Tecnológico de Monterrey

Microcontrollers

Professor: Dr. Gilberto Ochoa Ruiz

Topic 5: ARM Basic Programming - Interrupts

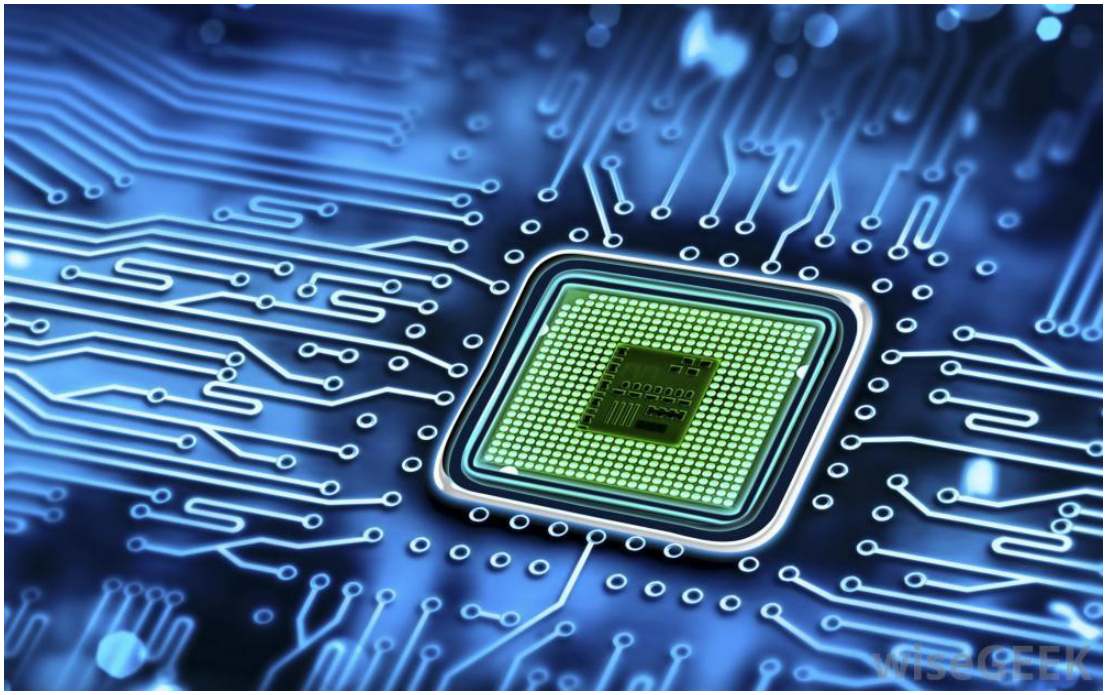
EIAD-204

Wednesday

April 29th 2020

6h30 – 9h30 PM

Guadalajara, Mexico



Basic Programming Techniques - Interrupts

Interrupts and Exceptions in ARM Cortex- M

In this section, first we examine the difference between polling and interrupt and then describe the various interrupts of the ARM Cortex.

Interrupts vs Polling

A single microprocessor can serve several devices. There are two ways to do that: interrupts or polling.

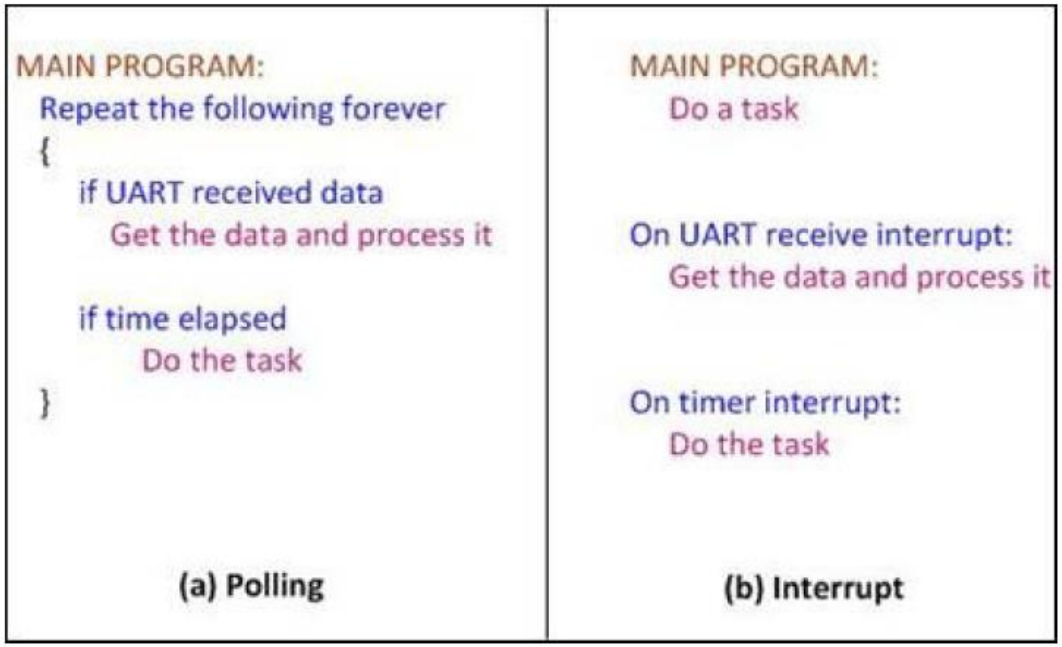
In the *interrupt* method, whenever any device needs service, the device notifies the CPU by sending it an interrupt signal.

Upon receiving an interrupt signal, the CPU interrupts whatever it is doing and serves the device.

Basic Programming Techniques - Interrupts

The program associated with the interrupt is called the *interrupt service routine* (ISR) or *interrupt handler*.

In *polling*, the CPU continuously monitors the status of a given device; when the status condition is met, it performs the service. After that, it moves on to monitor the next device until each one is serviced.



Interrupts vs Polling

Although polling can monitor the status of several devices and serve each of them as certain conditions are met, it is not an efficient use of the CPU time.

The polling method wastes much of the CPU's time by polling devices when they do not need service.

So in order to avoid tying down the CPU, interrupts are used.

Interrupts vs Polling

For example, in Timer we might wait until a determined amount of time elapses, and while we were waiting we cannot do anything else.

That is a waste of the CPU's time that could have been used to perform some useful tasks.

In the case of the Timer, if we use the interrupt method, the CPU can go about doing other tasks, and when the COUNT flag is raised the Timer will interrupt the CPU to let it know that the time is elapsed.

Interrupt Service Routine

For every interrupt there must be a program associated with it.

When an interrupt occurs this program is executed to perform certain service for the interrupt.

This program is commonly referred to as an *interrupt service routine* (ISR).

The interrupt service routine is also called the *interrupt handler*. When an interrupt occurs, the CPU runs the interrupt service routine.

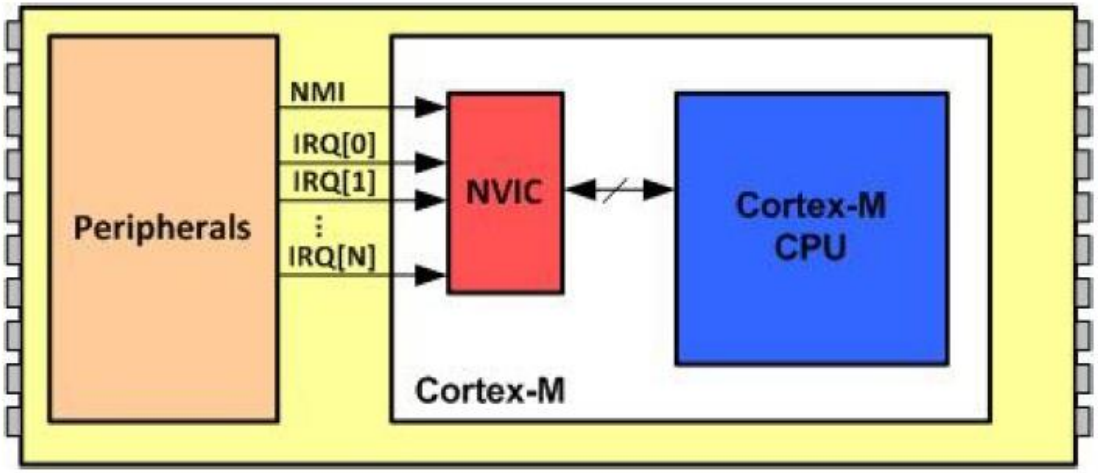
Now the question is how the ISR gets executed?

Basic Programming Techniques - Interrupts

Interrupt Service Routine

As shown in the figure, in the ARM CPU there are pins that are associated with hardware interrupts.

They are input signals into the CPU. When the signals are triggered, CPU pushes the PC register onto the stack and loads the PC register with the address of the interrupt service routine. This causes the ISR to get executed



Interrupt Service Routine

As can be seen from the table, for every interrupt there are four bytes of memory allocated in the interrupt vector table. These four memory locations provide the addresses of the interrupt service routine for which the interrupt was invoked.

Interrupt #	Interrupt	Memory Location (Hex)
	Stack Pointer initial value	0x00000000
1	Reset	0x00000004
2	NMI	0x00000008
3	Hard Fault	0x0000000C
4	Memory Management Fault	0x00000010
5	Bus Fault	0x00000014

Basic Programming Techniques - Interrupts

Interrupt Service Routine

6	Usage Fault (undefined instructions, divide by zero, unaligned memory access, …)	0x00000018
7	Reserved	0x0000001C
8	Reserved	0x00000020
9	Reserved	0x00000024
10	Reserved	0x00000028
11	SVCall	0x0000002C
12	Debug Monitor	0x00000030
13	Reserved	0x00000034
14	PendSV	0x00000038
15	SysTick	0x0000003C

Basic Programming Techniques - Interrupts

Interrupt Service Routine

16	IRQ for peripherals	0x00000040
17	IRQ for peripherals	0x00000044
...	...	...
255	IRQ for peripherals	0x000003FC

Interrupt Vector Table

Since there is a program (ISR) associated with every interrupt and this program resides in memory (RAM or ROM), there must be a look-up table to hold the addresses of these ISRs.

This look-up table is called *interrupt vector table*. In the ARM, the lowest 1024 bytes (256 × 4 = 1024) of memory space are set aside for the interrupt vector table and must not be used for any other function.

Of the 256 interrupts, some are used for software interrupts and some are for hardware IRQ interrupts.

NVIC (nested vector interrupt controller) In ARM Cortex-M

In the ARM Cortex series we have Cortex-A, Cortex-R and Cortex-M.

Currently only the Cortex-M has an on-chip interrupt controller called NVIC (Nested Vector Interrupt Controller).

This allows some degree of standardization among the ARM Cortex-Mx (M0, M1, M3, and M4) family members.

The classical ARM chips and Cortex-A and Cortex-R series do not have this NVIC interrupt controller, therefore ARM manufacturers' implementation of the interrupt handling varies.

Interrupt and Exception assignments in ARM Cortex-M

The NVIC of the ARM Cortex-M has room for the total of 255 interrupts and exceptions.

The interrupt numbers are also referred to INT type (or INT #) in which the type can be from 1 to 255 or 0x01 to 0xFF. That is INT 01 to INT 255 (or INT 0x01 to INT 0xFF.)

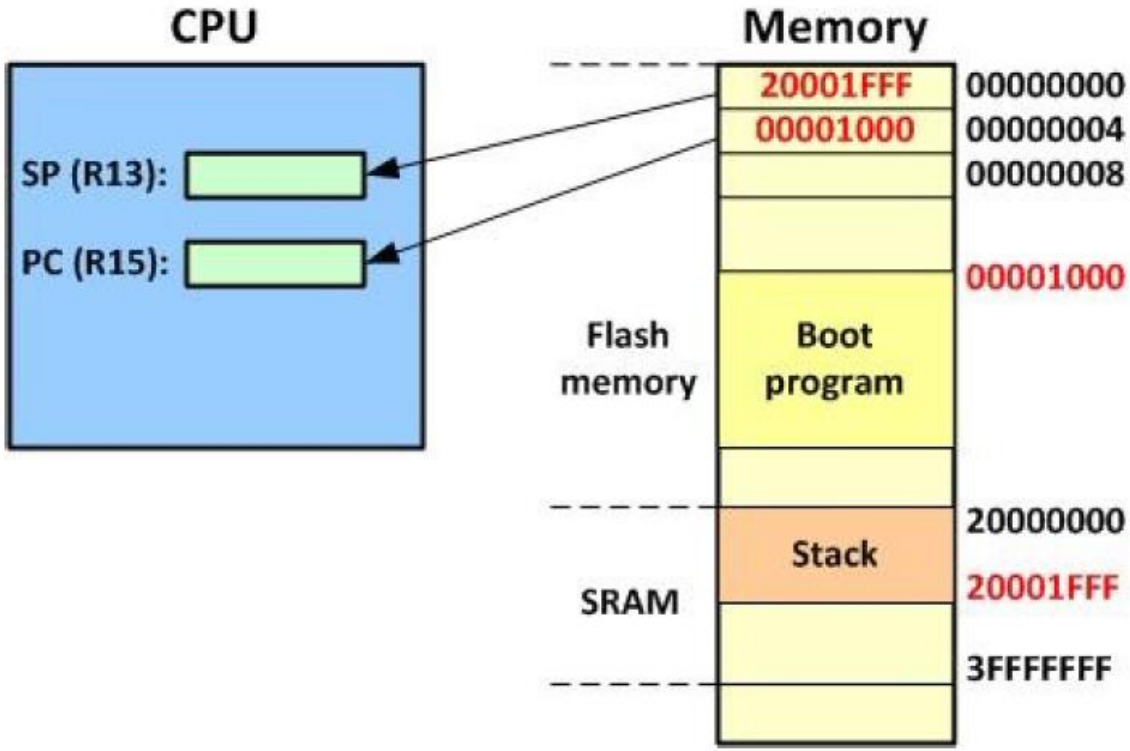
The NVIC in ARM Cortex-M assigns the first 15 interrupts for internal use.

The memory locations 0-3 are used to store the value to be loaded into the stack pointer when the device is coming out of reset

The predefined Interrupts (INT 0 to INT 15)

The followings are the first 15 interrupts in ARM Cortex-M:

Reset



Basic Programming Techniques - Interrupts

The ARM devices have a reset pin. It is usually tied to a circuit that keeps the pin low for a while when the power is coming on.

This is the power-up reset or power-on reset (POR). On the ARM trainer board, there is often a push-button switch to lower the signal too.

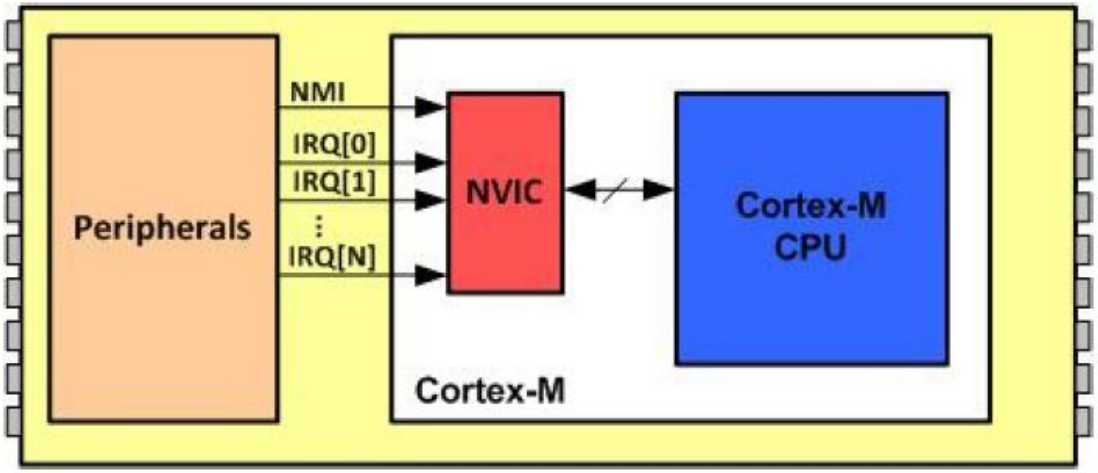
The reset signal is normally high after the power is on and when reset is activated during power-on or when the reset button is pressed, it goes low and the CPU goes to a known state with all the registers loaded with the predefined values.

When the device is coming out of reset, the ARM Cortex-M loads the program counter from memory location 0x00000004

Non-maskable Interrupt

There are pins in the ARM chip that are associated with hardware interrupts. They are IRQs and NMI (non-maskable interrupt).

IRQ is an input signal into the CPU, which can be masked (ignored) and unmasked through the use of software. However, NMI, which is also an input signal into the CPU, cannot be masked by software, and for this reason it is called a *non-maskable interrupt*.



Basic Programming Techniques - Interrupts

Introduction to counters and timers

ARM Cortex-M NVIC has embedded “INT 02” into the ARM CPU to be used only for NMI.

Whenever the NMI pin is activated, the CPU will go to memory location 0x0000008 to get the address of the interrupt service routine (ISR) associated with NMI.

Memory locations

0x00000008, 0x00000009, 0x0000000A, and 0x0000000B

contain the 4 bytes associated with the ISR belonging to NMI.

Basic Programming Techniques - Interrupts

Exceptions (Faults)

There is a group of interrupts belongs to the category referred to as *fault* or exception *interrupts*.

Internally, they are invoked by the microprocessor whenever there are conditions (exceptions) that the CPU is unable to handle.

One such situation is an attempt to execute an instruction that is not implemented in this CPU.

Since the result is undefined, and the CPU has no way of handling it, it automatically invokes the invalid instruction exception interrupt. This is commonly referred to as *exception or fault* in the ARM literature.

Exceptions (Faults)

Whenever an invalid instruction is executed, the CPU will go to memory location 0x00000018 to get the address of the ISR to handle the situation.

The undefined instruction fault is part of the *Usage Fault* exceptions.

Another exception is an attempt to divide a number by zero.

Since the result of dividing a number by zero is undefined, and the CPU has no way of handling such a result, it automatically invokes the divide error exception interrupt.

Exceptions (Faults)

There are many exceptions in the ARM Cortex., some of they are:

Hard Fault

The hard fault is an exception that occurs when the CPU having difficulties executing the ISR for any of the exceptions. One common cause of hard fault is trying to write to the registers of a peripheral before the clock is enabled for that peripheral.

Memory Management Fault

The memory manager unit fault is used for protection of memory from unwanted access. An example of memory management exception fault is when the access permission in MPU is violated by attempting to write into a region of memory designated as read-only.

Exceptions (Faults)

Bus Fault

The bus fault is an exception that occurs when there is an error in accessing the buses. This can be due to memory access problem during the fetch stage of an instruction or reading and writing to data section of memory. For example, if you try to access memory address location that has not been mapped to a memory chip or peripheral device the Bus Fault exception will occur.

Usage Fault

The ARM Cortex-M chip has implemented the divide-by-zero, unaligned memory access, undefined instruction, and so on as part of the Usage Fault exception. See your ARM Cortex-M data sheet.

Basic Programming Techniques - Interrupts

SVCall

An ISR can be called upon as a result of the execution of SVC (supervisor call) instruction.

This is referred to as a *software interrupt* since it was invoked from software

not from a fault exception, external hardware, or any peripheral IRQ interrupt!!!

Whenever the SVC instruction is executed, the CPU will go to memory location 0x0000002C to get the address of the ISR associated with SVC.

Basic Programming Techniques - Interrupts

The SVC is widely used by the operating system to call the OS kernel functions and services that can be provided only by the privileged access mode of the OS.

In many systems, the API and function calls needed by various User applications are handled by the SVCall to make sure the OS is protected.

In the classical ARM literature, SVC was called SWI (software interrupt), but the ARM Cortex-M has renamed it as SVC.

Again it must be noted that the SVC is an ARM Cortex-M instruction and can be used like any other ARM instruction.

Basic Programming Techniques - Interrupts

PendSV (pendable service call)

The PendSV (pendable service call) can be used to do the same thing as the SVC to get the OS services.

However, the SVC is an instruction and is executed right away just like all ARM instructions.

The PendSV is an interrupt and can wait until NVIC has time to service it when other urgent higher priority interrupts are being taken care.

We will examine the concept of nested interrupt and pending interrupts at end of this section to see how NVIC handles multiple pending interrupts.

Basic Programming Techniques - Interrupts

Debug Monitor

In executing a sequence of instructions, there is a need to examine the contents of the CPU's registers and system memory. This is often done by executing the program one instruction at a time and then inspecting registers and memory. This is commonly referred to as *single-stepping*, or performing a trace. ARM has designated INT 12, debug monitor, specifically for implementation of single-stepping.

SysTick

In the multitasking OS we need a real time interrupt clock to notify the CPU that it needs to service the task. The clock tick happens at a regular interval and is used mainly by the OS system. The SysTick in ARM Cortex is designed for this purpose.

Basic Programming Techniques - Interrupts

IRQ Peripheral interrupts

An ISR can be launched as a result of an event at the peripheral devices such as timer timeout or analog-to-digital converter (ADC) conversion complete.

The largest number of the interrupts in the ARM Cortex-M belongs to this category.

Notice from the table in Slide that ARM Cortex-M NVIC has set aside the first 15 interrupts (INT 1 to INT 15) for internal use and exceptions and is not available to chip designer.

For examples: Reset, NMI, undefined instructions, and so

Basic Programming Techniques - Interrupts

IRQ Peripheral interrupts

The rest of the interrupts can be used for peripherals.

Many of the INT 16 to INT 255 are used by the chip manufacturer to be assigned to various peripherals such as timers, ADC, Serial COM, external hardware interrupts, and so on.

Different manufacturers assign different interrupts to different peripherals and you need to examine the data sheet

Each peripheral device has a group of special function registers that must be used to access the device for configuration. For a given peripheral interrupt to take effect, the interrupt for that peripheral must be enabled.

Basic Programming Techniques - Interrupts

Fast context saving in task switching

Most of the interrupts are asynchronous, that means they may happen any time in the middle of program execution.

When the interrupt is acknowledged and the interrupt service routine is launched, the interrupt service routine will need some CPU resource, mainly the CPU registers, to execute the code.

In order not to corrupt the register content of the program that was running before interrupt occurs, these CPU registers need to be preserved.

This saving of the CPU contents before switching to interrupt handler is called context switching (or context saving).

Basic Programming Techniques - Interrupts

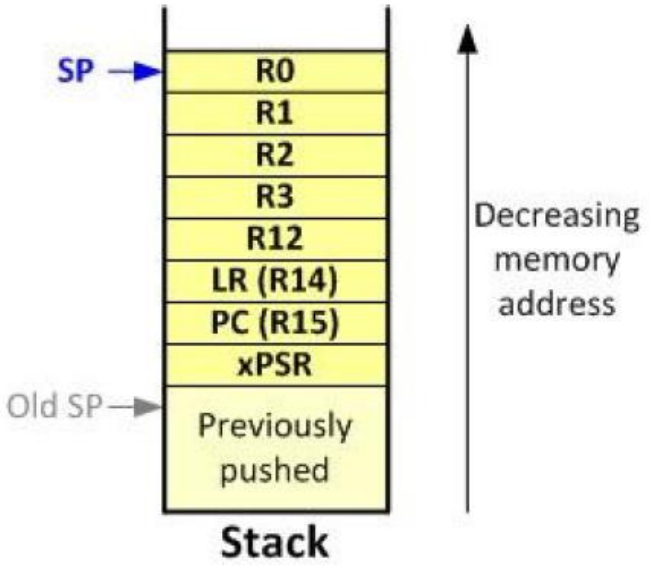
The use of the stack as a place to save the CPU's contents is tedious and time consuming. It takes time to save all the registers.

In executing an interrupt service routine, each task generally needs some key registers such as PC (R15), LR (R14), and CPSR (flag register), in addition to some working registers.

For that reason the ARM Cortex-M automatically saves the registers of

CPSR, PC, LR, R12, R3, R2, R1, and R0 on stack

when an interrupt is acknowledged.



Basic Programming Techniques - Interrupts

If the interrupt service routine needs to use more registers than those preserved, the program has to save the content before using the other registers.

The choice of the registers automatically saved adheres to the ARM Architecture Procedure Call Standard (AAPCS) so that an interrupt handler may be written as a plain C function without the need of any special provision.

When floating-point coprocessor is present, the FPU registers need to be saved too.

If the interrupt handler does not use floating point coprocessor, saving FPU registers is a waste of time.

Basic Programming Techniques - Interrupts

Processing interrupts in ARM Cortex-M

When the ARM Cortex-M processes any interrupt (from either Fault Exceptions or peripheral IRQs), it goes through the following steps:

1. The Current processor status register (CPSR) is pushed onto the stack and SP is decremented by 4, since CPSR is a 4-byte register.

2. The current PC (R15) is pushed onto the stack and SP is decremented

3. The current LR (R14) is pushed onto the stack and SP is decremented

4. The current R12 is pushed onto the stack and SP is decremented by 4.

5-8. R0- R3 are pushed onto the stack and SP is decremented by 4.

Basic Programming Techniques - Interrupts

Processing interrupts in ARM Cortex-M

9. Save Floating point coprocessor registers or move SP if lazy stacking is enabled.

10. The CPU goes into the Handler Mode (details will be described later). LR is loaded with a number with bit 31-5 all 1s.

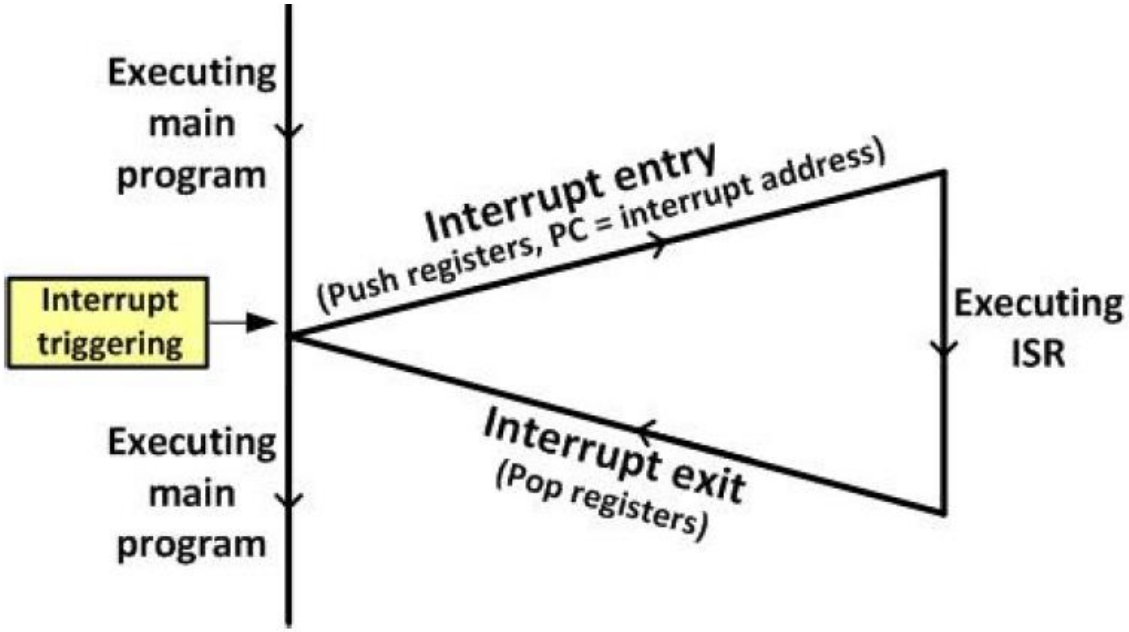
11. The INT number (type) is multiplied by 4 to get the address of the location within the vector table to fetch the program counter of the interrupt service routine (interrupt handler).

12. From the memory locations pointed to by this new PC, the CPU starts to fetch and execute instructions belonging to the ISR program.

Basic Programming Techniques - Interrupts

Processing interrupts in ARM Cortex-M

13. When one of the return instructions is executed in the interrupt service routine, the CPU recognizes that it is in the Handler Mode from the value of the LR.



It then restores the registers saved when entering ISR including the program counter from the stack and makes the CPU run the code where it left off when interrupt occurred.

Basic Programming Techniques - Interrupts

Difference between interrupt and a subroutine call

If the execution of an interrupt saves the program counter of the following instruction and jumps indirectly to the subroutine associated with the interrupt, what is the difference between that and a BL instruction, which also saves the program counter and jumps to the desired subroutine (procedure)?

The differences can be summarized as follows:

1. A “BL” instruction can take an argument and jump to any location within the 4-gigabyte address range of the ARM CPU, but “INT” goes to a fixed memory location in the interrupt vector table to get the address of the interrupt service routine.

Basic Programming Techniques - Interrupts

2. A “BL” instruction is used by the programmer in the sequence of instructions in the program but an externally activated hardware interrupt can come in at any time

requesting the attention of the CPU!!!

3. A “BL” instruction cannot be masked (disabled), but “INT#” belonging to externally activated hardware interrupts

4. A “BL” instruction automatically saves only PC of the next instruction on the stack,

On the other hand, amn “INT#” saves SP, R12, R3–R0, CPSR (flag register) in addition to PC of the next instruction.

Basic Programming Techniques - Interrupts

5. An interrupt puts the CPU in the Handler Mode while the “BL” instruction does not change the CPU execution mode.

6. When returning from the end of the subroutine that has been called by the “BL” instruction, the PC is restored to the address of the next instruction after the “BL” instruction.

When returning from the interrupt handler, the CPU will restore the registers saved when the CPU entered into ISR (the CPSR, R15, R14, R12, R3–R0 registers) from the top of stack.

TAREA ☺

Basic Programming Techniques - Interrupts

Interrupt priority

The next topic in this section is the concept of priority for exceptions and IRQs.

What happens if two interrupts want the attention of the CPU at the same time? Which has priority?

In the ARM Cortex-M the Reset, NMI and Hard Fault exceptions have fixed priority levels and are set by the ARM itself and not subject to change.

Among the Reset, NMI and Hard Fault, the Reset has the highest priority.

Basic Programming Techniques - Interrupts

As we can see from the table, the NMI and Hard Fault have lower priority than Reset, meaning if all three of them are activated at the same time, the Reset will be executed first.

Interrupt #	Interrupt	Priority Level
0	*Stack Pointer initial value*	
1	Reset	-3 Highest
2	NMI	-2
3	Hard Fault	-1
4	Memory Management Fault	Programmable
5	Bus Fault	Programmable
6	Usage Fault (undefined instructions, divide by zero, unaligned memory access, ….)	Programmable

Basic Programming Techniques - Interrupts

7	Reserved	Programmable
8	Reserved	Programmable
9	Reserved	Programmable
10	Reserved	Programmable
11	SVCall	Programmable
12	Debug Monitor	Programmable
13	Reserved	Programmable
14	PendSV	Programmable
15	SysTick	Programmable
16	IRQ for peripherals	Programmable
17	IRQ for peripherals	Programmable
...	...	Programmable
255	IRQ for peripherals	Programmable

Basic Programming Techniques - Interrupts

Programmable priority levels are values between 0 and 3 with 3 has the lowest priority.

The shows standard interrupt assignment for ARM Cortex. Not all Cortex-M chips have all the first 15 interrupts.

In some ARM Cortex-M chips if there is no memory management unit then its interrupt is reserved.

Again it must be emphasized that for the hardware IRQs coming through NVIC, the NVIC resolves priority depending on the way the NVIC is programmed.

Also, no all the interrupts are used in all the chips. The Freescale KL25Z uses only interrupt #1- #47 (or up to IRQ31).

Basic Programming Techniques - Interrupts

Interrupt latency

The time from the moment the event that triggers an interrupt signal to the moment the CPU starts to execute the ISR code is called the interrupt latency.

This latency depends on whether the source of the interrupt is an internal (e.g., exceptions) or external hardware (e.g., peripheral hardware IRQ) interrupt.

The duration of interrupt latency can also be affected by the type of the instruction which the CPU was executing when the interrupt occurs. It takes longer in cases where the instruction being executed lasts for many instruction cycles compared to the instructions that last for only one instruction cycle time.

Basic Programming Techniques - Interrupts

Interrupt latency

In the ARM Cortex-M, we also have extra clock cycles added to the latency due to the fact that it saves the content of registers CPSR, PC, LR, R12, and R0-R3 on stack.

See your ARM Cortex-M manual for the timing data sheet.

Another source of the interrupt latency is the interrupt priority.

As mentioned earlier, when several interrupts occur at the same time, the interrupt with the highest priority is acknowledged first.

All other interrupts have to wait.

Basic Programming Techniques - Interrupts

Interrupt inside an interrupt handler (nested interrupt)

What happens if the ARM is executing an ISR belonging to an interrupt and another interrupt is activated?

In such cases, a higher priority interrupt can preempt a lower priority interrupt.

The higher priority interrupt will stop the lower priority interrupt handler and launch the higher priority interrupt handler.

In the ARM Cortex-M systems, it is up to the software engineer to configure the priority level for each exception and IRQ device and set the policy of how to support nested interrupt.

Basic Programming Techniques - Interrupts

Interrupt inside an interrupt handler (nested interrupt)

What happens if the ARM is executing an ISR belonging to an interrupt and another interrupt is activated?

In such cases, a higher priority interrupt can preempt a lower priority interrupt.

The higher priority interrupt will stop the lower priority interrupt handler and launch the higher priority interrupt handler.

In the ARM Cortex-M systems, it is up to the software engineer to configure the priority level for each exception and IRQ device and set the policy of how to support nested interrupt.

Basic Programming Techniques - Interrupts

All interrupts happened at this time have to wait. If the interrupt service routine runs too long, there is a risk some INTs may be lost.

The interrupt service routine may unmask the interrupts. But in doing so, it will allow all the interrupts to preempt itself.

The ARM Cortex-M allows only the higher priority interrupts to preempt the lower priority interrupt service routine.

The programmer is responsible to assign the proper priority to each IRQ to determine whether an interrupt may preempt the other's interrupt handler. T

he NVIC in ARM Cortex-M has the ability to capture the pending interrupts and keeps track of each one until all are serviced.

Basic Programming Techniques - Interrupts

ARM Cortex-M Processor Modes

In this section we examine various operation modes in ARM Cortex-M.

ARM Cortex Thread (application) and Handler (exception) modes

In comparing the traditional ARM7 with ARM Cortex-M series we see some major changes in the ARM Cortex-M series. Among the changes are the CPU modes, stack, interrupt processing and many new instructions. These changes are meant to make the ARM Cortex-M systems to run programs faster and more efficiently.

The ARM Cortex-M can run in one of the two modes at any given time. They are: (1) Thread (Application) mode and (2) Handler (Exception) mode.

Basic Programming Techniques - Interrupts

The differences can be stated as follows:

1. When the ARM Cortex-M is powered on and coming out of reset, it automatically goes to the Thread mode. The Thread mode is the mode that vast majority of the applications programs are executed in. The CPU spends most of its time in Thread mode and gets interrupted only to execute ISR for exception faults or peripheral IRQs.

2. The ARM Cortex-M switches to Handler mode only when an exception fault (of course other than the Reset) or an IRQ interrupt from a peripheral is activated to get the attention of the CPU to execute an ISR (interrupt handler).
Upon returning from ISR, the CPU automatically changes from Handler mode back to Thread mode.

Basic Programming Techniques - Interrupts

There are two Stacks in ARM Cortex

The classical ARM has a single stack pointer (R13) to be used to point to RAM area for the purpose of stack.

With a multi-threaded operating system, every thread should have their own stack so does the operating system itself. It is much more efficient to have separate stack pointers for the system and the thread.

The ARM Cortex-M has two stack pointer registers. They are called PSP (processor stack pointer) and MSP (main stack pointer). Threads running in Thread mode should use the process stack and the kernel and exception handlers should use the main stack.

Basic Programming Techniques - Interrupts

The bit 1, ASP (active stack pointer), of the special function register called CONTROL register gives the option of choosing MSP or PSP for stack pointer.

Upon Reset the ASP=0, meaning that R13 is the Main Stack pointer (MSP) and its value come from the first 4 bytes of the interrupt vector table starting at 0x00000000 address location.

By making the ASP=1, the R13 is the same as PSP (processor stack pointer).

Basic Programming Techniques - Interrupts

nPRIV (Privilege): Defines the Thread mode privilege level

0: Privileged

1: Unprivileged

Active Stack Pointer (ASP): Defines the currently active stack pointer (ASP = SPSEL)

0: MSP is the current stack pointer.

1: PSP is the current stack pointer.

Floating Point Context Active (FPCA)

0: No floating point context active.

1: Floating point context active.

Basic Programming Techniques - Interrupts

Privileged and Unprivileged levels in ARM Cortex-M

The ARM Cortex-M series has a new feature that did not exist in the previous ARM products. This new feature is called privileged level. There are two privilege levels in ARM Cortex-M.

Processor Mode	Software	Privilege level
Thread	Applications	Privileged and Unprivileged
Handler	ISR for Exceptions and IRQs	Always Privileged
In Thread mode, use bit 0 of the CONTROL register to select Privileged or Unprivileged		

The Privileged level in ARM Cortex-M can be used to limit the CPU access to special registers and protected memory area to prevent the system from getting corrupted due to error in coding or malicious user.

Basic Programming Techniques - Interrupts

Here are summary of the Privileged level software:

1. Privileged level software has access to all registers including the special function registers for interrupts.

2. Privileged level software has access to every region of memory.

3. Privileged level software has access to system timer, NVIC, and system resources.

4. The Privileged level software can execute all the ARM Cortex-M instructions including the MRS, MSR, and CPS.

5. The Handlers for fault exceptions and IRQs can be executed only in Privileged level.

Basic Programming Techniques - Interrupts

6. Only the Privileged software can access the CONTROL register to see whether execution is in Privileged or Unprivileged mode.

In Unprivileged mode one can switch from Unprivileged level to Privileged level by using SVC instruction

Processor Mode	Software	Stack Usage
Thread	Applications	MSP or PSP
Handler	ISR for Exceptions and IRQs	MSP
Note: In Thread mode, use bit 1 of the Control register to select MSP or PSP for stack pointer.		

Processor Modes and Stack Usage in ARM Cortex-M

Basic Programming Techniques - Interrupts

Here are summary of the Unprivileged level software:

1. Unprivileged level software has no access to some registers such the special function registers for interrupts.

2. Unprivileged level software has limited access to some regions of memory.

3. Unprivileged level software is blocked from accessing system timer, NVIC, and system control block and resources.

4. The Unprivileged level software cannot execute some of the ARM instructions such as CPS. It has limited access to the MRS and MSR instructions.

Basic Programming Techniques - Interrupts

5. While Handler mode is always executed in the Privileged level, the Thread mode SW can be executed in Privileged or Unprivileged level.

6. In Unprivileged mode, one can use SVC instruction to make a supervisor call to switch from Unprivileged level to Privileged level.

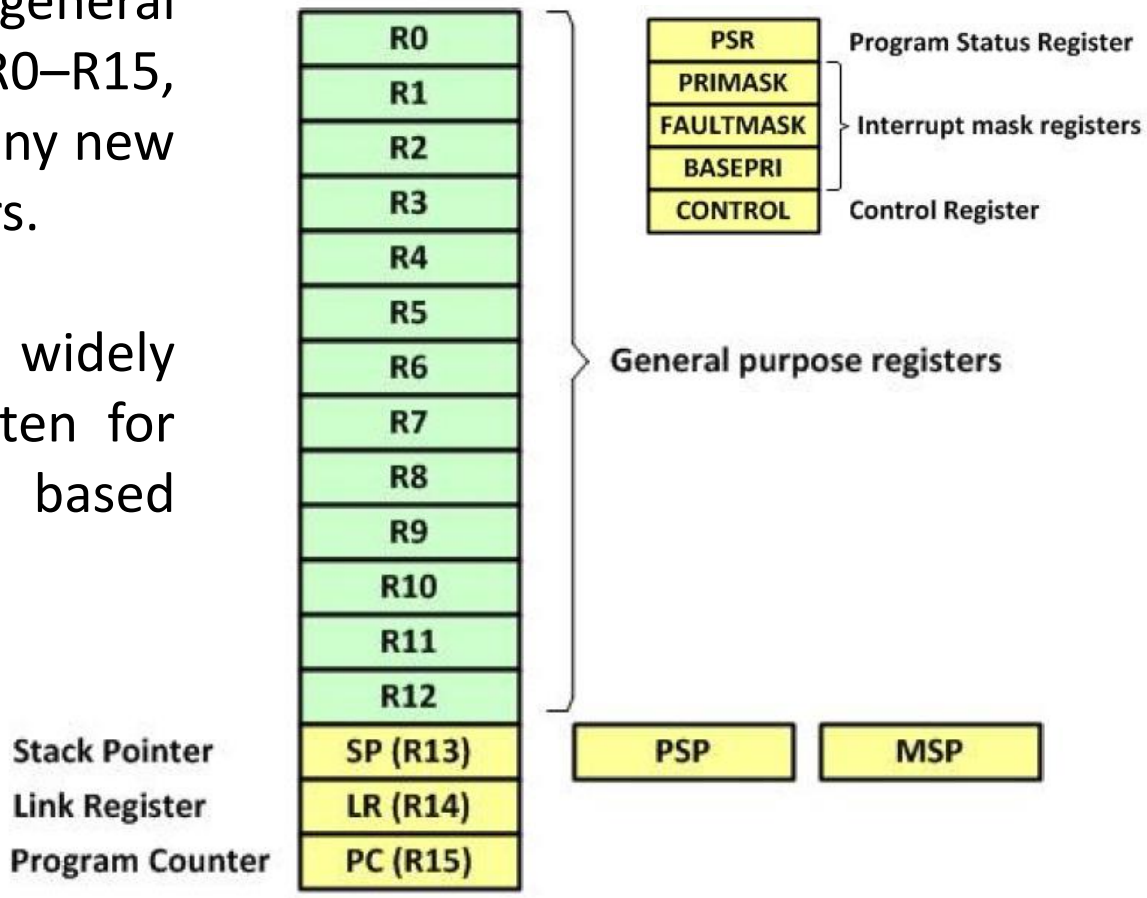
Mode	Privilege	Stack Pointer	Typical Example usage
Handler	Privileged	Main	Exception Handling
Handler	Unprivileged	Any	Reserved since Handler is always Privileged
Thread	Privileged	Main	Operating system kernel
Thread	Privileged	Process	
Thread	Unprivileged	Main	
Thread	Unprivileged	Process	Application threads

Basic Programming Techniques - Interrupts

Special Function register in ARM Cortex

Beside the traditional general purpose registers of R0–R15, the ARM Cortex has many new special function registers.

These registers are widely used in programs written for the Cortex-M based embedded systems.



Basic Programming Techniques - Interrupts

Special Function register in ARM Cortex

While the general purpose registers of R0–R15 can be accessed using the MOV, LDR, and STR instructions.

These new special function registers can be accessed only with the two new instructions MSR and MRS.

To manipulate (clear or set) the bits of special function registers, first we must use the MSR to move them to a general purpose register

After changing their values they are moved back by using MRS instruction

Basic Programming Techniques - Interrupts

Register name	Privilege Usage
MSP (main stack pointer)	Privileged
PSP (processor stack pointer)	Privileged or Unprivileged
PSR (Processor status register)	Privileged
APSR (application processor status register)	Privileged or Unprivileged
ISPR (interrupt processor status register)	Privileged
EPSR (execution processor status register)	Privileged
PRIMASK (Priority Mask register)	Privileged
FAULTMASK (fault mask register)	Privileged
BASEPRI (base priority register)	Privileged
CONTROL (control register)	Privileged

Basic Programming Techniques - Interrupts

Freescale I/O Port Interrupt Programming

Freescale KL25Z is an ARM Cortex-M chip. In previous classes, we saw to use GPIO ports for simple I/O.

We also showed a simple program getting (polling) an input switch and placing it on LED.

In this class, we show how to program the interrupt capability of the I/O ports in KL25Z chip.

Basic Programming Techniques - Interrupts

PORTA and PORTD Interrupt Registers

In this section, we will use a switch to show an example of external interrupt programming using GPIO pins.

However, before we do that, we need to examine the interrupt vector table for the Freescale KL25Z microcontroller.

For Freescale KL25Z, only PORTA and PORTD are capable of generating interrupts. The following table shows interrupt assignment in KL25Z used by FRDM board.

INT#	IRQ#	Vector location	Device
1-15	None	0000 0000 to 0000 003C	CPU Exception (set by ARM)

Basic Programming Techniques - Interrupts

INT#	IRQ#	Vector location	Device
16	0	0000 0040	DMA
17	1	0000 0044	DMA
18	2	0000 0048	DMA
19	3	0000 004C	DMA
20	4	0000 0050	–-
21	5	0000 0054	FTFA
22	6	0000 0058	PMC
23	7	0000 005C	LLWU

Basic Programming Techniques - Interrupts

INT#	IRQ#	Vector location	Device
24	8	0000 0060	I2C0
25	9	0000 0064	I2C1
26	10	0000 0068	SPI0
27	11	0000 006C	SPI1
28	12	0000 0070	UART0
29	13	0000 0074	UART1
30	14	0000 0078	UART2

Basic Programming Techniques - Interrupts

INT#	IRQ#	Vector location	Device
31	15	0000 007C	ADC0
32	16	0000 0080	CMP0
33	17	0000 0084	TPM0
34	18	0000 0088	TPM1
35	19	0000 008C	TPM2
36	20	0000 0090	RTC
37	21	0000 0094	RTC
38	22	0000 0098	PIT

Basic Programming Techniques - Interrupts

INT#	IRQ#	Vector location	Device
39	23	0000 009C	–
40	24	0000 00A0	USB OTG
41	25	0000 00A4	DAC0
42	26	0000 00A8	TSI0
43	27	-	MCG
44	28	0000 00B0	LPTMR0
45	29	0000 00B4	—–I
46	30	0000 00B8	I/O PORTA
47	31	0000-00BC	I/O PORTD

Basic Programming Techniques - Interrupts

This info can be found in the start-up header file of your C compiler.

Notice interrupt numbers 16 to 255 are assigned to the peripherals. In the Freescale KL25Z, the INT 46 is assigned to the I/O port of PORTA and INT47 to I/O PORTD.

Although PORTA has many pins, we have only one interrupt assigned to the entire PORTA. This is common in many microcontrollers.

In other words, when any of the PORTA pins trigger an interrupt, they all go to the same address location in the interrupt vector table.

It is the job of our interrupt Service Routine (ISR or interrupt Handler) to find out which pin caused the interrupt

Basic Programming Techniques - Interrupts

Next, we examine the registers associated with the PORTA interrupt.

Upon Reset, all the interrupts are disabled at the peripheral modules and NVIC but enabled globally.

To enable any interrupt we need these steps:

1) Enable the interrupt for a specific peripheral module.

2) Enable the interrupts at the NVIC module.

Next, we look at the details of each one.

Basic Programming Techniques - Interrupts

1) We need to enable the interrupt capability of a given peripheral at the module level. This should be done after other configurations of that peripheral are done.

In the case of I/O ports, each pin can be used as a source of external hardware interrupt. This is done with the PORTx_PCR register, as we will see soon.

PORTx_PCRn:

PORTx->PCR[n]

D31	D30	D29	D28	D27	D26	D25	D24	D23	D22	D21	D20	D19	D18	D17	D16
0							ISF	0				IRQC			

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
0					MUX			0	DSE	0	PFE	0	SRE	PE	PS

Basic Programming Techniques - Interrupts

Notice that, the IRQC field (bits D19-D16) of PORTx_PCR register are used to enable the interrupt capability of each pin of the I/O port.

They allow you to select the trigger of interrupt by an edge or a level (which we will describe in more details later).

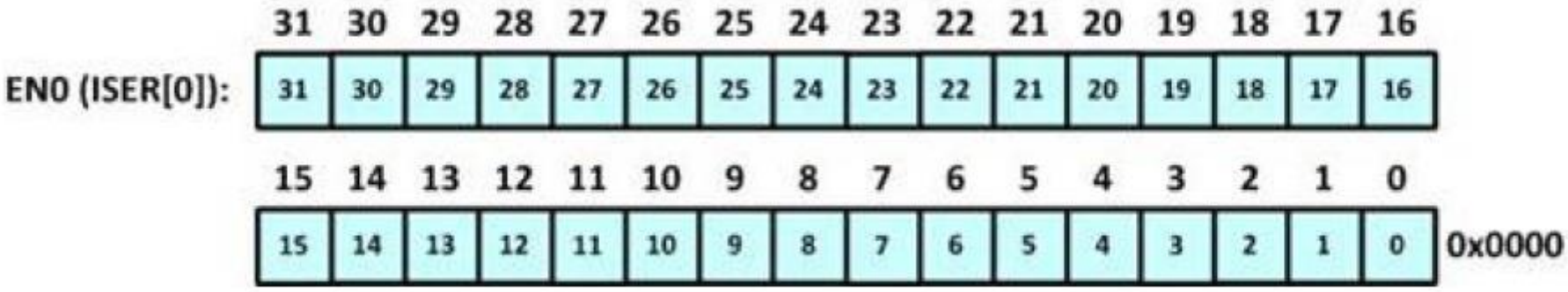
For example, to enable the interrupts for PTA1 on the falling edge, we will need the following:

```
PORTA->PCR[1] &= ~0xF0000; /* clear interrupt selection */

PORTA->PCR[1] |= 0xA0000; /* enable falling edge interrupt */
```

Basic Programming Techniques - Interrupts

2) In ARM Cortex, there is an interrupt enable for each entry in the interrupt vector table. These enable bits are in the registers in NVIC. Each register covers 32 IRQ interrupts.

For example, register EN0 controls the enable the interrupts for IRQ0 to IRQ31, EN1 for IRQ32 to IRQ63, and so on.



Notice, these registers are called Interrupt Set Enable and we have an array for all of them. The array is referred to as ISER[0], ISER[1], and so on. The KL25Z has a total of 32 IRQs and only ISER[0] is used.

Basic Programming Techniques - Interrupts

As we can see in the interrupt vector table Slides XYZ, the PORTA interrupt is assigned to IRQ30. Therefore, to enable the interrupt associated with PORTA in Vector table, we need the following:

```
NVIC->ISER[0] |= 0x40000000; /* enable INT30 (bit 30 of ISER[0]) */
```

The interrupts can be enabled using the following function, as well:

```
void NVIC_EnableIRQ(IRQn_Type IRQn);
```

The function is defined in the *core_cm0plus.h* file which is included in the *MKL25Z4.h* header file. To enable an interrupt using this function, the IRQ number of the interrupt should be passed as the argument to the function. For PORTA interrupt:

```
NVIC_EnableIRQ(30);
```

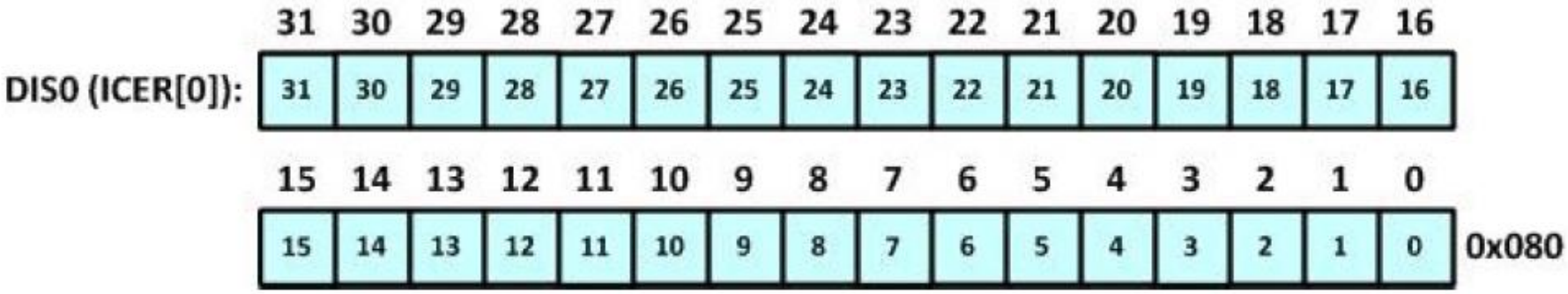
Tecnológico de Monterrey

Basic Programming Techniques - Interrupts

Since the IRQ numbers of all the interrupts are defined in the MKL25Z4.h, we can use their names instead of their numbers.

For example, to enable PORTA int. the following can be used as well:
NVIC_EnableIRQ(PORTA_IRQn);

For more information, open the MKL25Z4.h file and find “typedef enum IRQn” in the file. To disable interrupts there are other registers: ICER0 to ICER3. Because KL25Z device has 32 IRQs, only ICER0 is used.



Basic Programming Techniques - Interrupts

Each interrupt can be disabled by writing a 1 to the corresponding bit in the ICER registers. Writing 0 to the ICER registers has no effect on their values.

For example, the following instruction disables UART0 interrupt, keeping the other interrupts unchanged:

```
NVIC->ICER[0] = 0x1000; /*disable UART0 Interrupt */
```

The interrupts can be disabled using the following function, as well:

```
void NVIC_DisableIRQ(IRQn_Type IRQn);
```

For example, the following instruction disables the UART0 interrupt:

```
NVIC_DisableIRQ(UART0_IRQn);
```

Basic Programming Techniques - Interrupts

3) Global interrupt enable/disable allows us with a single instruction to mask all interrupts during the execution of some critical task such as manipulating a common pointer shared by multiple threads.

In ARM Cortex M, we do the global enable/disable of interrupts with assembly language instructions of CPSID I (Change processor state-disable interrupts) and CPSIE I (Change processor state-enable interrupts). In C language we use pseudo-functions:

```
__enable_irq(); /* Enable interrupt Globally */
```

and

```
__disable_irq(); /* Disable interrupt Globally */
```

Basic Programming Techniques - Interrupts

It is a good idea to disable all interrupts during the initialization of the program and enable interrupts after all the initializations are complete.

Now, using the following lines of code, we enable the interrupts for PTA1 pin at all three levels:

```
PORTA->PCR[1] &= ~0xF0000; /* clear interrupt selection */

PORTA->PCR[1] |= 0xA0000; /* enable falling edge interrupt */

NVIC->ISER[0] |= 0x40000000; /* enable INT30 (bit 30 of ISER[0]) */

__enable_irq(); /* global enable IRQs */
```

Basic Programming Techniques - Interrupts

IRQ Priority

As describe earlier, since ARM Cortex-M supports higher priority interrupt to preempt the lower interrupt handler, it is important that each interrupt be assigned a proper priority before they are enabled.

The IRQ interrupt priorities are controlled by the NVIC IPR registers. For each IRQ number, there is one byte corresponding to that IRQ to assign its priority.

The allowed priority levels are ranging from 0 to 3 in a KL25Z device and they are defined by two bits left justified in that byte. There are eight IPR registers to hold the priority of 32 IRQs.

One needs to identify the byte and the register to set the IRQ priority.

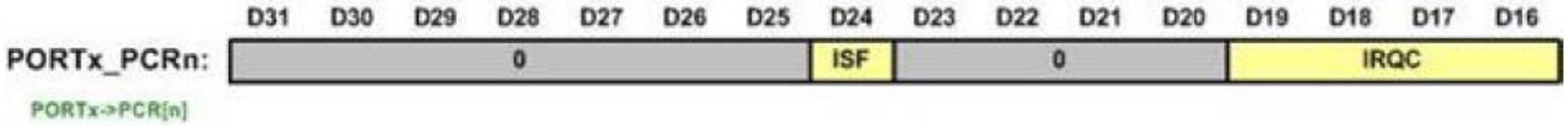
Basic Programming Techniques - Interrupts

Interrupt trigger point

When an input pin is connected to an external device to be used for interrupt, we have 5 choices for trigger point. They are:

1) low-level trigger (active Low level),
2) high-level trigger (active High level),
3) rising-edge trigger (positive-edge going from Low to High),
4) falling-Edge trigger (negative-edge going from High to Low),
5) Both edge (rising and falling) trigger.

In Freescale KL25Z, we must use PORTX_PCRn register to decide the level or edge for I/O interrupts.



Basic Programming Techniques - Interrupts

From above figure, we see the D19-D16 bits are used to decide low-level, high-level, falling-edge, rising-edge, or both-edge activation.

D19	D18	D17	D16	
1	0	0	0	Interrupt when logic zero (Active Low-level).
1	0	0	1	Interrupt on rising edge.
1	0	1	0	Interrupt on falling edge.
1	0	1	1	Interrupt on either edge.
1	1	0	0	Interrupt when logic one (Active High-level)

Basic Programming Techniques - Interrupts

Now, the following lines of code make the PTA1 interrupt trigger on logic zero.

```
PORTA->PCR[1] &= ~0xF0000; /* clear interrupt selection */

PORTA->PCR[1] |= 0x80000; /* enable low-level interrupt*/
```

We can also do both negative and positive edge trigger too, as we will see soon.

Basic Programming Techniques - Interrupts

Example 1

The main program toggles the red LED on PTB18 continuously.

When an interrupt comes from an external SW connected to PTA1, it toggles the green LED on PTB19 for a short period of time then it returns back to the main. The delay function is called several times in the interrupt handler to make the LED blink.

This is done only for demonstration purpose. It is generally a poor practice to do delay or to wait for some event to happen in the interrupt service routine because when interrupt service routine is active, the main program is halted as you can see that when the green LED is blinking, the red LED ceases to blink.

Basic Programming Techniques - Interrupts

```
/* p6_1: PORTA interrupt from a switch */
/* Upon pressing a switch connecting either PTA1 or PTA2 to ground, the green
LED will toggle for three times. */
/* Main program toggles red LED while waiting for interrupt from switches. */

#include “MKL25Z4.H”
void delayMs(int n);
int main(void) {

__disable_irq(); /* disable all IRQs */

SIM->SCGC5 |= 0x400; /* enable clock to Port B */
PORTB->PCR[18] = 0x100; /* make PTB18 pin as GPIO */
PORTB->PCR[19] = 0x100; /* make PTB19 pin as GPIO */
PTB->PDDR |= 0xC0000; /* make PTB18, 19 as output pin */
PTB->PDOR |= 0xC0000; /* turn off LEDs */
SIM->SCGC5 |= 0x200; /* enable clock to Port A */
```

Basic Programming Techniques - Interrupts

```
/* configure PTA1 for interrupt */

PORTA->PCR[1] |= 0x00100; /* make it GPIO */
PORTA->PCR[1] |= 0x00003; /* enable pull-up */
PTA->PDDR &= ~0x0002; /* make pin input */
PORTA->PCR[1] &= ~0xF0000; /* clear interrupt selection */
PORTA->PCR[1] |= 0xA0000; /* enable falling edge interrupt */

// configure PTA2 for interrupt

PORTA->PCR[2] |= 0x00100; /* make it GPIO */
PORTA->PCR[2] |= 0x00003; /* enable pull-up */
PTA->PDDR &= ~0x0004; /* make pin input */
PORTA->PCR[2] &= ~0xF0000; /* clear interrupt selection */
PORTA->PCR[2] |= 0xA0000; /* enable falling edge interrupt */
NVIC->ISER[0] |= 0x40000000; /* enable INT30 (bit 30 of ISER[0]) */
__enable_irq(); /* global enable IRQs */
```

Basic Programming Techniques - Interrupts

```
/* toggle the red LED continuously */
while(1) {
PTB->PTOR |= 0x40000; /* toggle red LED */
delayMs(500);
}
}
/* A pushbutton switch is connecting either PTA1 or PTA2 to ground to trigger
PORTA interrupt */
void PORTA_IRQHandler(void) {
int i;
/* toggle green LED (PTB19) three times */
for (i = 0; i < 3; i++) {
PTB->PDOR &= ~0x80000; /* turn on green LED */
delayMs(500);
PTB->PDOR |= 0x80000; /* turn off green LED */
delayMs(500);
}
PORTA->ISFR = 0x00000006; /* clear interrupt flag */
}
```

Basic Programming Techniques - Interrupts

Example 2

Notice that in the previous program, if we have two switches connected to PTA1 and PTA2, the program does not make any distinction which switch is pressed.

The reason is that only one interrupt (IRQ30) is associated with the entire PORTA. In other words, whichever pin of PORTA interrupt is activated it goes to the same interrupt handler belonging to PORTA.

Now, we can modify the program distinguish between various pins of PORTA. That means, by pressing SW1 (PTA1) we can toggle the green LED (PTB19) and when SW2 (PTA2) is pressed blue LED (PTD1) is toggled. Each of the PORTA pin interrupt sets a bit in the PORTA_ISFR.

Basic Programming Techniques - Interrupts

The interrupt handler reads the PORTA_ISFR register to find out which interrupt flag bit is posted and which pin is requesting interrupt, as we will see

This is like using a port for security system in which each pin is assigned to a window or a door.

A different message can be produced depending on which door or window is opened. Notice when both switches are pressed and released, both interrupts will be served sequentially.

The sequence the interrupts are served depends on the sequence their interrupt flags are checked in the interrupt handler.

Basic Programming Techniques - Interrupts

```
/* Main program toggles red LED while waiting for interrupt from switches */

#include “MKL25Z4.h”
void delayMs(int n);
int main(void)
{
__disable_irq(); /* disable all IRQs */
SIM->SCGC5 |= 0x400; /* enable clock to Port B */
SIM->SCGC5 |= 0x1000; /* enable clock to Port D */
PORTB->PCR[18] = 0x100; /* make PTB18 pin as GPIO */
PORTB->PCR[19] = 0x100; /* make PTB19 pin as GPIO */
PTB->PDDR |= 0xC0000; /* make PTB18, 19 as output pin */
PORTD->PCR[1] = 0x100; /* make PTD1 pin as GPIO */
PTD->PDDR |= 0x02; /* make PTD1 as output pin */
PTB->PDOR |= 0xC0000; /* turn off red/green LEDs */
PTD->PDOR |= 0x02; /* turn off blue LED */
SIM->SCGC5 |= 0x200; /* enable clock to Port A */
```

Basic Programming Techniques - Interrupts

```
// configure PTA1 for interrupt

PORTA->PCR[1] |= 0x00100; /* make it GPIO */
PORTA->PCR[1] |= 0x00003; /* enable pull-up */
PTA->PDDR &= ~0x0002; /* make pin input */
PORTA->PCR[1] &= ~0xF0000; /* clear interrupt selection */
PORTA->PCR[1] |= 0xA0000; /* enable falling edge interrupt */

// configure PTA2 for interrupt
PORTA->PCR[2] |= 0x00100; /* make it GPIO */
PORTA->PCR[2] |= 0x00003; /* enable pull-up */
PTA->PDDR &= ~0x0004; /* make pin input */
PORTA->PCR[2] &= ~0xF0000; /* clear interrupt selection */
PORTA->PCR[2] |= 0xA0000; /* enable falling edge interrupt */
NVIC->ISER[0] |= 0x40000000; /* enable INT30 (bit 30 of ISER[0]) */

__enable_irq(); /* global enable IRQs */
```

Basic Programming Techniques - Interrupts

```
/* toggle the red LED continuously */
while(1) {
PTB->PTOR |= 0x40000; /* toggle red LED */
delayMs(500); } }

void PORTA_IRQHandler(void) {
int i;
while (PORTA->ISFR & 0x00000006) {
if (PORTA->ISFR & 0x00000002) {
/* toggle green LED (PTB19) three times */
for (i = 0; i < 3; i++) {
PTB->PDOR &= ~0x80000; /* turn on green LED */
delayMs(500);
PTB->PDOR |= 0x80000; /* turn off green LED */
delayMs(500);
}
PORTA->ISFR = 0x00000002; /* clear interrupt flag */
}
```

Basic Programming Techniques - Interrupts

```
if (PORTA->ISFR & 0x00000004) {
/* toggle blue LED (PTD1) three times */
for (i = 0; i < 3; i++) {
PTD->PDOR &= ~0x02; /* turn on blue LED */
delayMs(500);
PTD->PDOR |= 0x02; /* turn off blue LED */
delayMs(500);
}
PORTA->ISFR = 0x00000004; /* clear interrupt flag */ } } }

/* Delay n milliseconds */
void delayMs(int n) {
int i;
int j;
for(i = 0 ; i < n; i++)
for (j = 0; j < 7000; j++) {}
}
```

Basic Programming Techniques - Interrupts

Example 3

The next example Program 6-3 is to connect a square wave of 3 V to PTD4 pin and let an LED toggle at the same rate as the input frequency.

The PORTD interrupt is configured to trigger on both rising edge and falling edge.

Because there is no delay in the interrupt handler and the interrupt is triggered by rising edge and falling edge, it is necessary to trigger the interrupts using a function generator.

If you connect PTD4 to a switch to the ground, the program does not work well because of contact bounce

Basic Programming Techniques - Interrupts

Example 3

```
/* Toggle blue LED by PTD4 interrupt on both edges */

/* PTD4 is configured to interrupt on both rising edge and falling edge. */
/*In the  interrupt handler, the blue LED (PTD1) is toggled. */

#include “MKL25Z4.h”

int main(void)
{
__disable_irq(); /* disable all IRQs */

SIM->SCGC5 |= 0x1000; /* enable clock to Port D */
PORTD->PCR[1] = 0x100; /* make PTD1 pin as GPIO */
PTD->PDDR |= 0x02; /* make PTD1 as output pin */
PTD->PDOR |= 0x02; /* turn off blue LED */
```

Basic Programming Techniques - Interrupts

Example 3

```
// configure PTD4 for interrupt

PORTD->PCR[4] |= 0x00100; /* make it GPIO */
PORTD->PCR[4] |= 0x00003; /* enable pull-up */
PTD->PDDR &= ~0x0010; /* make pin input */
PORTD->PCR[4] &= ~0xF0000; /* clear interrupt selection */
PORTD->PCR[4] |= 0xB0000; /* enable both edge interrupt */

NVIC->ISER[0] |= 0x80000000; /* enable INT31 (bit 31 of ISER[0]) */
__enable_irq(); /* global enable IRQs */

while(1) /* Do Nothing Really in Main*/
{ }
}
```

Basic Programming Techniques - Interrupts

Example 3

```
/* A pushbutton switch is connected to PTD4 pin to trigger PORTD interrupt */

void PORTD_IRQHandler(void)
{

if (PORTD->ISFR & 0x00000010) {

/* toggle blue LED (PTD1) */

PTD->PTOR |= 0x0002; /* toggle blue LED */
PORTD->ISFR = 0x0010; /* clear interrupt flag */

}
}
```

Tecnológico de Monterrey

Microcontrollers

Professor: Dr. Gilberto Ochoa Ruiz

Topic 4: ARM Basic Programming - Interrupts

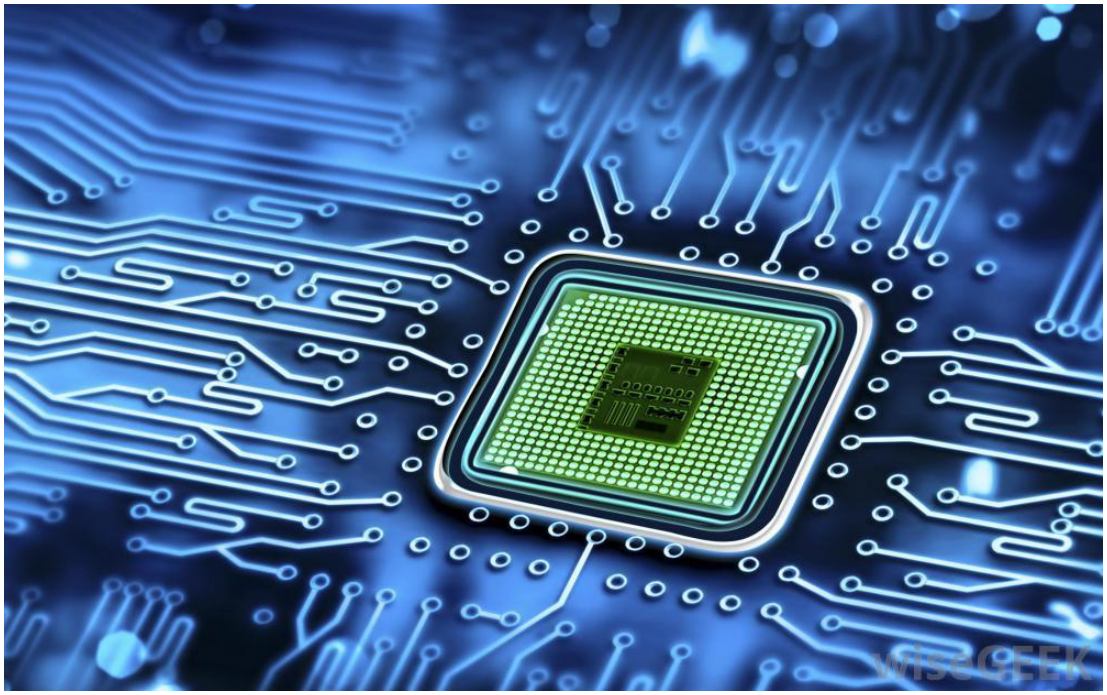
EIAD-204

Wednesday

May 5th 2020

6h30 – 9h30 PM

Guadalajara, Mexico



Timer Interrupt Programming

In the previous module, we showed how to program the timers.

In those programming examples, we used polling to see if a timeout event occurred (TOF flag status).

In this class, we will interrupt-based version of those programs.

We will examine the programs those programs and notice, we could run those programs only one at a time since we have to monitor the timer flag continuously.

Basic Programming Techniques - Timers

(Previously…) Making delays using the TPM timer

Example: Toggle blue LED (PTD1 pin) every 320 times TPM0_CNT matches the TPM0_MOD.

```
/* This program uses TPM0 to generate maximal delay to
toggle the blue LED.
MCGFLLCLK (41.94 MHz) is used as timer counter clock.
The Modulo register is set to 65,535. The timer counter
overflows at
41.94 MHz / 65,536 = 640 Hz
We put the time out delay in a for loop and repeat it
for 320 times before we
toggle the LED. This results in the LED flashing at half
second on and half
second off.
The blue LED is connected to PTD1. */
```

Basic Programming Techniques - Timers

(Previously…) Making delays using the TPM timer

```
#include <MKL25Z4.H>

int main (void) {
int i;
SIM->SCGC5 |= 0x1000; /* enable clock to Port D */
PORTD->PCR[1] = 0x100; /* make PTD1 pin as GPIO */
PTD->PDDR |= 0x02; /* make PTD1 as output pin */

SIM->SCGC6 |= 0x01000000; /* enable clock to TPM0 */
SIM->SOPT2 |= 0x01000000; /* use MCGFLLCLK as CNT clock */
TPM0->SC = 0; /* disable timer while configuring */
TPM0->MOD = 0xFFFF; /* max modulo value */
TPM0->SC |= 0x80; /* clear TOF */
TPM0->SC |= 0x08; /* enable timer free-running mode */
```

Basic Programming Techniques - Timers

(Previously…) Making delays using the TPM timer

```
while (1) {

for(i = 0; i < 320; i++) {

/* repeat timeout for 320 times */

while((TPM0->SC & 0x80) == 0) {}

/* wait until the TOF is set */

TPM0->SC |= 0x80; /* clear TOF */

PTD->PTOR = 0x02; /* toggle blue LED */
}
}
```

Basic Programming Techniques - Timers

(Previously…) Prescaler register for TPMx

```
/* Toggling blue LED using TPM0 delay (prescaler)
* This program uses TPM0 to generate maximal delay
to
* toggle the blue LED.
* MCGFLLCLK (41.94 MHz) is used as timer counter
clock.
* Prescaler is set to divided by 128 and the
Modulo register
* is set to 65,535. The timer counter overflows at
* 41.94 MHz / 128 / 65,536 = 5.0 Hz
*
* The blue LED is connected to PTD1.
*/
```

Basic Programming Techniques - Timers

(Previously…) Prescaler register for TPMx

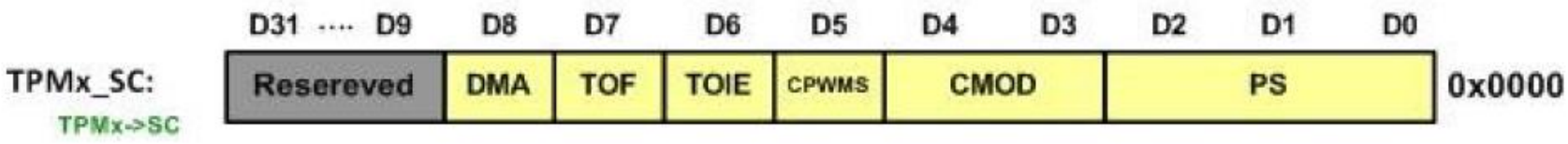
```
#include <MKL25Z4.H>

int main (void) {

SIM->SCGC5 |= 0x1000; /* enable clock to Port D */
PORTD->PCR[1] = 0x100; /* make PTD1 pin as GPIO */
PTD->PDDR |= 0x02; /* make PTD1 as output pin */
SIM->SCGC6 |= 0x01000000; /* enable clock to TPM0 */
SIM->SOPT2 |= 0x01000000; /* use MCGFLLCLK as timer counter
clock */
TPM0->SC = 0; /* disable timer while configuring */
TPM0->SC = 0x07; /* prescaler /128 */
TPM0->MOD = 0xFFFF; /* max modulo value */
TPM0->SC |= 0x80; /* clear TOF */
TPM0->SC |= 0x08; /* enable timer free-running mode */
```

(Previously…) Prescaler register for TPMx

```
while (1) {

while((TPM0->SC & 0x80) == 0) { }

/* wait until the TOF is set */

TPM0->SC |= 0x80; /* clear TOF */

PTD->PTOR = 0x02; /* toggle blue LED */

}

}
```

Timer Interrupt Programming

By using interrupt, we can run several of timer programs all at the same time. To do that, enable the timer interrupt using the TOIE (Timer Overflow Interrupt Enable) in TPMx_SC register.



From the table in slide XYZ of previous notes, notice that IRQ17, IRQ18, and IRQ19 are assigned to TPM0, TPM1, and TPM2. The following will enable these timers in NVIC:

```
NVIC->ISER[0] |= 0x0020000; /* enable IRQ17 (D17 of ISER[0]) */
NVIC->ISER[0] |= 0x0040000; /* enable IRQ18 (D18 of ISER[0]) */
NVIC->ISER[0] |= 0x0080000; /* enable IRQ19 (D19 of ISER[0]) */
```

Timer Interrupt Programming - Example

/* Toggling red and green LED using TPM0 and TPM1 interrupts

This program is modified version of the two previous programs but instead of using polling, the timeout interrupts are used.

TPM1 is configured to run twice the frequency of TPM0.

The infinite loop in main blinks the blue LED.

The TPM0 interrupt handler toggles the red LED and the TPM1 green LED. */

```
#include <MKL25Z4.H>
```

Timer Interrupt Programming - Example

```
void delayMs(int n);

int main (void) {
__disable_irq(); /* global disable IRQs */

SIM->SCGC5 |= 0x400; /* enable clock to Port B */
SIM->SCGC5 |= 0x1000; /* enable clock to Port D */
PORTB->PCR[18] = 0x100; /* make PTB18 pin as GPIO */
PTB->PDDR |= 0x40000; /* make PTB18 as output pin */
PORTB->PCR[19] = 0x100; /* make PTB19 pin as GPIO */
PTB->PDDR |= 0x80000; /* make PTB19 as output pin */
PORTD->PCR[1] = 0x100; /* make PTD1 pin as GPIO */
PTD->PDDR |= 0x02; /* make PTD1 as output pin */
```

Timer Interrupt Programming - Example

```
/* Configuring first timer*/
/* use MCGFLLCLK as timer counter clock */
SIM->SOPT2 |= 0x01000000;
SIM->SCGC6 |= 0x01000000; /* enable clock to TPM0 */
TPM0->SC = 0; /* disable timer while configuring */
TPM0->SC = 0x07; /* prescaler /128 */
TPM0->MOD = 0xFFFF; /* max modulo value */
TPM0->SC |= 0x80; /* clear TOF */
TPM0->SC |= 0x40; /* enable timeout interrupt */
TPM0->SC |= 0x08; /* enable timer */
NVIC->ISER[0] |= 0x00020000;
/* enable IRQ17 (bit 17 of ISER[0]) */
```

Timer Interrupt Programming - Example

```
/* Configuring Second Timer */
SIM->SCGC6 |= 0x02000000; /* enable clock to TPM1 */
TPM1->SC = 0; /* disable timer while configuring */
TPM1->SC = 0x07; /* prescaler /128 */
TPM1->MOD = 0x7FFF; /* half of the max modulo value */
TPM1->SC |= 0x40; /* enable timeout interrupt */
TPM1->SC |= 0x08; /* enable timer */
NVIC->ISER[0] |= 0x00040000; /* enable IRQ18 (bit 18 of
ISER[0]) */
__enable_irq(); /* global enable IRQs */
while (1) {
PTD->PTOR = 0x02; /* toggle blue LED */
delayMs(1500);  } }
```

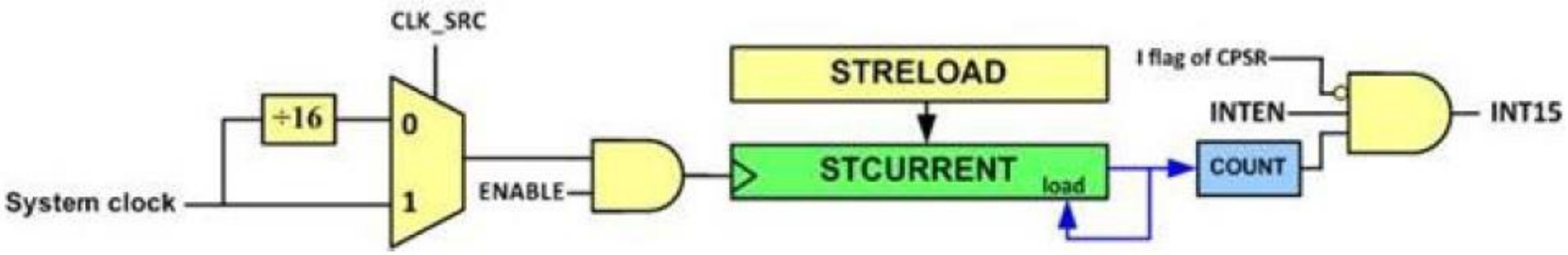
Basic Programming Techniques - Interrupts

Interrupts and Exceptions in ARM Cortex- M

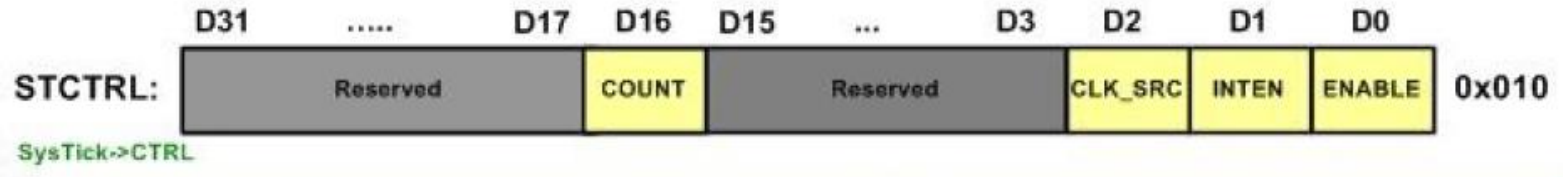
```
void TPM0_IRQHandler(void) {
PTB->PTOR = 0x40000; /* toggle red LED */
TPM0->SC |= 0x80; /* clear TOF */
}
void TPM1_IRQHandler(void) {
PTB->PTOR = 0x80000; /* toggle green LED */
TPM1->SC |= 0x80; /* clear TOF */
}
/* Delay n milliseconds *(
void delayMs(int n) {
int i; int j;
for(i = 0 ; i < n; i++)
for (j = 0; j < 7000; j++) {}
}
```

SysTick Programming and Interrupt

Another useful interrupt in ARM is the SysTick. The SysTick timer was discussed in previous class. Next, we will learn how to use the SysTick interrupt.



If INTEN=1, when the COUNT flag is set, it generates an interrupt. INTEN is D1 of the STCTRL register



SysTick Programming and Interrupt

bit	Name	Description
0	ENABLE	Enable (0: the counter is disabled, 1: enables SysTick to begin counting down)
1	INTEN	Interrupt Enable 0: Interrupt generation is disabled, 1: when SysTick counts to 0 an interrupt is generated
2	CLK_SRC	Clock Source 0: System clock divided by 16 1: System clock
16	COUNT	Count Flag 0: the SysTick has not counted down to zero since the last time this bit was read 1: the SysTick has counted down to zero *Note: this flag is cleared by reading the STRCTL or writing to STCURRENT register.*

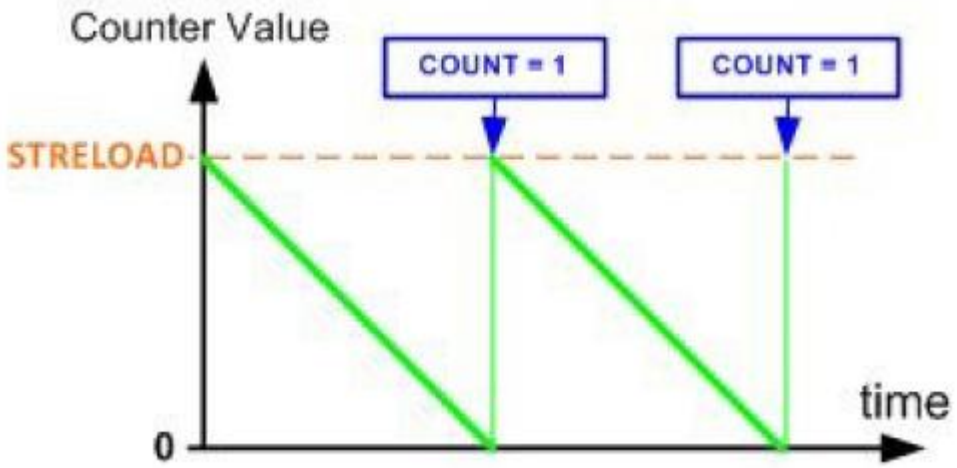
The SysTick interrupt can be used to initiate an action on a periodic basis.

This action is performed internally at a fixed rate without external signal.

Using Systick as an Interrupt

For example, in a given application we can use SysTick to read a sensor every 200 msec.

SysTick is used widely for an operating system so that the system software may interrupt the application software periodically (often at 10 ms interval) to monitor and control the system operations.



Using Systick as an Interrupt

Examining interrupt vector table for ARM Cortex, we see the SysTick is the interrupt #15.

The next program uses the SysTick to toggle the red LED of PTB18 every second. We need the RELOAD value of 41,940,000/16-1.

The CLK_SRC bit of CTRL register is cleared so system clock/16 is used as the clock source of SysTick.

The COUNT flag is raised every 41,940,000 clocks and an interrupt occurs. Then the RELOAD register is loaded with 41,940,000/16-1 automatically.

Using Systick as an Interrupt

Notice the interrupt is enabled in SysTick Control register.

Because SysTick is an interrupt below interrupt # 16, the enable/disable and the priority are not managed by the registers in the NVIC.

Its priority is controlled in the most significant byte of System Handler Priority 3 register (SHPR3) of System Control Block (SCB->SHP[1]).

There is no need to clear interrupt flag in the interrupt handler for SysTick.

Using Systick as an Interrupt

/* the red LED using the SysTick interrupt

- This program sets up the SysTick to interrupt at 1 Hz.

* The system clock is running at 41.94 MHz.

* In the interrupt handler, the red LED is toggled. */

```
#include <MKL25Z4.H>

int main (void) {
__disable_irq(); /* global disable IRQs */
```

Basic Programming Techniques - Interrupts

Interrupts and Exceptions in ARM Cortex- M

```
SIM->SCGC5 |= 0x400; /* enable clock to Port B */
PORTB->PCR[18] = 0x100; /* make PTB18 pin as GPIO */
PTB->PDDR |= 0x40000; /* make PTB18 as output pin */
SysTick->LOAD = 41940000/16-1; /* reload with number of
clocks per second */
SysTick->CTRL = 3; /* enable SysTick interrupt, use system
clock/16 */
__enable_irq(); /* global enable interrupt */
while (1) { /*wait here for interrupt */ } }

void SysTick_Handler(void) {
PTB->PTOR = 0x40000; /* toggle red LED */ }
```

Interrupt Priority Programming in Freescale ARM

The implementation of interrupt varies from vendor to vendor.

While ARM Holdings Co. has control over the standardization of the first 3 interrupts (INT0, INT1, and INT2), the ARM licensees are free to implement the interrupts of INT3-INT255.

The first three interrupts are Reset, NMI, and Hard Fault. For these three interrupts, Reset has the highest priority (with -3 priority number), then NMI (with -2), and Hard Fault (with -1), in that order.

In ARM, the lower value has higher priority.

Interrupt Priority Programming in Freescale ARM

All other interrupts have the priority number 0 meaning they have lower priority than Reset, NMI, and Hard Fault.

In the case of ARM Cortex, it groups several interrupts together with specific interrupt priority.

There are several special function registers dealing with the interrupts belonging to system exceptions of 4 to 15.

You can explore them by reading the ARM Cortex data sheet. In this section, we deal with the priority of peripheral interrupts of INT16 (IRQ0) to INT47 (IRQ31).

IRQ0 to IRQ31 in Freescale ARM KL25Z

The INT16 is assigned to IRQ0 since the first 16 interrupts (INT0-INT15) are used by the ARM core itself. Not all the IRQs are implemented in all ARM chips.

For example, The Freescale ARM KL25Z chip implements up to IRQ31 (or INT47). In other words, KL25Z has IRQ0 to IRQ31.

Notice that if we add 16 to IRQ# we get its INT#. We learned in Section 6.1 that, by multiplying the INT# by 4 we get its address in the interrupt vector table.

We must pay special attention to the IRQ# since this is used in the priority scheme used by the NVIC.

IRQ0 to IRQ31 in Freescale ARM KL25Z

According to ARM Cortex data sheet, an interrupt (exception) can be in one of the following four states:

Inactive. The exception is not active and not pending.
Pending. The exception is waiting to be serviced by the processor. An interrupt request from a peripheral or from software can change the state of the corresponding interrupt to pending.
Active. An exception that is being serviced by the processor but has not completed.
Note: An exception handler can interrupt the execution of another exception handler. Both exceptions are in the active state.
Active and Pending. The exception is being serviced by the uP, and there is a pending exception from the same source.

The IPR registers and Priority Grouping in ARM Cortex

The priority of an IRQ is assigned in one of the interrupt priority registers called *IPRx (Interrupt PRorityx)* in NVIC. If we do not assign a priority to an IRQ, by default, it has priority 0.

The IPR registers and Priority Grouping in ARM Cortex

Each IRQ uses one byte in an interrupt priority register. Therefore, each interrupt priority register holds priorities for four IRQ.

For example, IPR0 (IPR zero) holds the priorities of IRQ0, IRQ1, IRQ2 and IRQ3. In the same way, the priorities of IRQ4, IRQ5, IRQ6 and IRQ7 are assigned in IPR1.

For 32 IRQs, eight interrupt priority registers are used. The KL25Z device uses only the two most significant bits of the byte in the interrupt priority register.

With two bits, there can be four different priorities, 0 to 3.

The IPR registers and Priority Grouping in ARM Cortex

Notice, there is a pattern in the IPR# and IRQ# assignments. It follows the following formula:
IPRn IRQ(4n), IRQ(4n+1), IRQ(4n+2), and IRQ(4n+3)

In other words, we multiply the IPR# number by 4 (#x4) to get the first IRQ and from there we add 1, 2, and 3 to get all the three IRQs it supports.

To ease the calculation of finding the correct bits of the correct register to set the priority, the CMSIS has a macro NVIC_SetPriority defined in core_cm0plus.h for programmers to set the priority of an IRQ.

The IPR registers and Priority Grouping in ARM Cortex

For example, if we want to set the Timer 2 interrupt priority to 2, first we need to find out the IRQ number of Timer 2 interrupt, which is 19.

To locate the register for IRQ19, we will divide 19 by 4, which results in a quotient of 4 and remainder of 3. The byte that holds the priority of Timer 2 is byte 3 of IPR4. To get to byte 3, we need to shift the priority 24 bits (8 x 3) to the left and to get the two most significant bits, we need to shift it 6 more bits to the left. The statement will look like:

```
NVIC->IP[IRQn] |= PRIO << (8 * (IRQn % 4) + 6);
or
NVIC->IP[4] |= 2 << (8 * 3 + 6);
```

The IPR registers and Priority Grouping in ARM Cortex

This is tedious, it would be easier to use the macro:
NVIC_SetPriority (TPM2_IRQn, 2);
TPM2_IRQn is defined in MKL25Z4.h.

The next program illustrates two interrupts with different priority. In this example, delay function is called in the interrupt handler to demonstrate the preemption by higher priority interrupt

TPM0 is programmed to interrupt at 1 second interval. In the interrupt handler, the red LED is turned on for 500 ms. TPM1 is programmed to interrupt at 100 ms interval and in its interrupt handler, the green LED is turned on for 20 ms.

The IPR registers and Priority Grouping in ARM Cortex

Since TPM0 has higher priority, you will observe that the green LED is not blinking when TPM1 interrupt is running (when the red LED is on).

Now change the priority of the TPM1 to be higher than TPM0 by changing the line in TPM1_init( ) from

```
NVIC->IP[23] = 5 << 5; /* set timer2A interrupt priority to 5 */
```

to

```
NVIC->IP[23] = 3 << 5; /* set timer2A interrupt priority to 3 */
```

You will see that the green LED is blinking all the time so is the red LED because the TPM1 (green LED) preempts TPM0 interrupt handler (red LED).

The IPR registers and Priority Grouping in ARM Cortex

/* Testing nested interrupts
* Timer1 is setup to interrupt at 1 Hz.
* In timer interrupt handler, the red LED is turned on and a delay function of 350 ms is called. The LED is turned off at the end of the delay.
*
* Timer2 is setup to interrupt at 10 Hz.
* In timer interrupt handler, the blue LED is turned on and a delay function of 20 ms is called. The LED is turned off at the end of the delay.
*
* When Timer1 has higher priority, the Timer2 interrupts are blocked by Timer1 interrupt handler. You can see that when the red LED is on, the blue LED stops blinking.
* When Timer2 has higher priority, the Timer1 interrupt handler is preempted by Timer2 interrupts and the blue LED is blinking all the time.
* Timer2 interrupt handler also turns PD2 low and high so that it may be probed by the scope. */

The IPR registers and Priority Grouping in ARM Cortex

```
#include “MKL25Z4.h”
void Timer1_init(void);  void Timer2_init(void);  void delayMs(int n);
int main (void)
{ __disable_irq();
SIM->SCGC5 |= 0x400; /* enable clock to Port B */
SIM->SCGC5 |= 0x1000; /* enable clock to Port D */
PORTB->PCR[18] = 0x100; /* make PTB18 pin as GPIO */
PTB->PDDR |= 0x40000; /* make PTB18 as output pin */
PORTD->PCR[1] = 0x100; /* make PTD1 pin as GPIO */
PTD->PDDR |= 0x02; /* make PTD1 as output pin */
PORTD->PCR[2] = 0x100; /* make PTD2 pin as GPIO */
PTD->PDDR |= 0x04; /* make PTD2 as output pin */
Timer1_init();   Timer2_init();
__enable_irq();
while(1) /*wait here for interrupt */ { } }
```

Basic Programming Techniques - Interrupts

The IPR registers and Priority Grouping in ARM Cortex

```
void TPM1_IRQHandler(void)
{
PTD->PCOR |= 0x02; /* turn on blue LED */
delayMs(350);
PTD->PSOR |= 0x02; /* turn off blue LED */
TPM1->SC |= 0x80; /* clear TOF */
}
void TPM2_IRQHandler(void)
{
PTB->PCOR |= 0x40000; /* turn on red LED */
PTD->PCOR |= 0x04; /* make PTD2 low */
delayMs(50);
PTB->PSOR |= 0x40000; /* turn off red LED */
PTD->PSOR |= 0x04; /* make PTD2 high */
TPM2->SC |= 0x80; /* clear TOF */
}
```

The IPR registers and Priority Grouping in ARM Cortex

```
/* priority of TPM1 and TPM2 should be between 0 and 3 */

#define PRIOTPM1 2U
#define PRIOTPM2 3U

void Timer1_init(void)
{
SIM->SCGC6 |= 0x02000000; /* enable clock to TPM1 */
SIM->SOPT2 |= 0x03000000; /* use MCGIRCLK as timer counter clock */
TPM1->SC = 0; /* disable timer while configuring */
TPM1->SC = 0x02; /* prescaler /4 */
TPM1->MOD = 8192 - 1; //16384 - 1; /* modulo value */
TPM1->SC |= 0x80; /* clear TOF */
TPM1->SC |= 0x48; /* enable timer free-running mode and interrupt */
```

The IPR registers and Priority Grouping in ARM Cortex

```
/* set interrupt priority for TPM1 */
/* NVIC->IP[TPM1_IRQn / 4] |= PRIOTPM1 << ((TPM1_IRQn % 4) * 8 + 6); */
/* NVIC->IP[4] |= PRIOTPM1 << 22; */
NVIC_SetPriority(TPM1_IRQn, PRIOTPM1);
NVIC_EnableIRQ(TPM1_IRQn); /* enable Timer1 interrupt in NVIC */
}
void Timer2_init(void)
{
SIM->SCGC6 |= 0x04000000; /* enable clock to TPM2 */
SIM->SOPT2 |= 0x03000000; /* use MCGIRCLK as timer counter clock */
TPM2->SC = 0; /* disable timer while configuring */
TPM2->SC = 0x02; /* prescaler /4 */
TPM2->MOD = 819 - 1; /* modulo value */
TPM2->SC |= 0x80; /* clear TOF */
TPM2->SC |= 0x48; /* enable timer free-running mode and interrupt */
```

The IPR registers and Priority Grouping in ARM Cortex

```
/* set interrupt priority for TMP2 */
/* NVIC->IP[TPM2_IRQn / 4] |= PRIOTPM2 << ((TPM2_IRQn % 4) * 8 + 6); */
/* NVIC->IP[4] |= PRIOTPM2 << 30; */

NVIC_SetPriority(TPM2_IRQn, PRIOTPM2);
NVIC_EnableIRQ(TPM2_IRQn); /* enable Timer2 interrupt in NVIC */
}
// delay n milliseconds (16 MHz CPU clock)
void delayMs(int n)
{
int32_t i, j;
for(i = 0 ; i < n; i++)
for(j = 0; j < 7000; j++)
{} /* do nothing for 1 ms */
}
```