Section 3 – Interrupts

Tasks

In this section you will learn:

- The concept of Interrupts and polling
- The concept of NVIC in ARM Cortex M0+
- External Interrupt Controller (EIC) on SAMD20
- Servicing an Interrupt – An example
- Implementing SysTick Interrupt

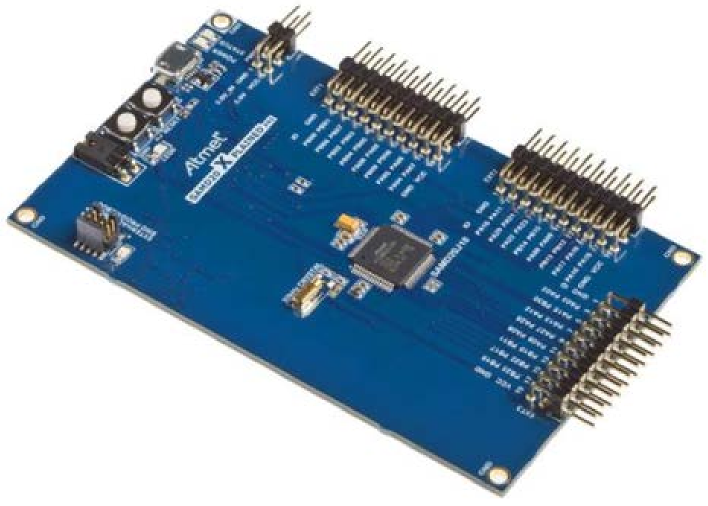


Table of Contents

1. It’s All in the Name

In its normal state of operation, a microcontroller executes a set of instructions in a loop. However, a certain “event” may occur that would require the microcontroller to “interrupt” its current instruction and perform an urgent task promptly, after which it returns to its normal state of operation. Imagine a passenger safety system in a car. As soon as a collision is detected, the first and immediate response should be the deployment of the airbag. Here the collision triggers an “interrupt” that deploys the airbag.

What exactly happens when an interrupt occurs? Here is the general sequence of events as depicted in Figure 1:

- The main code is running with interrupts enabled when an interrupt event occurs.
- The interrupt event sends an interrupt request to the CPU.
- After the current set of instructions is completed, the CPU responds. Within this response:
    - The current program counter value is saved in the stack. Some important register contents are also saved.
    - Program control jumps to the Interrupt Service Routine (ISR)[1] responsible for servicing the current interrupt.
    - ISR services the interrupt and executes a Return From Interrupt (RFI) instruction.
    - RFI restores the contents of the registers that were saved before the ISR is executed.
    - RFI recovers the program counter value.
- The main code continues to execute from the point where it stopped.

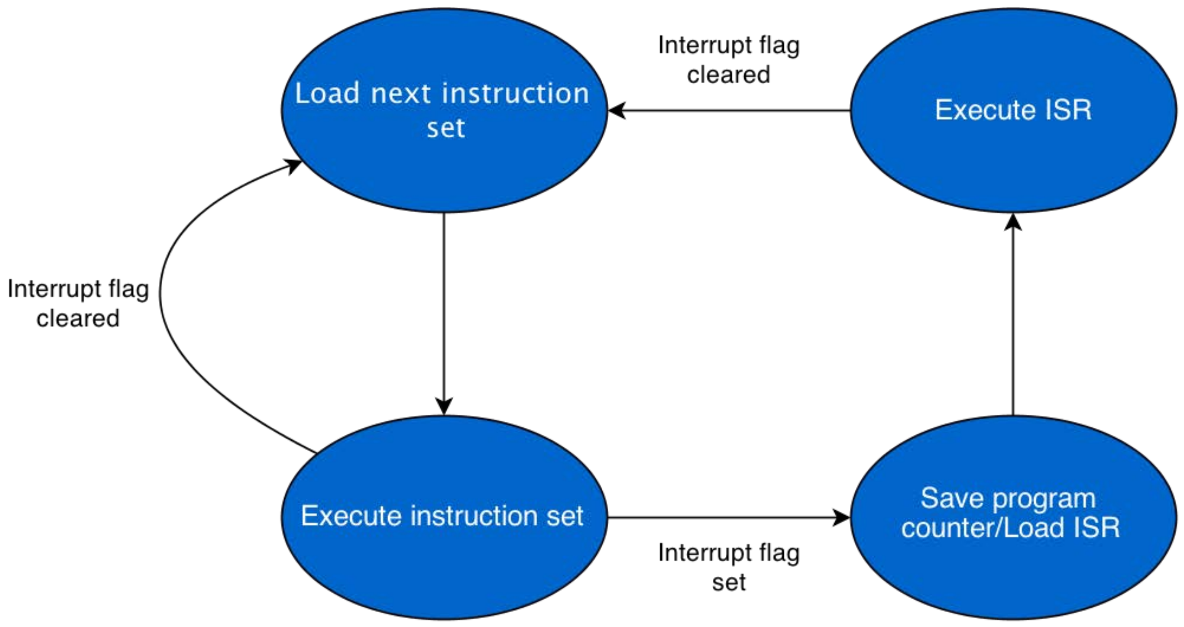


Figure 1: Interrupt state diagram

[1] ISR is explained in detail later in the section.

1.1 Types of Interrupts

There are two types of interrupts:

- Hardware Interrupt: An external device or peripheral sends an electrical signal to the CPU to signal that an event has occurred. For example - a button press or when the output of Analog Comparator reaches a certain value.
- Software Interrupt: This type of interrupt is a result of an error condition caused by the CPU itself. For example, when a portion of memory is accessed that does not exist. Such an interrupt is also called as “exception.” Another type of software interrupt similar to a “regular function” is often implemented to request services from low level system software such as device drivers. For example, computers often use software interrupt instructions to communicate with the disk controller to request data be read or written to a disk.

2. Interrupt vs. Polling - An Example

The concept of interrupt is basically to perform a certain task when a certain event occurs. Consider a system that controls a room heater:

When the temperature of a room rises to a set temperature, the heater should be turned off. Also, when the temperature of the room falls below a set temperature, the heater should be turned on. The room temperature is monitored by a temperature sensor.

There are two methods in which the above system can be implemented:

Method 1: The system interrupts its current activity and asks the sensor periodically: “Has the temperature risen above or fallen below the set range?” If the sensor answers yes, then the system either turns the heater on or off. This process is shown in Figure 2.

Method 2: The system continues to perform its current activities. When the temperature rises above or falls below the set range, the system is immediately informed, after which it immediately reads the current temperature and decides whether to turn the heater on or off. This process is shown in Figure 3.

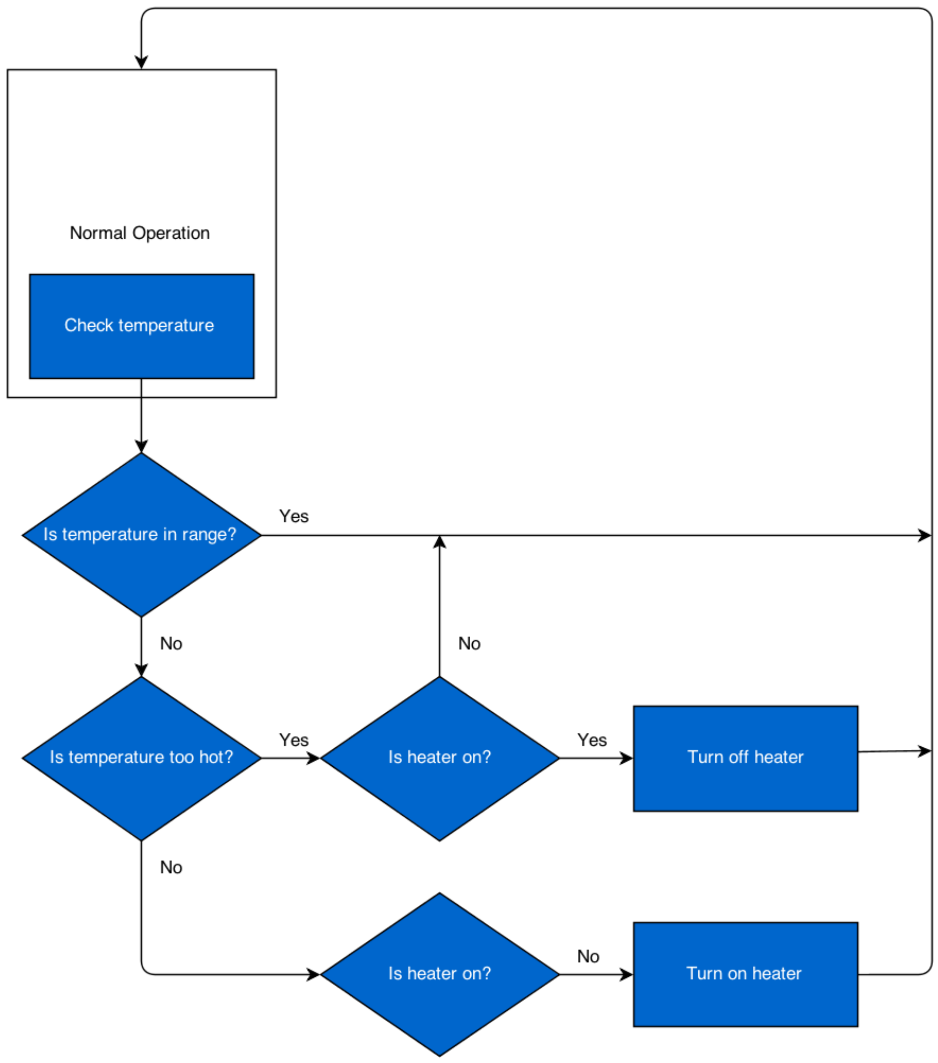


Figure 2: Temperature polling flow chart

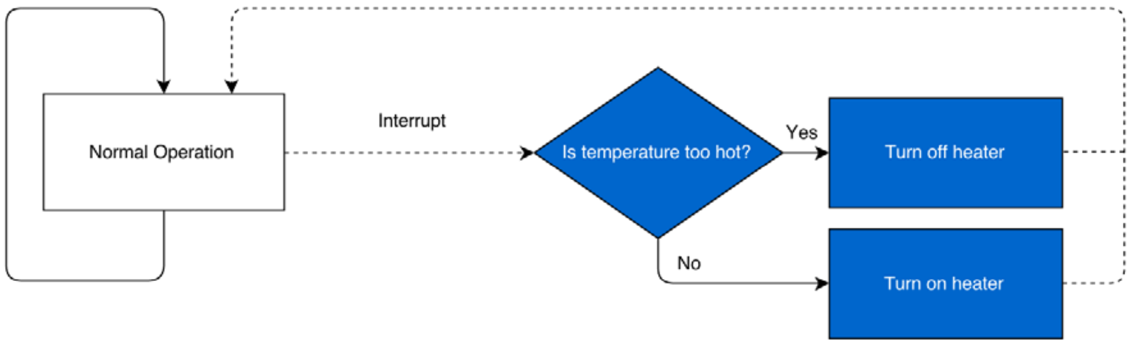


Figure 3: Temperature interrupt flow chart

Method 1 is called "Polling" and Method 2 is called "Interrupt."

CHECKPOINT What are the advantages and disadvantages of each method?

3. Nested Vectored Interrupt Controller

The Nested Vectored Interrupt Controller (NVIC) handles the different interrupts supported by the ARM Cortex M0+ microcontroller. We learned that when an interrupt occurs, the CPU branches from the current memory location to the Interrupt Service Routine (ISR). Each interrupt has its own ISR which is located in a unique memory location. The NVIC maintains a Vector Table which is a list of the memory locations of all the ISRs supported by the microcontroller. Let us take a closer look at some of these concepts.

3.1 Interrupt Service Routine

Every interrupt or exception has a handler called interrupt handler which is executed when the interrupt is fired. The handler is basically a C-like callback function. Below is a list of exception handlers in the startup_*device* file provided by ARM for Cortex M0+ core:

```
/* Default empty handler */
void Dummy_Handler(void);
/* Cortex-M0+ core handlers */
void NMI_Handler             ( void ) __attribute__ ((weak, alias("Dummy_Handler")));
void HardFault_Handler       ( void ) __attribute__ ((weak, alias("Dummy_Handler")));
void SVC_Handler             ( void ) __attribute__ ((weak, alias("Dummy_Handler")));
void PendSV_Handler          ( void ) __attribute__ ((weak, alias("Dummy_Handler")));
void SysTick_Handler         ( void ) __attribute__ ((weak, alias("Dummy_Handler")));
/* Peripherals handlers */
void PM_Handler              ( void ) __attribute__ ((weak, alias("Dummy_Handler")));
void SYSCTRL_Handler         ( void ) __attribute__ ((weak, alias("Dummy_Handler")));
void WDT_Handler             ( void ) __attribute__ ((weak, alias("Dummy_Handler")));
void RTC_Handler             ( void ) __attribute__ ((weak, alias("Dummy_Handler")));
void EIC_Handler             ( void ) __attribute__ ((weak, alias("Dummy_Handler")));
void NVMCTRL_Handler         ( void ) __attribute__ ((weak, alias("Dummy_Handler")));
void EVSYS_Handler           ( void ) __attribute__ ((weak, alias("Dummy_Handler")));
void SERCOM0_Handler         ( void ) __attribute__ ((weak, alias("Dummy_Handler")));
void SERCOM1_Handler         ( void ) __attribute__ ((weak, alias("Dummy_Handler")));
void SERCOM2_Handler         ( void ) __attribute__ ((weak, alias("Dummy_Handler")));
void SERCOM3_Handler         ( void ) __attribute__ ((weak, alias("Dummy_Handler")));
void SERCOM4_Handler         ( void ) __attribute__ ((weak, alias("Dummy_Handler")));
void SERCOM5_Handler         ( void ) __attribute__ ((weak, alias("Dummy_Handler")));
void TC0_Handler             ( void ) __attribute__ ((weak, alias("Dummy_Handler")));
void TC1_Handler             ( void ) __attribute__ ((weak, alias("Dummy_Handler")));
void TC2_Handler             ( void ) __attribute__ ((weak, alias("Dummy_Handler")));
void TC3_Handler             ( void ) __attribute__ ((weak, alias("Dummy_Handler")));
void TC4_Handler             ( void ) __attribute__ ((weak, alias("Dummy_Handler")));
void TC5_Handler             ( void ) __attribute__ ((weak, alias("Dummy_Handler")));
void TC6_Handler             ( void ) __attribute__ ((weak, alias("Dummy_Handler")));
void TC7_Handler             ( void ) __attribute__ ((weak, alias("Dummy_Handler")));
void ADC_Handler             ( void ) __attribute__ ((weak, alias("Dummy_Handler")));
void AC_Handler              ( void ) __attribute__ ((weak, alias("Dummy_Handler")));
void DAC_Handler             ( void ) __attribute__ ((weak, alias("Dummy_Handler")));
```

In the above list of interrupt handlers, each interrupt is defined as a “Dummy_Handler” which is marked as a “weak” implementation. Here the Dummy_Handler() function is a dummy function (as the name suggests) and is shown as:

```
/**
 * \brief Default interrupt handler for unused IRQs.
 */
void Dummy_Handler(void)
{
        while (1) {
        }
}
```

It has no interrupt handling code. Therefore, it is declared as “WEAK” as an indication to the *linker* to include the dummy implementation only if the *non*-weak implementation is not found. It is for the user to implement the actual ISR.

The ARM Cortex-M core implements a set of interrupt handlers:

• NMI_Handler:	This handler is used when response time is most critical and can never be masked or ignored. They are typically used to signal non-recoverable hardware errors.
• HardFault_Handler:	This ISR is entered if an error condition occurs. Hard faults are one of the most commonly occurring errors and are among the most difficult to debug.
• SVC_Handler:	SVC stands for supervisor call which are used to request privileged operations or access to system resources from an operating system. SVC_Handler services the SVC calls.
• SysTick_Hanlder:	This interrupt can be set to occur at a set time interval to provide timing functionality.

Each peripheral has an ISR assigned to it:

• PM_Handler:	ISR to service the Power Manager.
• SYSCTRL_Handler:	ISR to service the System Controller.
• WDT_Handler:	ISR to service the Watch Dog Timer.
• RTC_Handler:	ISR to service the Real Time Clock.
• EIC_Handler:	ISR to service the External Interrupt Controller.
• NVM_CTRL_Handler:	ISR to service the Non-Volatile Memory Controller.
• EVSYS_Handler:	ISR to service the Event System.
• SERCOMn_Handler:	ISR to service the Serial Communication Interface.
• TCn_Handler:	ISR to service the Timer Counter.
• ADC_Handler:	ISR to service the ADC.
• AC_Handler:	ISR to service the Analog Comparator.
• DAC_Handler:	ISR to service the DAC.

3.2 Interrupt Vector Table

The interrupt vector table contains the start addresses for all interrupt handlers. The start address is also known as the exception vector. The vector table is fixed at the base address 0x00000000. Each subsequent entry in the vector table is at a fixed offset from the base address. Figure 4 shows the order of the exception vectors in the vector table.

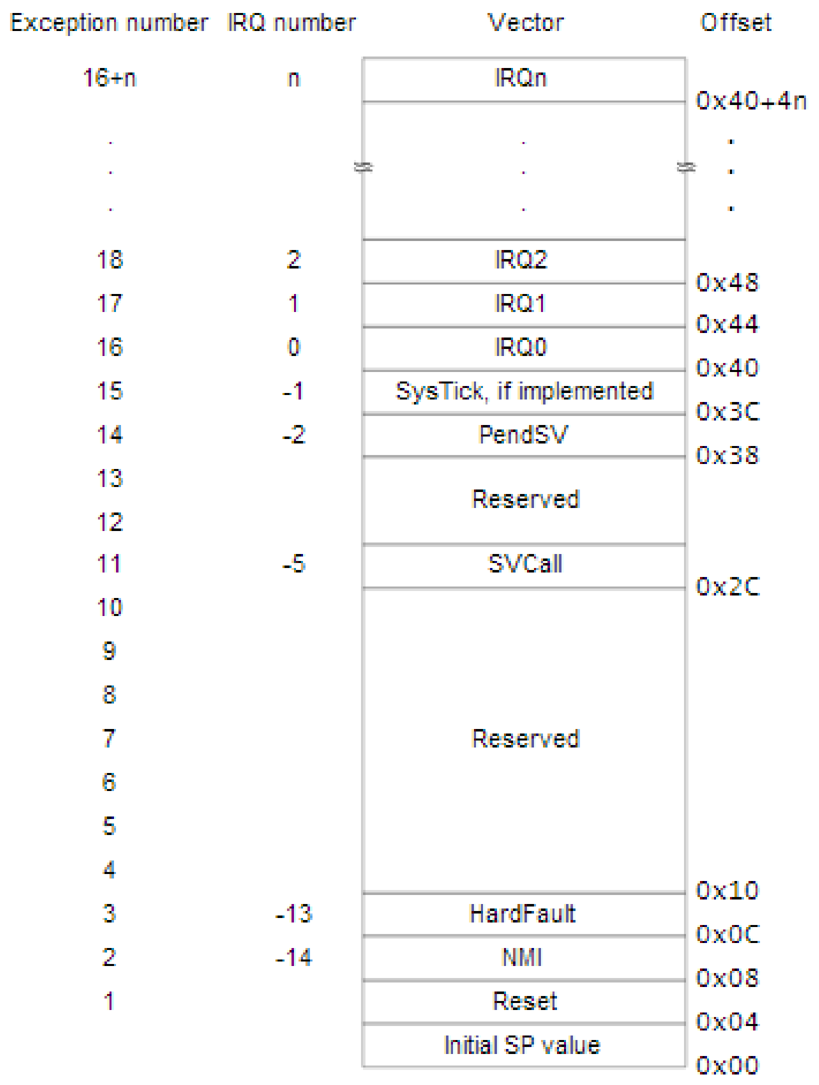


Figure 4: Vector Table

In Figure 4, the Exception number is a unique number assigned to each exception for the core to identify which interrupt/exception has occurred. The IRQ Number is a unique index assigned to each interrupt line to identify it.

The IRQ numbers for the different peripheral sources in SAMD20 are as shown in Table 1.

Table 1: IRQn for peripheral interrupts in SAMD20

Peripheral Source	NVIC Line
EIC NMI – External Interrupt Controller Non Maskable Interrupt	NMI
PM – Power Manager	0
SYSCTRL – System Controller	1
WDT – Watchdog Timer	2
RTC – Real Time Counter	3
EIC – External Interrupt Controller	4
NVMCTRL – Non-Volatile Memory Controller	5
EVSYS – Event System	6
SERCOM0 – Serial Communication Interface 0	7
SERCOM1 – Serial Communication Interface 1	8
SERCOM2 – Serial Communication Interface 2	9
SERCOM3 – Serial Communication Interface 3	10
SERCOM4 – Serial Communication Interface 4	11
SERCOM5 – Serial Communication Interface 5	12
TC0 – Timer/Counter 0	13
TC1 – Timer/Counter 1	14
TC2 – Timer/Counter 2	15
TC3 – Timer/Counter 3	16
TC4 – Timer/Counter 4	17
TC5 – Timer/Counter 5	18
TC6 – Timer/Counter 6	19
TC7 – Timer/Counter 7	20
ADC – Analog-to-Digital Converter	21
AC – Analog Comparator	22
DAC – Digital-to-Analog Converter	23
PTC – Peripheral Touch Controller	24

3.3 NVIC Registers

ARM specifies registers to control the NVIC, listed in the following:

- *Enable External Interrupt* (`NVIC_EnableIRQ`): This function enables an interrupt in the NVIC controller. The parameter to this function is the IRQ number to indicate which interrupt needs to be enabled.
- *Disable External Interrupt* (`NVIC_DisableIRQ`): This function disables an interrupt in the NVIC controller. The parameter to this function is the IRQ number to indicate which interrupt needs to be disabled.
- *Get Pending Interrupt* (`NVIC_GetPendingIRQ`): This function reads the pending register in the NVIC and returns the pending bit for the specified interrupt. The parameter to this function is the IRQ number.
- *Set Pending Interrupt* (`NVIC_SetPendingIRQ`): This function sets the pending bit of an external interrupt. The parameter to this function is the IRQ number.
- *Clear Pending Interrupt* (`NVIC_ClearPendingIRQ`): This function clears the pending bit of an external interrupt. The parameter to this function is the IRQ number.
- *Set Interrupt Priority* (`NVIC_SetPriority`): This function sets the priority of the interrupt. Note that priority cannot be set for every core interrupt. The parameter to this function is the IRQ number of the interrupt whose priority needs to be set.
- *Get interrupt Priority* (`NVIC_GetPriority`): This function reads the priority of an interrupt. The interrupt number can be positive to specify an external (device specific) interrupt, or negative to specify an internal (core) interrupt.

READ
& LEARN

Read and understand Section 11.22 "Interrupt Line Mapping" on page 28 of the SAMD20 datasheet.

3.4 Who goes first?

What happens when two or more interrupts occur at the same time? Clearly, only one interrupt can be serviced at a time. The solution is to assign each interrupt a priority level to determine which gets serviced first. The interrupt priority is selected based on the following criteria:

- Interrupt latency: The interrupt latency is the maximum time within which an interrupt must be serviced. If it is not serviced in this time, some event is lost or performance is seriously degraded.
- Interrupt execution time: The interrupt execution time is the number of machine cycles required to service the interrupt. Interrupts with a short interrupt latency time must have a short interrupt service time.

The priority of each interrupt is programmable, allowing the order in which interrupts are serviced to be dynamically changed. This is done by programming the values in the Vector Priority Registers for the corresponding interrupt line (`NVIC_SetPriority`). There are 16 programmable priority levels available, 0-15. By default, all interrupts are set to the lowest priority level (15), but each interrupt can be set to any priority level required.

If multiple interrupts are set to the same priority level or if the interrupt level is not set, the fixed hardware priority levels are used to determine the order in which the interrupts are serviced. The lower the priority number, higher the priority. Therefore, Interrupt 0 has the highest hardware priority.

Sometimes a hardware interrupt may be ignored by setting a bit in the corresponding interrupt's Interrupt Mask Register. However, this is done through the interrupt control register of the particular peripheral controller. Not all interrupts can be masked. Interrupts that lack an associated bit mask cannot be masked. Such interrupts are called Non-Maskable Interrupts (NMI).

4. External Interrupt Controller in SAMD20

The External Interrupt Controller (EIC) allows the hardware pins in the SAMD20 to be configured as an interrupt line and to act as an interrupt source. There is support for up to 16 external GPIO interrupts. Each interrupt is separately maskable. Here are some additional features of the EIC:

- An interrupt can be triggered on either rising edge or falling edge of the signal on a pin.
- An interrupt can be triggered on change in levels, either high or low, of a signal on a pin.

READ & LEARN

Read Section 20.6 on page 250 of the SAMD20 datasheet and understand the function of the EIC.

4.1 Port Configuration for EIC

In the "Port Control" Section, we learned that the hardware pins can be used either as a GPIO or act as a specific peripheral. For the hardware pin to be configured as an interrupt line, it must be configured to perform the peripheral function.

READ & LEARN

Read Section 6 "I/O Multiplexing and Considerations" on page 15 of the SAMD20 datasheet. Learn from Table 6-1 how the different hardware pins are multiplexed to serve different peripheral functions.

CHECKPOINT

For a hardware pin to perform as External Interrupt Source, which peripheral function must be chosen?

4.2 External Interrupt Controller Registers

The register mapping to control EIC is as shown in Table 2.

Table 2. EIC registry summary

Offset	Register	Name	Access
0x00	Control	CTRL	R/W
0x01	Status	STATUS	R
0x02	Non-Maskable Interrupt Control	NMICTRL	R/W
0x03	Non-Maskable Interrupt Flag Status and Clear	NMIFLAG	R/W
0x04	Event Control	EVCTRL	R/W
0x08	Interrupt Enable Clear	INTENCLR	R/W
0x0C	Interrupt Enable Set	INTENSET	R/W
0x10	Interrupt Flag Status and Clear	INTFLAG	R/W
0x14	Wake-Up Enable	WAKEUP	R/W
0x18+n*0x4 [n=0..1]	Configuration n	CONFIGn	R/W

READ & LEARN

Read the “Register Description” section on page 255 of Section 20 in the datasheet and learn the functions of the different registers.

4.3 Initializing the EIC

The basic steps involved in initializing the EIC are as follows:

1. Ensure that the multiplexing feature of the hardware pin is enabled so that it can perform a peripheral function. From Table 6-1 of the datasheet, we know that Peripheral A acts as EIC.
2. Enable the Generic CLK and APB CLK.
3. Set the right values to the EIC configuration registers (NMICTRL, EVCTRL, WAKEUP, CONFIGy) based on how you want it to function.
4. Enable EIC by setting the enable bit in the Control register.

5. Implementation of Interrupt – An Example

Now that we have learnt some basic concepts of interrupts, let us proceed to implement one.

We have already covered polling vs. interrupt method. Now let us recollect what we did in the Port Control section. One of the assignments was to toggle the LED when the button was pressed.

CHECKPOINT

How was the button press event handled - by Polling or Interrupts?

To help you recollect, here is the snippet of code that handled button press:

```
While(1)
{
       if(!( por->IN.reg & PORT_PA15))
       {
          por->OUTTGL.reg = PORT_PA14;
       }
       wait(500);
}
```

What we are actually doing is polling for the status of the pin PA15 in a continuous loop. If the pin went low, then the button was pressed and hence the button press event was handled, which in this case is to toggle the LED. Now let us implement the same thing using interrupts.

6. Task 1: Toggle LED0 When SW0 Button is Pushed Using Interrupts

CHECKPOINT Before you start the first task, you should be familiar with the following:

- The concept of interrupts and ISR
- The interrupts available in SAMD20. Which interrupt are you going to use to handle the button press event?
- The function of interrupt vector table
- The EIC in SAMD20 and how to use it

You will work on the same Studio 6.0 project template you used in the Port Control section.

TO DO Your task is to write a function that will correctly configure the SAMD20 registers in order to raise an interrupt event when the SW0 button is pushed. Also write the corresponding interrupt handler function.

6.1 Which registers enable EIC?

External Interrupt Controller allows us to use hardware pins to generate interrupts. Therefore, this is what we will use in the current task. Let us review the steps to configure and use the EIC to detect button push via interrupts:

1. *Enable Generic Clock and APB clock*: Since we have not explained clocks in SAMD20, the function to perform this will be provided to you.
2. *Port Configuration:* We already know how to configure the LED pin (PA14) and the SW0 push button (PA15) based on our exercise in Section 2. All we have to do is:
   a. Set the PMUXEN bit of the PINCFGy register.
   b. Configure the PMUXn register so that the SW0 pin serves as EXTINT.
3. *EIC Configuration:* The EIC related registers to be configured are as follows:
   a. Enable the EIC.
   b. Configure the EIC to perform edge (rising or falling) detection or level (high or low) detection.
   c. Enable the External Interrupt on SW0 (PA15) so that it is able to cause an interrupt event.
4. *Enable EIC function in ARM NVIC Interrupt Controller*

CHECKPOINT

In Step #2.b above, the PMUXn register for SW0 should serve as EXTINT. Based on the Table 6-1 on Page 15 of the SAMD20 datasheet, Peripheral A serves the EXTINT function. Therefore what should be the value of the PMUXn register?

TO DO

Go back to Table 2 and review all the registers available for EIC Control. Which ones do you think you can use to accomplish Step #3 mentioned above? Now complete Table 3 below.

Table 3: Registers to configure EIC

Register	Function

6.2 Understanding the SAMD20 System level header files

You have been provided with an Atmel Studio project template to work on. Let us take a look at some header files you will need to understand in order to configure the EIC registers and use the EIC interrupt handler.

6.2.1 component_eic.h

The component_eic.h header file has important definitions pertaining to the EIC module. It defines a structure called Eic that lists all registers related to EIC as shown below:

```
typedef struct {
  __IO EIC_CTRL_Type          CTRL;        /**< Offset: 0x00 (R/W  8) Control Register */
  __I  EIC_STATUS_Type        STATUS;      /**< Offset: 0x01 (R/   8) Status Register */
  __IO EIC_NMICTRL_Type       NMICTRL;     /**< Offset: 0x02 (R/W  8) NMI Control
                                                    Register */
  __IO EIC_NMIFLAG_Type       NMIFLAG;     /**< Offset: 0x03 (R/W  8) NMI Interrupt Flag
                                                    Register */
  __IO EIC_EVCTRL_Type        EVCTRL;      /**< Offset: 0x04 (R/W 32) Event Control
                                                    Register */
  __IO EIC_INTENCLR_Type      INTENCLR;    /**< Offset: 0x08 (R/W 32) Interrupt Enable
                                                    Clear Register */
  __IO EIC_INTENSET_Type      INTENSET;    /**< Offset: 0x0C (R/W 32) Interrupt Enable
                                                    Set Register */
  __IO EIC_INTFLAG_Type       INTFLAG;     /**< Offset: 0x10 (R/W 32) Interrupt Flag
                                                    Status and Clear Register */
  __IO EIC_WAKEUP_Type        WAKEUP;      /**< Offset: 0x14 (R/W 32) Wake-up Enable
                                                    Register */
  __IO EIC_CONFIG_Type        CONFIG[2];   /**< Offset: 0x18 (R/W 32) Config Register
                                                    [NUMBER_OF_CONFIG_REGS] */
} Eic;
```

You should already be familiar with these registers which were listed in Table 2. The comments alongside each register gives the register memory offset, read/write permissions, register size in bits, and description of the register.

Each register is assigned a “type” such as `EIC_CTRL_Type:`

```
typedef union {
  struct {
    uint8_t  SWRST:1;          /*!< bit:      0  Software Reset                     */
    uint8_t  ENABLE:1;         /*!< bit:      1  Enable                             */
    uint8_t  :6;               /*!< bit:  2.. 7  Reserved                           */
  } bit;                       /*!< Structure used for bit  access                  */
  uint8_t reg;                 /*!< Type      used for register access              */
} EIC_CTRL_Type;
```

You can access the control register using “`uint32_t reg`” in the following way:

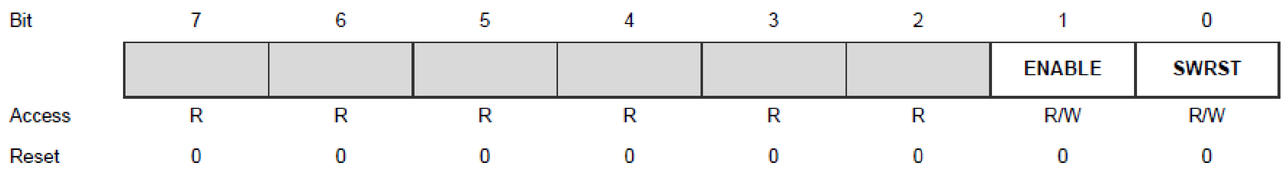
```
       eicP->CTRL.reg = <value to be assigned>;
```

Here `eicP` is the pointer variable to the `Eic` structure.

The reset value of each register is also defined in the header file. For example:

```
#define EIC_EVCTRL_OFFSET           0x04         /**< (EIC_EVCTRL offset) Event Control
                                                       Register */
#define EIC_EVCTRL_RESETVALUE       0x00000000   /**< (EIC_EVCTRL reset_value) Event Control
                                                       Register */
```

Finally, the bit-wise register description is provided in the header file. For example, here is the description of the Control Register from the datasheet:



Bit	7	6	5	4	3	2	1	0
							ENABLE	SWRST
Access	R	R	R	R	R	R	R/W	R/W
Reset	0	0	0	0	0	0	0	0

Figure 5: Description of the Control Register

This is described in the header file as below:

```
#define EIC_CTRL_SWRST_Pos          0            /**<  (EIC_CTRL) Software Reset */
#define EIC_CTRL_SWRST              (0x1u << EIC_CTRL_SWRST_Pos)
#define EIC_CTRL_ENABLE_Pos         1            /**<  (EIC_CTRL) Enable */
#define EIC_CTRL_ENABLE             (0x1u << EIC_CTRL_ENABLE_Pos)
#define EIC_CTRL_MASK               0x03u        /**<  (EIC_CTRL) MASK Register */
```

To summarize the component_eic.h completely defines the EIC related registers.

6.2.2 startup_samd20_gcc.c

The startup_samd20_gcc.c file is the Atmel variant of ARM startup code. It contains the following:

- Vectored Interrupt Table for Cortex-M0+ Core
- Interrupt handlers defined as dummy handlers with weak implementation (This concept was covered in Section 3.1)
- Startup code which is basically the reset handler

6.2.3 core_cm0plus.h

This is part of CMSIS (Cortex Microcontroller Software Interface Standard) software provided by ARM. CMSIS is a vendor-independent hardware abstraction layer for the Cortex-M processor series. The CMSIS enables consistent and simple software interfaces to the processor and the peripherals, simplifying software re-use, reducing the learning curve for new microcontroller developers and reducing the time to market for new devices[2].

This header file contains Core Register definitions such as:

- Core Register
- Core NVIC Register
- Core SCB Register
- Core SysTick Register
- Core MPU Register

For example here is the definition of the Core NVIC registers:

```
/** \ingroup    CMSIS_core_register
    \defgroup   CMSIS_NVIC  Nested Vectored Interrupt Controller (NVIC)
          Type definitions for the NVIC Registers
  @{
 */

/**   Structure type to access the Nested Vectored Interrupt Controller (NVIC).
 */
typedef struct
{
  __IO uint32_t ISER[1];                /*!< Offset: 0x000 (R/W)  Interrupt Set Enable
                                                 Register           */
       uint32_t RESERVED0[31];
  __IO uint32_t ICER[1];                /*!< Offset: 0x080 (R/W)  Interrupt Clear Enable
                                                 Register          */
       uint32_t RSERVED1[31];
  __IO uint32_t ISPR[1];                /*!< Offset: 0x100 (R/W)  Interrupt Set Pending
                                                 Register           */
       uint32_t RESERVED2[31];
  __IO uint32_t ICPR[1];                /*!< Offset: 0x180 (R/W)  Interrupt Clear Pending
                                                 Register         */
       uint32_t RESERVED3[31];
       uint32_t RESERVED4[64];
  __IO uint32_t IP[8];                  /*!< Offset: 0x300 (R/W)  Interrupt Priority
                                                 Register              */
}  NVIC_Type;
```

It also contains bit level description of the Core related registers. For example below is the description of the Systick Control and Status Register:

[2] *Arm-Products-Processors-Cortex*. (n.d.). Retrieved from Arm official website.

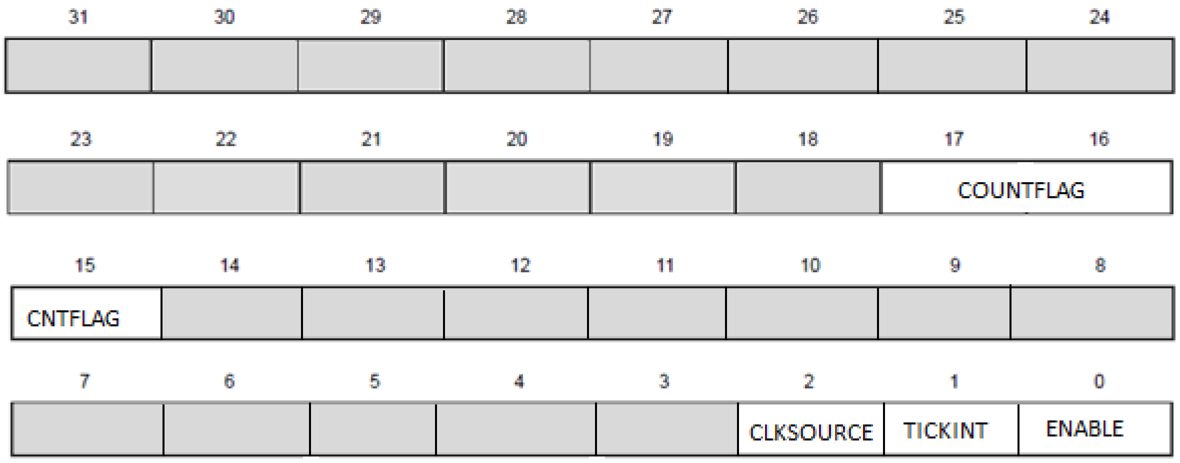


Figure 6: Description of the SysTick Control and Status Register

This is described in the header file as below:

```
/* SysTick Control / Status Register Definitions */
#define SysTick_CTRL_COUNTFLAG_Pos         16
/*!< SysTick CTRL: COUNTFLAG Position */
#define SysTick_CTRL_COUNTFLAG_Msk         (1UL << SysTick_CTRL_COUNTFLAG_Pos)
/*!< SysTick CTRL: COUNTFLAG Mask */
#define SysTick_CTRL_CLKSOURCE_Pos          2
/*!< SysTick CTRL: CLKSOURCE Position */
#define SysTick_CTRL_CLKSOURCE_Msk         (1UL << SysTick_CTRL_CLKSOURCE_Pos)
/*!< SysTick CTRL: CLKSOURCE Mask */
#define SysTick_CTRL_TICKINT_Pos            1
/*!< SysTick CTRL: TICKINT Position */
#define SysTick_CTRL_TICKINT_Msk           (1UL << SysTick_CTRL_TICKINT_Pos)
/*!< SysTick CTRL: TICKINT Mask */
#define SysTick_CTRL_ENABLE_Pos             0
/*!< SysTick CTRL: ENABLE Position */
#define SysTick_CTRL_ENABLE_Msk            (1UL << SysTick_CTRL_ENABLE_Pos)
/*!< SysTick CTRL: ENABLE Mask */
```

The header file also has function definitions to access the core registers. For example here is the function to access the NVIC register:

```
/**   Enable External Interrupt

    The function enables a device-specific interrupt in the NVIC interrupt controller.

    \param [in]      IRQn  External interrupt number. Value cannot be negative.
 */
__STATIC_INLINE void NVIC_EnableIRQ(IRQn_Type IRQn)
{
  NVIC->ISER[0] = (1 << ((uint32_t)(IRQn) & 0x1F));
}
```

Finally, the header file contains memory mapping of Cortex-M0+ Hardware:

```
#define SCS_BASE            (0xE000E000UL)
System Control Space Base Address */
#define SysTick_BASE        (SCS_BASE +  0x0010UL)
/*!< SysTick Base Address              */
#define NVIC_BASE           (SCS_BASE +  0x0100UL)
/*!< NVIC Base Address                 */
#define SCB_BASE            (SCS_BASE +  0x0D00UL)
/*!< System Control Block Base Address */
#define SCB                 ((SCB_Type       *)     SCB_BASE      )
/*!< SCB configuration struct            */
#define SysTick             ((SysTick_Type   *)     SysTick_BASE  )
/*!< SysTick configuration struct        */
#define NVIC                ((NVIC_Type      *)     NVIC_BASE     )
/*!< NVIC configuration struct           */
#if (__MPU_PRESENT == 1)
  #define MPU_BASE          (SCS_BASE +  0x0D90UL)
/*!< Memory Protection Unit              */
  #define MPU               ((MPU_Type       *)     MPU_BASE      )
/*!< Memory Protection Unit              */
#endif
```

In the link below is a complete list of Core registers provided by CMSIS found in Table 4.1 under:

Cortex-M0+ > **Revision: r0p1** > **Cortex-M0+ Technical Reference** > **System Control** > **System control register summary**.

http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0413c/Bhccjgga.html

6.3 Implementation of EIC interrupt

CHECKPOINT

- Which registers will you use to configure EIC?
- Have you understood how to use the Eic structure?

As mentioned, you have been provided with the code to enable the clocks required by EIC.

6.3.1 Accessing the Port Register

We have already configured the LED0 and SW0 pins in the Port Control Section. In that section, we used the pins as a simple GPIO. In addition, in this section we are going to use the EXTINT peripheral function of the SW0 pin so as to cause an interrupt.

CHECKPOINT How do you write code to enable peripheral function for SW0 pin?

TIPS Reuse the code from the previous assignment to configure the LED0 and SW0 pins and enable the PMUXEN bit in the PINCNFGy register. Also set the value in the PMUXn register so as to perform peripheral function A, which is EXTINT function.

6.3.2 Accessing the EIC Control Register

The EIC register defines all the registers available for EIC control.

```
typedef struct {
  __IO EIC_CTRL_Type             CTRL;         /**<  Offset: 0x00 (R/W  8) Control
                                                         Register */
  __I  EIC_STATUS_Type           STATUS;       /**<  Offset: 0x01 (R/   8) Status
                                                         Register */
  __IO EIC_NMICTRL_Type          NMICTRL;      /**< Offset: 0x02 (R/W  8) NMI Control
                                                         Register */
  __IO EIC_NMIFLAG_Type          NMIFLAG;      /**< Offset: 0x03 (R/W  8) NMI Interrupt
                                                         Flag Register */
  __IO EIC_EVCTRL_Type           EVCTRL;       /**< Offset: 0x04 (R/W 32) Event Control
                                                         Register */
  __IO EIC_INTENCLR_Type         INTENCLR;     /**< Offset: 0x08 (R/W 32) Interrupt
                                                         Enable Clear Register */
  __IO EIC_INTENSET_Type         INTENSET;     /**< Offset: 0x0C (R/W 32) Interrupt
                                                         Enable Set Register */
  __IO EIC_INTFLAG_Type          INTFLAG;      /**< Offset: 0x10 (R/W 32) Interrupt
                                                         Flag Status and Clear Register */
  __IO EIC_WAKEUP_Type           WAKEUP;       /**< Offset: 0x14 (R/W 32) Wake-up
                                                         Enable Register */
  __IO EIC_CONFIG_Type           CONFIG[2];    /**< Offset: 0x18 (R/W 32) Config
                                                         Register [NUMBER_OF_CONFIG_REGS] */
} Eic;
```

We could easily access the Eic register the following way:

```
Eic *eicP = EIC;
```

Here EIC is the base address definition:

```
#define EIC            (0x40001800U) /**< \brief (EIC) APB Base Address */
```

CHECKPOINT How will you use the structure variable `eicP` to do the following?:

1. Enable EIC
2. Perform Edge and level detection
3. Enable the interrupt pin on SW0

6.3.3 Enabling EIC in the Global NVIC

We have enabled the interrupt on SW0 pin via the `INTENSET` register. We now have to enable the global interrupts using the NVIC register – Interrupt Set-Enable Register. In the section 6.2.2 we have explained the “startup_samd20_gcc.c” file which provides a function to set/enable an interrupt:

```
NVIC_EnableIRQ (IRQn_Type IRQn)
```

Here the function parameter is the IRQ number of the interrupt line.

We have already covered the IRQ number of all the peripheral lines in Table 1. Also in Port Control Section 2, we have already seen that the samd20j18.h header file enumerates the IRQ numbers of all the peripherals.

6.3.4 Writing the EIC ISR

In Section 6.2.2, we have already mentioned that startup_samd20.c has “WEAK” dummy implementation of all the interrupt handlers. It is for the user to implement the actual interrupt handler. We implement this in the *main.c* file of the template project provided below:

```
/** Handler for the EXTINT hardware module interrupt. */
void EIC_Handler(void)
{
       Port *ports = PORT_INSTS;
       Eic *eicP = EIC_INSTS;
       PortGroup *por = &(ports->Group[0]);
       por->OUTTGL.reg = PORT_PA14;
       eicP->INTFLAG.reg = 1 << 15;
}
```

Your EIC Interrupt Handler should do the following:

1. Toggle the LED.
2. Read the SW0 interrupt flag in order to clear it.

An Interrupt Service Routine function is very similar to a normal function with some key differences.

TIPS

While writing an ISR function, keep the following things in mind:

- Keep the ISR as simple and short as possible. The idea is to quickly perform a special task and return to normal state of affairs. Don’t try to do too much in the ISR.
- Avoid print messages at all cost!
- Don’t ever call a delay() function from inside the ISR! This is very bad programming style.
- An ISR does not return any value. You may want to set a global variable instead.
- Inside an ISR, interrupts are disabled. Therefore don’t do anything that requires an interrupt to be serviced.
- Always clear the interrupt that is being serviced inside the ISR to avoid infinite call to the ISR.

7. Task 2: Implementation with SysTick_Handler

In Section 2 - Port Control, we handled the button push using the polling technique. In this section we used EIC controller to implement interrupt based handling of the bush button. As an additional exercise, let us use Systick to generate interrupt at a fixed periodic interval. In the SysTick ISR, we will check if the button SW0 was pushed and toggle the LED0 accordingly.

7.1 The SysTick Registers

Let us recollect what we learnt in Section 6.2.3. CMSIS defines a set of registers to access the System Timer – SysTick as shown in Table 4:

Table 4. System Timer Registers

Name	Address	Description
SysTick Control and Status	0xE000E010	Basic control of SysTick, e.g. enable, clock source, interrupt or poll
SysTick Reload Value	0xE000E014	Value to load Current Value register when 0 is reached
SysTick Current Value	0xE000E018	The current value of the count down
SysTick Calibration Value	0xE000E01C	Contains the number of ticks to generate a 10ms interval and other information, depending on the implementation

The SysTick can be polled to obtain current tick count or can be configured to generate an interrupt which is handled by the SysTick_Handler.

To Configure the SysTick to raise an interrupt every 1000k system clock cycles, we need to do the following:

1. The SysTick Reload Value register needs to be loaded with the interval required between SysTick events. The COUNTFLAG bit in the SysTick Control and Status register is set when the timer count value transitions from 1 to 0. Therefore the SysTick interrupt activates every n+1 clock ticks. For example, if a period of 1000 is required, 999 should be written to the SysTick Reload Value register. The SysTick Reload Value register supports values between 1 and 0x00FFFFFF.
2. The CLKSOURCE bit of the SysTick Control and status Register needs to be set to use the core clock. It should be cleared to select an external clock source as reference.
3. The TICKINT bit of the SysTick Control and status Register needs to be set to generate an interrupt whenever the count has reached 0.
4. Finally the SysTick Timer is enabled by setting the ENABLE bit of the SysTick Control and status Register.
5. The CURRENT bit of the SysTick Current Value Register can be cleared to restart the timer. This will automatically clear the COUNTFLAG bit in the SysTick Control and Status register.

TIPS

Fortunately, the core_cm0plus.h implements a function that does all of the above configurations to the SysTick Registers. It only requires as a parameter the time interval for interrupt to occur in system clock cycles.

CHECKPOINT

- Were you able to implement EIC based interrupt to toggle an LED?
- Are you familiar with CMSIS software that allows access to Core registers and peripherals (covered in Section 6.2.3)?[3]

[3] You may also want to refer the following site for information on CMSIS: http://www.vr.ncue.edu.tw/esa/b1011/CMSIS_Core.htm

- Are you familiar with the core_cm0plus.h header file which defines some Core related Control Registers (such as NVIC Register, SysTick Register, etc.) and Core related function definition/implementation such as NVIC_EnableIRQ(), SysTick_Config(), etc...?

TO DO

Your task is to write a function that will correctly configure the SysTick register in order to raise an interrupt every 1000K system clock source cycles. Also write the corresponding interrupt handler that toggles LED0 whenever SW0 is pushed.

8. Conclusion

In this section we studied concepts of interrupts and ISRs. We are familiar with the ARM NVIC and registers available to control the NVIC. We have covered the External Interrupt Controller in SAMD20 in detail. We are now familiar with the following:

1. How interrupts are supported by the ARM Cortex M0+ microcontroller.
2. The external interrupt control (EIC) registers available to configure and use the SAMD20 EIC.
3. We used the EIC to detect button press and toggle LED. Therefore we know the actual implementation of EIC including port control, EIC register settings and Global NVIC register settings. We also know how to write an ISR.
4. We also used the SysTick interrupt so we are now familiar with the Systick register settings.
5. We also know the difference between polling and interrupt technique, including which is better and why.

Atmel | Enabling Unlimited Possibilities®

Atmel Corporation
1600 Technology Drive
San Jose, CA 95110
USA
Tel: (+1)(408) 441-0311
Fax: (+1)(408) 487-2600
www.atmel.com

Atmel Asia Limited
Unit 01-5 & 16, 19F
BEA Tower, Millennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
HONG KONG
Tel: (+852) 2245-6100
Fax: (+852) 2722-1369

Atmel Munich GmbH
Business Campus
Parkring 4
D-85748 Garching b. Munich
GERMANY
Tel: (+49) 89-31970-0
Fax: (+49) 89-3194621

Atmel Japan G.K.
16F Shin-Osaki Kangyo Bldg.
1-6-4 Osaki, Shinagawa-ku
Tokyo 141-0032
JAPAN
Tel: (+81)(3) 6417-0300
Fax: (+81)(3) 6417-0370