

Vectored Interrupt Programming

Copyright (c) 2023 - 2014 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

Based on

ARM System-on-Chip Architecture, 2nd ed, Steve Furber

Introduction to ARM Cortex-M Microcontrollers
– Embedded Systems, Jonathan W. Valvano

Digital Design and Computer Architecture,
D. M. Harris and S. L. Harris

ARM assembler in Raspberry Pi
Roger Ferrer Ibáñez

<https://thinkingeek.com/arm-assembler-raspberry-pi/>

Standard Interrupt Controller

The **standard** interrupt controller sends an interrupt signal to the processor core when an external device requests servicing.

It can be programmed to ignore or mask an individual device or set of devices.

The **interrupt handler** determines which device requires servicing by reading a device bitmap register in the interrupt controller.

standard interrupt controller

programmable mask

*interrupt source
by reading a device register*

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

Vectored Interrupt Controller

The VIC is more powerful than the standard interrupt controller

- prioritizes interrupts
- simplifies the determination of interrupt source (of which device caused the interrupt)
- a priority is associated with a handler address for each interrupt request
- the VIC asserts an interrupt signal to the core only when the priority of a new interrupt is higher than that of the currently executing interrupt handle

VIC = IVT + Priority

*Vectored Interrupt Controller (VIC)
Interrupt Vector Table (IVT)*

priority – handler address

preemptive interrupt handling

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

Multiple ISR handlers

Usually in the old generation controllers,
there is only one ISR
that handles multiple interrupt sources.

the **ISR** checks the particular register
to find the interrupt source
– who is interrupting the processor.

large interrupt latency

to reduce interrupt latency,
ARM has come up with an idea of
a **vector interrupt controller (VIC)**
where each interrupt can have separate ISR's

each **ISR** address will be
stored in the **Interrupt Vector Table**.

Default ISR

ISR determines the interrupt source

One ISR – multiple interrupt sources

Large latency

Separate ISR's, IVT

IRQ source 1	----	ISR 1 address
IRQ source 2	----	ISR 2 address
IRQ source 3	----	ISR 3 address
...		...

If IRQ source i , then jump to ISR_i

<https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/>

IRQ sources and ISR addresses

The VIC provides a software interface to the interrupt system.

In a standard interrupt controller, **software** must determine the **source** that is requesting service where its **ISR** is loaded.

In a vectored interrupt controller, **hardware** supplies the **starting address**, or **vector address**, of the ISR corresponding to the interrupt **source** that has the highest priority

SW (ISR) determines the interrupt source

Jump address must be loaded

HW determines the interrupt source

A table lookup of IVT provides the jump address (the specific ISR)

<https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/>

Interrupt Vector Table

Interrupt vector table (IVT)

contains the address of the IRQ handlers of every interrupt.

directs the PC where to go, when an interrupt occurs.

refers early generation of VIC, because they just point the address when an interrupt occurs.

priority was not fully applied

$$VIC = IVT + Priority$$

Old VIC = IVT only, no priority

<https://www.quora.com/What-is-the-difference-between-ARMs-nested-vectored-interrupt-controller-and-an-interrupt-vector-table-which-seems-to-be-used-by-the-other-proc>

VIC interrupt handling types (1)

the VIC interrupt handling types

- make the core jump directly to the handler address for the device **(Vectored IRQ)**
- either call the standard interrupt exception handler, which can load the handler address for the device from the VIC **(Non-Vectored IRQ)**

$VIC = IVT + Priority$

Vectored IRQ
unique ISR
jump to the address

Non-vectored IRQ
default ISR
load the address

in Ipc214x

VICVectAddr : holds the **address** of the associated **ISR** i.e the one which is currently active.

VICDefVectAddr : stores the **address** of the “**default/common**” ISR for a **Non-Vectored IRQ** occurs

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

VIC interrupt handling types (2)

VIRQ (Vectored IRQ) has
dedicated IRQ service routine
for *each* Vectored interrupt source

NVIRQ (Non-Vectored IRQ) has
the same IRQ service routine
for *all* Non-Vectored Interrupts.

VIC = IVT + Priority

Vectored IRQ
dedicated ISR for each IRQ source

Non-vectored IRQ
default ISR for all IRQ sources

in lpc214x

VICVectAddr : holds the **address** of the associated **ISR** i.e the one which is currently active.

VICDefVectAddr : stores the **address** of the “**default/common**” ISR for a **Non-Vectored IRQ** occurs

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

VIC interrupt handling types (3)

Vectored means that
the CPU is aware of the **address** of the **ISR**
when the interrupt occurs

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

For the **Vectored Interrupt Controller**,
the system internally maintains a table
IVT (Interrupt Vector Table)

which contains the information
about **Interrupts sources**
and their corresponding **ISR address**.

VIC = IVT + Priority

Vectored IRQ
ISR address is known
jump to the address

Non-vectored IRQ
IRQ address not known
load the address

<i>IRQ source 1</i>	<i>ISR 1 address</i>
<i>IRQ source 2</i>	<i>ISR 2 address</i>
<i>IRQ source 3</i>	<i>ISR 3 address</i>
<i>...</i>	<i>...</i>

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

ARM FIQ

In an ARM system, two levels of interrupts are available:

Fast Interrupt reQuest (FIQ)

- For fast, **low latency** interrupt handling.

Interrupt ReQuest (IRQ)

- For more **general** interrupts.

- a single **FIQ source** system for a **low-latency** interrupt

Single FIQ source system

- the **ISR** is executed directly
without determining the **source** of the interrupt
- this reduces the **interrupt latency**

Register bank

- the **banked registers** of **FIQ mode**
can be used more efficiently,
without incurring a **context save** overhead

<https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/>

3 categories of IRQ's

The ARM Vectored Interrupt Controller (VIC) takes 32 interrupt request inputs and programmably assigns them into 3 categories,

- **FIQ**
- **vectored IRQ**
- **non-vectored IRQ.**

<https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/>

Nested Vectored Interrupt Controller

A Nested Vectored Interrupt Controller (NVIC) is used to manage the interrupts from multiple interrupt sources.

NVIC is closely integrated with the processor core to achieve low-latency interrupt processing and efficient processing of late arriving interrupts.

<https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/>

NVIC vs. VIC (1)

every interrupt with certain priority levels

each interrupt is serviced / processed
with its own priority level.

Servicing / processing the interrupt means
the processing of the part of codes
inside the IRQ handler
of the respective interrupt.

Interrupt handling of

- Nested Vectored Interrupt Controller (NVIC)
- Vectored Interrupt Controller (VIC)
- Interrupt Vector Table (IVT)

<https://www.quora.com/What-is-the-difference-between-ARMs-nested-vectored-interrupt-controller-and-an-interrupt-vector-table-which-seems-to-be-used-by-the-other-proc>

NVIC vs. VIC (2)

Example assumption :

Priority 1 (P1) - highest

Priority 2 (P2) - second highest

There are two different interrupts X and Y with priority levels P1 and P2 respectively.

- interrupts X and Y occur at the same time.
- interrupt Y (P2) has occurred first and while servicing interrupt Y (P2) interrupt X (P1) occurs

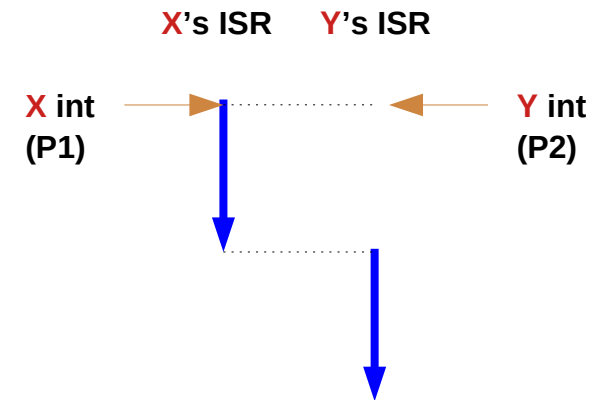
<https://www.quora.com/What-is-the-difference-between-ARMs-nested-vectored-interrupt-controller-and-an-interrupt-vector-table-which-seems-to-be-used-by-the-other-proc>

Nested VIC handling (1)

- If interrupts X and Y occur at the same time.

first X (P1) is processed,
Y (P2) is put on hold.

After processing X,
Y is processed.



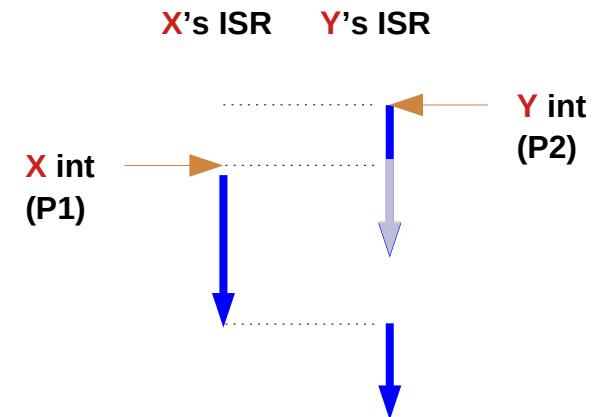
<https://www.quora.com/What-is-the-difference-between-ARMs-nested-vectored-interrupt-controller-and-an-interrupt-vector-table-which-seems-to-be-used-by-the-other-proc>

Nested VIC handling (2)

- If interrupt Y (P2) has occurred first and interrupt X (P1) occurs while servicing interrupt Y (P2)

Then, the controller puts the interrupt Y's IRQ handler on hold and processes the interrupt X's IRQ handler completely and then resumes the interrupt Y's IRQ handler

So, it processes interrupt by nesting them within each other.



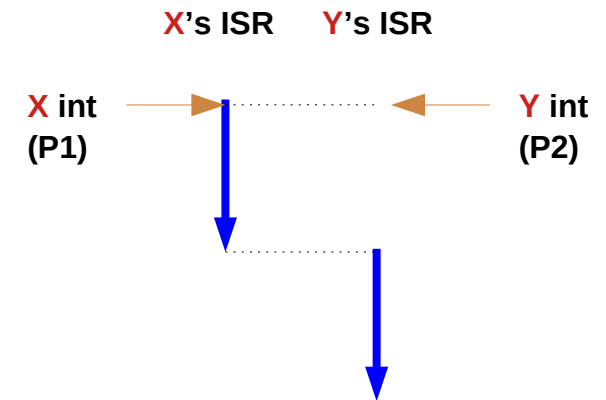
<https://www.quora.com/What-is-the-difference-between-ARMs-nested-vectored-interrupt-controller-and-an-interrupt-vector-table-which-seems-to-be-used-by-the-other-proc>

VIC handling (1)

- If interrupts X and Y occur at the same time.

first X (P1) is processed,
Y (P2) is put on hold.

After processing X,
Y is processed.



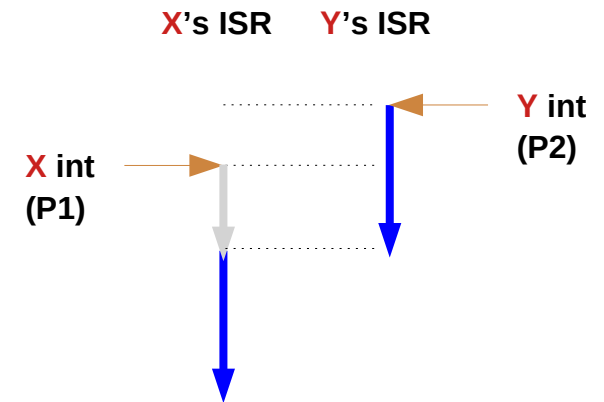
<https://www.quora.com/What-is-the-difference-between-ARMs-nested-vectored-interrupt-controller-and-an-interrupt-vector-table-which-seems-to-be-used-by-the-other-proc>

VIC handling (2)

- If interrupt Y (P2) has occurred first and interrupt X (P1) occurs while servicing interrupt Y (P2)

Then, the controller processes the interrupt Y's IRQ handler completely

and then the processes the interrupt X's IRQ handler



<https://www.quora.com/What-is-the-difference-between-ARMs-nested-vectored-interrupt-controller-and-an-interrupt-vector-table-which-seems-to-be-used-by-the-other-proc>

NVIC features in cortex M (1)

- external interrupts (1 ~ 240)
- bits of priority (3 ~ 8)
- a dynamic **re-prioritization** of interrupts.
- **priority grouping** enables the selection of
 - preempting** interrupt levels
 - non-preempting** interrupt levels.
- support for **tail-chaining** and **late arrival** of interrupts.
This enables back-to-back interrupt processing
without the overhead of state saving and restoration
between interrupts.

<https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vector-interrupt-controller-vic-nvic/>

NVIC features in cortex M (2)

- processor state automatically saved on interrupt entry,
restored on interrupt exit,
with no instruction overhead.
- Optional Wake-up Interrupt Controller (WIC),
providing ultra-low-power sleep mode support.
- Vector table can be located in either RAM or flash.

All interrupts including the core exceptions
are managed by the **NVIC**.

The **NVIC** maintains knowledge of
the stacked, or nested, interrupts
to enable tail-chaining of interrupts.

<https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/>

Nest VIC (1)

- In a microcontroller, such as those at the heart of industrial motion controllers, interrupts serve as a way to immediately divert the CPU from its current task to another, more important task.
- An interrupt can be triggered internally from the microcontroller (MCU) or externally, by a peripheral.
- the interrupt alerts the CPU to an occurrence such as a time-based event
 - a specified amount of time has elapsed or
 - a specific time is reached, for example,
- a change of state, or
- the start or end of a process.

<https://www.motioncontroltips.com/what-is-nested-vector-interrupt-control-nvic/>

Nest VIC (2-1)

- Another method of monitoring a timed event or change of state is referred to as “polling.”
- With polling, the status of a timer or state change is periodically checked.
- The downsides of polling are the risk of excessive latency (delay) between the actual change and its detection, the possibility of missing a change altogether, and the increased processing time and power it requires.
-
-
-

<https://www.motioncontroltips.com/what-is-nested-vector-interrupt-control-nvic/>

Nest VIC (2-2)

- When an interrupt occurs, an interrupt signal is generated, which causes the CPU to stop its current operation, save its current state, and begin the processing program
 - referred to as an interrupt service routine (ISR) or interrupt handler
 - associated with the interrupt.
-
- When the interrupt processing is complete,
- the CPU restores its previous state and resumes where it left off.

<https://www.motioncontroltips.com/what-is-nested-vector-interrupt-control-nvic/>

Nest VIC (3)

- Nested vector interrupt control (NVIC) is a method of prioritizing interrupts, improving the MCU's performance and reducing interrupt latency.
- NVIC also provides implementation schemes for handling interrupts that occur when other interrupts are being executed or when the CPU is in the process of restoring its previous state and resuming its suspended process.
-
- The term “nested” refers to the fact that in NVIC, a number of interrupts can be defined (up to several hundred in some processors), and each interrupt is assigned a priority, with “0” being the highest priority.
- In addition, the most critical interrupt can be made non-maskable, meaning it cannot be disabled (masked).

<https://www.motioncontroltips.com/what-is-nested-vector-interrupt-control-nvic/>

Nest VIC (4)

- One function of NVIC is to ensure that higher priority interrupts are completed before lower-priority interrupts, even if the lower-priority interrupt is triggered first.
- For example, if a lower-priority interrupt is being registered* or executed and a higher-priority interrupt occurs, the CPU will stop the lower-priority interrupt and process the higher-priority one first.
-
- * A register is a special, dedicated memory circuit within the CPU that can be written and read much more quickly than regular memory.
-
- The register is used to store information such as calculation results, CPU execution states, or other critical program information.

<https://www.motioncontroltips.com/what-is-nested-vector-interrupt-control-nvic/>

Nest VIC (5)

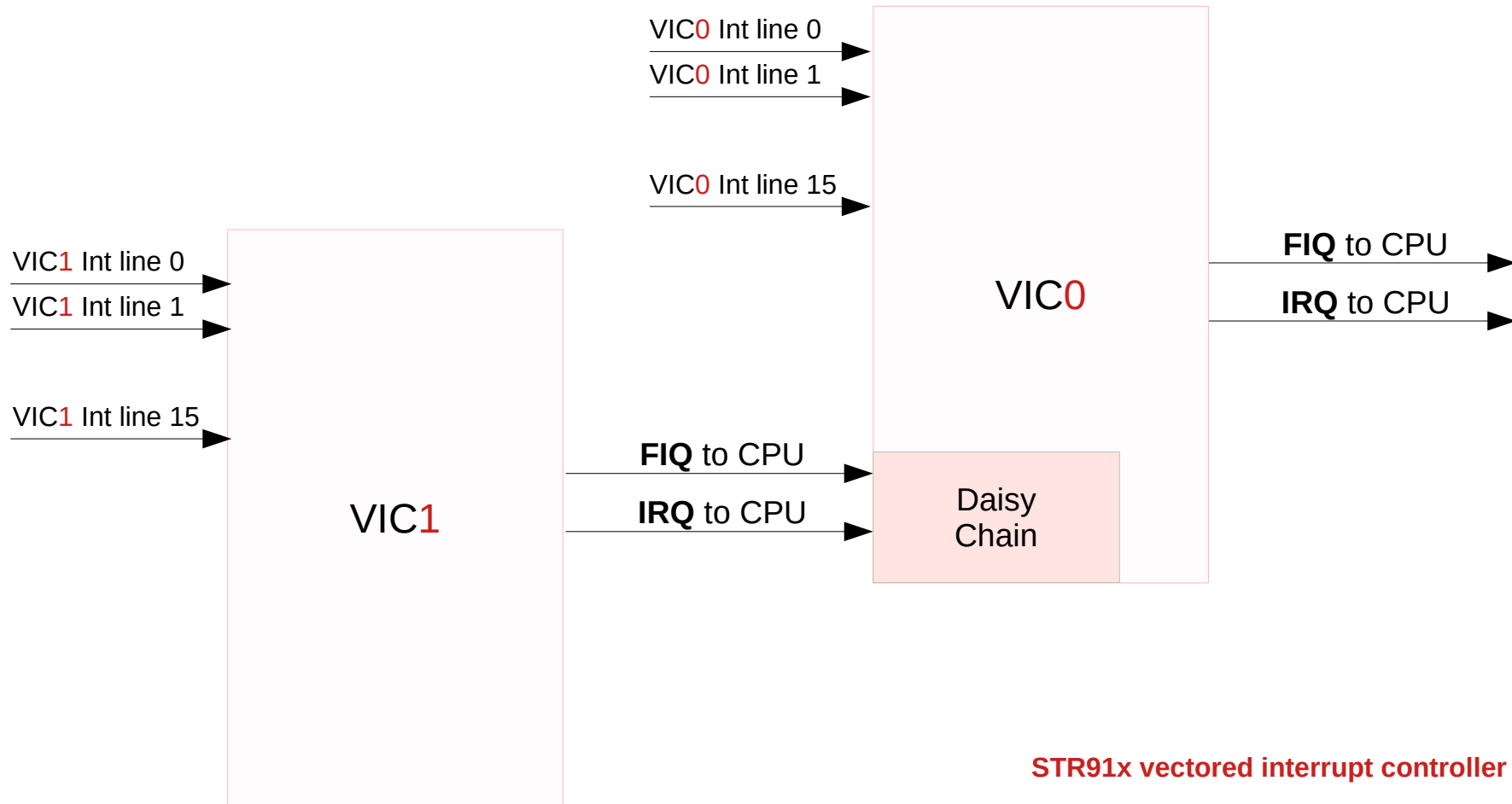
- Similarly, a handling scheme referred to as “tail-chaining” specifies that if an interrupt is pending while the ISR for another, higher-priority another interrupt completes, the processor will immediately begin the ISR for the next interrupt, without restoring its previous state.
-
- The term “vector” in nested vector interrupt control refers to the way in which the CPU finds the program, or ISR, to be executed when an interrupt occurs.
-
-

<https://www.motioncontroltips.com/what-is-nested-vector-interrupt-control-nvic/>

Nest VIC (6)

- Nested vector interrupt control uses a vector table that contains the addresses of the ISRs for each interrupt.
- When an interrupt is triggered, the processor gets the address from the vector table.
-
- The prioritization and handling schemes of nested vector interrupt control reduce the latency and overhead that interrupts typically introduce and
- ensure low power consumption, even with high interrupt loading on the controller.
-

<https://www.motioncontroltips.com/what-is-nested-vector-interrupt-control-nvic/>



https://www.st.com/resource/en/application_note/an2593-str91x-interrupt-management-stmicroelectronics.pdf

- for FIQ there are no priority levels.
- when an enabled FIQ interrupt occurs, the VIC signals it *directly* to the ARM core by asserting the FIQ interrupt line.
- then the ARM core switches to FIQ mode, and goes to address 0x1C where the FIQ interrupt handler resides

no priority levels

directly assert FIQ interrupt line

FIQ mode

FIQ interrupt handler at 0x1C

Interrupt	Configured Priority
VIC0 FIQ / VIC1 FIQ	NA
VIC0 IRQ	0
VIC0 IRQ	1
VIC0 IRQ	15
VIC1 IRQ	0
VIC1 IRQ	1
VIC1 IRQ	15



Highest priority

Lowest priority

<https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/>

- normally in order to minimize FIQ interrupt latency only one interrupt should be configured as FIQ.
- But it is possible to configure several interrupts as FIQ, and in this case the application software must read the FIQ status registers of both VIC0 and VIC1 in order to determine the FIQ interrupt source
- when the interrupt flag is cleared in the peripheral(s) that generated the interrupt, the VIC then will stop asserting the FIQ interrupt to CPU and the flag will be cleared in the VIC FIQ status register

one interrupt source

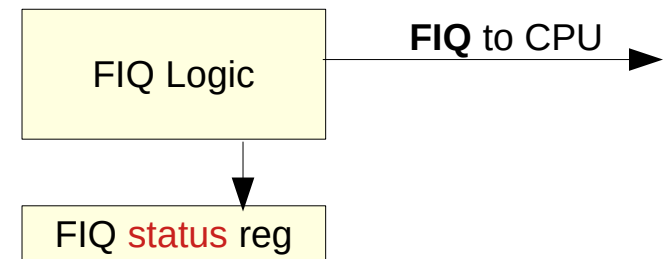
multiple interrupt sources

read the FIQ status reg


determine the FIQ source

interrupt flag in the peripherals

interrupt flag in the VIC status reg



<https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vector-interrupt-controller-vic-nvic/>

- 
1. Vectored IRQ handling
 2. Simple (Non vectored) IRQ handling

Vectored handling ensures the *best interrupt latency*

the hardware priority management of the VIC
- small latency

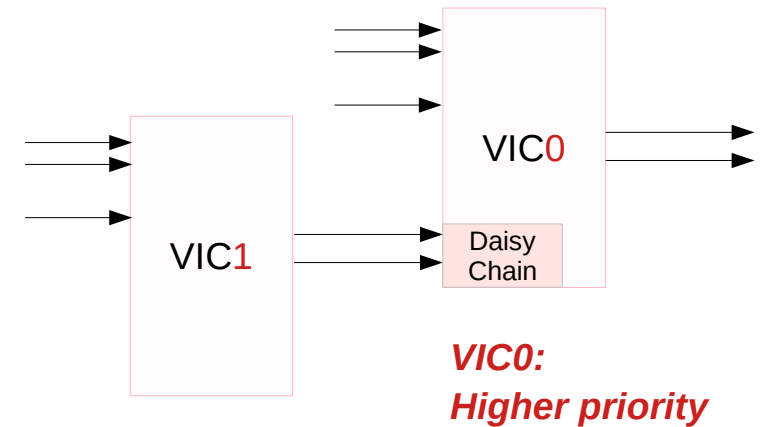
the software priority management of the VIC
- simple handling

<https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/>

Although a software priority management increases the interrupt latency, it can be useful in special cases where a **VIC1** interrupt has to be configured with a **higher** priority level than a **VIC0** interrupt,

- this is not possible when using the **hardware** priority management due to the **hardwired** priority between **VIC0** and **VIC1**

Interrupt	Configured Priority
VIC0 FIQ / VIC1 FIQ	NA
VIC0 IRQ	0
VIC0 IRQ	1
VIC0 IRQ	15
VIC1 IRQ	0
VIC1 IRQ	1
VIC1 IRQ	15



<https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/>

3. After handling the interrupt including the clearing of the interrupt flags, you must **write** any value in the **VIC0_VAR** if the interrupt originates from VIC0, or in the **VIC1_VAR** if the interrupt is from VIC1, in order to indicate to the VIC that interrupt processing has finished, so it can update the priority logic: then a same or lower level interrupt will be able to interrupt the CPU

<https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/>

Non-vector handling method of IRQ interrupts does not use the VIC hardware priority management,

so this means you do not have to read or write the VIC0_VAR or VIC1_VAR registers to update the hardware priority logic.

This method can be used when there is a need to give higher priority to a VIC1 interrupt over a VIC0 interrupt.

<https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectorized-interrupt-controller-vic-nvic/>

The flow for simple (non vectored) IRQ handling is the following:

1. An **IRQ** interrupt occurs.
2. Branch to the **interrupt handler**.
3. Read the VICs **IRQ Status** registers to determine the **source** that generated the interrupt, and **prioritize** the interrupts if there are multiple active interrupt sources.
4. Branch to the *corresponding* **ISR**.
5. Execute the **ISR**.
6. Clear the interrupt. If a **software interrupt** generated the request, you must write to the **VICx_SWINTCR** register.
7. Check the **IRQ Status** registers of both VICs to ensure that no other interrupt is active. If there is an active request, go to Step 4.
8. Return from the interrupt.

<https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/>

the sequence for the vectored interrupt flow:

- **VICVectAddr** Register
read to branch to the **ISR** (interrupt service routine),
which is currently active.
write to **clear** the respective interrupt
- **VICSoftIntClear** Register
to **clear** the software interrupt request triggered by **VICSoftInt**

<https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/>

the sequence for the vectored interrupt flow:

- When an interrupt occurs, The ARM processor branches to either the **IRQ** or **FIQ interrupt vector**.
- If the interrupt is an **IRQ**, read the **VICVectAddr Register** and branch to the **ISR** (interrupt service routine).
- **Stack** the workspace so that you can re-enable IRQ interrupts.
- **Enable** the **IRQ interrupts** so that a **higher priority** can be serviced.
- **Execute** the Interrupt Service Routine (**ISR**).
- **Clear** the requesting interrupt in the peripheral, or write to the **VICSoftIntClear Register** if the request was generated by a **software interrupt**.
- **Disable** the interrupts and **restore** the workspace.
- **Write** to the **VICVectAddr Register**. This **clears** the respective interrupt in the internal interrupt priority hardware.
- **Return** from the interrupt. This **re-enables** the interrupts.

<https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/>

Interrupt Source

VicVecCntl0	IRQ source 0
VicVecCntl1	IRQ source 1
VicVecCntl2	IRQ source 2
VicVecCntl3	IRQ source 3
VicVecCntl4	IRQ source 4
VicVecCntl5	IRQ source 5
VicVecCntl6	IRQ source 6
VicVecCntl7	IRQ source 7
VicVecCntl8	IRQ source 8
VicVecCntl9	IRQ source 9
VicVecCntl10	IRQ source 10
VicVecCntl11	IRQ source 11
VicVecCntl12	IRQ source 12
VicVecCntl13	IRQ source 13
VicVecCntl14	IRQ source 14
VicVecCntl15	IRQ source 15

Service Routine

VicVecAddr0	ISR 0 address
VicVecAddr1	ISR 1 address
VicVecAddr2	ISR 2 address
VicVecAddr3	ISR 3 address
VicVecAddr4	ISR 4 address
VicVecAddr5	ISR 5 address
VicVecAddr6	ISR 6 address
VicVecAddr7	ISR 7 address
VicVecAddr8	ISR 8 address
VicVecAddr9	ISR 9 address
VicVecAddr10	ISR 10 address
VicVecAddr11	ISR 11 address
VicVecAddr12	ISR 12 address
VicVecAddr13	ISR 13 address
VicVecAddr14	ISR 14 address
VicVecAddr15	ISR 15 address



Highest priority

Lowest priority

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

the '*magnitude*' : the interrupt source ID
the 'source' of the currently pending IRQ

VicVecCntl0~15

the '*direction*' : the corresponding ISR
vectored IRQ '*points to*' its own unique ISR

VicVecAddr0~15

Non-Vectored IRQs does not point to a unique ISR
instead, **default / common** ISR

In LPC214x, **VICDefVectAddr** register is used
The user must assign the address of the **default** ISR

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

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 LPC2148 ARM7 Implementation.

VicVecCntl0~15

VicVecAddr0~15

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.

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

Bit 0 : WDT

Bit 1 : N/A

Bit 2 : ARMC0

Bit 3 : ARMC1

Bit 4 : TIMR0

Bit 5 : TIMR1

Bit 6 : UART0

Bit 7 : UART1

Bit 8 : PWM

Bit 9 : I2C0

Bit10 : I2C0

Bit11 : SPI1/SSP

Bit12 : PLL

Bit13 : RTC

Bit14 : EINT0

Bit15 : EINT1

Bit16 : EINT2

Bit17 : EINT3

Bit18 : AD0

Bit19 : I2C1

Bit20 : BOD

Bit21 : AD1

Bit22 : USB

Interrupt Source Encoding

22 requests

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

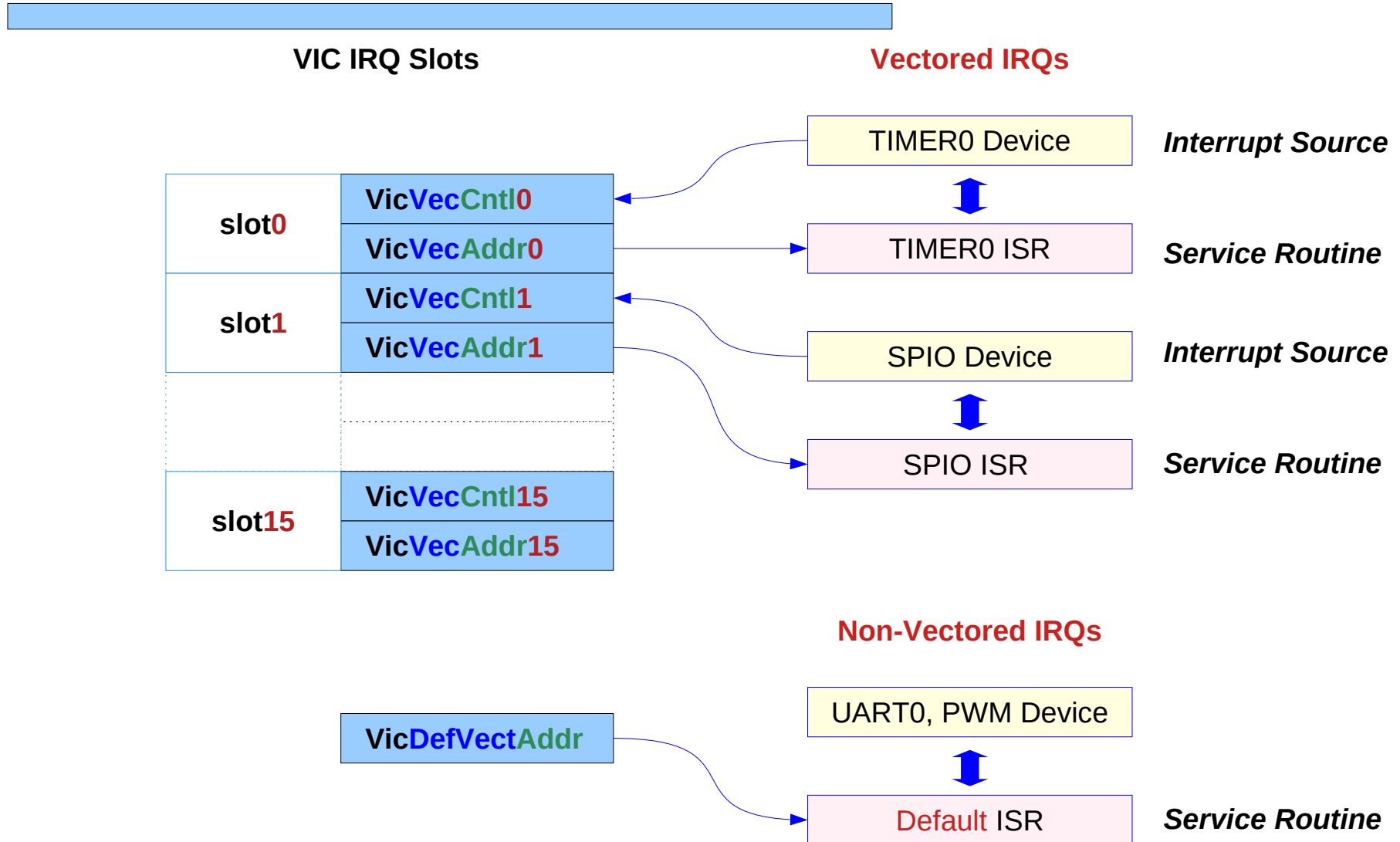
For example if you working with 2 interrupt sources
UART0 and TIMER0.

Now if you want to give TIMER0 a higher priority than UART0
then assign TIMER0 interrupt a lower number slot than UART0 .

eg. TIMER0 to slot 0 and UART0 to slot 1 or
TIMER0 to slot 4 and UART to slot 9 and so on.

The number of the slot doesn't matter
as long TIMER0 slot is lower than UART0 slot.

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>



<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>



VIC has plenty of registers.

Most of the registers that are used to configure interrupts or read status

each **bit** corresponds to a particular interrupt **source** and this correspondence is same for all of these registers.

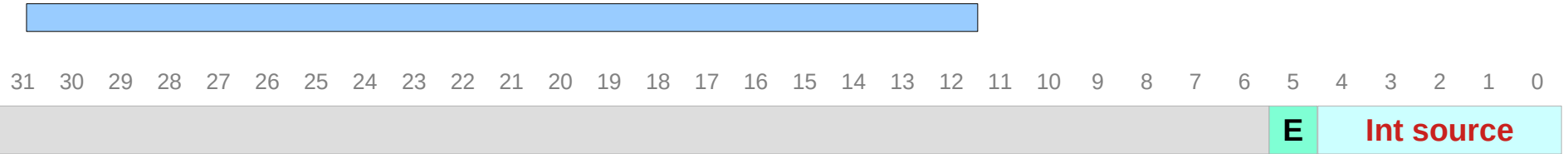
For example

- 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.

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

- 1) **VICIntSelect** (R/W) : used to **select** an interrupt as IRQ or as FIQ
- 2) **VICIntEnable** (R/W): used to **enable** interrupts
- 3) **VICIntEnClr** (R/W) : used to **disable** interrupts
- 4) **VICIRQStatus** (R) : used for reading the current **status** of the enabled **IRQ** interrupts.
- 5) **VICFIQStatus** (R) : used for reading the current **status** of the enabled **FIQ** interrupts
- 6) **VICSoftInt** : used to generate interrupts using **software** i.e the program itself
- 7) **VICSoftIntClear** : used to **clear** the interrupt request that was triggered(forced) using **VICSoftInt**.
- 8) **VICVectCntl0 ~15** : used to assign a particular interrupt source to a particular slot.
- 9) **VICVectAddr0 ~15** : store the **address** of the function that must be called when an interrupt occurs
- 10) **VICVectAddr** : holds the **address** of the associated ISR i.e the one which is currently active.
- 11) **VICDefVectAddr** : stores the **address** of the “**default**/common” ISR for a **Non-Vectored IRQ** occurs

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>



VICVectCntl0 ~ 15 : used to assign a particular interrupt source to a particular slot.

VICVectCntl0 - the highest priority VICVectCntl15 - the lowest priority

Bit4 ~ Bit0 contain the number of the interrupt request which is assigned to this slot.

Bit5 is used to enable the **vectored IRQ** slot by writing a 1

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

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

defining the ISR

explicitly tell the compiler that the function is not a normal function but an ISR

a special keyword called “`__irq`”
: a **function qualifier**.

use this keyword with the **function definition**

an example of defining an ISR in Keil :

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

// or equivalently

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

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

for ARM based microcontrollers like lpc2148.

in order to assign **TIMER0** IRQ and ISR to slot **X**.

Assign **TIMER0** Interrupt to Slot number **0**

```
// Enable TIMER0 IRQ  
// 5th bit must 1 to enable the slot  
// Vectored-IRQ for TIMER0 has been configured
```

```
VICIntEnable |= (1<<4) ;  
VICVectCntl0 = (1<<5) | 4 ;  
VICVectAddr0 = (unsigned) myISR;
```

- 2) **VICIntEnable** (R/W) : used to **enable** interrupts
- 8) **VICVectCntl0** ~15 : used to assign a particular interrupt source to a particular slot.
- 9) **VICVectAddr0** ~15 : store the **address** of the function that must be called when an interrupt occurs

Bit 0	: WDT
Bit 1	: N/A
Bit 2	: ARMC0
Bit 3	: ARMC1
Bit 4	: TIMER0
Bit 5	: TIMR1
Bit 6	: UART0
Bit 7	: UART1
Bit 8	: PWM
Bit 9	: I2C0
Bit10	: I2C0

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

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

T0IR (TIMER0 Interrupt Register)

4 bits for the *timer match* interrupts

4 bits for the *timer capture* interrupts

The **high** bit in the IR signifies that an **interrupt** is generated

Writing a logic **one** to the corresponding IR bit will **reset** the **interrupt**.

Writing a **zero** has no effect.

U0IIR (UART0 Interrupt Identification Register)

The U0IIR provides a status code that denotes the **priority** and **source** of a **pending interrupt**.

The interrupts are **frozen** during an **U0IIR access**.

If an interrupt occurs during an U0IIR access, the interrupt is **recorded** for the next U0IIR access

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

The **IR** can be read to **identify** which of 8 possible **interrupt** sources are **pending**.

The **IR** can be written to **clear** interrupts.

TIMER/ COUNTER0 **T0IR**
TIMER/ COUNTER1 **T1IR**

The **Interrupt Register** consists of four bits for the **match interrupts** and four 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

Bit 0	: MR0 Interrupt	flag for match channel 0
Bit 1	: MR1 Interrupt	flag for match channel 1
Bit 2	: MR2 Interrupt	flag for match channel 2
Bit 3	: MR3 Interrupt	flag for match channel 3
Bit 4	: CR0 Interrupt	flag for capture channel 0 event
Bit 5	: CR1 Interrupt	flag for capture channel 1 event
Bit 6	: CR2 Interrupt	flag for capture channel 2 event
Bit 7	: CR3 Interrupt	flag for capture channel 3 event

A **high bit** signifies the **interrupt** is generated

```
#define MR0I_FLAG (1<<0)            // 0x00000001
#define MR1I_FLAG (1<<1)            // 0x00000002
#define MR2I_FLAG (1<<2)            // 0x00000004
***
regVal = T0IR;

if( T0IR & MR0I_FLAG )            {
    *** MR0 match ***
} else if ( T0IR & MR1I_FLAG ) {
    *** MR1 match ***
} else if ( T0IR & MR2I_FLAG ) {
    *** MR2 match ***
}

T0IR = regval;
***
```

https://www.keil.com/dd/docs/datashts/philips/user_manual_ipc214x.pdf

Note than UART0's Interrupt Register (**U0IIR**) is a lot different than TIMER0's (**T0IR**).

The first Bit **UART0[0]** in **U0IIR** indicates whether any interrupt is pending or not and its Active LOW!

The next 3 bits **UART0[3:1]** give the Identification for any of the 4 Interrupts if enabled.

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

Bit0 **Interrupt Pending**
the U0IIR[0] is active low
the pending interrupt can be determined by evaluating U0IIR[3:1]

Bit3:1 **Interrupt Identification**
U0IIR[3:1] identifies an interrupt corresponding to the UART0 Rx FIFO
All other combinations of U0IIR[3:1] not list are reserved
(000, 100, 101, 111)

- 011 1 RLS (Receive Line Status)
- 010 2a RDA (Receive Data Available)
- 110 2b CTI (Character Time-Out Indicator)
- 001 3 THRE (Transmitter Holding Register Empty)

```
__irq void myDefault_ISR(void)
{
    U0RegVal = U0IIR; // read the current value
    ...
    if (!(U0RegVal & 0x1)) // active low
    {
    }
    ...
    VICVectAddr = 0x0; // The ISR has finished!
}
```

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

IOPIN 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

IODIR GPIO Port **Direction control** register.

This register individually controls the direction of each port pin.

IOSET GPIO Port **Output Set** register.

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.

IOCLR GPIO Port **Output Clear** register.

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.

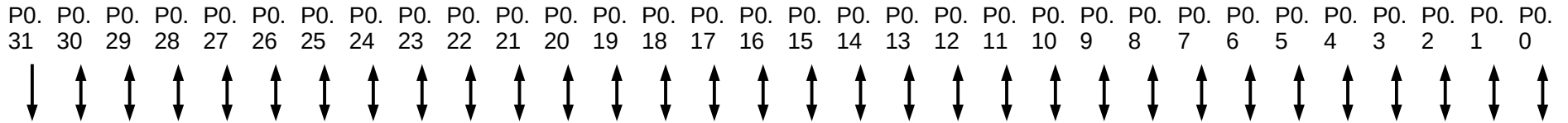
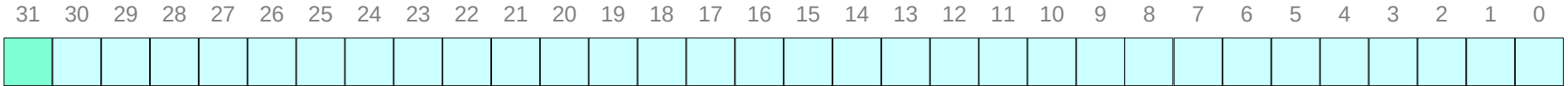
the **legacy GPIO** referred as "the slow" GPIO

the **enhanced GPIO** referred as "the fast" GPIO.

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

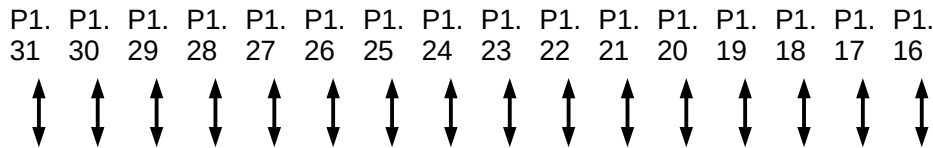
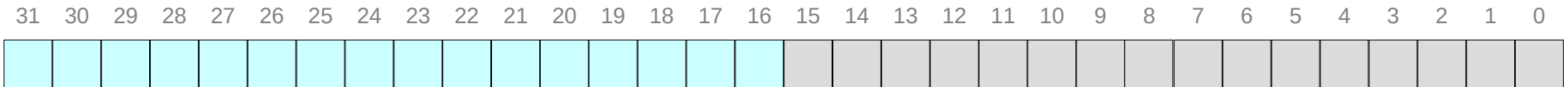
IODIR0 for Port 0

1 for output / 0 for input



IODIR1 for Port 1

1 for output / 0 for input



<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

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

FIOMASK Fast Mask register for port.
Writes, sets, clears, and reads to port alter or return only the bits enabled by **zeros** in this register.
(done via **writes** to **FIOPIN**, **FIOSET**, and **FIOCLR**, and **reads** of **FIOPIN**)

FIOPIN Fast Port Pin value register using **FIOMASK**.
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 is not configured as an input to ADC).
The value read is **masked** by ANDing with **FIOMASK**.
Writing to this register places corresponding values in all bits enabled by **ones** in **FIOMASK**.

FIOSET Fast Port Output Set register using **FIOMASK**.
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.
Only bits enabled by **ones** in **FIOMASK** can be **altered**

FIOCLR Fast Port Output Clear register using **FIOMASK**.
This register controls the state of output pins.
Writing 1s produces **lows** at the corresponding port pins.
Writing 0s has no effect.
Only bits enabled by **ones** in **FIOMASK** can be **altered**.

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

```
IO0DIR = 0xFFFFFFFF; // Configure all pins on Port 0 as Output
IO0PIN = 0x0;
```

```
IO0PIN = ~IO0PIN; // Toggle all pins in Port 0
```

```
IO0PIN ^= (1<<0); // xor 2^0 Toggle GPIO0 PIN0 .. P0.0
IO0PIN ^= (1<<1); // xor 2^1 Toggle GPIO0 PIN1 .. P0.1
IO0PIN ^= (1<<2); // xor 2^2 Toggle GPIO0 PIN2 .. P0.2
```

```
IO0PIN ^= (1<<2); // Toggle 3rd Pin (PIN2) in GPIO0 .. P0.2
IO0PIN ^= (1<<3); // Toggle 4th Pin (PIN3) in GPIO0 .. P0.3
```

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

This must not be confused with the set of 16 `VICVecAddr0 ~15` 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.

```
__irq void myDefault_ISR(void)
{
    U0RegVal = U0IIR;    // read the current value
    ...
    if( !(U0RegVal & 0x1) )    // active low
    {
    }
    ...
    VICVectAddr = 0x0;    // The ISR has finished!
}
```

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

consider two simple cases for coding an ISR

Use **TIMER0** for generating IRQs

Case #1)

only one 'internal' source of interrupt in **TIMER0**
i.e an **MR0** match event which raises an IRQ.

Case #2)

multiple 'internal' source of interrupt in **TIMER0**
i.e. say a match event for **MR0** , **MR1** & **MR2**
which raise an IRQ.

TOIR for **TIMER0**
T0's Interrupt Register

```
regVal = TOIR;  
    *** MR0 ***  
TOIR = regval;
```

```
regVal = TOIR;  
  
if( TOIR & MR0I_FLAG ) {  
    *** MR0 match ***  
} else if ( TOIR & MR1I_FLAG ) {  
    *** MR1 match ***  
} else if ( TOIR & MR2I_FLAG ) {  
    *** MR2 match ***  
}  
  
TOIR = regval;
```

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

Since only one source is triggering an **interrupt** we don't need to identify it
– though its a good practice to explicitly identify it.

```
__irq void myISR(void)
{
    long int regVal;

    // read the current value in
    // TIMERO's Interrupt Register
    regVal = TOIR;

    //... MR0 match event has occurred
    // .. do something here

    // write back to clear the interrupt flag
    TOIR = regval;

    VICVectAddr = 0x0; // The ISR has finished!
}
```

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

Even in case #2 things are simple unless we need to identify the 'actual' source of interrupt.

```
#define MR0_FLAG (1<<0)
#define MR1_FLAG (1<<1)
#define MR2_FLAG (1<<2)

__irq void myISR(void)
{
    long int regVal;

    // read the current value
    // in TIMER0's Interrupt Register
    regVal = TOIR;

    // write back to clear the interrupt flag
    TOIR = regVal;

    // Acknowledge that ISR has finished execution
    VICVectAddr = 0x0;
}
```

```
if( TOIR & MR0_FLAG ) {
    //do something for MR0 match
} else if ( TOIR & MR1_FLAG ) {
    //do something for MR1 match
} else if ( TOIR & MR2_FLAG ) {
    //do something for MR2 match
}
```

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

Case #2 actually provides a general method of using **Timers** as **PWM** generators!

You can use any one of the match registers as PWM Cycle generator and then use other 3 match registers to generate 3 PWM signals!

Since LPC214x already has PWM generator blocks on chip I don't see any use of Timers being used as PWM generators.

But for MCUs which don't have PWM generator blocks this is very useful.

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

Both of them deal with IRQs from different blocks
: **TIMER0** and **UART0**.

Case #3)

Multiple **Vectored IRQs** from different devices.
Hence **Priority** comes into picture here.

myTimer0_ISR()
myUart0_ISR

Case #4)

Multiple **Non-Vectored IRQs** from different devices.

myDefault_ISR

TOIR for **TIMER0**
T0's Interrupt Register

UOIR for **UART0**
U0's Interrupt ID Register

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

Case #3

TIMER0 and UART0 generating interrupts with TIMER0 having higher priority.

2 different Vectored ISRs

– one for TIMER0 and one for UART0.

`myTimer0_ISR()`

`myUart0_ISR`

assume only 1 internal source inside both TIMER0 and UART0

```
__irq void myTimer0_ISR(void)
{
    long int regVal;

    regVal = TOIR;           // read the current value

    TOIR = regval;          // write back to clear
                            // the interrupt flag

    VICVectAddr = 0x0;

}
```

```
__irq void myUart0_ISR(void)
{
    long int regVal;

    regVal = UOIR;          // read the current value

    //Something inside UART0 has raised an IRQ

    VICVectAddr = 0x0;

}
```

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

Case #4

TIMERO and UART0 generating interrupts

But here both of them are **Non-Vectored** and hence will be serviced by a common **Non-Vectored ISR**.

Hence, here we will need to check the actual **source** i.e **device** which triggered the interrupt and proceed accordingly.

This is quite similar to Case #2.

T0's Interrupt Register

U0's(Uart 0) Interrupt Identification Register

```
__irq void myDefault_ISR(void)
{
    long int T0RegVal , U0RegVal;

    T0RegVal = T0IR;    // read the current value
    U0RegVal = U0IIR;   // read the current value

    if( T0RegVal )
    {
        //do something for TIMERO Interrupt

        T0IR = T0RegVal;    // write back to clear
                            // the interrupt flag
    }

    if( !(U0RegVal & 0x1) )    // active low
    {
        // do something for UART0 Interrupt
        // No need to write back to U0IIR
        // since reading it clears it
    }

    VICVectAddr = 0x0;    // The ISR has finished!
}
```

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

Well , you can think FIQ as a promoted version of a Vectored IRQ.

To promote or covert a Vectored IRQ to FIQ
just make the bit for corresponding IRQ
in VICIntSelect register to 1
and it will be become an FIQ.

Also Note that its recommended
that you only have one FIQ in your system.

FIQs have low latency than VIRQs
and usually used in System Critical Interrupt Handling.

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

```
/*  
*****  
/* This file is part of the uVision/ARM development tools */  
/* Copyright KEIL ELEKTRONIK GmbH 2002-2005 */  
*****  
/*  
/* LPC214X.H: Header file for Philips LPC2141/42/44/46/48 */  
/*  
*****  
  
/* Vectored Interrupt Controller (VIC) */  
#define VICIRQStatus ((volatile unsigned long *) 0xFFFFF000)  
#define VICFIQStatus ((volatile unsigned long *) 0xFFFFF004)  
#define VICRawIntr ((volatile unsigned long *) 0xFFFFF008)  
#define VICIntSelect ((volatile unsigned long *) 0xFFFFF00C)  
#define VICIntEnable ((volatile unsigned long *) 0xFFFFF010)  
#define VICIntEnClr ((volatile unsigned long *) 0xFFFFF014)  
#define VICSoftInt ((volatile unsigned long *) 0xFFFFF018)  
#define VICSoftIntClr ((volatile unsigned long *) 0xFFFFF01C)  
#define VICProtection ((volatile unsigned long *) 0xFFFFF020)  
#define VICVectAddr ((volatile unsigned long *) 0xFFFFF030)  
#define VICDefVectAddr ((volatile unsigned long *) 0xFFFFF034)
```

<https://www.keil.com/dd/docs/arm/philips/lpc214x.h>


```
#define VICVectAddr0    (*((volatile unsigned long *) 0xFFFFF100))
#define VICVectAddr1    (*((volatile unsigned long *) 0xFFFFF104))
    ...
#define VICVectAddr15  (*((volatile unsigned long *) 0xFFFFF13C))

#define VICVectCntl0    (*((volatile unsigned long *) 0xFFFFF200))
#define VICVectCntl1    (*((volatile unsigned long *) 0xFFFFF204))
    ...
#define VICVectCntl15  (*((volatile unsigned long *) 0xFFFFF23C))
```

<https://www.keil.com/dd/docs/arm/philips/lpc214x.h>

```
/* Timer 0 */
#define T0IR      (*((volatile unsigned long *) 0xE0004000)) // Interrupt Register (IR)
#define T0TCR    (*((volatile unsigned long *) 0xE0004004)) // Timer Control Register (TCR)
#define T0TC     (*((volatile unsigned long *) 0xE0004008)) // Timer Counter (TC)
#define T0PR     (*((volatile unsigned long *) 0xE000400C)) // Prescale Register (PR)
#define T0PC     (*((volatile unsigned long *) 0xE0004010)) // Prescale Counter Register (PC)
#define T0MCR    (*((volatile unsigned long *) 0xE0004014)) // Match Control Register (MCR)
#define T0MR0    (*((volatile unsigned long *) 0xE0004018)) // Match Register 0 (MR0)
#define T0MR1    (*((volatile unsigned long *) 0xE000401C)) // Match Register 1 (MR1)
#define T0MR2    (*((volatile unsigned long *) 0xE0004020)) // Match Register 2 (MR2)
#define T0MR3    (*((volatile unsigned long *) 0xE0004024)) // Match Register 3 (MR3)
#define T0CCR    (*((volatile unsigned long *) 0xE0004028)) // Capture Control Register (CCR)
#define T0CR0    (*((volatile unsigned long *) 0xE000402C)) // Capture Register 1 (CR1)
#define T0CR1    (*((volatile unsigned long *) 0xE0004030)) // Capture Register 2 (CR2)
#define T0CR2    (*((volatile unsigned long *) 0xE0004034)) // Capture Register 3 (CR3)
#define T0CR3    (*((volatile unsigned long *) 0xE0004038)) // Capture Register 4 (CR4)
#define T0EMR    (*((volatile unsigned long *) 0xE000403C)) // External Match Register (EMR)
#define T0CTCR   (*((volatile unsigned long *) 0xE0004070)) // Counter Control Register (CTCR)
```

<https://www.keil.com/dd/docs/arm/philips/lpc214x.h>

```
/* Universal Asynchronous Receiver Transmitter 0 (UART0) */
#define U0RBR      (*((volatile unsigned char *) 0xE000C000)) // Receiver Buffer Register (RBR)
#define U0THR      (*((volatile unsigned char *) 0xE000C000)) // Transmit Holding Register (THR)
#define U0IER      (*((volatile unsigned long *) 0xE000C004)) // Interrupt Enable Register (IER)
#define U0IIR      (*((volatile unsigned long *) 0xE000C008)) // Interrupt Identification Register (IIR)
#define U0FCR      (*((volatile unsigned char *) 0xE000C008)) // FIFO Control Register (FCR)
#define U0LCR      (*((volatile unsigned char *) 0xE000C00C)) // Line Control Register (LCR)
#define U0MCR      (*((volatile unsigned char *) 0xE000C010)) // Modem Control Register (U1MCR)
#define U0LSR      (*((volatile unsigned char *) 0xE000C014)) // Line Status Register (LSR)
#define U0MSR      (*((volatile unsigned char *) 0xE000C018)) // Modem Status Register (U1MSR)
#define U0SCR      (*((volatile unsigned char *) 0xE000C01C)) // Scratch Pad Register (SCR)
#define U0DLL      (*((volatile unsigned char *) 0xE000C000)) // Divisor Latch LSB Register (DLL)
#define U0DLM      (*((volatile unsigned char *) 0xE000C004)) // Divisor Latch MSB Register (DLM)
#define U0ACR      (*((volatile unsigned long *) 0xE000C020)) // Auto-baud Control Register (ACR)
#define U0FDR      (*((volatile unsigned long *) 0xE000C028)) // Fraction Divisor Register (FDR)
#define U0TER      (*((volatile unsigned char *) 0xE000C030)) // Transmit Enable Register (TER)
```

<https://www.keil.com/dd/docs/arm/philips/lpc214x.h>

```
/* General Purpose Input/Output (GPIO) */
```

```
#define IOPIN0      (*((volatile unsigned long *) 0xE0028000))
#define IOSET0     (*((volatile unsigned long *) 0xE0028004))
#define IODIR0     (*((volatile unsigned long *) 0xE0028008))
#define IOCLR0     (*((volatile unsigned long *) 0xE002800C))
#define IOPIN1     (*((volatile unsigned long *) 0xE0028010))
#define IOSET1     (*((volatile unsigned long *) 0xE0028014))
#define IODIR1     (*((volatile unsigned long *) 0xE0028018))
#define IOCLR1     (*((volatile unsigned long *) 0xE002801C))
#define IO0PIN     (*((volatile unsigned long *) 0xE0028000)) // alias
#define IO0SET     (*((volatile unsigned long *) 0xE0028004)) // alias
#define IO0DIR     (*((volatile unsigned long *) 0xE0028008)) // alias
#define IO0CLR     (*((volatile unsigned long *) 0xE002800C)) // alias
#define IO1PIN     (*((volatile unsigned long *) 0xE0028010)) // alias
#define IO1SET     (*((volatile unsigned long *) 0xE0028014)) // alias
#define IO1DIR     (*((volatile unsigned long *) 0xE0028018)) // alias
#define IO1CLR     (*((volatile unsigned long *) 0xE002801C)) // alias
#define FIO0DIR    (*((volatile unsigned long *) 0x3FFFC000))
#define FIO0MASK   (*((volatile unsigned long *) 0x3FFFC010))
#define FIO0PIN    (*((volatile unsigned long *) 0x3FFFC014))
#define FIO0SET    (*((volatile unsigned long *) 0x3FFFC018))
#define FIO0CLR    (*((volatile unsigned long *) 0x3FFFC01C))
#define FIO1DIR    (*((volatile unsigned long *) 0x3FFFC020))
#define FIO1MASK   (*((volatile unsigned long *) 0x3FFFC030))
#define FIO1PIN    (*((volatile unsigned long *) 0x3FFFC034))
#define FIO1SET    (*((volatile unsigned long *) 0x3FFFC038))
#define FIO1CLR    (*((volatile unsigned long *) 0x3FFFC03C))
```

<https://www.keil.com/dd/docs/arm/philips/lpc214x.h>

```
/*
(C) Umang Gajera | Power_user_EX - www.ocfreaks.com 2011-13.
More Embedded tutorials @ www.ocfreaks.com/cat/embedded

Source for Interrupt Tutorial Case #1.
License : GPL.
*/

#include <lpc214x.h>

#define MR0I          (1<<0)    // Interrupt When TC matches MR0
#define MR0R          (1<<1)    // Reset TC when TC matches MR0

#define DELAY_MS      500       // 0.5 Second(s) Delay
#define PRESCALE      60000     // 60000 PCLK clock cycles to increment TC by 1

void initClocks(void);
void initTimer0(void);
__irq void TOISR(void);
void initClocks(void);
```

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

```
int main(void)
{
    initClocks(); //Initialize CPU and Peripheral Clocks @ 60Mhz
    initTimer0(); //Initialize Timer0
    IOODIR = 0xFFFFFFFF; //Configure all pins on Port 0 as Output
    IOOPIN = 0x0;

    T0TCR = 0x01; //Enable timer

    while(1); //Infinite Idle Loop

    //return 0; //normally this wont execute ever :P
}
```

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

```
void initTimer0(void)
{
    /*Assuming that PLL0 has been setup with CCLK = 60Mhz and PCLK also = 60Mhz.*/

    //-----Configure Timer0-----
    T0CTCR = 0x0;

    T0PR = PRESCALE-1; //(Value in Decimal!) - Increment T0TC at every 60000 clock cycles
        //Count begins from zero hence subtracting 1
        //60000 clock cycles @60Mhz = 1 mS

    T0MRO = DELAY_MS-1; //(Value in Decimal!) Zero Indexed Count - hence subtracting 1

    T0MCR = MR0I | MR0R; //Set bit0 & bit1 to High which is to : Interrupt & Reset TC on MR0

    //-----Setup Timer0 Interrupt-----
    VICVectAddr4 = (unsigned) T0ISR; //Pointer Interrupt Function (ISR)

    VICVectCntl4 = 0x20 | 4; //0x20 (i.e bit5 = 1) -> to enable Vectored IRQ slot
        //0x4 (bit[4:0]) -> this the source number - here its timer0 which has VIC channel mask # as 4
        //You can get the VIC Channel number from Lpc214x manual R2 - pg 58 / sec 5.5

    VICIntEnable = 0x10; //Enable timer0 int

    T0TCR = 0x02; //Reset Timer
}
http://www.ocfreaks.com/lpc2148-interrupt-tutorial/
```

```
__irq void TOISR(void)
{
    long int regVal;
    regVal = T0IR;           // Read current IR value

    IO0PIN = ~IO0PIN;      // Toggle all pins in Port 0

    T0IR = regVal;         // Write back to IR to clear Interrupt Flag
    VICVectAddr = 0x0;     // This is to signal end of interrupt execution
}

void initClocks(void)
{
    // This function is used to config PPL0 and setup both
    // CPU and Peripheral clock @ 60Mhz
    // You can find its definition in the attached files or case #2 source
}
```

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>


```
/*
(C) Umang Gajera | Power_user_EX - www.ocfreaks.com 2011-13.
More Embedded tutorials @ www.ocfreaks.com/cat/embedded

LPC2148 Interrupt Example.
License : GPL.
*/

#include <lpc214x.h>

#define PLOCK                0x00000400
#define MR0I                 (1<<0)    // Interrupt When TC matches MR0
#define MR1I                 (1<<3)    // Interrupt When TC matches MR1
#define MR2I                 (1<<6)    // Interrupt When TC matches MR2
#define MR2R                 (1<<7)    // Reset TC when TC matches MR2

#define MR0I_FLAG            (1<<0)    // Interrupt Flag for MR0
#define MR1I_FLAG            (1<<1)    // Interrupt Flag for MR1
#define MR2I_FLAG            (1<<2)    // Interrupt Flag for MR2

#define MR0_DELAY_MS        500       // 0.5 Second(s) Delay
#define MR1_DELAY_MS        1000      // 1 Second Delay
#define MR2_DELAY_MS        1500      // 1.5 Second(s) Delay
#define PRESCALE             60000     // 60000 PCLK clock cycles to increment TC by 1
```

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

```
void delayMS(unsigned int milliseconds);
void initClocks(void);
void initTimer0(void);
__irq void myTimer0_ISR(void);

void setupPLL0(void);
void feedSeq(void);
void connectPLL0(void);

int main(void)
{
    initClocks();           // Initialize CPU and Peripheral Clocks @ 60Mhz
    initTimer0();          // Initialize Timer0
    IO0DIR = 0xFFFFFFFF;   // Configure all pins on Port 0 as Output
    IO0PIN = 0x0;

    TOTCR = 0x01;         // Enable timer

    while(1);             // Infinite Idle Loop

    //return 0;           // normally this wont execute ever :P
}
```

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

```
void initTimer0(void)
{
    /*Assuming that PLL0 has been setup with CCLK = 60Mhz and PCLK also = 60Mhz.*/

    //-----Configure Timer0-----
    T0CTCR = 0x0;
    T0PR = PRESCALE-1;           // 60000 clock cycles @60Mhz = 1 mS
    T0MR0 = MR0_DELAY_MS-1;     // 0.5sec (Value in Decimal!) Zero Indexed Count - hence subtracting 1
    T0MR1 = MR1_DELAY_MS-1;     // 1sec
    T0MR2 = MR2_DELAY_MS-1;     // 1.5secs
    T0MCR = MR0I | MR1I | MR2I | MR2R; // Set the Match control register

    //-----Setup Timer0 Interrupt----- // I've just randomly picked-up slot 4
    VICVectAddr4 = (unsigned) myTimer0_ISR; // Pointer Interrupt Function (ISR)
    VICVectCntl4 = 0x20 | 4;
    VICIntEnable = 0x10;        // Enable timer0 int
    T0TCR = 0x02;              // Reset Timer
}
```

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

```
__irq void myTimer0_ISR(void)
{
    long int regVal;
    regVal = T0IR;           // read the current value in T0's Interrupt Register

    if( T0IR & MR0I_FLAG ) {
        //do something for MR0 match

        IO0PIN ^= (1<<0);   // Toggle GPIO0 PIN0 .. P0.0
    }
    else if ( T0IR & MR1I_FLAG ) {
        //do something for MR1 match

        IO0PIN ^= (1<<1);   // Toggle GPIO0 PIN1 .. P0.1
    }
    else if ( T0IR & MR2I_FLAG ) {
        //do something for MR0 match

        IO0PIN ^= (1<<2);   // Toggle GPIO0 PIN2 .. P0.2
    }

    T0IR = regVal;         // write back to clear the interrupt flag
    VICVectAddr = 0x0;    // Acknowledge that ISR has finished execution
}
```

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

```
void initClocks(void)
{
    setupPLL0();
    feedSeq();           // sequence for locking PLL to desired freq.
    connectPLL0();
    feedSeq();           // sequence for connecting the PLL as system clock

    // SysClock is now ticking @ 60Mhz!

    VPBDIV = 0x01;      // PCLK is same as CCLK i.e 60Mhz

    // PLL0 Now configured!
}
```

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

```
//-----PLL Related Functions :-----
```

```
// Using PLL settings as shown in : http://www.ocfreaks.com/lpc214x-pll-tutorial-for-cpu-and-peripheral-clock/
```

```
void setupPLL0(void)
```

```
{
```

```
    // Note : Assuming 12Mhz Xtal is connected to LPC2148.
```

```
    PLL0CON = 0x01;
```

```
    PLL0CFG = 0x24;
```

```
}
```

```
void feedSeq(void)
```

```
{
```

```
    PLL0FEED = 0xAA;
```

```
    PLL0FEED = 0x55;
```

```
}
```

```
void connectPLL0(void)
```

```
{
```

```
    while( !( PLL0STAT & PLOCK ));
```

```
    PLL0CON = 0x03;
```

```
}
```

```
http://www.ocfreaks.com/lpc2148-interrupt-tutorial/
```

```
//-----Setup TIMER0 Interrupt----- // Using Slot 0 for TIMER0
VICVectAddr0 = (unsigned) myTimer0_ISR; // Pointer Interrupt Function (ISR)

VICVectCntl0 = 0x20 | 4;

VICIntEnable |= (1<<4); // Enable timer0 int , 4th bit=1

//-----Setup UART0 Interrupt----- //Any Slot with Lower Priority than TIMER0's slot will suffice
VICVectAddr1 = (unsigned) myUart0_ISR; // Pointer Interrupt Function (ISR)

VICVectCntl1 = 0x20 | 6;

VICIntEnable |= (1<<6); //Enable Uart0 interrupt , 6th bit=1
```

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

```
__irq void myTimer0_ISR(void)
{
    long int regVal;
    regVal = T0IR;           // read the current value in T0's Interrupt Register

    IO0PIN ^= (1<<2);       // Toggle 3rd Pin in GPIO0 .. P0.2

    T0IR = regVal;         // write back to clear the interrupt flag
    VICVectAddr = 0x0;     // Acknowledge that ISR has finished execution
}
```

```
__irq void myUart0_ISR(void)
{
    long int regVal;
    regVal = U0IIR;        // Reading U0IIR also clears it!

    //Receive Data Available Interrupt has occurred
    regVal = U0RBR;       // dummy read
    IO0PIN ^= (1<<3);     // Toggle 4th Pin in GPIO0 .. P0.3

    VICVectAddr = 0x0;    // Acknowledge that ISR has finished execution
}
```

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>


```
VICDefVectAddr = (unsigned) myDefault_ISR;    // Pointer to Default ISR

//-----Enable (Non-Vectored) TIMER0 Interrupt-----
VICIntEnable |= (1<<4);                      // Enable timer0 int , 4th bit=1

//-----Enable (Non-Vectored) UART0 Interrupt-----
VICIntEnable |= (1<<6);                      // Enable Uart0 interrupt , 6th bit=1
```

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

```
__irq void myDefault_ISR(void)
{
    long int T0RegVal , U0RegVal;
    T0RegVal = T0IR;           // read the current value in T0's Interrupt Register
    U0RegVal = U0IIR;

    if( T0IR )
    {
        IO0PIN ^= (1<<2);     // Toggle 3rd Pin in GPIO0 .. P0.2
        T0IR = T0RegVal;     // write back to clear the interrupt flag
    }

    if( !(U0RegVal & 0x1) )
    {
        //Recieve Data Available Interrupt has occured
        U0RegVal = U0RBR;     // dummy read
        IO0PIN ^= (1<<3);     // Toggle 4th Pin in GPIO0 .. P0.3
    }

    VICVectAddr = 0x0; // Acknowledge that ISR has finished execution
}
```

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

<http://www.ocfreaks.com/lpc2148-interrupt-tutorial/>

References

- [1] http://wiki.osdev.org/ARM_RaspberryPi_Tutorial_C
- [2] <http://blog.bobuhiro11.net/2014/01-13-baremetal.html>
- [3] <http://www.valvers.com/open-software/raspberry-pi/>
- [4] <https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/downloads.html>