 60Mhz which is the maximum frequency of operation for lpc214x MCUs.

FOSC	**=> frequency from the crystal oscillator(XTAL)/external clock**
FCCO	**=> frequency of the PLL Current Controlled Oscillator(CCO)**
CCLK	**=> PLL output frequency (CPU Clock)**
M	**=> PLL Multiplier value from the MSEL bits in the PLLCFG register**
P	**=> PLL Divider value from the PSEL bits in the PLLCFG register**
PCLK	**=> Peripheral Clock which is derived from CCLK**

PLL output frequency (CPU Clock)is given by the formula:

- **CCLK = M x FOSC**

  or

- **CCLK = FCCO / (2 x P)**

Frequency of the PLL Current Controlled Oscillator(CCO) is given by:

- **FCCO = CCLK x 2 x P**

  or

- **FCCO = FOSC x M x 2 x P**

The PLL inputs and settings must meet the following:

- FOSC is in the range of **10 MHz to 25 MHz.**
- CCLK is in the range of **10 MHz to Fmax** (the maximum allowed frequency for the microcontroller – determined by the system microcontroller is embedded in).

Multiplier and Divider

To play with PLLs we are given a Multiplier and a Divider which decide the output frequency/clock form PLL block. But we must play cautiously with PLLs since the CPU and other MCU peripherals operate on the clock provided buy it.

Feed Sequence

To access PLL we need to have a **key** in order to use or configure it. The key here is the '**Feed Sequence'**. Feed Sequence is nothing but assignment of 2 particular 'fixed' values to a register related to the PLL block. This register is called '**PLL0FEED**'. And those fixed values **are 0xAA and 0x55 are 'in order'!.** Hence the code for feed sequence must be :

```
PLL0FEED = 0xAA;
PLL0FEED = 0x55;
```

Registers for PLL

1)PLL0CON: This is the PLL Control register used to enable and 'connect' the PLL.

The first bit is used to 'Enable' PLL.
The second bit is used to 'Connect' the PLL.

During Boot Up

Note #1: Initially when the MCU boots-up it uses its internal RC Oscillator as clock source.

#2When the PLL is setup and enabled we must switch from internal RC Oscillator to the PLL's output clock. This is referred to as 'Connecting' the PLL.

#3PLL is only connected when both 'Enable' and 'Connect' bits are 1 and a valid feed sequence is applied.

2)PLL0CGF: The multiplier and divider values are stored in this register.

The 1st 5 bits (called **MSEL**) are used to store the **Multiplier(M).**
Next 2 bits i.e 5,6 (called **PSEL**) are used to store the value of the **Divider(P).**

3)PLL0STAT: This is a read only register and contains current status of PLL enable , connect , M and P values. You can read more about the bits in datasheet.

We are more interested in the 10th bit of PLL0STAT since it contains the lock status. When we setup and enable the PLL it takes a while for PLL to latch on the target frequency. After PLL has latched or locked to target frequency this bit becomes 1. Only after this we can connect PLL to the CPU. Hence , ***we need to wait for PLL to lock on to target frequency.***

The **Order** of setting up PLLs is **strictly** as follows :

1. ***Setup PLL***
2. ***Apply Feed Sequence***
3. ***Wait for PLL to lock and then Connect PLL***
4. ***Apply Feed Sequence***

Calculating PLL Settings

#1) PLL0

1. Select the CPU Frequency. This selection may be based many factors and end application. On Chip peripheral devices may be running on a lower clock than the processor.
2. Choose an oscillator frequency (FOSC). CCLK must be the whole (non-fractional) multiple of FOSC. Which leaves us with a few values of Xtal for the selected CPU frequency (CCLK).
3. Calculate the value of M to configure the MSEL bits. M = CCLK / FOSC. M must be in the range of 1 to 32. The value written to the MSEL bits in PLLCFG is M − 1.
4. Find a value for P to configure the PSEL bits, such that FCCO is within its defined frequency limits. FCCO is calculated using the equation given above. P must have one of the values 1, 2, 4, or 8.

Values of PSEL for P are :

P	Value**(binary) in** PSEL Bit(5,6)
1	00
2	01
4	10
8	11

Now , Since there is no point in running the MCU at lower speeds than 60Mhz (except in some situations) ill put a table giving values for M and P for different crystal frequencies i.e FOSC.

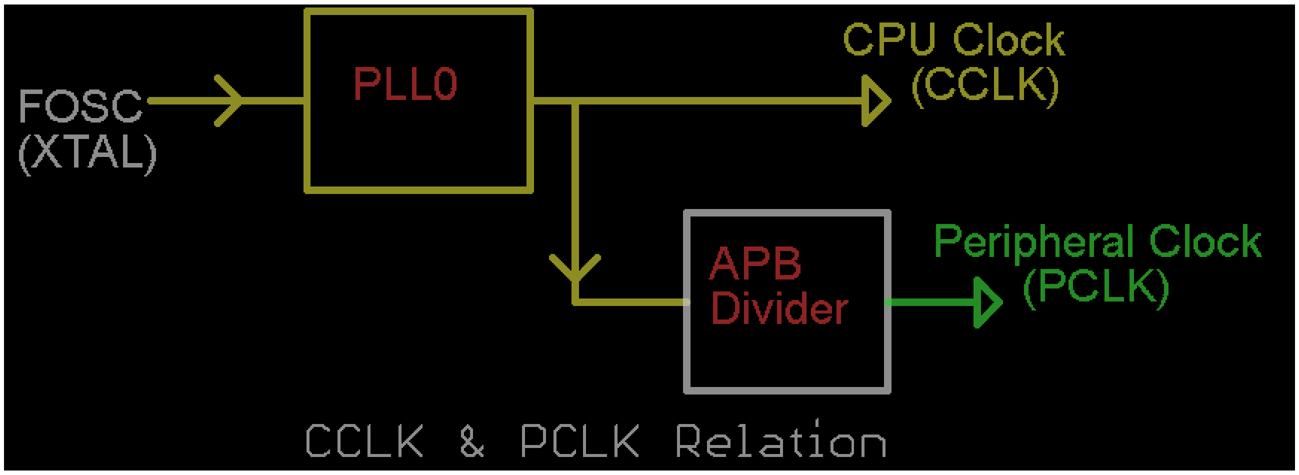
The Value of P depends on the selection of CCLK. So , for CCLK=60Mhz we get P=2 from the equation P = FCCO/(2xCCLK).
Since we want FCCO in range 156Mhz to 320Mhz first substitute FCCO = 156 and we get P =

1.3. Now substituting the maximum value for FCCO i.e 320Mhz we get P = 2.67. Now , P must be an integer between 1.3 and 2.67. So we will use P=2.

FOSC	M	Value in MSEL (M-1)	P	Value in PSEL	Final Value in PLL0CFG
5Mhz	12	11 = [0xB]	2	01	0x2B
10Mhz	6	5 = [0x5]	2	01	0x25
12Mhz	5	4 = [0x4]	2	01	0x24
15Mhz	4	3 = [0x3]	2	01	0x23
20Mhz	3	2 = [0x2]	2	01	0x22

Setting-up Peripheral Clock (PCLK)

The Peripheral Clock i.e. PCLK is derived from CPU Clock i.e. CCLK. The APB Divider decides the operating frequency of PCLK. The input to APB Divider is CCLK and output is PCLK.



By Default PCLK runs at 1/4th the speed of CCLK. To control APB Divider we have a register called VPBDIV. The value in VPBDIV controls the division of CCLK to generate PCLK as shown below:

VPBDIV=**0x00;**	APB bus clock (PCLK) is one fourth of the processor clock (CCLK)
VPBDIV=**0x01;**	APB bus clock (PCLK) is the same as the processor clock (CCLK)
VPBDIV=**0x02;**	APB bus clock (PCLK) is one half of the processor clock (CCLK)
VPBDIV=**0x03;**	Reserved. If this value is written to the APBDIV register, it has no effect (the previous setting is retained).

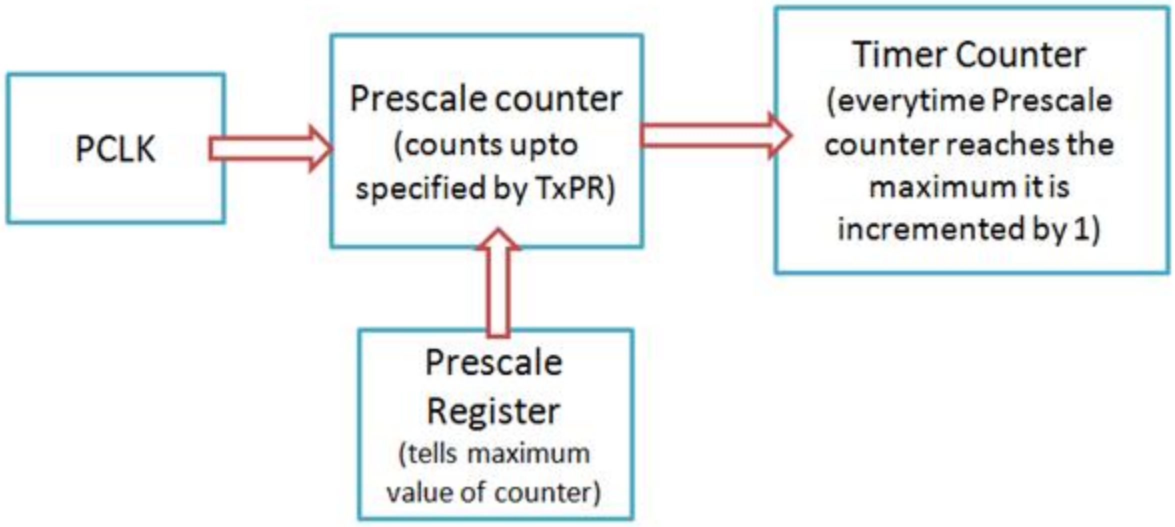
PLL Code:

PLL.c

Timer

- LPC2148 comes loaded with two 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.

- A timer has a **Timer Counter (TC)** and **Prescale Register (PR)** associated with it. When Timer is Reset and Enabled TC is set to 0 and incremented by 1 every **'PR+1'** clock cycles. When it reaches its maximum value it gets reset to 0 and hence restarts counting.


PCLK
Prescale counter (counts upto specified by TxPR)
Timer Counter (everytime Prescale counter reaches the maximum it is incremented by 1)
Prescale Register (tells maximum value of counter)

- Prescale Register is used to define the **resolution** of the timer. If **PR=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**.

The Register Specifics

What is a Match Register anyways ?

Ans: 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.

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.

Up to four external outputs corresponding to match registers with following capabilities

- Set low on match
- Set high on match
- Toggle on match
- Do nothing on match

Now let’s see some of the important registers concerned mainly with timer operation.

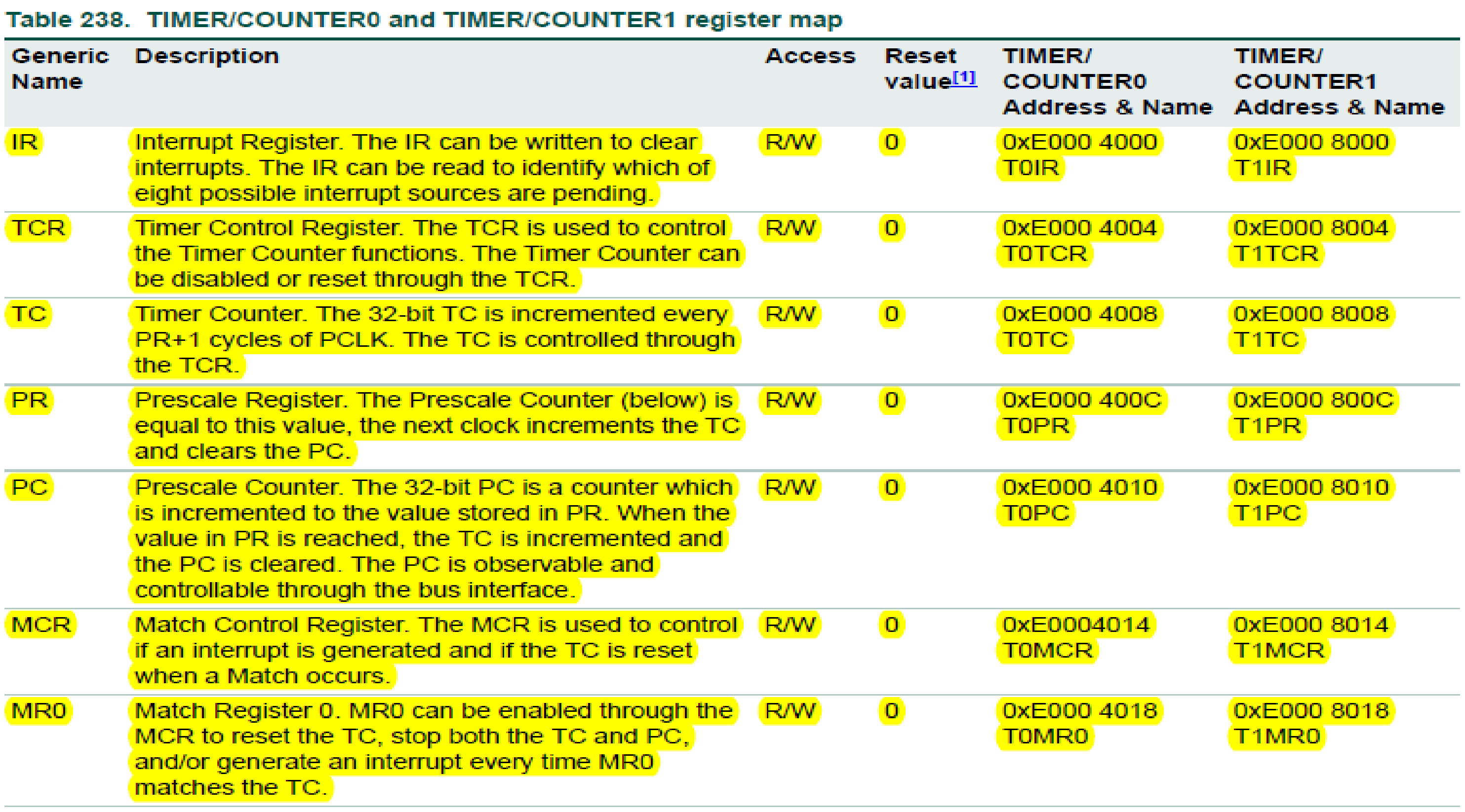
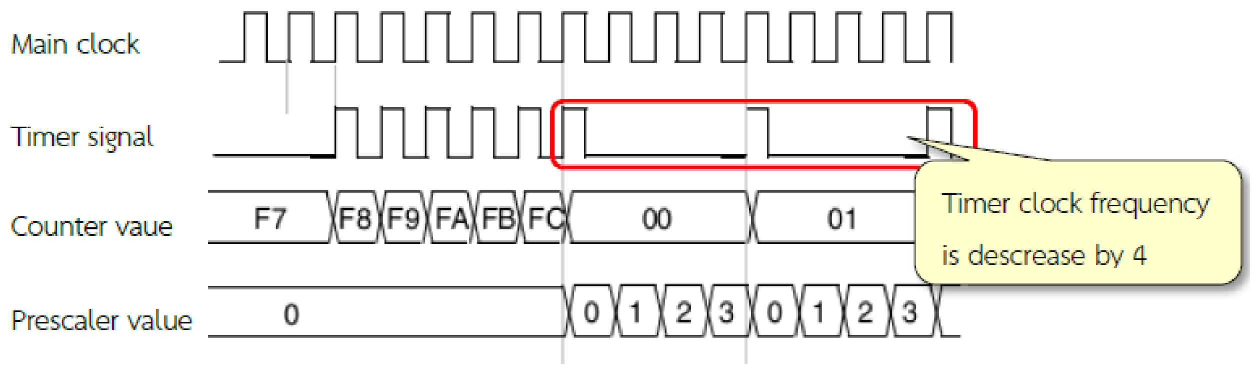


Table 238. TIMER/COUNTER0 and TIMER/COUNTER1 register map

Generic Name	Description	Access	Reset value[1]	TIMER/ COUNTER0 Address & Name	TIMER/ COUNTER1 Address & Name
IR	Interrupt Register. The IR can be written to clear interrupts. The IR can be read to identify which of eight possible interrupt sources are pending.	R/W	0	0xE000 4000 T0IR	0xE000 8000 T1IR
TCR	Timer Control Register. The TCR is used to control the Timer Counter functions. The Timer Counter can be disabled or reset through the TCR.	R/W	0	0xE000 4004 T0TCR	0xE000 8004 T1TCR
TC	Timer Counter. The 32-bit TC is incremented every PR+1 cycles of PCLK. The TC is controlled through the TCR.	R/W	0	0xE000 4008 T0TC	0xE000 8008 T1TC
PR	Prescale Register. The Prescale Counter (below) is equal to this value, the next clock increments the TC and clears the PC.	R/W	0	0xE000 400C T0PR	0xE000 800C T1PR
PC	Prescale Counter. The 32-bit PC is a counter which is incremented to the value stored in PR. When the value in PR is reached, the TC is incremented and the PC is cleared. The PC is observable and controllable through the bus interface.	R/W	0	0xE000 4010 T0PC	0xE000 8010 T1PC
MCR	Match Control Register. The MCR is used to control if an interrupt is generated and if the TC is reset when a Match occurs.	R/W	0	0xE0004014 T0MCR	0xE000 8014 T1MCR
MR0	Match Register 0. MR0 can be enabled through the MCR to reset the TC, stop both the TC and PC, and/or generate an interrupt every time MR0 matches the TC.	R/W	0	0xE000 4018 T0MR0	0xE000 8018 T1MR0

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:

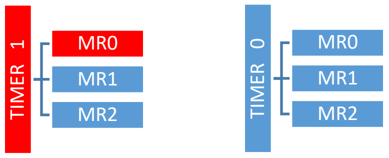
For MR0:
Bit 0 : Interrupt on MR0 i.e trigger an interrupt when MR0 matches TC. Interrupts are enabled when set to 1 and disabled when set to 0.

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.



Scenario of Polling

Case 1

```
while(T1TC < milliseconds); //wait until timer counter reaches the desired delay

T1TCR = 0x00; //Disable timer
```

Case 2

```
T1MCR =(1<<5);

T1MR1 =milliseconds;

T1TCR = 0x02; //Reset Timer

T1TCR = 0x01; //Enable timer

 while(!(T1TC==milliseconds));

 T1TCR = 0x00; //Disable timer
```

Case 3

```
while(T1TC<T1MR0);

 T1TCR = 0x00; //Disable timer
```

Code Explanation

Timer_Interrupt   Timer_Polling.c   Timer_polling.c

Vector Interrupt Controller

The difference between Vectored IRQ(VIRQ) and Non-Vectored IRQ(NVIRQ) is that VIRQ has dedicated IRQ service routine for each interrupt source which while NVIRQ has the same IRQ service routine for all Non-Vectored Interrupts. You will get a clear picture when I cover some examples in examples section.

Note : VIC (in ARM CPUs & MCUs) , as per its design , can take **32** interrupt request inputs but only **16** requests can be assigned to Vectored IRQ interrupts in its LCP2148 ARM7 Implementation. We are given a set of **16 vectored IRQ slots** to which we can assign any of the **22** requests that are available in LPC2148. The slot numbering goes from **0 to 15** with **slot no. 0 having highest priority and slot no. 15 having lowest priority**.

Example: For example if you working with 2 interrupt sources say.. UART0 and TIMER0. Now if you want to give TIMER0 a higher priority than UART0 .. then assign TIMER0 interrupt a lower number slot than UART0 Like .. hook up TIMER0 to slot 0 and UART0 to slot 1 or TIMER0 to slot 4 and UART to slot 9 or whatever slots you like. The number of the slot doesn't matter as long TIMER0 slot is lower than UART0 slot.

VIC has plenty of registers. Most of the registers that are used to configure interrupts or read status have each bit corresponding to a particular interrupt source and **this mapping is same for all of *these* registers**. What I mean is that bit 0 in these registers corresponds to Watch dog timer interrupt , bit 4 corresponds to TIMER0 interrupt , bit 6 corresponds to UART0 interrupt .. and so on.

Here is the complete table which says which bit corresponds to which interrupt source as given in Datasheet:

Table 1:

Bit#	22	21	20	19	18	17	16	15	14	13	12	11
IRQ	USB	AD1	BOD	I2C1	AD0	EINT3	EINT2	EINT1	EINT0	RTC	PLL	SPI1/SSP
Bit#	10	9	8	7	6	5	4	3	2	1	0	
IRQ	SPI0	I2C0	PWM	UART1	UART0	TIMR1	TIMR0	ARMC1	ARMC0	N/A	WDT	

Note : TIMR0 = TIMER0 , TIMR1 = TIMER1 , ARMC1 = ARMCore1 , ARMC2 = ARMCore2.

LPC2148 ARM 7 Interrupt Related Registers

Now we will have a look at some of the important Registers that are used to implement interrupts in lpc214x:

1) VICIntSelect (R/W) :Not USED MUCH

This register is used to select an interrupt as IRQ or as FIQ. Writing a 0 at a given bit location(as given in Table 1) will make the corresponding interrupt as IRQ and writing a 1 will make it FIQ. For e.g if you make Bit 4 as 0 then the corresponding interrupt source i.e TIMER0 will be IRQ else if you make Bit 4 as 1 it will be FIQ instead. Note than by default all interrupts are selected as IRQ. Note that here IRQ applies for both Vectored and Non-Vectored IRQs. *[Refer Table 1]*

2) VICIntEnable (R/W) :

This is used to enable interrupts. Writing a 1 at a given bit location will make the corresponding interrupt Enabled. If this register is read then 1's will indicated enabled interrupts and 0's as disabled interrupts. Writing 0's has no effect. *[Refer Table 1]*

3) VICIntEnClr (R/W) :

This register is used to disable interrupts. This is similar to VICIntEnable expect writing a 1 here will disabled the corresponding Interrupt. This has an effect on VICIntEnable since writing at bit given location will clear the corresponding bit in the VICIntEnable Register. Writing 0's has no effect. *[Refer Table 1]*

4) VICIRQStatus (R) :

This register is used for reading the current status of the enabled IRQ interrupts. If a bit location is read as 1 then it means that the corresponding interrupt is enabled and active. Reading a 0 is unless here lol.. *[Refer Table 1]*

5) VICFIQStatus (R) :

Same as VICIRQStatus except it applies for FIQ. *[Refer Table 1]*

6) VICSoftInt :

This register is used to generate interrupts using software i.e manually generating interrupts using code i.e the program itself. If you write a 1 at any bit location then the correspoding interrupt is triggered i.e. it forces the interrupt to occur. Writing 0 here has no effect. *[Refer Table 1]*

7) VICSoftIntClear :

This register is used to clear the interrupt request that was triggered(forced) using VICSoftInt. Writing a 1 will release(or clear) the forcing of the corresponding interrupt. *[Refer Table 1]*

8) VICVectCntl0 to VICVectCntl15 (16 registers in all) :

These are the Vector Control registers. These are used to assign a particular interrupt source to a particular slot. As mentioned before slot 0 i.e VICVectCntl0 has highest priority and

VICVectCntl15 has the lowest. Each of this registers can be divided into 3 parts : **{Bit0 to bit4} , {Bit 5} , {and rest of the bits}**.

The first 5 bits i.e Bit 0 to Bit 4 contain the number of the interrupt request which is assigned to this slot. The interrupt source numbers are given in the table below :

Table 2:

Interrupt Source	**Source number In Decimal**	**Interrupt Source**	**Source number In Decimal**
WDT	0	PLL	12
N/A	1	RTC	13
ARMCore0	2	EINT0	14
ARMCore1	3	EINT1	15
TIMER0	4	EINT2	16
TIMER1	5	EINT3	17
UART0	6	ADC0	18
UART1	7	I2C1	19
PWM	8	BOD	20
I2C0	9	ADC1	21
SPI0	10	USB	22
SPI1	11		

The **5th bit** is used to **enable** the vectored IRQ slot by writing a 1.

Attention! : Note that if the vectored IRQ slot is disabled it will not disable the interrupt but will change the corresponding interrupt to Non-Vectored IRQ. Enabling the slot here means that it can generate the address of the 'dedicated Interrupt handling function (ISR)' and disabling it will generate the address of the 'common/default Interrupt handling function (ISR)' which is for Non-Vectored ISR. In simple words if the slot is enabled it points to 'specific and dedicated interrupt handling function' and if its disable it will point to the 'default function'. This will get more clear as we do some examples.

Note : The Interrupt Source Number is also called as *VIC Channel Mask*. For this tutorial I'll be calling it as "Interrupt Source Number" to keep things simple and clear. (^_^)

The rest of the bits are reserved.

9) VICVectAddr0 to VICVectAddr15 (16 registers in all) :

For Vectored IRQs these register store the address of the function that must be called when an interrupt occurs. Note – If you assign slot 3 for TIMER0 IRQ then care must be taken that you assign the address of the interrupt function to corresponding address register .. i.e VICVectAddr3 in this example.

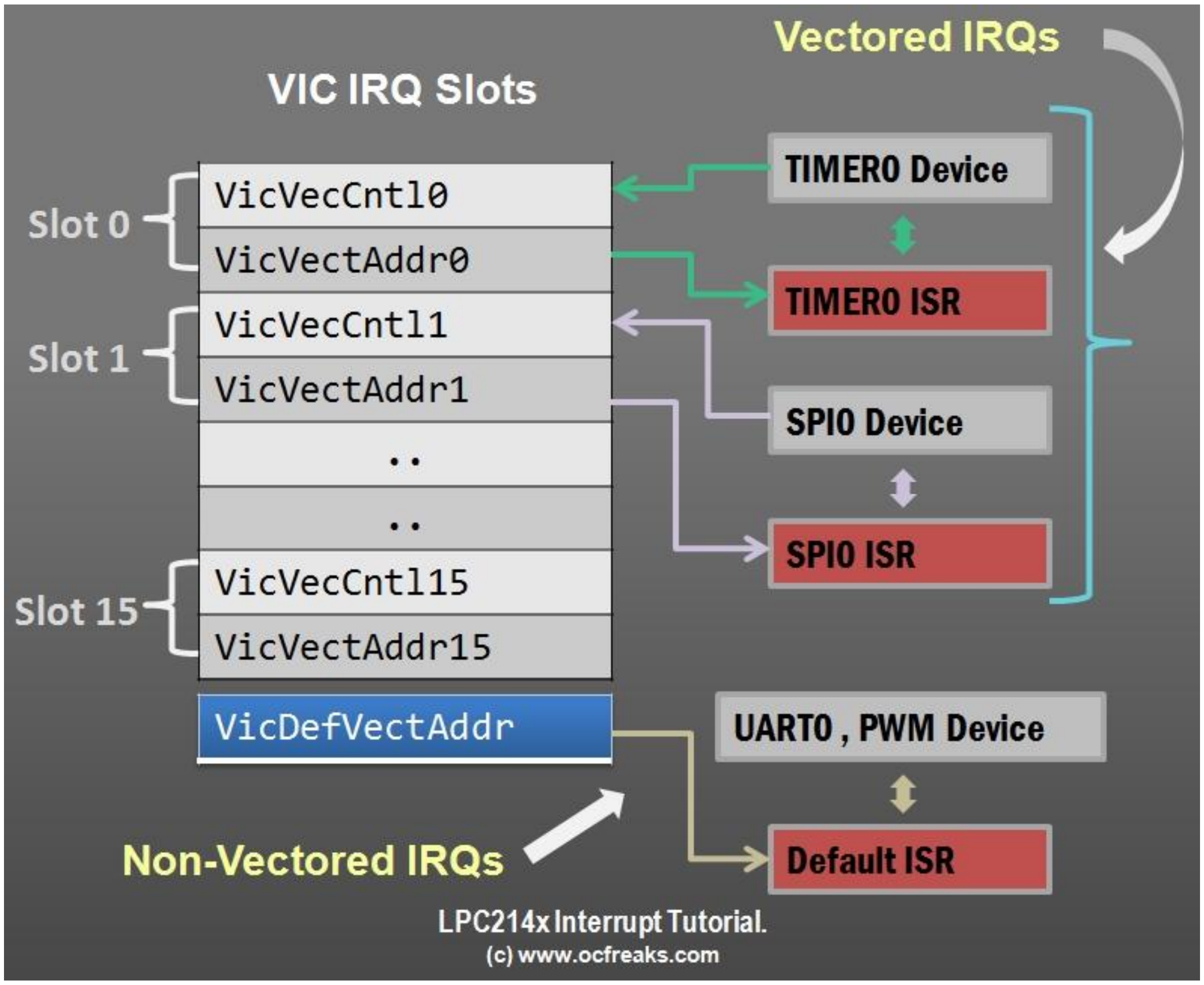
10) VICVectAddr :

This **must not be confused** with the above set of 16 VICVecAddrX registers. When an interrupt is Triggered this register holds the address of the associated ISR i.e the one which is currently active. Writing a value i.e dummy write to this register indicates to the VIC that current Interrupt has finished execution. In this tutorial the only place we'll use this register .. is at the **end of the ISR** to signal end of ISR execution.

11) VICDefVectAddr :

This register stores the address of the "default/common" ISR that must be called when a Non-Vectored IRQ occurs.

Now , that we've seen all the Registers , lets consider another Example with a Simple Diagram : Here we have 4 IRQs Configured. TIMER0 and SPI0 are configured as Vectored IRQs with TIMER0 having Highest Priority. UART0 and PWM IRQs are setup as Non-Vectored IRQs. The whole setup is as given below:



Configuring and Programming Interrupts (ISR)

To explain How to configure Interrupts and define ISRs I'll use Timers as an example , since this Timers are easy to play. Hence , I Assume you are comfortable with using timers with LPC214x ;).

First lets see how to define the **ISR** so that **compiler handles it carefully**. For this we need to explicitly tell the compiler that the function is not a normal function but an ISR. For this we'll use a special keyword called "**__irq**" which is a **function qualifier**. If you use this keyword with the function definition then compiler will automatically treat it as an ISR. Here is an example on how to define an ISR in Keil :

```
__irq void myISR (void)
{
 ...
}

//====OR Equivalently====

void myISR (void) __irq
{
 ...
}
```

Note that both are perfectly valid ways of defining the ISR. I prefer to use __irq before the function name.

Now lets see how to actually setup the interrupt for ARM based microcontrollers like lpc2148. Consider we wanna assign TIMER0 IRQ and ISR to slot X. Here is a simple procedure like thing to do that :

A Simple 3 Step Process to Setup / Enable a Vectored IRQ

1. First we need to enable the TIMER0 IRQ itself! Hence , from **Table 1** we get the bit number to **Enable** TIMER0 Interrupt which is Bit number 4. Hence we must make bit 4 in VICIntEnable to '1'.
2. Next , from **Table 2** we get the interrupt source number for TIMER0 which is **decimal** 4 and **OR** it with **(1<<5)** [i.e 5th bit=1 which enables the slot] and assign it to VICVectCntlX.
3. Next assign the address of the related ISR to VICVectAddrX.

Here is a simple Template to do it:

Replace **X** by the **slot number** you want .., then Replace **Y** by the **Interrupt Source Number** as given in **Table 2** and finally replace **myISR** with your own **ISR's function Name** .. and you'r Done!

```
VICIntEnable |= (1<<Y) ;
VICVectCntlX = (1<<5) | Y ;
VICVectAddrX = (unsigned) myISR;
```

Using above steps we can Assign TIMER0 Interrupt to Slot number say.. 0 as follows :

```
VICIntEnable |= (1<<4) ; // Enable TIMER0 IRQ
VICVectCntl0 = (1<<5) | 4 ; //5th bit must 1 to enable the slot (see the
register definition above)
VICVectAddr0 = (unsigned) myISR;
//Vectored-IRQ for TIMER0 has been configured
```

Attention! : Note the Correspondence between the Bit number in Table 1 and the Source Number in Table 2. The **Bit Number** now becomes the **Source Number in decimal**.

Having done that now its time to write the code for the ISR.

More Attention plz! : First thing to note here is that each device(or block) in lpc214x has only 1 IRQ associated with it. But **inside each device there may be different sources** which can raise an interrupt (or an IRQ). Like the TIMER0 block(or device) has **4 match + 4 capture registers** and any one or more of them can be configured to **trigger** an interrupt. Hence such devices have a **dedicated interrupt register** which contains a **flag bit** for each of these source(For Timer block its '**T0IR**'). So , when the ISR is called first we need to identify the actual source of the interrupt using the Interrupt Register and then proceed accordingly. Also just before , when the main ISR code is finished we also need to acknowledge or signal that the ISR has finished executing for the current IRQ which triggered it. This is done by clearing the the **flag(i.e the particular bit) in the device's interrupt register** and then by writing a zero to VICVectAddr register which signifies that interrupt has ISR has finished execution successfully.

External interrupt inputs

The LPC2141/2/4/6/8 includes four External Interrupt Inputs as selectable pin functions.
The External Interrupt Inputs can optionally be used to wake up the processor from Power-down mode.

4.5.1 Register description

The external interrupt function has four registers associated with it.
The **EXTINT** register contains the interrupt flags
The **EXTWAKEUP** register contains bits that enable individual external interrupts to wake up the microcontroller from Power-down mode.
The **EXTMODE** and **EXTPOLAR** registers specify the level and edge sensitivity parameters.

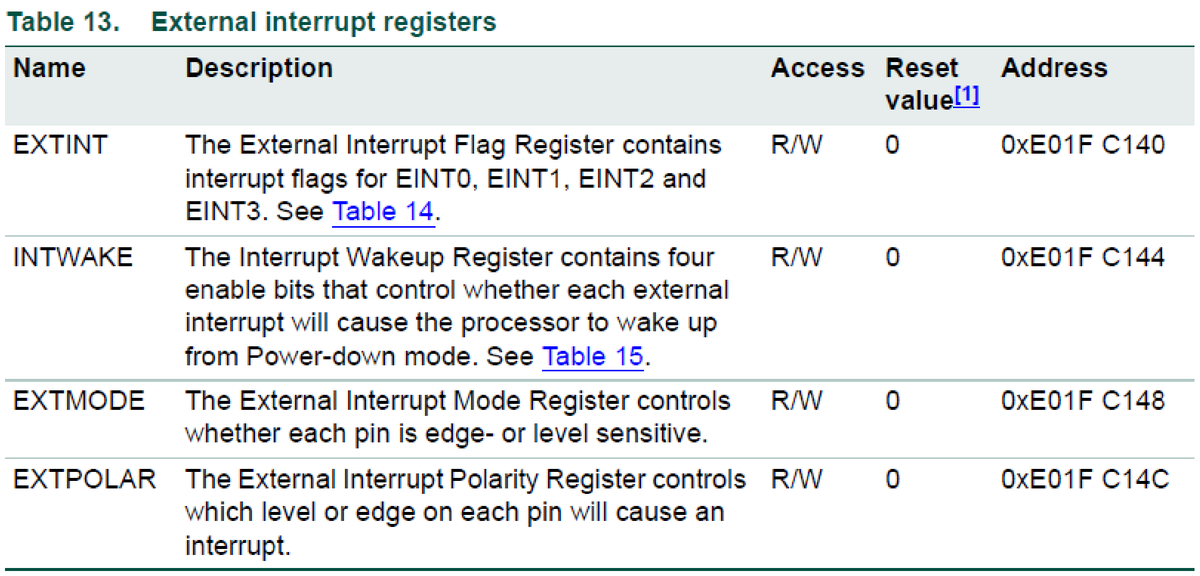


Table 13. External interrupt registers

Name	Description	Access	Reset value[1]	Address
EXTINT	The External Interrupt Flag Register contains interrupt flags for EINT0, EINT1, EINT2 and EINT3. See Table 14.	R/W	0	0xE01F C140
INTWAKE	The Interrupt Wakeup Register contains four enable bits that control whether each external interrupt will cause the processor to wake up from Power-down mode. See Table 15.	R/W	0	0xE01F C144
EXTMODE	The External Interrupt Mode Register controls whether each pin is edge- or level sensitive.	R/W	0	0xE01F C148
EXTPOLAR	The External Interrupt Polarity Register controls which level or edge on each pin will cause an interrupt.	R/W	0	0xE01F C14C

External Interrupt Flag register (EXTINT - 0xE01F C140)

When a pin is selected for its external interrupt function, the level or edge on that pin (***selected by its bits in the EXTPOLAR and EXTMODE registers***) will set its interrupt flag in this register. This asserts the corresponding interrupt request to the VIC, which will cause an interrupt if interrupts from the pin are enabled.

Writing ones to bits EINT0 through EINT3 in EXTINT register clears the corresponding bits. In level-sensitive mode this action is efficacious only when the pin is in its inactive state.

Once a bit from EINT0 to EINT3 is set and an appropriate code starts to execute (handling wakeup and/or external interrupt), this bit in EXTINT register must be cleared. Otherwise the event that was just triggered by activity on the EINT pin will not be recognized in the future.

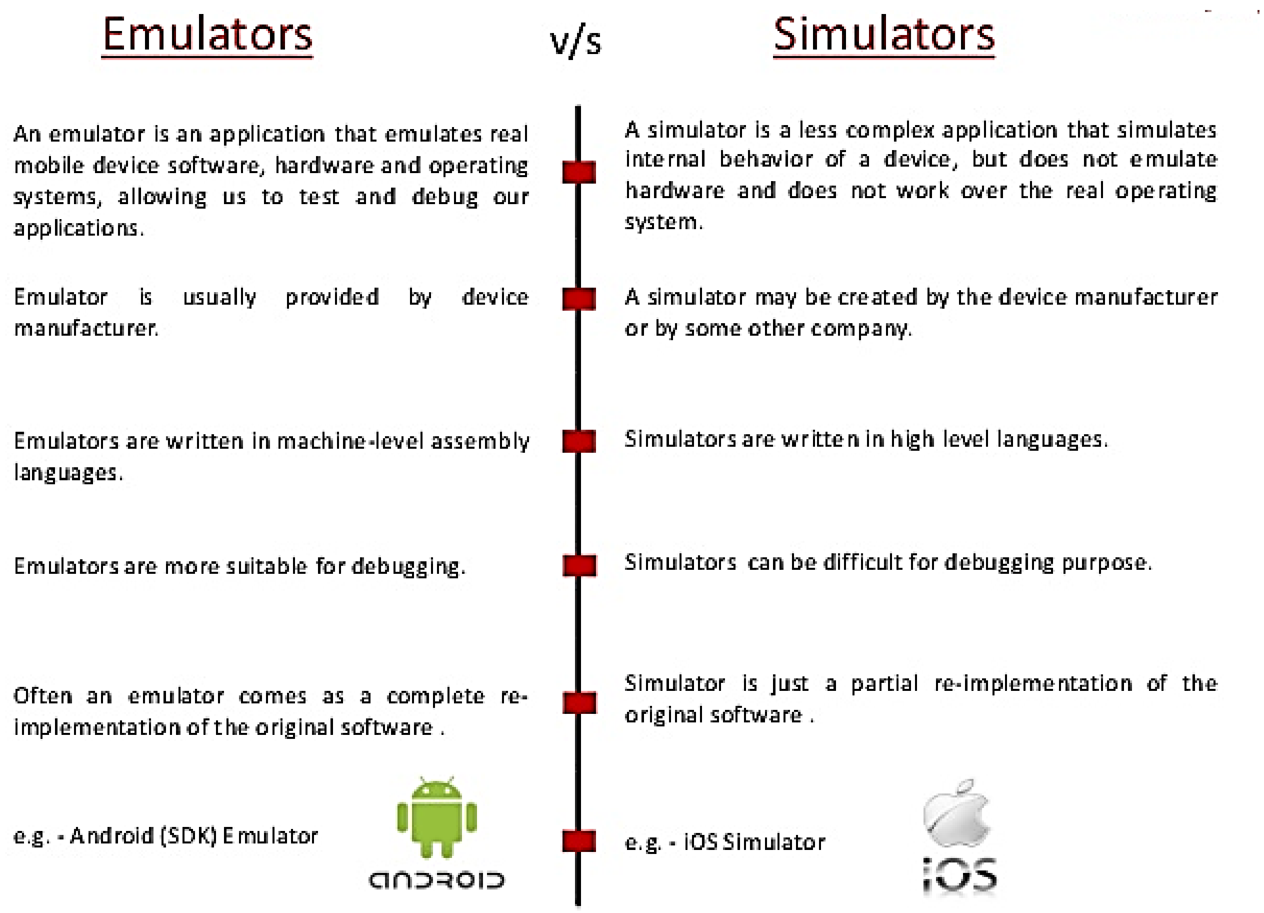
```
EXTINT – To Select Interrupt Pin →EXTINT |= (1<<2); // EINT2
EXTMODE – To Select Mode Level or Edge →EXTMODE |= (1<<2); //Edge Sensitivity
EXTPOLAR – To Select Polarity Rising or Falling Edge → EXTPOLAR |= (1<<2); //Rising Edge

VICVectAddr0 = (unsigned long)your_isr;
VICVectCntl0 = 0X20|16;
VICIntEnable |= (1<<16);

void your_isr(void) __irq
{
        //your code to set reset Buzzer using static variable concept
        EXTINT |= (1<<2); //Clearing your Interrupt
        VICVectAddr = 0;
}

        PINSEL2 &= ~(1<<3);(Optional needed for some internal config)
        PINSEL0 |= (1<<31);//Switch Is Connected to GPIO Pin15 which is EINT2
```

31:30	P0.15	00	GPIO Port 0.15	0
		01	Reserved[1][2] or RI (UART1)[3]	
		10	EINT2	
		11	Reserved[1][2] or AD1.5[3]	

Code

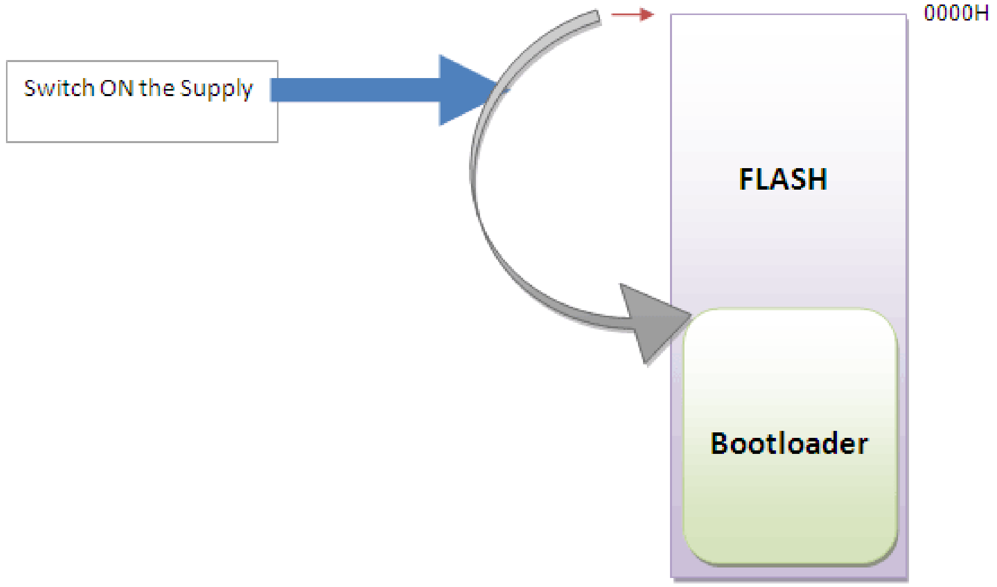
ExternalInterrupt.c

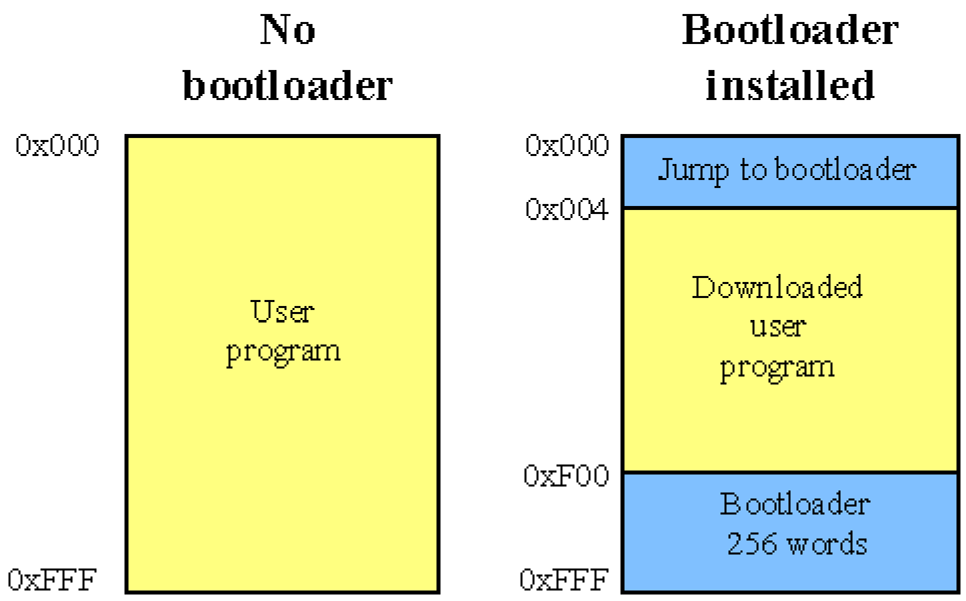
Simulator vs Emulator

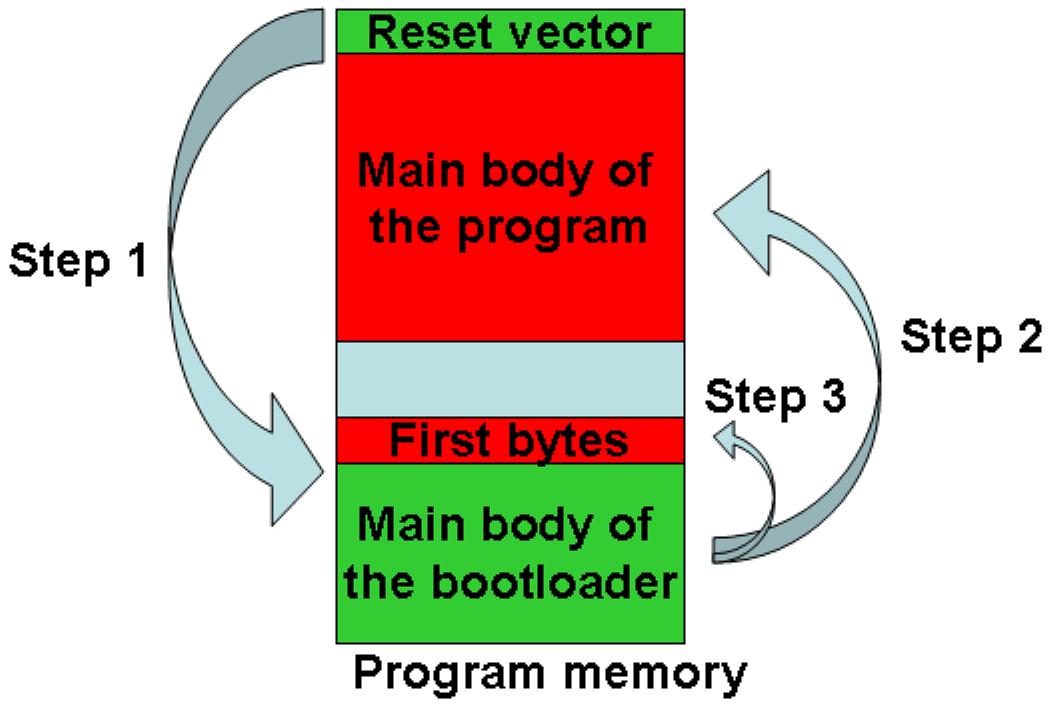


Emulators	v/s	Simulators
An emulator is an application that emulates real mobile device software, hardware and operating systems, allowing us to test and debug our applications.		A simulator is a less complex application that simulates internal behavior of a device, but does not emulate hardware and does not work over the real operating system.
Emulator is usually provided by device manufacturer.		A simulator may be created by the device manufacturer or by some other company.
Emulators are written in machine-level assembly languages.		Simulators are written in high level languages.
Emulators are more suitable for debugging.		Simulators can be difficult for debugging purpose.
Often an emulator comes as a complete re-implementation of the original software .		Simulator is just a partial re-implementation of the original software .
e.g. - Android (SDK) Emulator		e.g. - iOS Simulator

Bootloader


Switch ON the Supply
0000H
FLASH
Bootloader


No bootloader
0x000
User program
0xFFF
Bootloader installed
0x000
Jump to bootloader
0x004
Downloaded user program
0xF00
Bootloader 256 words
0xFFF



Bootloader is a piece of code that runs before any operating system is running.

Bootloader are used to boot other operating systems, usually each operating system has a set of bootloaders specific for it.

Bootloaders usually contain several ways to boot the OS kernel and also contain commands for debugging and/or modifying the kernel environment.

OS Concepts linking to arm

Logical address or virtual address and physical address memory mapping
Virtual memory ram example
Scheduler single dual core example
Signals are software interrupts
Preemptive by nested interrupt vector controller since we have single core
Shared memory
Mutual Exclusion
Semaphore
Mutex
Threads

VPBDIV

VPBDIV.rar