Microcontroller Interrupts

An Online Continuing Education Course for Engineers

Course Number: IC-2010

Credit: 2 Hours / 2 PDH / 2 CPD

Microcontroller Interrupts

Stuart Ball, P.E.

Introduction

This course covers the basic concepts and techniques of using interrupts in microcontrollers. Interrupts provide a means of tailoring microcontroller program execution to real-world events. This course assumes a basic knowledge of computer or microcontroller operation and a basic understanding of software concepts.

Glossary

CPU: Central processing unit, the processing core of the microcontroller.

Compiler/linker: The toolset that accepts source code in C or some other language and produces machine code that can be executed by the CPU.

Machine Code: Binary instructions and data that are stored in program memory and executed by the CPU.

Peripheral: An internal device within the microcontroller that performs some function such as a serial interface, timer, analog-to-digital converter, or another purpose.

Program Memory: The memory from which the CPU gets instructions to be executed. In a microcontroller, this is usually nonvolatile memory (flash, ROM, or similar type of memory).

Program Counter: A register internal to the microcontroller that contains the address of the instruction to be executed. As code execution progresses, the program counter increments to the next address, and can be modified with a new address in response to a program execution or can be replaced with the contents at the top of the stack.

Pseudocode: Informal, English description of an algorithm or routine, not specific to any computer coding language.

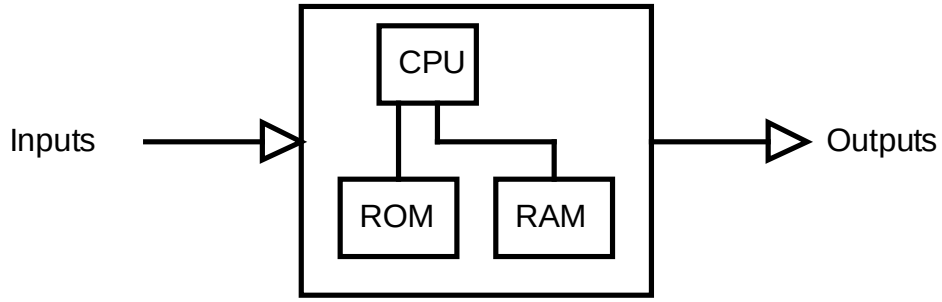
ROM and RAM: ROM is a read-only memory, normally used for storing the program and maybe flash, fixed ROM, or other programmable memory. RAM is random-access memory, is read-write, randomly addressable, and is usually volatile in that the contents are lost when power is removed.

Source Code: The human-readable statements, created by a programmer, that are compiled into machine code by the compiler/linker. For a microcontroller, typically C or assembly statements.

Stack: A region of memory, used as a LIFO (last-in-first-out), and that is dedicated to saving and restoring register values and addresses. The stack usually grows downward in the memory space as items are “pushed” onto it. The current address of the stack is normally maintained in a Stack Pointer register that automatically decrements when a new item is pushed onto the stack.

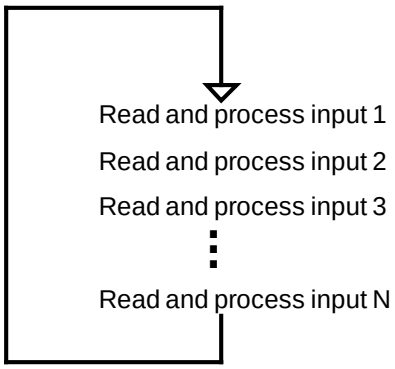
Although the most recent item added to the stack is normally at the lowest address, it is referred to as the "top" of the stack.

Why interrupts: A microcontroller is a basic computer with a CPU, ROM, RAM, and some I/O (input/output) capability. A simple microcontroller block diagram is shown below. This is a simple circuit that has some inputs, does some processing in software, and produces outputs. The inputs might be sensors from an industrial process or data from a control computer or some other kind of sensor or interface.



In this simple example, the software, in ROM, supplies instructions to the CPU, which reads the inputs, do whatever processing is needed, and generates or writes the outputs. The inputs and outputs could be simple signals such as a single digital bit, a complex voltage waveform, a stream of binary bits, or a multi-bit binary word. The inputs could be connected to simple ports that are read by the microcontroller CPU as binary words, or they could be connected to internal microcontroller peripherals such as timers, analog-to-digital converters, or interface peripherals such as UART (universal asynchronous receiver/transmitter) peripherals, or other interfaces such as Ethernet or Bluetooth.

If we assume there are N inputs of various types in this system, one way of implementing the code for this example would be to use a repeating loop, like this:



The software processes each input, one at a time, until it has processed all of them. Then it loops back to the beginning and does it again. But imagine that input 1 sometimes takes a long time to process while input 3 needs immediate attention when something changes. For example, input 1 might be a voltage that represents the light from a sensor, and that it requires a lot of mathematical operations to process the value. But input 3 is data from a high-speed interface that needs to be read and processed quickly. Maybe the processing from input 1 takes so long that some of the data from input 3 are lost.

What is needed is a way to notify the CPU that input 3 needs immediate attention so that it can be given priority and the processing time for input 1 doesn't cause data from input 3 to be lost. We do that with interrupts.

What are interrupts? Interrupts are a way of *interrupting* the normal software execution so that the microcontroller software can service or process an important event. In the example, input 3 might be a signal indicating that a byte was received by some internal peripheral such as a UART and that the microcontroller needs to read the byte and process it in some way. If the UART can't consistently service the UART and read the byte before the next byte is received, then the received byte will be overwritten by the next byte and will be lost. This could happen if input 1 takes too long. It is also possible that the UART doesn't have a way to directly poll the state of the receiver, to detect when a byte has to be read.

In cases like this, an interrupt is normally used. An interrupt is a signal to the microcontroller CPU that whatever is currently being processed must be *interrupted* so as to process something else – in this example to read the byte from the UART. In this situation, we want the CPU to stop what it's doing, read the byte from the UART, do something with it, and then resume whatever it was doing before the interrupt. Typical uses for interrupts include:

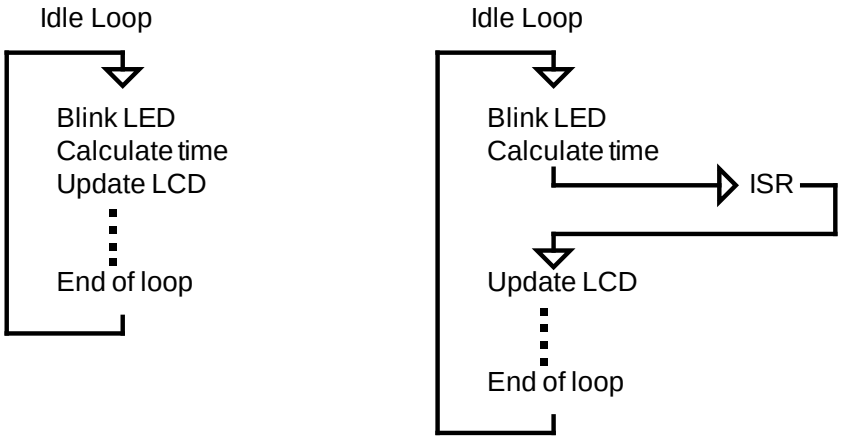
- A regular timer tick for timekeeping or for task switching when using a real-time operating the system.
- Notification of transmit buffer empty or receive buffer full for UARTs, SPI (Serial Peripheral Interface), Ethernet, USB, and other interface peripherals.
- Notification of pin change from input pins which might be detecting zero-crossing of an AC signal or a ready condition from an external device such as a display.

General requirements for interrupts: There are some things that are required for all interrupts, and these are a function of both the hardware and the software:

- Recognize the interrupting event
- Divert processing from the current task or operation to service the event
- After event servicing is complete, restore processing control to the original task or operation in such a way that the interrupted process is not affected by the interrupt

The function that services the interrupt is called the ISR (Interrupt Service Routine). Typically, notification to the CPU is handled by an internal interrupt controller that recognizes the interrupt and diverts the CPU to the ISR.

The two figures below illustrate this; in the figure on the left, the CPU is running some kind of idle loop over and over, blinking an LED, calculating some time, updating an LCD (liquid crystal display), and other operations. At the end of the loop, execution repeats from the beginning. In the figure on the right, an interrupt occurs between the time calculation and the LCD update step. Perhaps the interrupt indicates that the UART peripheral has received a byte of data that needs to be read. The CPU breaks out of the continuous loop, the interrupt is serviced by the ISR, and then control returns to the loop at the same place where it exited.



Interrupt Vector Table: To get to the ISR, The interrupt controller must know where the ISR is located in memory. The ISR is compiled as part of the software and can be located anywhere. The addresses of all the ISRs are normally stored in an interrupt vector table. A vector table for a simple, hypothetical microcontroller might look like this:

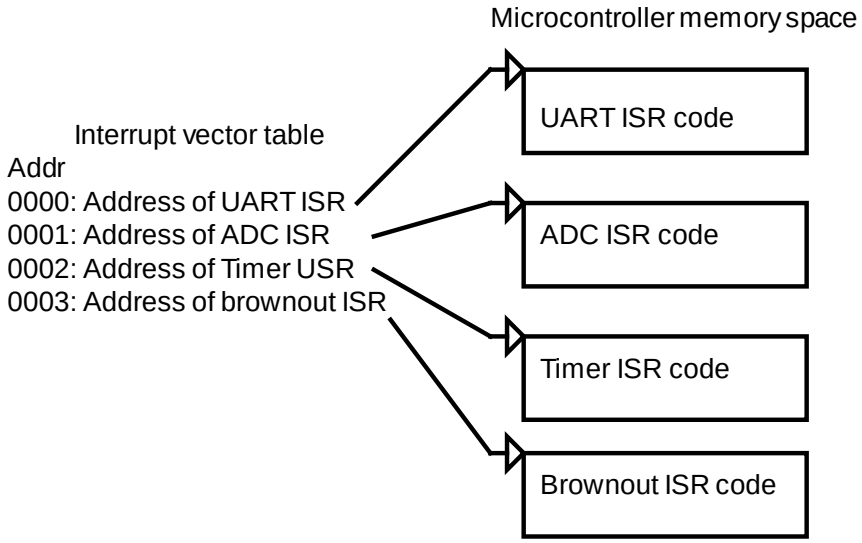
UART – gets control when a UART interrupt is received.

ADC – gets control when an analog-to-digital-converter has a word ready

Timer – gets control when a timer event, such as timeout, occurs

Brownout – gets control when the supply voltage drops too low

Each entry in the table would be large enough to hold the address of the routine that services its respective interrupt. For example, if the CPU address space was 32 bits wide, then the addresses for these five interrupts could be at word locations 0, 1, 2, and 3, each word being 4 bytes long. The ISR code is located somewhere in the execution space of the microcontroller, and the vector table contains the address of the start of each ISR, as shown below.



The interrupt vector table may just be a list of ISR addresses, or it may consist of a jump instruction for each interrupts to be used. If it is a list of jump instructions, then the interrupt controller just forces the CPU to start execution at that location in the table, which then jumps to the ISR. On devices where the table is just a list of addresses, the controller must read the address and then force the CPU to begin executing at the ISR address.

On a microcontroller with numerous internal peripherals and sophisticated ability to accept interrupts from input pins, there may be dozens of possible interrupt sources and the interrupt vector table maybe hundreds of bytes long. In most cases, when using a high-level language such as C, creation of the interrupt table will be performed by the compiler/linker when the ISR is defined in the source code.

The simple example here has a fixed vector table starting at memory address 0; some microcontrollers have a register that can move the base address of the vector table to different locations. Some devices can move the table from nonvolatile ROM to RAM.

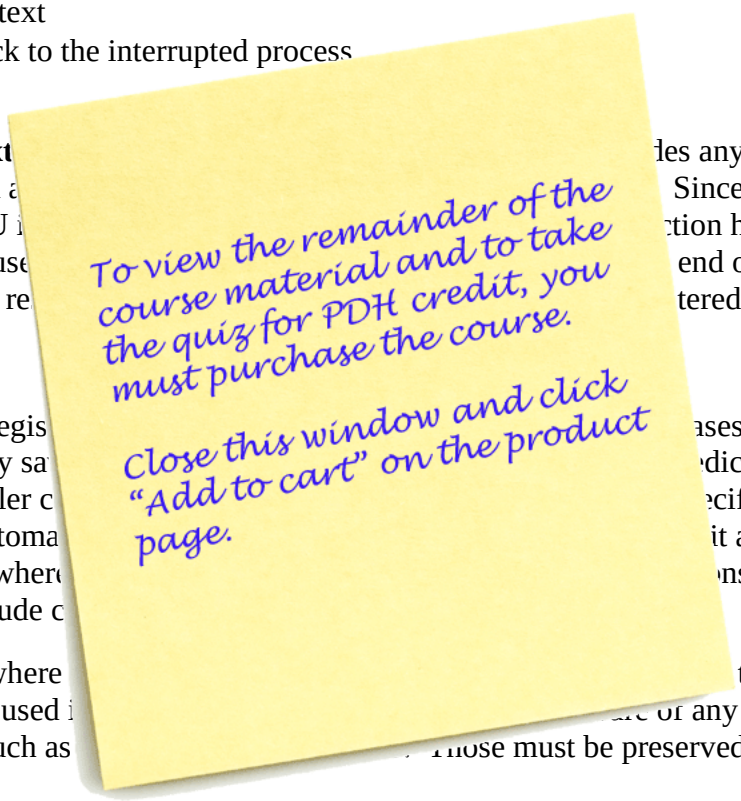
In the hypothetical UART example, when the UART byte is received, an interrupt signal will be sent to the interrupt controller. The interrupt controller will stop the CPU, save the program counter on the stack, get the address of the UART ISR from location 0 in the vector table, and then force the CPU to start executing at that address.

Once the CPU is in the UART ISR, a few things have to happen in software:

- Save the system context so that everything is restored to the proper state at the end of the ISR.
- Read and process the new UART byte
- Reset the interrupt signal
- Restore context
- ISR exit back to the interrupted process

What is the context

les any CPU registers that are used by the ISR a Since there is no way to know what the CPU i ction has to preserve and restore anything it use end of the ISR, all the registers need to be re tered.



In most cases, the regis ases it might be necessary to directly sa edicated memory area. Many microcontroller c ecify that a function is an ISR, and will automa it at the end. However, in cases where nsibility of the programmer to include c

Note that in cases where the context, it will save CPU registers used i of any other items that need to be saved, such as Those must be preserved by the ISR software.