Basic Communication with Peripheral Devices

Prepared by Sean Collins
September 2018

Peripherals

Peripheral devices are specialized hardware modules that extend the processor's capabilities. For example, some peripherals provide circuitry for enabling the processor to communicate with the outside world. Additionally, there are many other auxiliary functions provided by peripherals, such as measuring time.

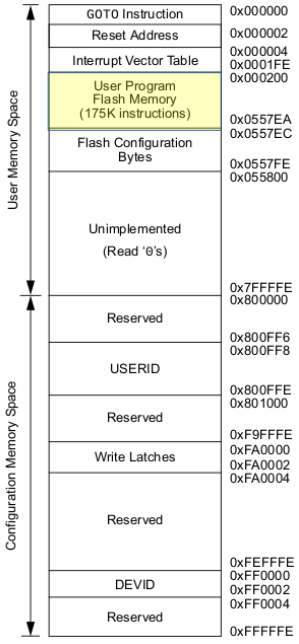
For cost efficiency, embedded processors are typically designed to serve the needs of a particular problem domain. Embedded systems are used in a wide variety of problem domains, so there is a correspondingly wide variety of processor designs on the market. Each of these designs may offer different sets of peripherals. Therefore, in order to figure out which peripherals a processor includes, you must consult its datasheet.

Processor/Peripheral Interface

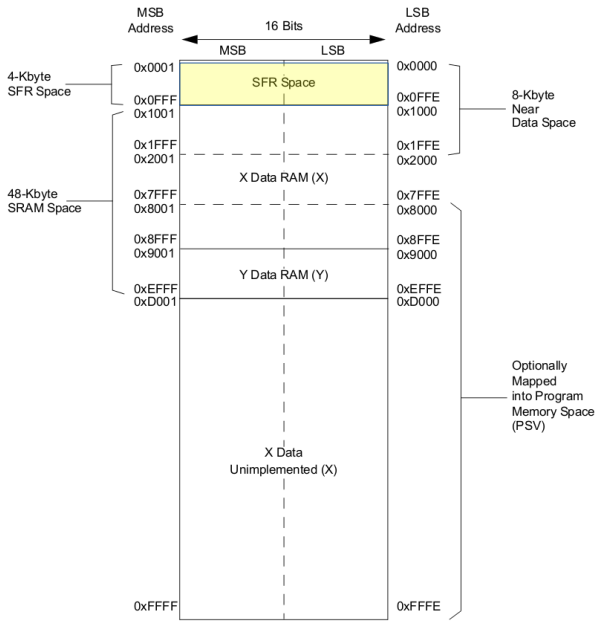
For the convenience of programmers and hardware designers, peripherals are often memory-mapped, which means that each peripheral has registers that are associated with addresses in the processor's memory. Consequently, a processor can read and write data to its peripherals as though it were reading or writing to memory locations. To implement memory-mapped peripherals, processor designers must reserve portions of the processor's address space for special registers, called Control and Status Registers. Microchip refers to this section in memory as the Special Function Register (SFR) space.

Memory Map

Every processor has a memory map, which specifies how memory is divided into sections and what the purpose of each of section is. From a big picture point of view, Microchip divides the memory for the dsPIC33E architecture into two spaces: a "Program Address Space" and a "Data Address Space." Our code's instructions, for example, are stored in a sub-section of the Program Address Space. Special Function Registers, on the other hand, are stored in a sub-section of the Data Address Space. Microchip provides detailed diagrams of its memory maps (like the ones below) in a particular section of the datasheet entitled **Memory Organization**.



Program Memory Space



Data Memory Space

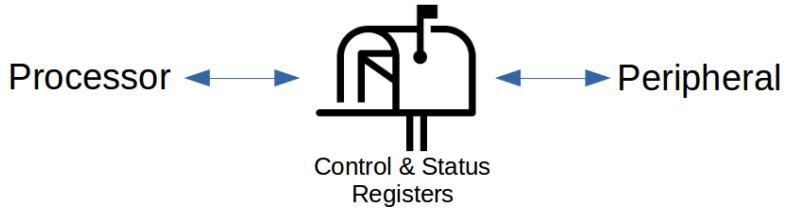
Control and Status Registers

Control and Status Registers, which serve as the interfaces between a processor and its peripherals, are documented in Microchips datasheets in a sub-section of the "Memory Organization" section, called "Special Function Register Maps." This section lists all of the registers associated with peripherals, the addresses of these registers, and the meanings of these registers. It also indicates the meanings of the individual bits within each register.

Note: the register maps contain many cryptic acronyms, and the meanings of these acronyms are not necessarily clear; however, you can determine the meanings of these acronyms by consulting the section of the datasheet corresponding to the peripheral with which you are working.

- **Control Registers** are memory locations where we can pass commands to the hardware by writing data.
- **Status Registers** are memory locations which we can read from in order to query the state of the hardware.

Communicating with Peripherals



Processor/Peripheral Interface

We have learned that the Control and Status Registers serve as tiny "mailboxes," where the peripheral hardware can deposit messages to be read later on by the processor, or vice versa. But how does the processor know when there is a message available? There are two basic strategies for addressing this problem: polling and interrupts.

In each of these techniques, the basic procedure is similar: the processor commands the peripheral to perform some task (by writing to a control register) and then waits for the peripheral to complete. The difference, however, lies it what the processor is doing during the waiting period.

Polling

"Polling" is a technique in which the processor repeatedly checks the status of some flag or register in order to see whether a task has been completed. Let's look at a simple example of a polling strategy. In the following code, suppose that the variable "status_register" contains the status of a task that was assigned to some peripheral. When the peripheral completes its task, let's assume that it will write a value of 1 to the register. As you can see, the processor will remain in the loop until the peripheral is finished.

```
int main () {
    run_task(); // processor sends command to peripheral

    while (status_register != 1) {
        // keep looping until task is done
    }
    // task is done

    return 0;
}
```

The main disadvantage of polling is that it holds up the processor, preventing it from doing useful work while it's waiting for the peripheral to finish. (Note: Even if we place some useful work inside the body of the while loop, the processor will still waste a lot of time querying the status register.) Polling, however, is sometimes useful for implementing extremely fast responses to messages from peripherals (because the processor is constantly checking for messages).

Interrupts

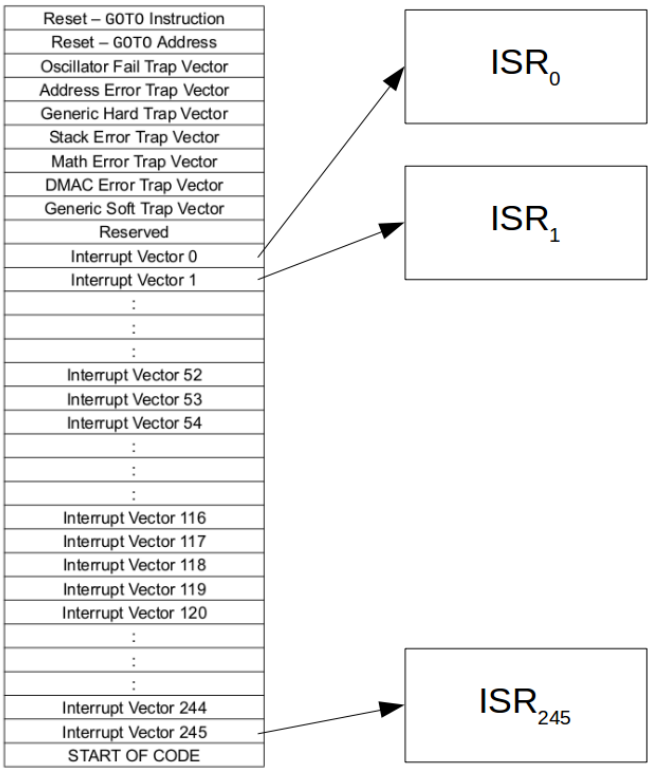
Interrupts – asynchronous, electronic signals sent from peripherals to the processor – provide a second mechanism with which a peripheral can request attention from the processor. Similar to polling, the processor initially sends a command to a peripheral; however, while the peripheral is working, the processor can continue to do useful work. As soon as the peripheral is finished with its task, it sends an interrupt signal to the processor, causing the processor to temporarily pause whatever it is doing in order to respond to the interrupt.

Since the interrupt technique enables the processor and the peripheral to perform work in parallel, it makes much more efficient use of the processor. Interrupts, however, incur some overhead because the processor must save its current state before executing code for responding to an interrupt. Consequently, polling outperforms interrupts when extremely fast response time is required.

Interrupt Service Routines

As programmers, we can control the way that the processor responds to interrupts by writing functions called "Interrupt Service Routines" (ISRs) and associating each of these routines with an interrupt. As soon as the processor detects the interrupt, it temporarily stops whatever it's doing and executes our routine.

Inside of memory, there is some data referred to as the Interrupt Vector Table, which manages associations between interrupts and their ISRs. Each row in the table corresponds to one type of interrupt, and the value in this row is a pointer to whichever ISR is assigned. The "Interrupt Controller" section of the datasheet for the dsPIC33EPGP504 contains the following diagram of the Interrupt Vector Table:



Interrupt Vector Table - Pointers to ISRs

It is up to the programmer to initialize this table by writing Interrupt Service Routines. We write these routines using special syntax and special names, which enable the compiler to distinguish ISRs from regular functions. In the next section, we'll discuss how to write these routines.

Interrupts in MPLABX

Special Syntax for Interrupt Service Routines

In order to define an Interrupt Service Routine, we must attach an "attribute" to a function by placing it between the function's return type and name. This attribute should appear in both the function declaration and the function definition. The purpose of the attribute is to provide the compiler with extra information. Here is an example:

```
// Declaration of "_INT0Interrupt"
void __attribute__ (( interrupt, no_auto_psv )) _INT0Interrupt ();

// Definition of "_INT0Interrupt"
void __attribute__ (( interrupt, no_auto_psv )) _INT0Interrupt ()
{
    // Interrupt-handling code goes here
    // ...
    // Reset the interrupt flag
}
```

You will notice that there are two arguments ("interrupt" and "no_auto_psv") to the attribute in the example above. The "interrupt" argument tells the compiler that we are defining an ISR. The "no_auto_psv" argument tells the compiler that we do not need any "program space visibility." (Program space visibility is useful when our ISR needs to access a bunch of constant data – because constant data typically gets stored in the Program Address Space. So program space visibility is a way to map some of the Program Address Space into the Data Address Space, where it is quickly accessible from an ISR. We often don't need this feature, so we'll tell the compiler not to use it for now.)

Finally, the name of the interrupt service routine is predefined by the compiler. In other words, you can't give your ISR any name you want. The reason that we must use predefined interrupt names is because the compiler needs to know where to store the address of our ISR routine in the Interrupt Vector Table. The predefined names indicate to the compiler which type of interrupt should trigger the given routine. Note: To view all of the interrupt names corresponding to your processor, open the "dashboard" in MPLABX (Window → Dashboard), click on "Compiler Help" (a blue circle with a question mark), and open the "Interrupt Vector Tables Reference."

Resetting the Interrupt Flag

Whenever an interrupt occurs, an interrupt flag corresponding to that interrupt is set to 1. This indicates to the processor that the interrupt has occurred. Consequently, **every interrupt service routine must reset (i.e. "clear") its interrupt flag**. Otherwise, the processor would get stuck handling interrupts forever! You must program this step explicitly by setting the value of the interrupt flag bit to 0. Refer to the Interrupt Controller Register Map – inside the Special Function Register Map section of the datasheet – to see the addresses and positions of the interrupt flag bits.

Fortunately, Microchip's header files define useful bitsets, which we can use to clear the interrupt flag, as in the following example code:

```
void __attribute__ (( interrupt, no_auto_psv )) _INT0Interrupt ()
{
    // Interrupt-handling code goes here
    // ...

    IFS0bits.INT0IF = 0; // uses bitset to clear interrupt flag
}
```

Configuring an Interrupt

Before we can start using a particular interrupt source, we must configure the interrupt. Three of the most important settings that we must initialize are:

1) Global Interrupt Enable

   In order to cause the processor to execute our ISR code, we must enable interrupts both globally and locally. First, to enable interrupts globally, we will set the Global Interrupt Enable bit, which is located in one of the Interrupt Control registers in the Interrupt Controller Register Map. Refer to the Special Function Register Map section of the datasheet for the address of this register. Fortunately, Microchip's headers provide a useful bitset.

2) (Local) Interrupt Enable

   Each particular interrupt also has an Interrupt Enable flag, which is used to turn the interrupt source ON and OFF. If the interrupt source is OFF, then our ISR code will not execute. We must enable interrupts before we can use them. Again, the addresses of all of the interrupt control registers are documented in the Special Function Register Map section of the datasheet. You can also use one of the bitsets provided in Microchip's header files.

3) Interrupt Priority

   Finally, the programmer can assign up to 7 levels of priority to the various sources of interrupts. This basic tutorial will not cover interrupt priority in detail; however, you should know that if multiple interrupts occur simultaneously, the processor will service the interrupt with the highest priority first.

Here's some example code, in which I enable global interrupts and enable the Timer1 interrupt:

```
void configure_interrupts ()
{
    INTCON2bits.GIE = 1; // enable global interrupts
    IEC0bits.T1IE   = 1; // enable Timer1 interrupt
}
```

Summary

Peripherals

- Peripherals are hardware devices that extend the functionality of a processor.
- Processor passes data to/from peripherals via the memory-mapped Control and Status Registers
- There are two main techniques for coordinating processor/peripheral communication:
  - Polling
  - Interrupts

Interrupts

- Interrupts are asynchronous signals passed by external hardware to the processor. They provide a mechanism with which peripheral devices can request the processor's attention, and they are often used to inform the processor that a peripheral has completed some task.
- The Interrupt Vector Table is a data structure in the processor's memory that contains an array of pointers to Interrupt Service Routines (ISRs). Programmers populate this table by explicitly defining ISRs.
- Interrupts must be enabled both individually and also globally in order for the processor to execute their corresponding ISRs
- Programmers can assign priorities to interrupts in order to indicate how to resolve conflicts.