Why Structured Layout?

LPC Memory map can be accessed by using structures, but the "LPC17xx.h" file may be confusing at first to find your LPC registers. The memory map is accessed by structure names followed by structure members. Let's work through an example in which you are trying to find out how to access PINSEL0 mentioned in your LPC datasheet:

* Open up "**LPC17xx.h**" (it is in Project/L0 folder)
* Search for "**PINSEL0**"

You will see that this is inside a struct called **LPC\_PINCON\_TypeDef**:

* Now search for "**LPC\_PINCON\_TypeDef**" with a #define in the same line.

You will see that **LPC\_PINCON** is a pointer of this struct

#define **LPC\_PINCON** ((LPC\_PINCON\_TypeDef \*) LPC\_PINCON\_BASE)

* You can now access **PINSEL0** by :

LPC\_PINCON->PINSEL0 = XYZ;

At first this may get tedius, but once you get more experience, you won't open the **LPC17xx.h** file very often.

The first register to be set is the PINSEL register (add: 0x4002C000). It can be directly accessed by following method

Data = \*((volatile unsigned long \*)0x4002C000)

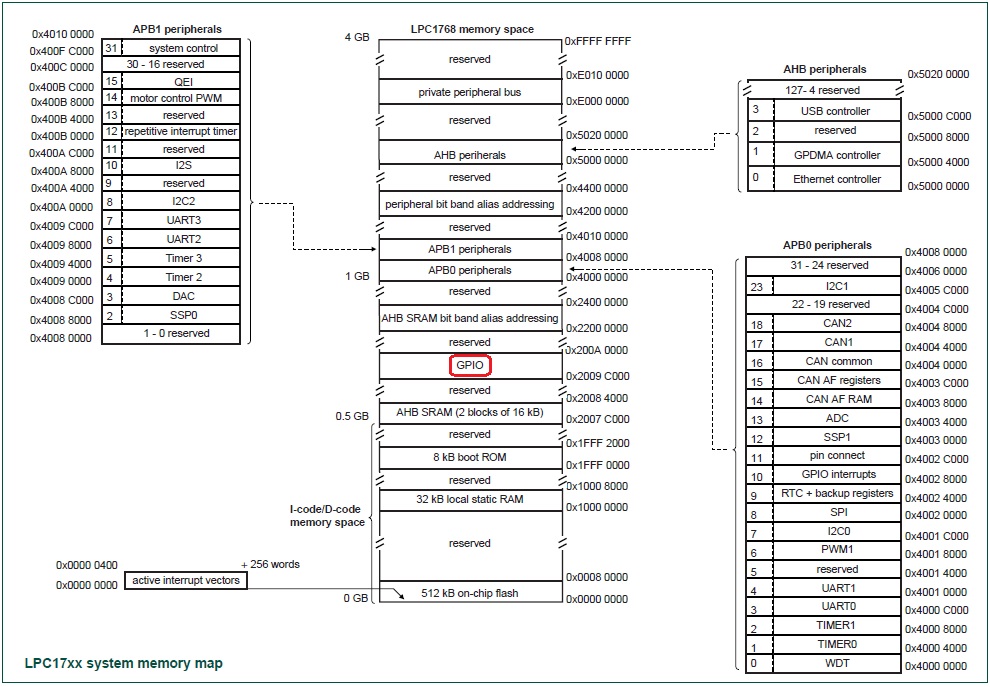
It becomes cumbersome to use the above mentioned method and looking into the datasheet again and again to check the address (copy and paste). It is better to declare all the addresses in a header file for convenience like :

#define PINSEL \*((volatile unsigned long \*)0x4002C000)

Now it is easy to use the declaration

Data = PINSEL;

# **LPC1768 Memory Map**

As it is 32-bit architecture it can access 2^32 locations(4GB).  
This 4Gb of addressable locations are divided into ROM,RAM,GPIO,AHB Peripherals as shown in the below image.   
In LPC1768 the GPIO registers are mapped to memory location 0x2009 C000 - 0x2009 FFFF.   
  
[](https://exploreembedded.com/wiki/File:Lpc1768_Memory_Map.jpg)

# **LPC GPIO**

LPC176x/5x General Purpose Input/Output (GPIO) Programming

 ARM is 32-bit architecture and provides 32 bit GPIO ports. In this tutorial, we are going to cover about GPIO pins, how to use them, how to configure GPIO registers and an example how microcontroller can interact with outside world with GPIO pins. For this tutorial we are taking LPC1769 as reference and with the use of CMSIS library.

 In order to get started with GPIO ports, we need to look into the five ‘registers’ that controls the port pins: FIODIR, FIOMASK, FIOPIN, FIOSET and FIOCLR. Each of these registers is explained in detail below with some basic examples of how they work

GPIO port Direction register FIODIR (FIO0DIR to FIO4DIR)

 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. For example, if we want to use our GPIO pin to send signals ‘out’ from the microcontroller to some external device, we need to set the pin as output (‘1’).

 Consider the below example to understand more about this register. Suppose we want to set 0th pin of port0 as input and 0th pin of port1 as output, the code will be as follows.

LPC\_GPIO0->FIODIR=0x0;

LPC\_GPIO1->FIODIR=0x1;

The first line shows how to set the 0th pin of port0 as input to receive information from the outside world. In the second line, we set the 0th pin of port1 as output to send information to the outside world.

 We can configure more than one pins as input or output by just setting the register values.

LPC\_GPIO0->FIODIR=0x0;

LPC\_GPIO1->FIODIR=0xFF;

GPIO port output Set register FIOSET (FIO0SET to FIO4SET)

 This register is used to produce a HIGH level output at the port pins configured as GPIO in an OUTPUT mode. Writing 1 produces a HIGH level at the corresponding port pins. Writing 0 has no effect. If any pin is configured as an input or a secondary function, writing 1 to the corresponding bit in the FIOxSET has no effect.

GPIO port output Clear register FIOCLR (FIO0CLR to FIO4CLR)

 This register is used to produce a LOW level output at port pins configured as GPIO in an OUTPUT mode. Writing 1 produces a LOW level at the corresponding port pin and clears the corresponding bit in the FIOxSET register. Writing 0 has no effect. If any pin is configured as an input or a secondary function, writing to FIOxCLR has no effect.

LPC\_GPIO2->FIOCLR = 0x000000FF;

This line of code clear the port2 (configured as output port using FIODIR) lowest 8 bits. If your GPIO pin is set as Output (set using the FIODIR register), you can use FIOSET to set your GPIO pin to high or FIOCLR to set it to low.

GPIO port Pin value register FIOPIN (FIO0PIN to FIO4PIN)

 You can use the FIOPIN register to read the current logic state of every GPIO pin in the pin block regardless whether the pin is configured as input or output. FIOPIN returns the current state of ALL 32 pins in the pin block, you have to do a little bit of extra work to determine the value of one specific pin. That we can know by using this line of code.

Value = LPC\_GPIO0->FIOPIN & 0x03

# LPC Timer

LPC176x/5x Timer Programming

 There are different ways to add delay in ARM controller. One of the simplest way to add delay is to use an empty for loop. The loop executes as many times as the count value specified. But this kind of delay is not precise. For precise timing applications, we use timers. LPC1769 comes loaded with four 32 bit general purpose timers. Each Timer can be used as a ‘Timer’ 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. Prescale Register is used to set the count 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.

MATCH REGISTER(MR)

 A Match Register is a Register which contains a specific value set by the user. When the Timer starts – TC incremented every PR+1 clock cycles. If it matches with the count value stored in match register, it can reset the timer or can generate an interrupt as defined by the user.

Match Registers can be configured to:

• Stop Timer on Match and trigger an optional interrupt.

• Reset Timer on Match and trigger an optional interrupt.

• To count continuously and trigger an interrupt on match.

CAPTURE REGISTER

 Capture register is used to Capture Input signal. When a transition event occurs on a Capture pin, it can be used to copy the value of TC into any of the 4 Capture Register or to generate an Interrupt.

Basic configuration

 The Timer 0, 1, 2, and 3 peripherals are configured using the following registers:

1. Power: In the PCONP register, set bits PCTIM0/1/2/3. On reset, Timer0/1 are enabled (PCTIM0/1 = 1), and Timer2/3 are disabled (PCTIM2/3 = 0).

2. Peripheral clock: In the PCLKSEL0 register, select PCLK\_TIMER0/1; in the PCLKSEL1 register, select PCLK\_TIMER2/3.

3. Pins: Select timer pins through the PINSEL registers. Select the pin modes for the port pins with timer functions through the PINMODE registers.

4. Interrupts: Interrupts are enabled in the NVIC using the appropriate Interrupt Set Enable register.

5. DMA: Up to two match conditions can be used to generate timed DMA requests.

Registers used for ADC programming in LPC17xx:

Interrupt Register (IR)

 The Interrupt Register consists of 4 bits for the match interrupts and 4 bits for the capture interrupts. If an interrupt is generated then the corresponding bit in the IR will be high. Otherwise, the bit will be low. Writing a logic one to the corresponding IR bit will reset the interrupt. Writing a zero has no effect. The act of clearing an interrupt for a timer match also clears any corresponding DMA request.

Timer Control Register (TCR)

  The Timer Control Register (TCR) is used to control the operation of the Timer/Counter. Bit0 is used to enable the timer and Bit1 is used to reset the timer.

Count Control Register (CTCR)

 The Count Control Register (CTCR) is used to select between Timer and Counter mode and in Counter mode to select the pin and edge(s) for counting. When Counter Mode is chosen as a mode of operation, the CAP input (selected by the CTCR bits 3:2) is sampled on every rising edge of the PCLK clock. After comparing two consecutive samples of this CAP input, one of the following four events is recognized: rising edge, falling edge, either of edges or no changes in the level of the selected CAP input. Only if the identified event occurs and the event corresponds to the one selected by bits 1:0 in the CTCR register, will the Timer Counter register be incremented. Bit[1:0] decides the mode of operation.

00 Timer Mode: the TC is incremented when the Prescale Counter matches the Prescale Register. The Prescale counter is incremented on every rising PCLK edge.

01 Counter Mode: TC is incremented on rising edges on the CAP input selected by bits 3:2.

10 Counter Mode: TC is incremented on falling edges on the CAP input selected by bits 3:2.

11 Counter Mode: TC is incremented on both edges on the CAP input selected by bits 3:2.

Timer Counter registers (TC)

 The 32-bit Timer Counter register is incremented when the prescale counter reaches its terminal count. Unless it is reset before reaching its upper limit, the Timer Counter will count up through the value 0xFFFF FFFF and then wrap back to the value 0x0000 0000. This event does not cause an interrupt, but a match register can be used to detect an overflow if needed.

Prescale register (PR)

 The 32-bit Prescale register specifies the maximum value for the Prescale Counter.

Prescale Counter register (PC)

 The 32-bit Prescale Counter controls division of PCLK by some constant value before it is applied to the Timer Counter. This allows control of the relationship of the resolution of the timer versus the maximum time before the timer overflows. The Prescale Counter is incremented on every PCLK. When it reaches the value stored in the Prescale register, the Timer Counter is incremented and the Prescale Counter is reset on the next PCLK. This causes the Timer Counter to increment on every PCLK when PR = 0, every 2 pclks when PR = 1, etc.

Match Registers (MR0 - MR3)

 The Match register values are continuously compared to the Timer Counter value. When the two values are equal, actions can be triggered automatically. The action possibilities are to generate an interrupt, reset the Timer Counter, or stop the timer. Actions are controlled by the settings in the MCR register.

Match Control Register (MCR)

 The Match Control Register is used to control what operations are performed when one of the Match Registers matches the Timer Counter.

Configuring a Timer

 Now let’s see how to configure and setup a timer.

1. Reset the timer by configuring TCR.

2. Set Prescale value in PR

3. Set the count value in Match register(MR)

4. Set appropriate value in MCR.

5. Enable the timer by configuring TCR.

6. Enable timer interrupt if needed.

LPC\_TIM0->TCR = 0x02; /\* reset timer \*/

LPC\_TIM0->PR = 0x00; /\* set prescaler to zero \*/

LPC\_TIM0->MR0 = 2400; /\* set the count value \*/

LPC\_TIM0->IR = 0xff; /\* reset all interrupts \*/

LPC\_TIM0->MCR = 0x04; /\* stop timer on match \*/

LPC\_TIM0->TCR = 0x01; /\* start the timer \*/

# LPC ADC

****

# **Register Configuration**

As all the LPC1768 SFRs(Special Function Registers) are defined in lpc17xx.h, this has to be included at the beginning of our project/code.

LPC1768 has its GPIOs divided into five ports PORT0 - PORT4, although many of them are not physically 32bit wide. Refer the data sheet for more info. The Below registers will be used for Configuring and using the GPIOs registers for sending and receiving the Digital signals. A structure LPC\_GPIOn(n= 0,1,2,3) contains all the registers for required for GPIO operation. Refer lpc17xx.h file for more info on the registers.

[](https://exploreembedded.com/wiki/File:0_Lpc1768_Gpio.JPG)

**PINSEL:** GPIO Pins Select Register  
Almost all the LPC1768 pins are multiplexed to support more than 1 function. Every GPIO pin has a minimum of one function and max of four functions. The required function can be selected by configuring the PINSEL register. As there can be up to 4 functions associated with a GPIO pin, two bits for each pin are available to select the function. This implies that we need two PINSEL registers to configure a PORT pins. By this the first 16(P0.0-P0.16) pin functions of PORT0 can be selected by 32 bits of PINSELO register. The remaining 16 bits(P0.16-P0.32) are configured using 32bits of PINSEL1 register. As mentioned earlier every pin has max of four functions. Below table shows how to select the function for a particular pin using two bits of the PINSEL register.

|  |  |  |
| --- | --- | --- |
| Value | Function | Enumeration |
| 00 | Primary (default) function, typically GPIO port | PINSEL\_FUNC\_0 |
| 01 | First alternate function | PINSEL\_FUNC\_1 |
| 10 | Second alternate function | PINSEL\_FUNC\_2 |
| 11 | Third alternate function | PINSEL\_FUNC\_3 |

**FIODIR:**Fast GPIO Direction Control Register.  
This register individually controls the direction of each port pin.

|  |  |
| --- | --- |
| Values | Direction |
| 0 | Input |
| 1 | Output |

**FIOSET:**Fast Port Output Set Register.  
This register controls the state of output pins. Writing 1s produces highs at the corresponding port pins. Writing 0s has no effect. Reading this register returns the current contents of the port output register not the physical port value.

|  |  |
| --- | --- |
| Values | FIOSET |
| 0 | No Effect |
| 1 | Sets High on Pin |

**FIOCLR:**Fast Port Output Clear Register.  
This register controls the state of output pins. Writing 1s produces lows at the corresponding port pins. Writing 0s has no effect.

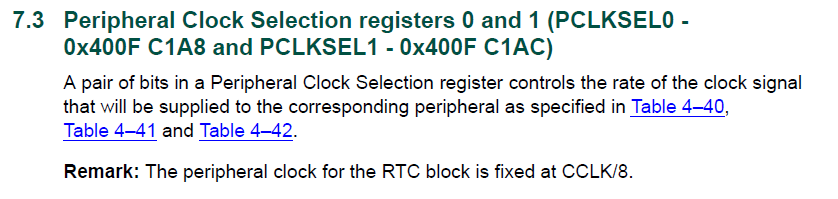
|  |  |
| --- | --- |
| Values | FIOCLR |
| 0 | No Effect |
| 1 | Sets Low on Pin |

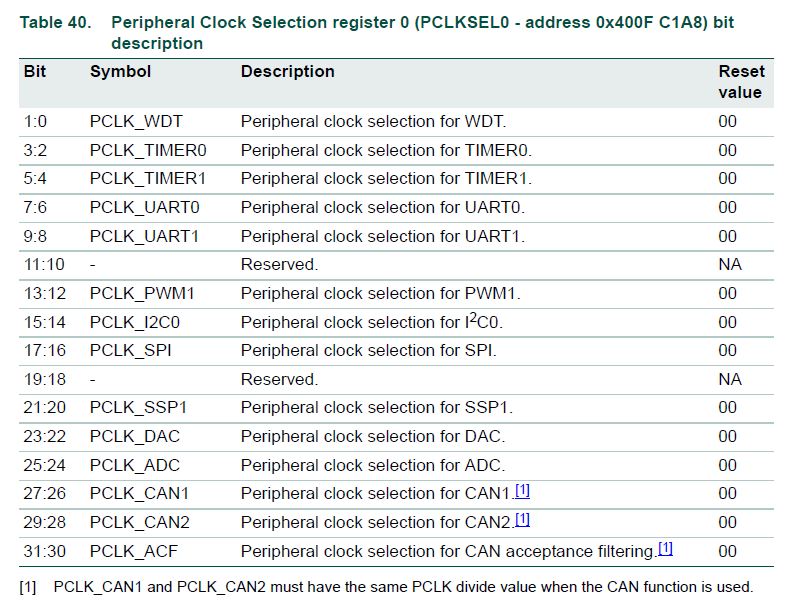
**FIOPIN:**Fast Port Pin Value Register.  
This register is used for both reading and writing data from/to the PORT.  
**Output:** Writing to this register places corresponding values in all bits of the particular PORT pins.  
**Input:** The current state of digital port pins can be read from this register, regardless of pin direction or alternate function selection (as long as pins are not configured as an input to ADC).  
**Note:**It is recommended to configure the PORT direction and pin function before using it.

# UART

Step 1 PINSEL Selection

Step 2 Set peripheral clock : LPC\_SC->PCLKSEL0 |= (1 << 7);







Step3 Set Baud Rate

LPC\_UART0->DLL = 235;

LPC\_UART0->DLM |=0X00;

LPC\_UART0->LCR &= ~(1<<7); //DLAB bit reset

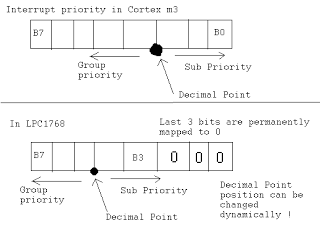
LPC\_UART0->FDR |= (1<<4);

LPC\_UART0->TER |= (1<<7);



# Interrupt Priorities

In the LPC17xx user manual as of date, NVIC is explained in chapter 6. In page 87 of the manual , we have the register description for interrupt priorities.We see that each interrupt has a 5 bit priority level associated with it. What does this 5 bit number mean ? Strangely enough, the answer is not in chapter 6 which covers NVIC.  
  
This piece of info is presented in the appendix of the manual ! Chapter 34 . More specifically, 34.3.3.6 Interrupt priority grouping in page 749.  
The 5 bit priority field for each interrupt has 2 parts separated by a decimal point. In fact, Cortex M3 provides 8 bit priority field for each interrupt. But LPC1768 has implemented only 5 bits. the 3 LSB bits are permanently mapped to 0.

[](http://3.bp.blogspot.com/_8SBLZpKyvFA/TS25Tw197nI/AAAAAAAAAA8/k4znYLvp_iw/s1600/interrupt_group.png)

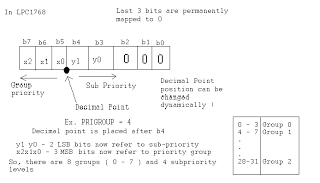
Interrupts are assigned to different groups and interrupts in each group have different priority within the group. Lets see what the implications are:  
  
1. An interrupt belonging to a lower group (number) may preempt an interrupt of a higher group(number). Suppose Interrupt A is in group 4 and is running. Interrupt B in group 2 arrives while Int A ISR is executing. Now, interrupt A ISR is preempted and ISR for Interrupt B is executed. After ISR B is done, execution of ISR A continues.  
2. If interrupt C which is also in group 4 arrives while ISR A is executing, ISR A is NOT preempted. It has to wait. After execution of ISR A, ISR C is executed.  
3. If interrupt A and C arrive simultaneously, the one with lower sub-priority value is executed first.

Here is another sweet thing. The decimal point ( to be accurate, binary point ) demarcating group priority number from sub priority number may be positioned as the user wishes. This means we can choose and have a compromise on the number of groups vs. subpriority.

Lets see how we go about doing this. Have a look at section 34.4.3.6.1 Binary Point in page  772 of the manual.

NVIC\_SetPriorityGrouping(0x04);

Setting PRIGROUP to 4 puts the binary point after b4 as illustrated below. This will create 8 priority groups( 3 bits for priority group ) and 4 sub priority levels ( 2 bits for sub level ) in each group.

[](http://4.bp.blogspot.com/_8SBLZpKyvFA/TS3TN63ZBlI/AAAAAAAAABE/aTHn0kEjhgU/s1600/pri.png)

Now, we can set the priority level for an interrupt like this  
  
NVIC\_SetPriority(UART3\_IRQn,5);  
  
This will assign UART3 IRQ to priority group 1 ( see the table in the figure above ) and give it a sub priority level of 1 within the group.

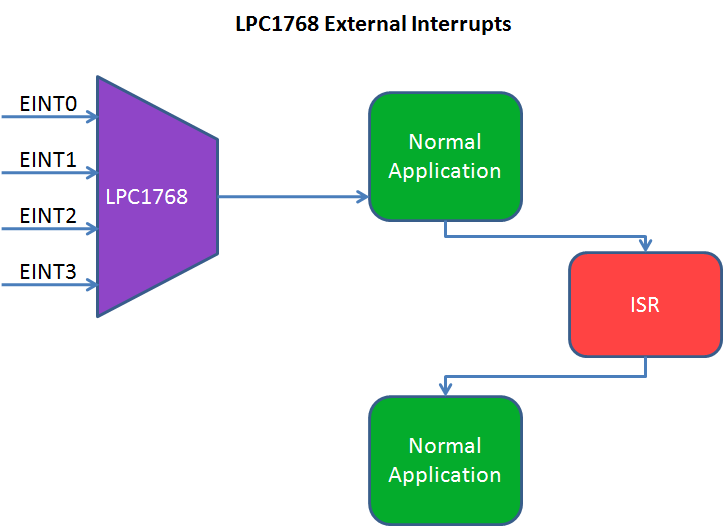
# **LPC1768: External Interrupts**

In this tutorial we will discuss how to configure and use the LPC1768 external interrupts(EINT0-EINT3).  
At the end of tutorial we will see how to use the ExploreEmbedded external interrupt library. 

# **EINTx Pins**

LPC1768 has four external interrupts EINT0-EINT3.  
As LPC1768 pins are multi functional, these four interrupts are available on multiple pins.  
Below table shows mapping of EINTx pins.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Port Pin** | **PINSEL\_FUNC\_0** | **PINSEL\_FUNC\_1** | **PINSEL\_FUNC\_2** | **PINSEL\_FUNC\_3** |
| P2.10 | GPIO | **EINT0** | NMI |  |
| P2.11 | GPIO | **EINT1** | I2STX\_CLK |  |
| P2\_12 | GPIO | **EINT2** | I2STX\_WS |  |
| P2.13 | GPIO | **EINT3** | I2STX\_SDA |  |

[](https://exploreembedded.com/wiki/File:Lpc1768_external_interrupts.png)

# **EINT Registers**

Below table shows the registers associated with LPC1768 external interrupts.

|  |  |
| --- | --- |
| **Register** | **Description** |
| PINSELx | To configure the pins as External Interrupts |
| EXTINT | External Interrupt Flag Register contains interrupt flags for EINT0,EINT1, EINT2 & EINT3. |
| EXTMODE | External Interrupt Mode register(Level/Edge Triggered) |
| EXTPOLAR | External Interrupt Polarity(Falling/Rising Edge, Active Low/High) |

|  |
| --- |
| EXTINT |
| 31:4 | 3 | 2 | 1 | 0 |
| RESERVED | EINT3 | EINT2 | EINT1 | EINT0 |

**EINTx:** Bits will be set whenever the interrupt is detected on the particular interrupt pin.  
If the interrupts are enabled then the control goes to ISR.  
Writing one to specific bit will clear the corresponding interrupt. 

|  |
| --- |
| EXTMODE |
| 31:4 | 3 | 2 | 1 | 0 |
| RESERVED | EXTMODE3 | EXTMODE2 | EXTMODE1 | EXTMODE0 |

**EXTMODEx:** This bits is used to select whether the EINTx pin is level or edge Triggered  
0: EINTx is Level Triggered.  
1: EINTx is Edge Triggered. 

|  |
| --- |
| EXTPOLAR |
| 31:4 | 3 | 2 | 1 | 0 |
| RESERVED | EXTPOLAR3 | EXTPOLAR2 | EXTPOLAR1 | EXTPOLAR0 |

**EXTPOLARx:** This bits is used to select polarity(LOW/HIGH, FALLING/RISING) of the EINTx interrupt depending on the EXTMODE register.  
0: EINTx is Active Low or Falling Edge (depending on EXTMODEx).  
1: EINTx is Active High or Rising Edge (depending on EXTMODEx).

# **Steps to Configure Interrupts**

1. Configure the pins as external interrupts in PINSELx register.
2. Clear any pending interrupts in EXTINT.
3. Configure the EINTx as Edge/Level triggered in EXTMODE register.
4. Select the polarity(Falling/Rising Edge, Active Low/High) of the interrupt in EXTPOLAR register.
5. Finally enable the interrputs by calling NVIC\_EnableIRQ() with IRQ number.

# **Code**

Below example shows the toggling of Leds depending on the external interrupts.

|  |  |
| --- | --- |
|  | #include <lpc17xx.h> |
|  |  |
|  | #define PINSEL\_EINT0 20 |
|  | #define PINSEL\_EINT1 22 |
|  |  |
|  | #define LED1 0 |
|  | #define LED2 1 |
|  |  |
|  | #define SBIT\_EINT0 0 |
|  | #define SBIT\_EINT1 1 |
|  |  |
|  | #define SBIT\_EXTMODE0 0 |
|  | #define SBIT\_EXTMODE1 1 |
|  |  |
|  | #define SBIT\_EXTPOLAR0 0 |
|  | #define SBIT\_EXTPOLAR1 1 |
|  |  |
|  |  |
|  | void EINT0\_IRQHandler(void) |
|  | { |
|  | LPC\_SC->EXTINT = (1<<SBIT\_EINT0); /\* Clear Interrupt Flag \*/ |
|  | LPC\_GPIO2->FIOPIN ^= (1<< LED1); /\* Toggle the LED1 everytime INTR0 is generated \*/ |
|  | } |
|  |  |
|  |  |
|  | void EINT1\_IRQHandler(void) |
|  | { |
|  | LPC\_SC->EXTINT = (1<<SBIT\_EINT1); /\* Clear Interrupt Flag \*/ |
|  | LPC\_GPIO2->FIOPIN ^= (1<< LED2); /\* Toggle the LED2 everytime INTR1 is generated \*/ |
|  | } |
|  |  |
|  |  |
|  |  |
|  | int main() |
|  | { |
|  | SystemInit(); |
|  |  |
|  | LPC\_SC->EXTINT = (1<<SBIT\_EINT0) | (1<<SBIT\_EINT1); /\* Clear Pending interrupts \*/ |
|  | LPC\_PINCON->PINSEL4 = (1<<PINSEL\_EINT0) | (1<<PINSEL\_EINT1); /\* Configure P2\_10,P2\_11 as EINT0/1 \*/ |
|  | LPC\_SC->EXTMODE = (1<<SBIT\_EXTMODE0) | (1<<SBIT\_EXTMODE1); /\* Configure EINTx as Edge Triggered\*/ |
|  | LPC\_SC->EXTPOLAR = (1<<SBIT\_EXTPOLAR0)| (1<<SBIT\_EXTPOLAR0); /\* Configure EINTx as Falling Edge \*/ |
|  |  |
|  | LPC\_GPIO2->FIODIR = (1<<LED1) | (1<<LED2); /\* Configure LED pins as OUTPUT \*/ |
|  | LPC\_GPIO2->FIOPIN = 0x00; |
|  |  |
|  | NVIC\_EnableIRQ(EINT0\_IRQn); /\* Enable the EINT0,EINT1 interrupts \*/ |
|  | NVIC\_EnableIRQ(EINT1\_IRQn); |
|  |  |
|  | while(1) |
|  | { |
|  | // Do nothing |
|  | } |
|  | } |

[**view raw**](https://gist.github.com/SaheblalBagwan/dc02144bfe786cc4169100cda17245d5/raw/50f0e4f70cf041a4c2a718389668345cb5a1960c/lpc1768_ExternalInterrupt0.c)[**lpc1768\_ExternalInterrupt0.c**](https://gist.github.com/SaheblalBagwan/dc02144bfe786cc4169100cda17245d5#file-lpc1768_externalinterrupt0-c) hosted with ❤ by [**GitHub**](https://github.com/)

# **Using ExploreEmbedded Libraries**

In the above tutorial we discussed how to configure and use the LPC1768 external interrupts.  
Now we will see how to use the ExploreEmbededd external interrupt libraries.

|  |  |
| --- | --- |
|  | #include <lpc17xx.h> |
|  | #include "stdutils.h" |
|  | #include "gpio.h" |
|  | #include "extintr.h" |
|  |  |
|  | #define LED1 P2\_0 |
|  | #define LED2 P2\_1 |
|  |  |
|  | void myExtIntrIsr\_0(void) |
|  | { |
|  | GPIO\_PinToggle(LED1); /\* Toggle the LED1 (P2\_0) \*/ |
|  | } |
|  |  |
|  | void myExtIntrIsr\_1(void) |
|  | { |
|  | GPIO\_PinToggle(LED2); /\* Toggle the LED2 (P2\_1) \*/ |
|  | } |
|  |  |
|  |  |
|  | int main (void) |
|  | { |
|  | SystemInit(); |
|  |  |
|  | GPIO\_PinDirection(LED1,OUTPUT); /\* Configure the pins as Output to blink the Leds\*/ |
|  | GPIO\_PinDirection(LED2,OUTPUT); |
|  |  |
|  | EINT\_AttachInterrupt(EINT0,myExtIntrIsr\_0,FALLING); /\* myExtIntrIsr\_0 will be called by EINT0\_IRQHandler \*/ |
|  | EINT\_AttachInterrupt(EINT1,myExtIntrIsr\_1,FALLING); /\* myExtIntrIsr\_1 will be called by EINT1\_IRQHandler \*/ |
|  |  |
|  | while(1) |
|  | { |
|  | //do nothing |
|  | } |
|  | } |

[**view raw**](https://gist.github.com/SaheblalBagwan/ee07ec3913b23f580a651c5f33dcd3af/raw/88d30f814b8044b1829d153a695418110885efc0/lpc1768_ExternalInterrupt1.c)[**lpc1768\_ExternalInterrupt1.c**](https://gist.github.com/SaheblalBagwan/ee07ec3913b23f580a651c5f33dcd3af#file-lpc1768_externalinterrupt1-c) hosted with ❤ by [**GitHub**](https://github.com/)

The below example demonstrates the difference between the edge triggered and level triggered interrupt.  
EINT0 is configured as FALLING edge and counter will be incremented whenever there is a high-low pulse on EINT0.   
EINT1 is configured as ACTIVE low and counter will be inctermented as long as EINT1 is LOW.  
EINT1 counter increments fast as the ISR will be executed multiple times.

|  |  |
| --- | --- |
|  | #include <lpc17xx.h> |
|  | #include "stdutils.h" |
|  | #include "gpio.h" |
|  | #include "extintr.h" |
|  | #include "lcd.h" |
|  |  |
|  | volatile uint32\_t cnt1=0,cnt2=0; |
|  |  |
|  | void myExtIntrIsr\_0(void) |
|  | { |
|  | cnt1++; /\* Increment cnt1 every time EINT0 is detected \*/ |
|  | } |
|  |  |
|  | void myExtIntrIsr\_1(void) |
|  | { |
|  | cnt2++; /\* Increment cnt2 every time EINT1 is detected \*/ |
|  | } |
|  |  |
|  |  |
|  | int main (void) |
|  | { |
|  | SystemInit(); |
|  |  |
|  | /\* RS RW EN D0 D1 D2 D3 D4 D5 D6 D7 P\_NC(Not connected)\*/ |
|  | LCD\_SetUp(P2\_0,P2\_1,P2\_2,P\_NC,P\_NC,P\_NC,P\_NC,P1\_24,P1\_25,P1\_26,P1\_27); |
|  | LCD\_Init(2,16); |
|  |  |
|  | /\* EINT0 is configured as FallingEdge interrupt and myExtIntrIsr\_0 will be called by EINT0\_IRQHandler \*/ |
|  | EINT\_AttachInterrupt(EINT0,myExtIntrIsr\_0,FALLING); |
|  |  |
|  | /\* EINT1 is configured as Active Low interrupt and myExtIntrIsr\_1 will be called by EINT1\_IRQHandler \*/ |
|  | EINT\_AttachInterrupt(EINT1,myExtIntrIsr\_1,LOW); |
|  |  |
|  | while(1) |
|  | { |
|  | LCD\_GoToLine(0); |
|  | LCD\_Printf("EINT0=%8u EINT1:%8u",cnt1,cnt2); /\* Display the occurrence of EINT0 and EINT1 \*/ |
|  | } |
|  | } |

Group Priority Doc