ECE 2534

Introduction to

Interrupts

Taking care of peripherals

- By default, a CPU needs to **poll** its peripherals to find out when they need attention

- Example: a timer rolls over
- Example: an input buffer is full
- Example: input pin X on an I/O port has changed

Taking care of peripherals

- By default, a CPU needs to **poll** its peripherals to find out when they need attention

```
while (1)
{
        // read inputs
        if (periph1 needs attention)
                … // take care of periph1
        if (periph2 needs attention)
                … // take care of periph2
        . . .
}
```

- *Problem*: CPU could be doing other, useful things, if no peripherals need attention
- *Problem*: CPU could miss an important event on periph1, while servicing periph2

Interrupts

- ❑ Most modern processors allow peripherals to **interrupt** the normal sequence of instruction execution

- ❑ Example: a timer rolls over
  → the timer hardware generates an “interrupt request”
- ❑ Example: an input buffer is full
  → the I/O hardware generates an “interrupt request”

- ❑ In response to an interrupt request, the processor executes an **Interrupt Service Routine** (ISR), also called an **Interrupt Handler**

A simple interrupt handler (from Lab 1)

```c
unsigned sec1000;              // global variable

// Interrupt handler - respond to timer-generated interrupt
#pragma interrupt InterruptHandler_2534 ipl1 vector 0
void InterruptHandler_2534( void )
{
   if( INTGetFlag(INT_T2) )   // Verify source of interrupt
   {
      sec1000++;               // Body of interrupt handler
      INTClearFlag(INT_T2);   // Acknowledge interrupt
   }
}
```

Recall the “fetch-execute cycle” for a CPU

- REPEAT:
  - FETCH the next instruction
  - Advance the Program Counter (PC)
  - DECODE the just-fetched instruction
  - EXECUTE the just-decoded instruction

- This usual sequence can be altered:
  - *Before* fetching the next instruction,
    if interrupts have been enabled, the CPU’s hardware checks to see if a peripheral has raised an interrupt request
  - If so, the CPU’s hardware takes special actions that involve saving and changing the PC

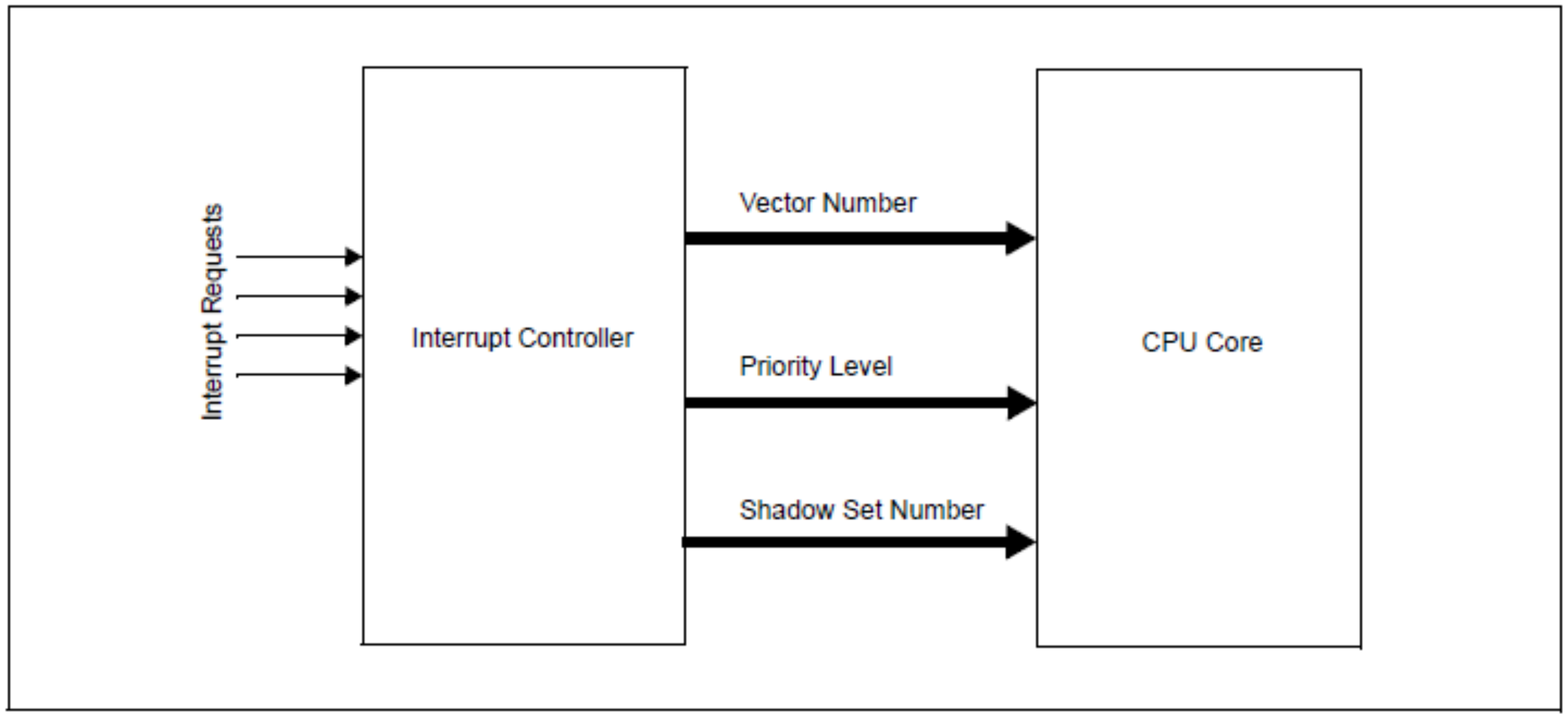
Usual sequence of events

1. Processor is executing instructions in some function F
2. A peripheral raises an interrupt request
3. Processor completes the current machine instruction and saves the current "context"
   - Context usually refers to the current CPU register values
   - The program counter (PC), in particular, will be saved for later
4. The address of the relevant ISR is written to the PC (the address is sometimes called an "interrupt vector")
5. Processor executes the ISR, which is like a subroutine; its purpose is to take care of the peripheral device that generated the interrupt
6. When the ISR exits, the saved processor context is reloaded
7. Re-loading the (previously saved) PC value causes the processor to resume normal execution in function F, continuing where it left off before being interrupted

The interrupt controller

- ❑ The interrupt mechanism of a CPU requires some nontrivial hardware
- ❑ The PIC32 contains a hardware subsystem called the interrupt controller, which takes input from interrupt sources, resolves priority issues, and keeps track of interrupt vectors

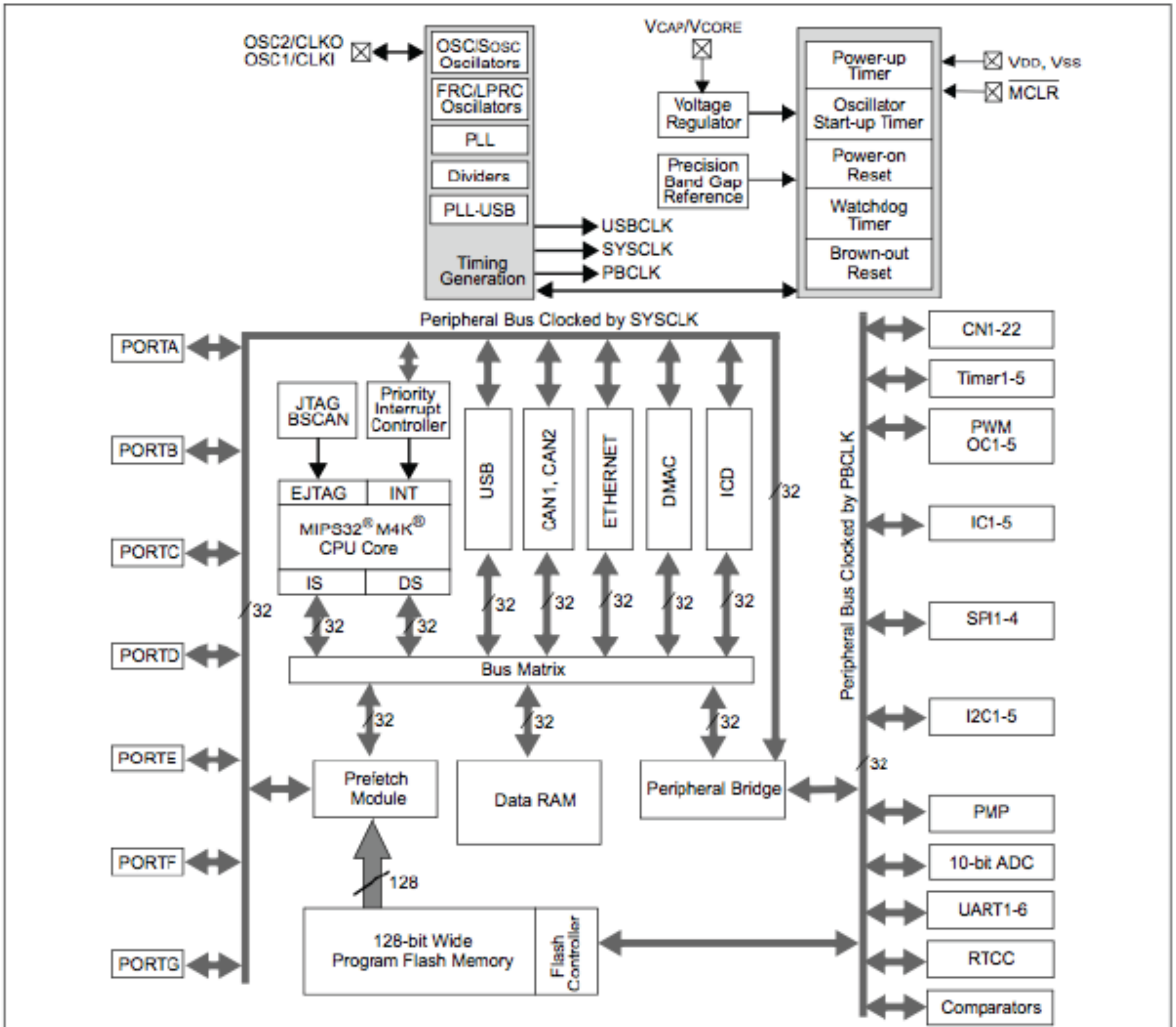
FIGURE 7-1: INTERRUPT CONTROLLER MODULE



PIC32

(PIC32 datasheet, section 1)

FIGURE 1-1: BLOCK DIAGRAM(1,2)

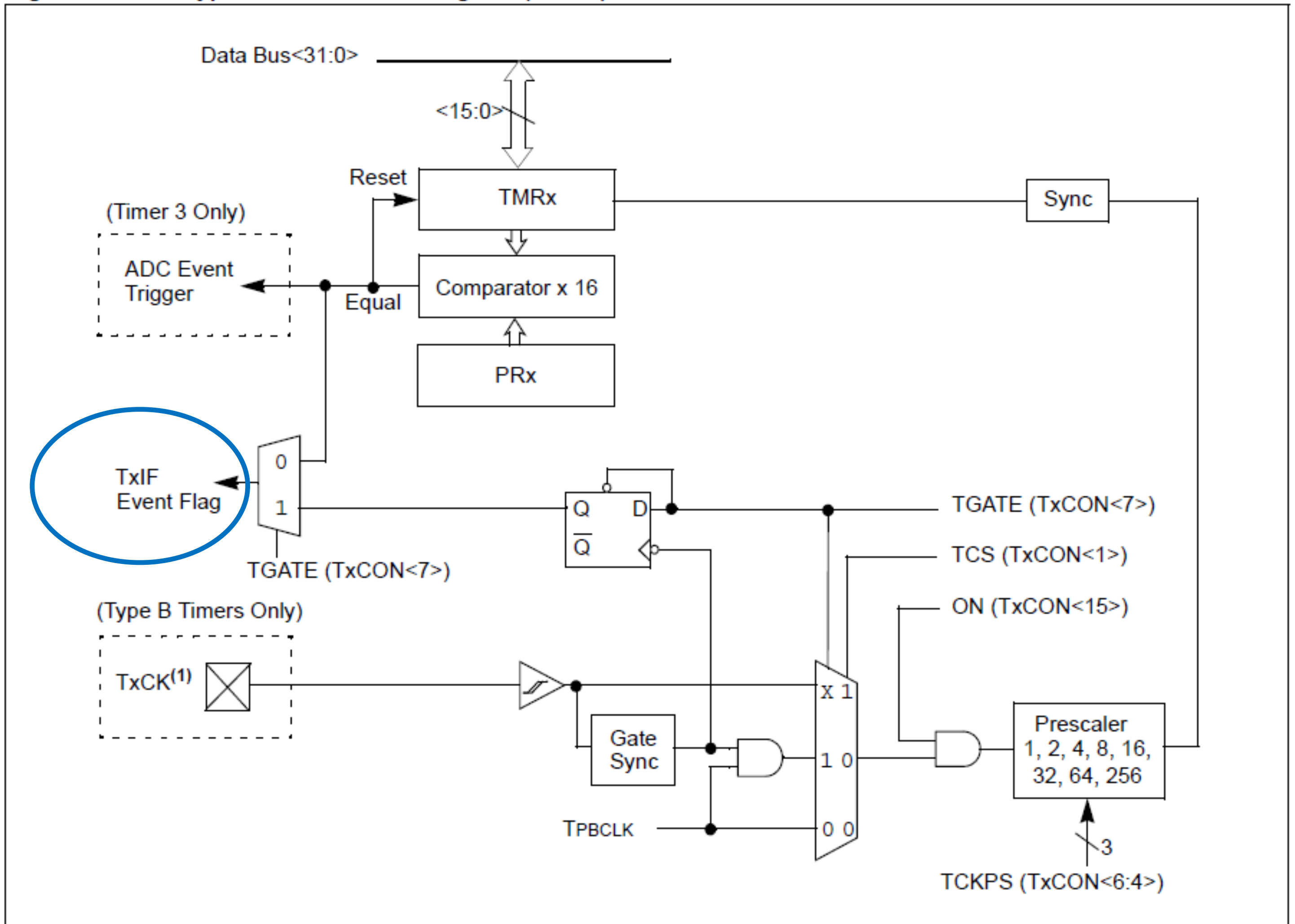


Note 1: Some features are not available on all device variants.

2: BOR functionality is provided when the on-board voltage regulator is enabled.

Timer 2 / 3 / 4 / 5 (16-bit operation)

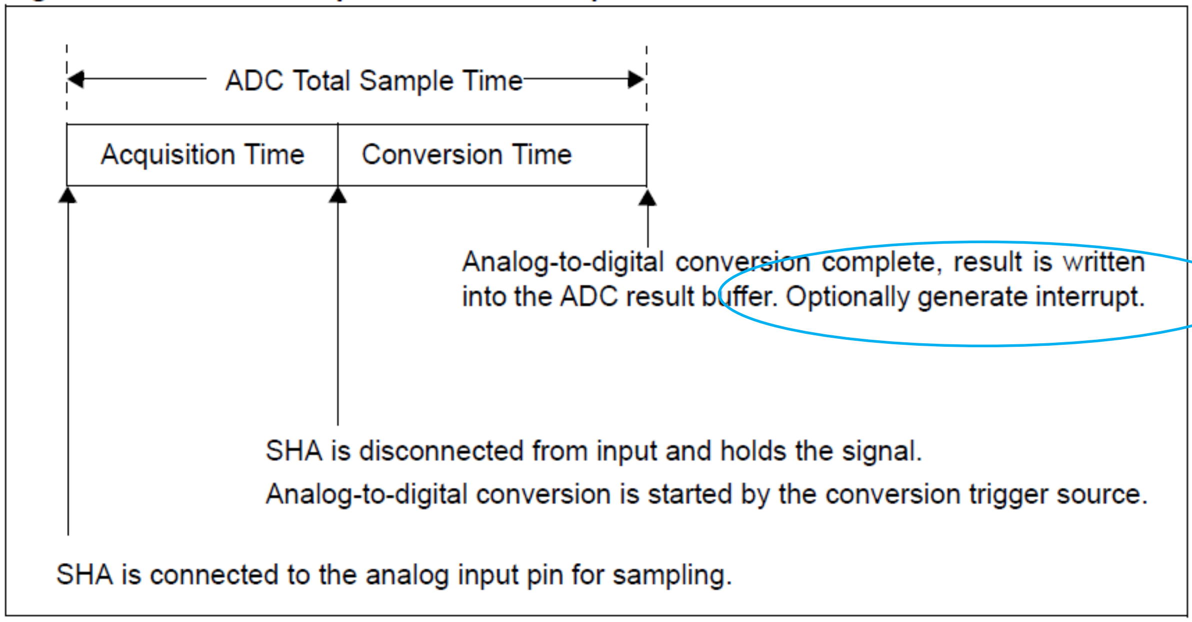
Figure 14-2: Type B Timer Block Diagram (16-Bit)

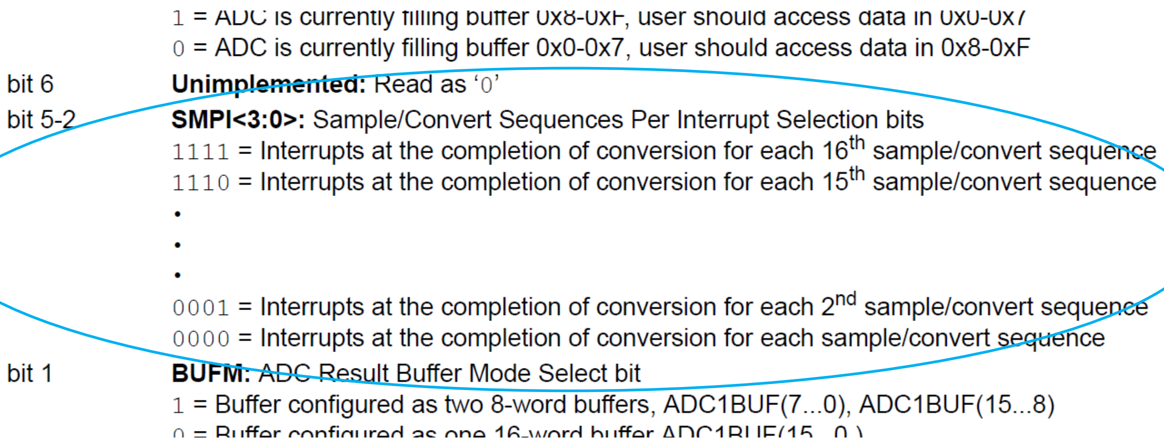


Note 1: The TxCK pin is not available in certain variants of the PIC32MX device family. In such cases, the timer must use the peripheral clock as its input clock. Refer to the specific device data sheet for the I/O pin details.

From the ADC manual

Figure 17-2: ADC Sample/Conversion Sequence


ADC Total Sample Time
Acquisition Time
Conversion Time
Analog-to-digital conversion complete, result is written into the ADC result buffer. Optionally generate interrupt.
SHA is disconnected from input and holds the signal.
Analog-to-digital conversion is started by the conversion trigger source.
SHA is connected to the analog input pin for sampling.



1 = ADC is currently filling buffer 0x8-0xF, user should access data in 0x0-0x7
0 = ADC is currently filling buffer 0x0-0x7, user should access data in 0x8-0xF

bit 6 **Unimplemented:** Read as '0'

bit 5-2 **SMPI<3:0>:** Sample/Convert Sequences Per Interrupt Selection bits
1111 = Interrupts at the completion of conversion for each 16th sample/convert sequence
1110 = Interrupts at the completion of conversion for each 15th sample/convert sequence
•
•
•
0001 = Interrupts at the completion of conversion for each 2nd sample/convert sequence
0000 = Interrupts at the completion of conversion for each sample/convert sequence

bit 1 **BUFM:** ADC Result Buffer Mode Select bit
1 = Buffer configured as two 8-word buffers, ADC1BUF(7...0), ADC1BUF(15...8)
0 = Buffer configured as one 16-word buffer ADC1BUF(15...0.)

The interrupt controller

- Within the interrupt controller,
each interrupt source is associated with an Interrupt Enable (IE) bit and an Interrupt Flag (IF) bit
  - At power-up, all **IE** bits are cleared to 0
  - Whenever an **IE** bit is 0, the interrupt controller will ignore interrupt requests from the associated device
  - Whenever an **IE** bit is set to 1, the interrupt controller will typically interrupt the CPU whenever an interrupt request is generated by that device
  - If an **IF** bit is set to 1, it effectively means that the associated source has issued an interrupt request
(your code could use this flag to determine which device, among several, has raised an interrupt request)
  - The ISR needs to “clear” the **IF** bit (change it to 0) to allow detection of further interrupts

How to enable interrupts

- The initialization procedure is different on every type of processor, and for different peripheral devices

- See the following example for the PIC32 . . .

Initialization for interrupt-driven operation (from Lab 1)

```
// Set up timer 2 to roll over every ms
OpenTimer2(T2_ON          |
           T2_IDLE_CON    |
           T2_SOURCE_INT  |
           T2_PS_1_16     |
           T2_GATE_OFF,
           624);  // freq = 10MHz/16/625 = 1 kHz

// Set up CPU to respond to interrupts from Timer2
INTSetVectorPriority(INT_TIMER_2_VECTOR, INT_PRIORITY_LEVEL_1);
INTClearFlag(INT_T2);
INTEnable(INT_T2, INT_ENABLED);
INTConfigureSystem(INT_SYSTEM_CONFIG_SINGLE_VECTOR);
INTEnableInterrupts();
```

Interrupt sources

- There are many potential sources of interrupts
(The PIC32 architecture supports up to ~~96~~ 256 possible interrupt sources!)
- Internal sources
  - Timer
  - UART
  - ADC (analog-to-digital converter)
  - DMA (direct memory access) controller
  - . . .
  - Software – special instructions can cause interrupts
- External sources
  - Level-change activity on I/O port pins
  - . . .

- Potential interrupt sources for the PIC32MX

(From the PIC32MX5XX/6XX/7XX data sheet)

- Interrupt vector table
  = a list of addresses of ISRs
- “Vector number”
  = an index into that table

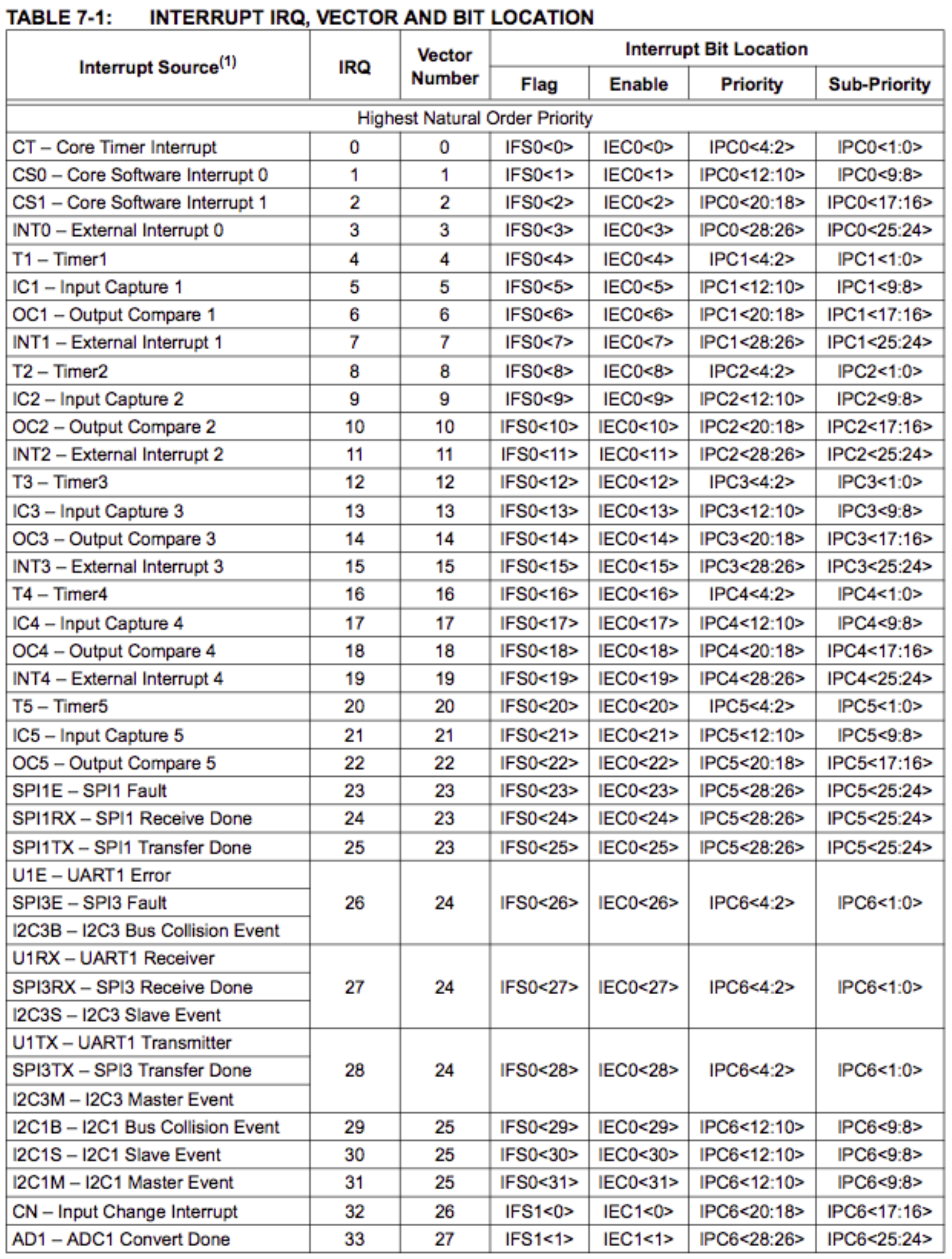


TABLE 7-1: INTERRUPT IRQ, VECTOR AND BIT LOCATION

Interrupt Source[1]	IRQ	Vector Number	Interrupt Bit Location			
			Flag	Enable	Priority	Sub-Priority
Highest Natural Order Priority						
CT – Core Timer Interrupt	0	0	IFS0<0>	IEC0<0>	IPC0<4:2>	IPC0<1:0>
CS0 – Core Software Interrupt 0	1	1	IFS0<1>	IEC0<1>	IPC0<12:10>	IPC0<9:8>
CS1 – Core Software Interrupt 1	2	2	IFS0<2>	IEC0<2>	IPC0<20:18>	IPC0<17:16>
INT0 – External Interrupt 0	3	3	IFS0<3>	IEC0<3>	IPC0<28:26>	IPC0<25:24>
T1 – Timer1	4	4	IFS0<4>	IEC0<4>	IPC1<4:2>	IPC1<1:0>
IC1 – Input Capture 1	5	5	IFS0<5>	IEC0<5>	IPC1<12:10>	IPC1<9:8>
OC1 – Output Compare 1	6	6	IFS0<6>	IEC0<6>	IPC1<20:18>	IPC1<17:16>
INT1 – External Interrupt 1	7	7	IFS0<7>	IEC0<7>	IPC1<28:26>	IPC1<25:24>
T2 – Timer2	8	8	IFS0<8>	IEC0<8>	IPC2<4:2>	IPC2<1:0>
IC2 – Input Capture 2	9	9	IFS0<9>	IEC0<9>	IPC2<12:10>	IPC2<9:8>
OC2 – Output Compare 2	10	10	IFS0<10>	IEC0<10>	IPC2<20:18>	IPC2<17:16>
INT2 – External Interrupt 2	11	11	IFS0<11>	IEC0<11>	IPC2<28:26>	IPC2<25:24>
T3 – Timer3	12	12	IFS0<12>	IEC0<12>	IPC3<4:2>	IPC3<1:0>
IC3 – Input Capture 3	13	13	IFS0<13>	IEC0<13>	IPC3<12:10>	IPC3<9:8>
OC3 – Output Compare 3	14	14	IFS0<14>	IEC0<14>	IPC3<20:18>	IPC3<17:16>
INT3 – External Interrupt 3	15	15	IFS0<15>	IEC0<15>	IPC3<28:26>	IPC3<25:24>
T4 – Timer4	16	16	IFS0<16>	IEC0<16>	IPC4<4:2>	IPC4<1:0>
IC4 – Input Capture 4	17	17	IFS0<17>	IEC0<17>	IPC4<12:10>	IPC4<9:8>
OC4 – Output Compare 4	18	18	IFS0<18>	IEC0<18>	IPC4<20:18>	IPC4<17:16>
INT4 – External Interrupt 4	19	19	IFS0<19>	IEC0<19>	IPC4<28:26>	IPC4<25:24>
T5 – Timer5	20	20	IFS0<20>	IEC0<20>	IPC5<4:2>	IPC5<1:0>
IC5 – Input Capture 5	21	21	IFS0<21>	IEC0<21>	IPC5<12:10>	IPC5<9:8>
OC5 – Output Compare 5	22	22	IFS0<22>	IEC0<22>	IPC5<20:18>	IPC5<17:16>
SPI1E – SPI1 Fault	23	23	IFS0<23>	IEC0<23>	IPC5<28:26>	IPC5<25:24>
SPI1RX – SPI1 Receive Done	24	23	IFS0<24>	IEC0<24>	IPC5<28:26>	IPC5<25:24>
SPI1TX – SPI1 Transfer Done	25	23	IFS0<25>	IEC0<25>	IPC5<28:26>	IPC5<25:24>
U1E – UART1 Error	26	24	IFS0<26>	IEC0<26>	IPC6<4:2>	IPC6<1:0>
SPI3E – SPI3 Fault						
I2C3B – I2C3 Bus Collision Event						
U1RX – UART1 Receiver	27	24	IFS0<27>	IEC0<27>	IPC6<4:2>	IPC6<1:0>
SPI3RX – SPI3 Receive Done						
I2C3S – I2C3 Slave Event						
U1TX – UART1 Transmitter	28	24	IFS0<28>	IEC0<28>	IPC6<4:2>	IPC6<1:0>
SPI3TX – SPI3 Transfer Done						
I2C3M – I2C3 Master Event						
I2C1B – I2C1 Bus Collision Event	29	25	IFS0<29>	IEC0<29>	IPC6<12:10>	IPC6<9:8>
I2C1S – I2C1 Slave Event	30	25	IFS0<30>	IEC0<30>	IPC6<12:10>	IPC6<9:8>
I2C1M – I2C1 Master Event	31	25	IFS0<31>	IEC0<31>	IPC6<12:10>	IPC6<9:8>
CN – Input Change Interrupt	32	26	IFS1<0>	IEC1<0>	IPC6<20:18>	IPC6<17:16>
AD1 – ADC1 Convert Done	33	27	IFS1<1>	IEC1<1>	IPC6<28:26>	IPC6<25:24>

TABLE 7-1: INTERRUPT IRQ, VECTOR AND BIT LOCATION (CONTINUED)

Interrupt Source[(1)]	IRQ	Vector Number	Interrupt Bit Location			
			Flag	Enable	Priority	Sub-Priority
PMP – Parallel Master Port	34	28	IFS1<2>	IEC1<2>	IPC7<4:2>	IPC7<1:0>
CMP1 – Comparator Interrupt	35	29	IFS1<3>	IEC1<3>	IPC7<12:10>	IPC7<9:8>
CMP2 – Comparator Interrupt	36	30	IFS1<4>	IEC1<4>	IPC7<20:18>	IPC7<17:16>
U3E – UART2A Error SPI2E – SPI2 Fault I2C4B – I2C4 Bus Collision Event	37	31	IFS1<5>	IEC1<5>	IPC7<28:26>	IPC7<25:24>
U3RX – UART2A Receiver SPI2RX – SPI2 Receive Done I2C4S – I2C4 Slave Event	38	31	IFS1<6>	IEC1<6>	IPC7<28:26>	IPC7<25:24>
U3TX – UART2A Transmitter SPI2TX – SPI2 Transfer Done IC4M – I2C4 Master Event	39	31	IFS1<7>	IEC1<7>	IPC7<28:26>	IPC7<25:24>
U2E – UART3A Error SPI4E – SPI4 Fault I2C5B – I2C5 Bus Collision Event	40	32	IFS1<8>	IEC1<8>	IPC8<4:2>	IPC8<1:0>
U2RX – UART3A Receiver SPI4RX – SPI4 Receive Done I2C5S – I2C5 Slave Event	41	32	IFS1<9>	IEC1<9>	IPC8<4:2>	IPC8<1:0>
U2TX – UART3A Transmitter SPI4TX – SPI4 Transfer Done IC5M – I2C5 Master Event	42	32	IFS1<10>	IEC1<10>	IPC8<4:2>	IPC8<1:0>
I2C2B – I2C2 Bus Collision Event	43	33	IFS1<11>	IEC1<11>	IPC8<12:10>	IPC8<9:8>
I2C2S – I2C2 Slave Event	44	33	IFS1<12>	IEC1<12>	IPC8<12:10>	IPC8<9:8>
I2C2M – I2C2 Master Event	45	33	IFS1<13>	IEC1<13>	IPC8<12:10>	IPC8<9:8>
FSCM – Fail-Safe Clock Monitor	46	34	IFS1<14>	IEC1<14>	IPC8<20:18>	IPC8<17:16>
RTCC – Real-Time Clock and Calendar	47	35	IFS1<15>	IEC1<15>	IPC8<28:26>	IPC8<25:24>
DMA0 – DMA Channel 0	48	36	IFS1<16>	IEC1<16>	IPC9<4:2>	IPC9<1:0>
DMA1 – DMA Channel 1	49	37	IFS1<17>	IEC1<17>	IPC9<12:10>	IPC9<9:8>
DMA2 – DMA Channel 2	50	38	IFS1<18>	IEC1<18>	IPC9<20:18>	IPC9<17:16>
DMA3 – DMA Channel 3	51	39	IFS1<19>	IEC1<19>	IPC9<28:26>	IPC9<25:24>
DMA4 – DMA Channel 4	52	40	IFS1<20>	IEC1<20>	IPC10<4:2>	IPC10<1:0>
DMA5 – DMA Channel 5	53	41	IFS1<21>	IEC1<21>	IPC10<12:10>	IPC10<9:8>
DMA6 – DMA Channel 6	54	42	IFS1<22>	IEC1<22>	IPC10<20:18>	IPC10<17:16>
DMA7 – DMA Channel 7	55	43	IFS1<23>	IEC1<23>	IPC10<28:26>	IPC10<25:24>
FCE – Flash Control Event	56	44	IFS1<24>	IEC1<24>	IPC11<4:2>	IPC11<1:0>
USB – USB Interrupt	57	45	IFS1<25>	IEC1<25>	IPC11<12:10>	IPC11<9:8>
CAN1 – Control Area Network 1	58	46	IFS1<26>	IEC1<26>	IPC11<20:18>	IPC11<17:16>
CAN2 – Control Area Network 2	59	47	IFS1<27>	IEC1<27>	IPC11<28:26>	IPC11<25:24>
ETH – Ethernet Interrupt	60	48	IFS1<28>	IEC1<28>	IPC12<4:2>	IPC12<1:0>
IC1E – Input Capture 1 Error	61	5	IFS1<29>	IEC1<29>	IPC1<12:10>	IPC1<9:8>
IC2E – Input Capture 2 Error	62	9	IFS1<30>	IEC1<30>	IPC2<12:10>	IPC2<9:8>
IC3E – Input Capture 3 Error	63	13	IFS1<31>	IEC1<31>	IPC3<12:10>	IPC3<9:8>
IC4E – Input Capture 4 Error	64	17	IFS2<0>	IEC2<0>	IPC4<12:10>	IPC4<9:8>

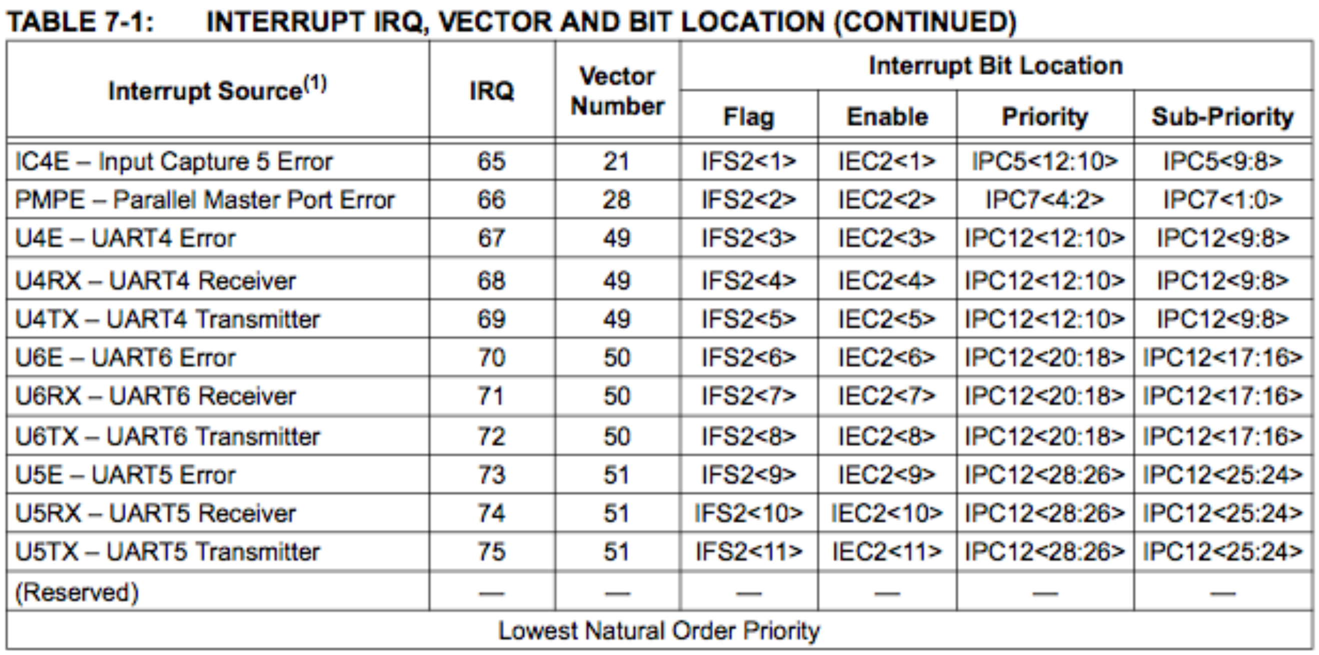


TABLE 7-1: INTERRUPT IRQ, VECTOR AND BIT LOCATION (CONTINUED)

Interrupt Source[(1)]	IRQ	Vector Number	Interrupt Bit Location			
			Flag	Enable	Priority	Sub-Priority
IC4E – Input Capture 5 Error	65	21	IFS2<1>	IEC2<1>	IPC5<12:10>	IPC5<9:8>
PMPE – Parallel Master Port Error	66	28	IFS2<2>	IEC2<2>	IPC7<4:2>	IPC7<1:0>
U4E – UART4 Error	67	49	IFS2<3>	IEC2<3>	IPC12<12:10>	IPC12<9:8>
U4RX – UART4 Receiver	68	49	IFS2<4>	IEC2<4>	IPC12<12:10>	IPC12<9:8>
U4TX – UART4 Transmitter	69	49	IFS2<5>	IEC2<5>	IPC12<12:10>	IPC12<9:8>
U6E – UART6 Error	70	50	IFS2<6>	IEC2<6>	IPC12<20:18>	IPC12<17:16>
U6RX – UART6 Receiver	71	50	IFS2<7>	IEC2<7>	IPC12<20:18>	IPC12<17:16>
U6TX – UART6 Transmitter	72	50	IFS2<8>	IEC2<8>	IPC12<20:18>	IPC12<17:16>
U5E – UART5 Error	73	51	IFS2<9>	IEC2<9>	IPC12<28:26>	IPC12<25:24>
U5RX – UART5 Receiver	74	51	IFS2<10>	IEC2<10>	IPC12<28:26>	IPC12<25:24>
U5TX – UART5 Transmitter	75	51	IFS2<11>	IEC2<11>	IPC12<28:26>	IPC12<25:24>
(Reserved)	—	—	—	—	—	—
Lowest Natural Order Priority						

- Some observations:
  - Up to 96 IRQs (interrupt sources)
  - At most 64 vector table entries
  - More IRQs than vectors, so some IRQs share the same vector table entries

Interrupt characteristics

- Interrupt vector
  - = starting address of first instruction of the ISR for a particular interrupt source
  - PIC32 supports up to 64 interrupt vectors
- Priority
  - PIC32 provides priority levels 1 (lowest) to 7 (highest)
  - Whenever two interrupts occur simultaneously, the one with the higher priority will be processed first
  - Priority level 0: ignore the interrupt
  - By default, the priority level of a normal CPU program is 1; when an ISR of higher priority is processed, the priority level of the CPU increases
- Subpriority
  - PIC32 provides subpriority levels 0 to 3
  - Within a given priority level, higher subpriority levels are handled first

- The PIC32 works in 2 different “modes”:

  - Single-vector mode
    Only one vector (address) is used.
    All interrupt sources will use the same interrupt vector.

  - Multi-vector mode
    Different interrupt sources can map to different interrupt vectors

- Notice that different interrupt sources can share the same vector, and have the same priority level
→ The ISR will need to figure out which source caused the interrupt

Interrupt latency

- Interrupt latency is the delay from the time an interrupt request is issued to the time that the CPU begins to execute the ISR

- Notice that lower priority can mean longer latency

Control Registers

The Interrupts module consists of the following Special Function Registers (SFRs):

- **INTCON: Interrupt Control Register**
- **PRISS: Priority Shadow Select Register**
- **INTSTAT: Interrupt Status Register**
- **IPTMR: Interrupt Proximity Timer Register**
- **IFSx: Interrupt Flag Status Register** ← Has device X requested an interrupt?
- **IECx: Interrupt Enable Control Register** ← Is device X allowed to request an interrupt?
- **IPCx: Interrupt Priority Control Register**
- **OFFx: Interrupt Vector Address Offset Register**

Shadow Registers – *low-level detail*

- The PIC32 family of devices employs two register sets:
  - a primary register set for normal program execution, and
  - a shadow register set for highest priority interrupt processing.
- Register set selection is automatically performed by the interrupt controller, based on the mode of operation
- Example: In Single Vector mode, the SS0 bit (INTCON<16> or PRISS<0>) determines which register set to be used.
  - If SS0 bit is set to ‘1’, the interrupt controller will instruct the CPU to use a shadow register set for all interrupts.
  - If SS0 bit is set to ‘0’, the interrupt controller will instruct the CPU to use only the first register set.

Writing ISRs in C

- An ISR resembles a C function, with the following differences
  - It accepts no arguments
  - No return value
  - We use a pragma in the source code to tell the compiler that it is an ISR

```
#pragma interrupt InterruptHandler_2534 ipl1 vector 0
void InterruptHandler_2534( void )
{
   if( INTGetFlag(INT_T2) )
   {
      sec1000++;
      INTClearFlag(INT_T2);
   }
}
```

Verify source of interrupt

Body of interrupt handler
(ISRs should execute quickly)

Acknowledge (clear) the source that generated the interrupt

Programming with ISRs – single-vector case

- In single-vector interrupt mode, all interrupt sources are “wired” to the same ISR
- The PLIB contains several functions to help the programmer

  .../pic32mx/include/peripheral/int.h

  .../pic32mx/include/peripheral/int_5xx_6xx_7xx.h

```
INTGetFlag()
INTClearFlag()

INTSetVectorPriority()
INTClearFlag()
INTEnable()
INTConfigureSystem()
INTEnableInterrupts()
```

Setting priority for a particular peripheral

```
void INTSetVectorPriority( INT_VECTOR vector, INT_PRIORITY priority)

INT_T2,                    // Timer2
INT_U1RX,                  // UART1 RX Event
INT_U1TX,                  // UART1 TX Event
INT_AD1,                   // ADC1 Convert Done
. . .

          INT_PRIORITY_LEVEL_7   ← highest priority
          . . .
          INT_PRIORITY_LEVEL_1   ← lowest priority
```

Enabling interrupts from a particular peripheral

```
void  INTEnable( INT_SOURCE source, INT_EN_DIS enable)

INT_T2,                  // Timer2
INT_U1RX,                // UART1 RX Event
INT_U1TX,                // UART1 TX Event
INT_AD1,                 // ADC1 Convert Done
. . .

                         INT_ENABLED
                         INT_DISABLED
```

Enabling interrupts at the system level

A program must call functions similar to these before any interrupts will be handled.

```
void   INTConfigureSystem( INT_SYSTEM_CONFIG  config)

                 INT_SYSTEM_CONFIG_SINGLE_VECTOR
                 INT_SYSTEM_CONFIG_MULT_VECTOR


unsigned int   INTEnableInterrupts(void)
```

Summary

- Polling vs. interrupt-driven code
- Interrupt Service Handler (ISR) = Interrupt Handler
- An ISR closely resembles a subroutine,
  but it is “called” by a hardware mechanism