CpE 213
Digital Systems Design

Lecture 21

Thursday 11/6/2003

UMR UNIVERSITY OF MISSOURI-ROLLA
The Name. The Degree. The Difference.

Announcements

- Exam 2: Thursday Nov. 13th
  - Topics to be covered and practice exams have been posted.
  - Will cover everything up to, but not including interrupts. Emphasis will be on material not covered in exam 1.
  - Open book and open notes.
- Assignment 7 has been posted.
  - Due: Nov. 11th at 11 am.
  - Will be solved in review session.
- Review session: Tuesday Nov. 11th, from 7 to 9 pm in EECH G31 (this room).
- Suggest review topics in addition to HW 7.

UMR Distinguished Speaker Series in Trustworthy Systems

The Future of Wireless Security

Mr. Bruce Potter

WEDNESDAY NOV. 12, 2003 3:30 – 4:30 PM, ROOM 111 Emerson Electric Co Hall

Over the last decade, wireless communications have changed the way we live. From cellular phones to wireless LANs to satellite television, wireless is everywhere. It has enabled us to be in contact with our family, friends, coworkers, and the public at large on a continuous basis. However, this technology is not without its risks. Wireless technology lacks one of the core building blocks of information assurance: physical security. An attacker can listen in on signals destined for another party an in attempt to gain knowledge, cause harm, or for personal advancement. Attackers have become increasingly sophisticated recently, driven largely by the fact that wireless communication equipment has become incredibly inexpensive and is being deployed with general purpose operating systems such as Linux and Symbian.

This talk will cover the dangers facing wireless devices. Issues such as client-side trust, poor encryption techniques, and weak operating systems will be discussed to determine how these weaknesses effect the security of complete systems. Specific examples in the cellular, LAN, and PAN arenas will be given in order to render the dangers concrete to the audience. The talk will also discussion what can be done to defend against an ever evolving attacker.

Bruce Potter is a Senior Software Security Consultant at Cigital. Mr. Potter is founder and president of Capitol Area Wireless Network, a non-profit community wireless networking initiative based in Washington, D.C. His areas of expertise include wireless security, large-scale network architectures, smartcards and promotion of secure software engineering practices. Mr. Potter coauthored the book 802.11 Security, published in 2003 by O'Reilly and Associates as well as Mac OS X Security published by New Riders. He also has published articles in various security–related publications including Elsevier's Network Security and IEEE's Security and Privacy. Mr. Potter was trained in computer science at the University of Alaska, Fairbanks.

Refreshment will be served before the lecture at 3:00 outside of Room 111.

UMR Distinguished Speaker Series in Trustworthy Systems

Outline

- Structured program development
- Final advice on coding
- Interrupts
  - Interrupt handling basics
  - Interrupts on the 8051

Structured programming
Chapter 8 of ISM

Structured Program Development

- Many early programs were very unstructured (spaghetti-like) due to the use of goto statements and hence difficult to follow and maintain.
- Structured program design is based on the work of Bohm and Jacopini in mid 60s as an improvement on the incomprehensible ‘spaghetti code’.
- It is defined as a method of programming in which programs are constructed with easy-to-follow logic, attempt to use only three basic control structures and are divided into modules.
- Nowadays, all modern higher-level languages are designed to support structured programming.

Some slides dapted from ET2710 slides by K. T. Ng

Characteristics of Structured Programming

- Program is made of small, understandable and manageable modules.
- A module is a part of a program that is dedicated to performing one action. (e.g. each function, subroutine etc. is a kind of module).
- Any program, regardless of its complexity can be formed by three basic elements of control structure:
  - Sequence
  - Selection/Alternation/Decision
  - Iteration/Repetition/Looping
- Each block of code (i.e. sequence of statements) has a single entry point and a single exit point. Thus we can chain sequences of statements together in an orderly fashion.

Code Blocks: Single Entry/Exit

- Sequence of N statements
  - Start as statement 1
  - End at statement N
- Selection structure:
  - Start with If or Select
  - End with End If or End Select
- Repetition structure:
  - While condition Do
  - . . .
  - End While

Advantages of Structured Programming

- The advantages of adopting a structured approach to programming include the following:
  - It makes the program
    - easier to read
    - easier to understand
    - easier to debug
    - easier to maintain
    - more portable
    - easier to reuse
  - A structured programming approach lends itself to team programming.

Flowcharts and Pseudocode

- Structured program languages lend themselves to flowcharts and pseudocode.
- A flowchart is a pictorial representation of a structured program. A flowchart is designed to visually represent the flow of execution through a program.
- Although flowcharts are used quite widely in a variety of situations, they are not often used in algorithms due to ambiguities in their structure. In this course we use pseudocode instead.
- Pseudocode literally means “false code”. It is an English-like, natural language description which concentrates on the logic behind in a program --- not the syntax of a programming language. It is considered as the “first draft” of the actual program.

Structure Programming in Assembly Language

- Assembly language does not formally support structured techniques such as those of high level languages.
- Though the resulted code is often slightly longer, it is possible to implement these structures in assembly language.
- This results in the programs being easier to understand, document, and maintain.

The Concept of Sequence

- Certain events must occur in a particular order - for example, we should get out of bed before showering.
- Other events may occur in any order and do not affect the overall solution.
- In the solution of any problem, it is necessary to decide whether any steps must come before or after other steps.
- Often the effectiveness of a solution may depend on whether or not this happens.
- The same deliberations do not have to occur when the order of steps is not important.

The Sequence Structure

- The sequence is a linear structure in which instructions are executed consecutively.
- The most common statement used in high-level languages for this structure is the assignment statement.
  - For example: result = A + B
- To implement this as closely as possible in assembly language:
  - Firstly, a MOV instruction is needed for at least one of the right hand side variables to move it into a register.
  - Then an arithmetic or logical instruction is needed for each operation to be performed.
  - Finally, a MOV is needed to store the calculated result into the variable whose name is on the left-hand side.

The Sequence Structure Example

- Pseudocode

  W = (X + Y) * Z

  We assume all the variables W, X, Y and Z are in the 8051's internal memory and have been declared by the EQU directives.

- 8051 ASM code

```
MOV A,X     ;Get X
ADD A,Y     ;Compute X + Y
MOV B,Z     ;Get Z to B
MUL AB      ;Compute (X + Y) * Z
MOV W,A     ;Store result in W
```

(For simplicity's sake, it is assumed that the product is ≤ 255)

The Selection Structure

- High-level languages (HLLs) use selection structures such as IF-THEN, IF-THEN-ELSE and SWITCH statements to control block of code execution.
- Selection structure that are supported in assembly language is very simple, such as testing if the value of a variable (stored in a register) is zero or if the addition of two variables produces a carry or not.
- More complex selection conditions must be translated into a sequence of simple assembly language instructions whose final result will be an equivalent condition test.

The Switch Statement

- The SWITCH statement is a handy variation of the IF-THEN-ELSE statement.
- It is used when one statement from many must be chosen as determine by a value.
- Pseudo-code:

```
SWITCH (selector) {
    case value1: Statement 1;
                 break;
    case value2: Statement 2;
                 break;
              . . . . . .
    case valuen: Statement n;
                 break;
  default:       Default statement;
}
```

Example of SWITCH Statement

- e.g. Implement the following pseudo-code in 8051 assembly language. Assume GRADE and SCORE are predefined variables.

```
SWITCH (GRADE) {
  CASE 'A':   SCORE = 90;
                BREAK;
  CASE 'B':   SCORE = 70;
                BREAK;
  CASE 'C':   SCORE = 50;
                BREAK;
  DEFAULT: SCORE = 25;
}
```

Example of SWITCH Statement (Cont.)

8051 assembly code

```
            MOV    A,GRADE          ;Get GRADE to A
CASE0:      CJNE   A,#'A',CASE1     ;If GRADE ≠ 'A' then CASE1
            MOV    SCORE,#90        ;assign SCORE to 90 for case 0
            SJMP   CONTINUE
CASE1:      CJNE   A,#'B',CASE2     ;If GRADE ≠ 'B' then CASE2
            MOV    SCORE,#70        ;assign SCORE to 70 for case 1
            SJMP   CONTINUE
CASE2:      CJNE   A,#'C',DEFAULT   ;If GRADE ≠ 'C' then DEFAULT
            MOV    SCORE,#50        ;assign SCORE to 50 for case 2
            SJMP   CONTINUE
DEFAULT:    MOV    SCORE,#25        ; assign SCORE to 25 for default
CONTINUE:   . . . . . .
```

Group Exercise

Circle the correct answers:

- C is (hard/easy) to code and debug, but ASM code can be (smaller/larger) and (slower/faster).
- In ASM, you have (less/more) control over the hardware.
- 90/10 rule: 10% of your code is executed 90% of the time. (C/ASM) is suitable for this 10%, the other (10/90)% can be written in (C/ASM).

Final advice about coding

- Table lookup may be more efficient than calculation.
- Only use necessary precision; nothing more.
- Uss the smallest data type possible; avoid large types such as “float.”
- Read examples in your lecture handout.

CpE 213
Example ISM67 – Table lookup

Purpose

This is an elaboration on example 3-23 in the text. Table look-up is a common alternative to complex calculations when programming a simple processor like the 8051. This example illustrates use of table look-up as an alternative to a complex floating point math calculation.

Math operations on the 8051

Embedded system designers occasionally need to perform math operations on sensor data. Typical 'real world' data limits are 4-20 mA, 0-5 V, -5 to +5 V, etc, and are normally limited to a few bits of precision. Common values are 8 or 12 bits and rarely exceed 16 bits. The temptation is to resort to use of floating point values for the calculations, but processors like the 8051 lack floating point hardware. Compilers such as C51 are usually supplied with a math library, which includes routines for performing floating point arithmetic as well as the usual sqrt, sin, cos, etc functions.

The following C program might be used, for example, to read a value into port 1 representing a voltage between 0 and 5 V, calculate the square root of the voltage, and output the result to port 2. The input voltage is scaled so that 00H represents 0 V, 0FFH represents 5 V, etc. P1 is thus 51*Vin. We would like P2 to be set equal to the square root of the input voltage scaled up by the same value of 51. In other words, if the input voltage is 2.5v, P1 would be read as 127. The square root of 2.5v is 1.58 so we would output int(1.58*51)=80. How the voltage is turned into a digital value and vice versa isn't important right now. Later we'll talk about a device called an *analog to digital converter* and another called a *digital to analog converter* that can do such conversions.

```
#include <math.h>
#include <reg51.h>
void main(void){
  while(1){
  P2=(unsigned char) 51.0*sqrt(5.0*(float)(P1/255.));
 }
}
```

The problem with this rather simple program is that it takes about 3.665 mS per loop and occupies 1104 bytes of code (half the entire 8xC752 code space!). The following code accomplishes exactly the same function in about 10 uS and 279 bytes. That's 25% of the original code and 366 times faster!

```
#include <reg51.h>
void main(void){
 unsigned char code sqrt[]= {
0,7,10,12,14,16,17,19,20,21,23,24,25,26,27,28,29,29,30,31,32,33,33,
34,35,36,36,37,38,38,39,40,40,41,42,42,43,43,44,45,45,46,46,47,47,
48,48,49,49,50,50,51,51,52,52,53,53,54,54,55,55,56,56,57,57,58,58,
58,59,59,60,60,61,61,61,62,62,63,63,63,64,64,65,65,65,66,66,67,67,
67,68,68,68,69,69,70,70,70,71,71,71,72,72,72,73,73,74,74,74,75,75,
75,76,76,76,77,77,77,78,78,78,79,79,79,80,80,80,80,81,81,81,82,82,
82,83,83,83,84,84,84,84,85,85,85,86,86,86,87,87,87,87,88,88,88,89,
89,89,89,90,90,90,91,91,91,91,92,92,92,93,93,93,93,94,94,94,94,95,
95,95,96,96,96,96,97,97,97,97,98,98,98,98,99,99,99,99,100,100,100,
100,101,101,101,101,102,102,102,102,103,103,103,103,104,104,104,104,
105,105,105,105,106,106,106,106,107,107,107,107,108,108,108,108,109,
109,109,109,109,110,110,110,110,111,111,111,111,112,112,112,112,112,
113,113,113,113,114,114,114};
```

```
while(1){
  P2=sqrt[P1];
  }
}
```

The table was generated using an Excel spread sheet to calculate all 256 possible values of the scaled square root function and exported as a comma delimited file which was then included in my C code. P1 is used as an index into the table and the resulting value is ready to be output to P2. You can run these programs using dScope with a watch set on P2 (ws P2,10) and interactively type new values for P1 (P1=127) in the command window and watch the value of P2 change.

The M51 list file (foo.M51) produced by the linker is helpful for determining the size of each program since it will show the size(s) of any library routines included as well as your own code. The size of the first version is quite small compared to the second but it's the sqrt library function and the floating point routines which are the killers.

Summary

This example has shown:

- The penalty to be paid as the price for using floating point arithmetic on the 8051,
- How to use table look-up as an alternative to math operations on the 8051.

Questions

Answer the following questions.

1. What is the sample rate of the two programs in this example? In other words, how many times per second is P1 read?
2. Suppose a table containing 256 samples of one cycle of a sine wave is used to generate a sine wave output on P2. Assuming a 12 Mhz clock, what is the shortest possible period of the signal generated by P2?
3. What is the highest frequency signal you can generate using the direct calculation approach? Assume 256 samples per cycle and a 12 Mhz clock to be consistent.
4. Compile the programs above and run through one complete iteration to see how many cycles (states) each version takes.

Interrupts
Chapter 6 of ISM

Interrupt Handling Basics

- Interrupt: An unexpected (asynchronous) hardware-induced subroutine call (may also be referred to as an event or as an exception)
- Real-Time: Handling of interrupts ("events") must be completed within fixed time constraints ("mission critical timing")

Interrupt Handling Basics

- Sources of interrupts: Can either be on-chip devices (e.g., timers, communications controllers, analog-to-digital converter) or off-chip devices (e.g., printer, sensor)
- Vector: Locations in memory that "point to where to go" in order to service an interrupt request, i.e., a pointer to an interrupt service routine

Interrupt Handling Basics

- Non-maskable
  - a CPU interrupt input/source that cannot be ignored (or "masked out") by the processor
  - usually reserved for catastrophic system failures (e.g., bus error or power failure)
  - usually denoted "NMI" (non-maskable interrupt).
  - 8051 does not have one!

Interrupt Handling Basics

- Maskable
  - an interrupt that can be (temporarily) ignored by the processor (e.g., if it is currently performing a higher priority task than the incoming interrupt request)
  - typically referred to as "application interrupt requests" and denoted "IRQ" or "INT"
  - some processors, including 8051, provide multiple, prioritized application interrupts

Interrupt Handling Basics

- Context switch
  - CPU activity associated with changing from one task (e.g., a "main-line program") to another task (e.g., an interrupt service routine)
  - the machine state or "context" (typically consisting of all CPU registers) needs to be saved for the task being "rolled out" and loaded for the task being "rolled in"
  - How does the 8051 hardware facilitate context switch?

Interrupt Sequencing

1) An interrupting event actually occurs
   - Can happen at any time
   - Independent of internal processor operation
   - Sometimes called an *imprecise interrupt*
2) The interrupt event is detected by the processor
   - Only occurs at discrete sample times
3) The interrupt(s) is (are) polled and highest priority one selected for service.
   - Delay from event to service is called *latency*.

Interrupt Servicing Latency

- the amount of time that expires between when a device asserts the CPU's interrupt request input and when the CPU fetches the first instruction of the interrupt service routine
- consists of two components

Instruction Completion Latency

- the amount of time for the CPU to complete the current instruction in progress
- note that most CPUs cannot be interrupted mid-instruction
- this time is non-deterministic since:
  - instructions are typically variable length in execution cycles
  - interrupts are asynchronous with respect to the CPU clock
- reasonable bounds on the instruction completion latency can be estimated

Processor Latency

- the time between when the interrupt input signal is recognized by the CPU and when the first instruction of the interrupt service routine is fetched
- most CPUs examine their interrupt inputs at the beginning of each fetch cycle
- the processor latency is also referred to as the “context switch” overhead

Interrupt Prioritization

- Prioritization is the assignment of relative priorities to the various interrupt signals
- the 8051 provides a way to establish the priority (high or low) of each interrupt source
- interrupts at the same priority level are serviced on a first-come first-serve basis
- 8051 has a fixed polling sequence that essentially orders interrupts of the same priority

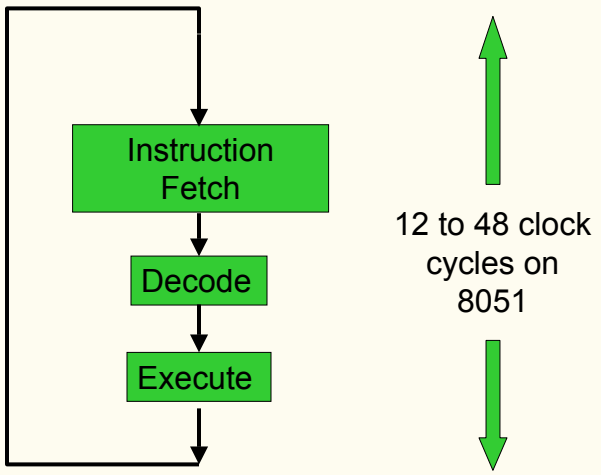
Interrupt Handling Basics

- Level-sensitive interrupt input
  - a CPU interrupt input signal that must be latched externally
  - this latch, which must then be cleared under software control by the interrupt service routine, is called a device flag
  - most CPU interrupt inputs are level sensitive

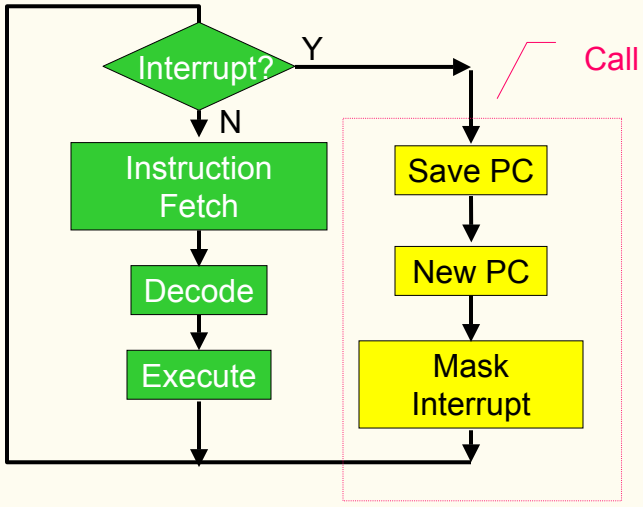
Interrupt Handling Basics

- Edge-sensitive interrupt input
  - a CPU interrupt input that requires no external latch
  - on some CPUs, specific interrupt inputs can be programmed to be edge sensitive

Basic instruction execution



Add interrupt polling



Hardware Generated Call

- Some considerations:
  - Where to save the PC?
  - Where does the call's address field come from?
  - What about the rest of the processor's state?
- PC is pushed on the stack
- The address depends on the interrupt
  - This address is called an interrupt vector
- Registers can be saved by switching banks

Interrupts on the 8051

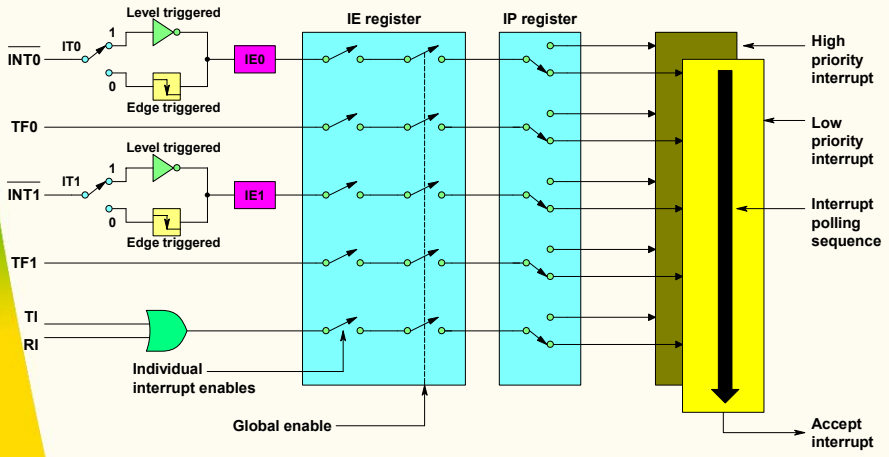
Interrupt Sources for 8051

- The 8051 has 5 sources of interrupts:
  - 2 external and 3 internal sources
  - External INT0 pin (P3.2) — can connect to any external hardware.
  - Timer 0 — TF0 flag (internal).
  - External INT1 pin (P3.3) — can connect to any external hardware.
  - Timer 1 — TF1 flag (internal).
  - Serial port: just finished transmitting a character or have just received a character — RI&TI (internal).
- A signal from any of these sources can trigger execution of an ISR.
- Some manufacturers consider RESET to be an interrupt source as well.

8051 Interrupt Organization

- Each of these interrupt sources can be individually enabled or disabled in the Interrupt Enable (IE) register.
- IE contains a global disable bit, which can disable all interrupts at once.
- External interrupts can be programmed for edge or level sensitivity.
- Each interrupt type can be programmed to one of two priority levels.
- All of the bits that generate interrupts are under software control!

Overview of 8051 Interrupt Structure



Interrupt Enable Register (IE)

IE Address = A8H Reset Value = 0000 0000B
Bit Addressable

Bit:	7	6	5	4	3	2	1	0
	EA	- - -	- - -	ES	ET1	EX1	ET0	EX0

Symbol	Function
EA	Global disable bit. If EA = 0, all interrupts are disabled. If EA = 1, each interrupt can be individually enabled or disabled by setting or clearing its enable bit.
- - -	Reserved for future used.
- - -	Reserved for future used.
ES	Serial port interrupt enable bit.
ET1	Timer 1 interrupt enable bit.
EX1	External interrupt 1 enable bit.
ET0	Timer 0 interrupt enable bit.
EX0	External interrupt 0 enable bit.
	Enable Bit = 1 enables the interrupt. Enable Bit = 0 disables it.

Group Exercise

- Show the instructions to:
  - a) enable INTS, TF0, and INT1
  - b) disable (mask) TF0
  - c) disable all interrupts with a single instruction

Interrupt Priority

- Each interrupt source can be individually programmed to one of two priority levels, by setting or clearing a bit in the IP register.
- A low priority interrupt can be interrupted by a higher priority interrupt, but not by another low priority interrupt.
- A high priority interrupt cannot be interrupted by any other interrupt source.
- If two requests of different priority levels are received simultaneously,
- Default is all low priority so only one interrupt at a time.

Simultaneous Interrupts of Same Priority

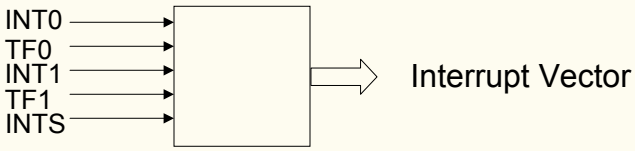
- If interrupts of the same priority level are received at the same time, an internal polling sequence determines which interrupt is serviced.
- The “priority within level” structure is only used to resolve simultaneous requests of the same priority level.

Interrupt Priority within Level Polling Sequence

Priority Within Level	Source
1 (Highest)	External Interrupt 0
2	Timer 0
3	External Interrupt 1
4	Timer 1
5 (Lowest)	Serial Port

Tie breaker

- Essentially a priority encoder
- Interrupt sources in, interrupt vector out



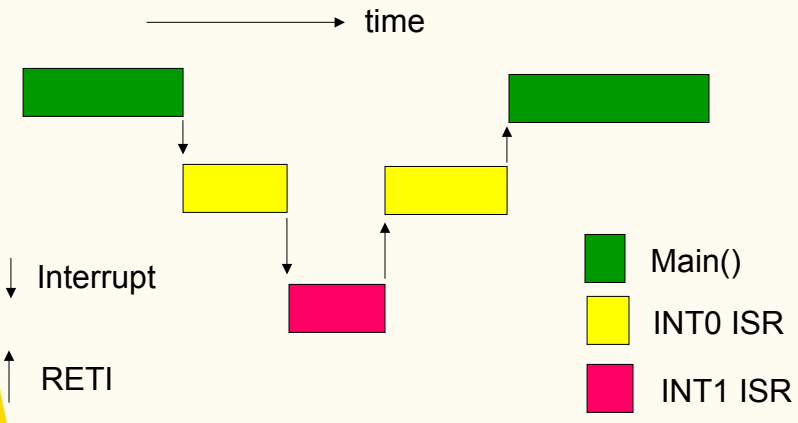
Priority Encoder

INT0	TF0	INT1	TF1	INTS	Vector
1	-	-	-	-	0x0003
0	1	-	-	-	0x000B
0	0	1	-	-	0x0013
0	0	0	1	-	0x001B
0	0	0	0	1	0x0023

Interrupt Priority Example

- INT0 - low priority, INT1 - high priority
- Both INT0 and INT1 occur at same time
- Result:

Interrupt operation



Interrupt Priority Register (IP)

IP **Address = B8H** **Reset Value = XX00 0000B**
Bit Addressable

Bit:	7	6	5	4	3	2	1	0
	- - -	- - -	- - -	PS	PT1	PX1	PT0	PX0

Symbol	Function
- - -	Reserved for future used.
- - -	Reserved for future used.
PS	Serial port interrupt priority bit.
PT1	Timer 1 interrupt priority bit.
PX1	External interrupt 1 priority bit.
PT0	Timer 0 interrupt priority bit.
PX0	External interrupt 0 priority bit.
	Priority Bit = 1 assigns high priority. Priority Bit = 0 assigns low priority.

Group Exercise

- a) Program the IP register to assign the highest priority to INT1.
- b) Discuss what happens then if INT0, INT1, and TF0 are activated at the same time.

Group Exercise

- Assume that after reset, the interrupt priority is set by the instruction "`MOV IP,#00001100B`." Discuss the sequence in which the interrupts are serviced.

Return from ISR

- ISR - Interrupt Service Routine
- A normal RET can't be used!
- RETI re-enables interrupts at that level and returns to interrupted program

Programming ISRs in Keil

- Keil compiler implements a function extension that explicitly declares a function as an interrupt handler.
- The extension is **interrupt**, and it must be followed by an integer specifying which interrupt the handler is for.
- For example:

```
/* This is a function that will be called
whenever a serial interrupt occurs. Note that
before this will work, interrupts must be
enabled.*/
void serial_int (void) interrupt 4 {
  ...
}
```

Interrupts in Keil

- Number following **interrupt** keyword is calculated by subtracting 3 from the interrupt vector address and dividing by 8.

Interrupt	Vector Address	Interrupt Number
External 0	0003h	0
Timer 0	000Bh	1
External 1	0013h	2
Timer 1	001Bh	3
Serial	0023h	4

Interrupts in Keil

- Optional extension to interrupt: **using**
- Tells compiler to change to a new register bank prior to executing ISR instead of pushing all registers to stack.
- Reduces interrupt latency, should be used when quick execution is important.
- Example:

```
void serial_int (void) interrupt 4 using 1{
...}
```

- Interrupts of the same priority can use the same register bank. Why?

Excerpt from Writing C Code for the 8051

by Matthew Kramer

(Available on-line at: http://ubermensch.org/Computing/8051/8051-c/#appa)

Example 2 INT.C

```
/*****************************************************************************
* int.c - A demonstration of how to write interrupt-driven code for an
*  8051 using the Keil C compiler.  The same techniques may work in other
*  8051 C compilers with little or no modifcation.  This program will
*  combine the functionality of both basic.c and serial.c to allow serial
*  communications to be entirely in the background, driven by the serial
*  interrupt.  This allows the main() function to count on Port 0 without
*  being aware of any ongoing serial communication.
*/

/* included headers */

#include

/* function declarations */

char getCharacter (void);       /* read a character from the serial port */
void sendCharacter (char);      /* write a character to the serial port */

/*
 * Interrupt handlers:
 *  Here the code for the interrupt handler will be placed.  In this
 *  example, a handler for the serial interrupt will be written.
 *  Examination of the 8051 specs will show that the serial interrupt is
 *  interrupt 4.  A single interrupt is generated for both transmit and
 *  receive interrupts, so determination of the exact cause (and proper
 *  response) must be made within the handler itself.
 *  To write an interrupt handler in Keil, the function must be declared
 *  void, with no parameters.  In addition, the function specification
 *  must be followed by a specification of the interrupt source it is
 *  attached to.  The "using" attribute specifies which register bank
 *  to use for the interrupt handler.
 */

void serial_int (void) interrupt 4
{
            static charchr = '\0';/* character buffer */

            /*
             * The interrupt was generated, but it is still unknown why.  First,
             * check the RI flag to see if it was because a new character was
             * received.
             */
```

```
        if (RI == 1)            /* it was a receive interrupt */
        {
                chr = SBUF;             /* read the character into our local buffer */
                RI = 0;                 /* clear the received interrupt flag */
                TI = 1;                 /* signal that there's a new character to send */
        }
        else if (TI == 1)       /* otherwise, assume it was a transmit interrupt */
        {
                TI = 0;         /* clear the transmit interrupt flag */
                if (chr != '\0')        /* if there's something in the local buffer... */
                {
                        if (chr == '\r') chr = '\n';        /* convert  to  */
                        SBUF = chr;                         /* put the character into the
transmit buffer */
                        chr = '\0';
                }
        }
}

/* functions */

/*****************************************************************************
* main - Program entry point.  This program sets up the timers and
*  interrupts, then simply receives characters from the serial port and
*  sends them back.  Notice that nowhere in the main function is Port 0
*  incremented, nor does it call any other function that may do so.
*  main() is free to do solely serial communications.  Port 0 is handled
*  entirely by the interrupt handler (aside from initialization).
*
* INPUT: N/A
* RETURNS: N/A
*/

main()
{
        /* Before the serial port may be used, it must be configured. */

        /* Set up Timer 0 for the serial port */

        SCON = 0x50;        /* mode 1, 8-bit uart, enable receiver */
        TMOD = 0x20;        /* timer 1, mode 2, 8-bit reload */
        TH1  = 0xFE;        /* reload value for 2400 baud */
        ET0  = 0;           /* we don't want this timer to make interrupts */
        TR1  = 1;           /* start the timer */
        TI   = 1;           /* clear the buffer */

        /*
         * The compiler automatically installs the interrupt handler, so
         * all that needs to be done to use it is enable interrupts.  First,
         * speficially enable the serial interrupt, then enable interrupts.
         */

        ES   = 1;           /* allow serial interrupts */
```

```
        EA   = 1;               /* enable interrupts */

        /* initialize Port 0 to 0, as in basic.c */

    P0 = 0;

        /*
         * Loop forever, increasing Port 0.  Again, note nothing is done
         * with the serial port in this loop.  Yet simulations will show
         * that the software is perfectly capable of maintaining serial
         * communications while this counting proceeds.
         */

        while (1==1)
    {
                unsigned int        i;
                for (i = 0; i < 60000; i++) {;} /* delay */
                P0 = P0 + 1;        /* increment Port 0 */
        }
}
```