Interrupts and Using Them in C

These lecture notes created by Mr. James B. (Jim) Carlson, NCSU

In These Notes . . .

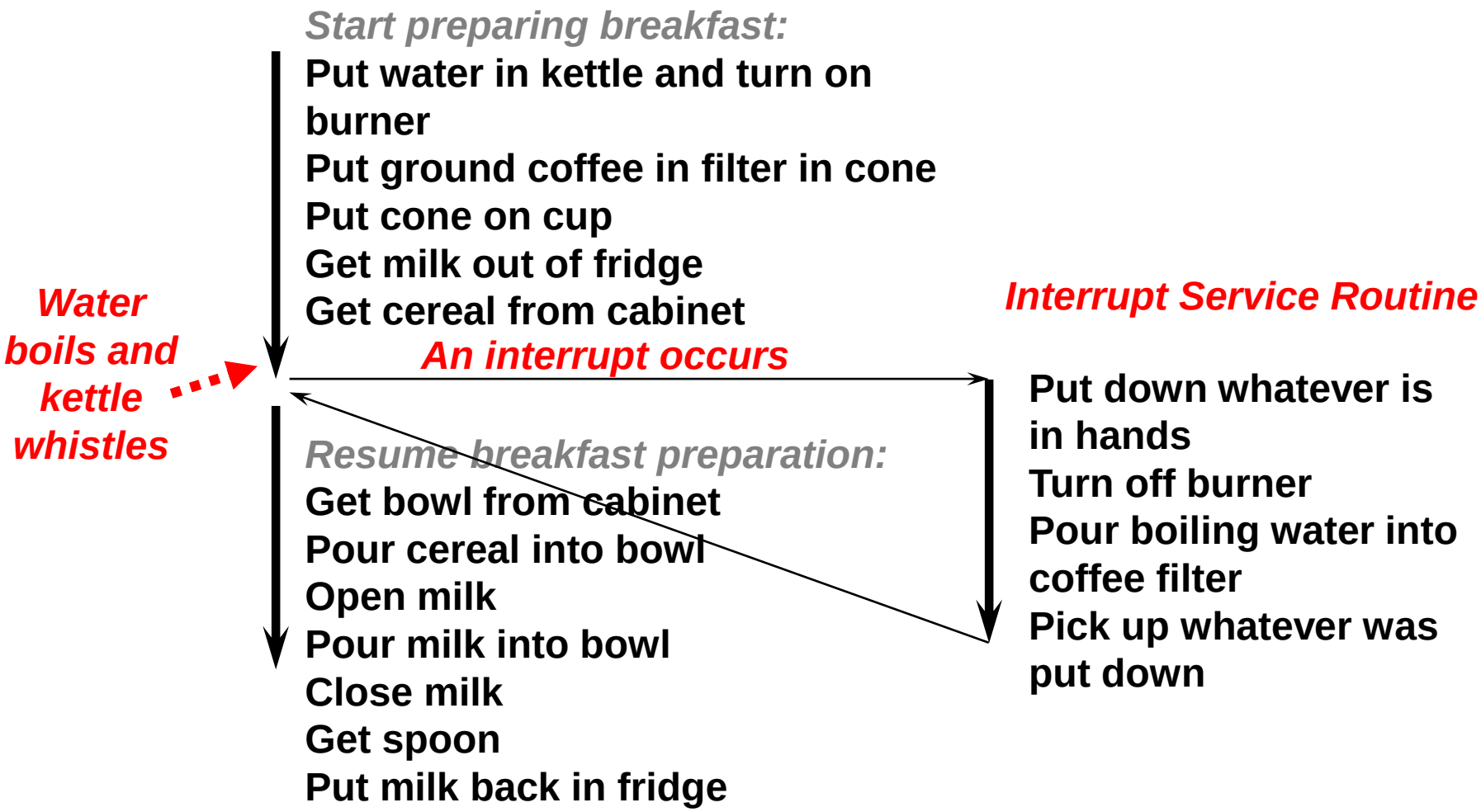
- Interrupts
  - How they work
  - Creating and debugging C interrupt routines
- Readings
  - MSP430FR57xx Family User's Guide, Chapter 1.3 Interrupts
  - If not already done, consider reading …
    - “Introduction to Interrupts,” Russell Massey
    - “Interrupt Latency,” Jack Ganssle
    - “Introduction to Interrupt Debugging,” Stuart Ball

Interrupts and Polling

- Consider the task of making coffee
  - We need to boil water, but don't know exactly how long it will take to boil

- How do we detect the water is boiling?
  - Keep watching the pot until we see bubbles
    - This is called ***polling***
    - Wastes time – can't do much else
  - Put the water in a kettle which will whistle upon boiling
    - The whistle is an ***interrupt***
    - Don't need to keep watching water. Instead you can **do something else until the kettle whistles**.
    - Much more efficient

Breakfast Timeline



Microcontroller Interrupts

- Types of interrupts
  - Hardware interrupts
    - ***Asynchronous***: not related to what code the processor is currently executing
    - Examples: Port Pin input becomes active as with the switches, character is received on serial port, or the most common, a timer expires.
  - Software interrupts
    - ***Synchronous***: are the result of specific instructions executing
    - Examples: BRK, INT, undefined instructions, overflow occurs for a given instruction
  - We can enable and disable (*mask*) most interrupts as needed *(maskable)*, others are *non-maskable*
- Interrupt service routine (ISR)
  - Subroutine which processor is ***forced to execute*** in respond to a ***specific event***
  - After ISR completes, MCU goes back to previously executing code

Interrupt Processing Sequence for Enabled Interrupt

- Other code (main – or function off of main) is running
- Interrupt trigger occurs
- Processor does some hard-wired processing
- Processor executes ISR (foreground), including return-from-interrupt instruction at end
- Processor resumes other code

Main Code (Background)

ISR (Foreground)

Interrupt Considerations

- Vectors
- Prioritization
- Saving and Restoring Context
- Response time
- Maximum interrupt rate

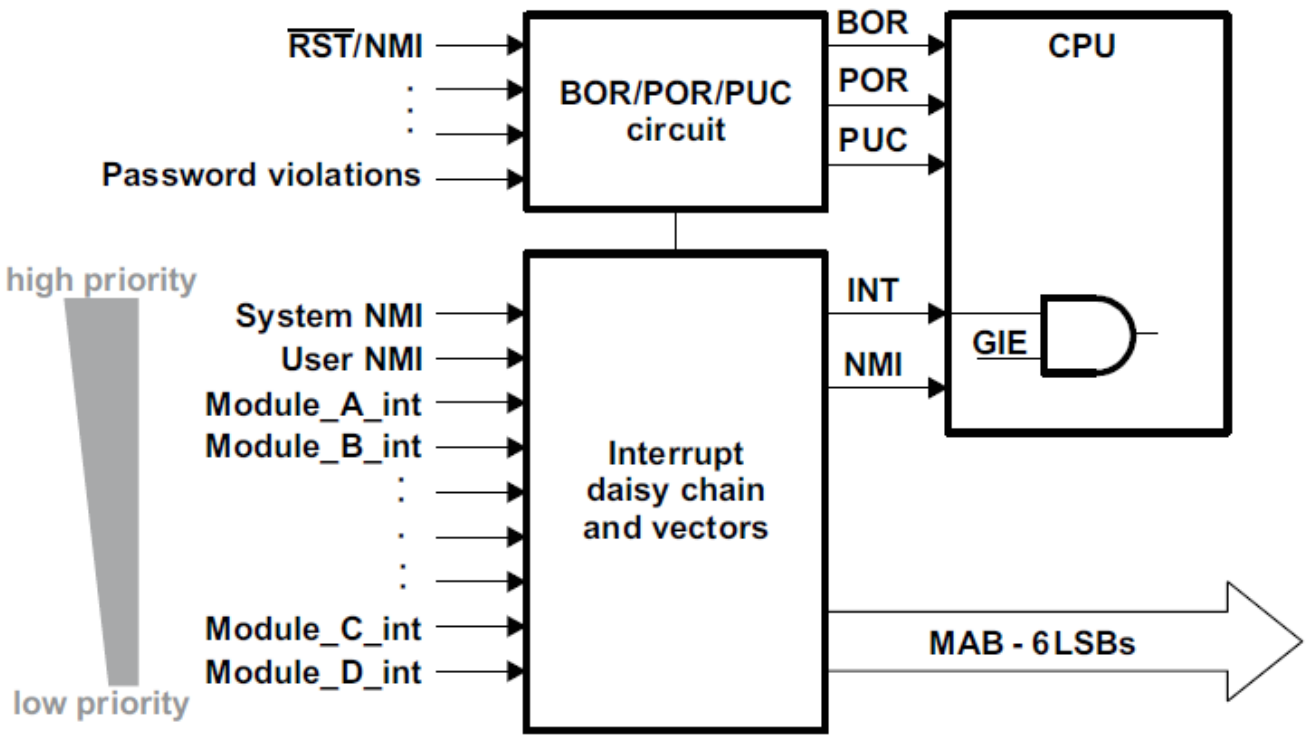
MSP430 Vectors Details

- The interrupt vectors are located in the address range 0FFFFh to 0FF80h, for a maximum of 64 interrupt sources. A vector is programmed by the user and points to the start location of the corresponding interrupt service routine. Table below is an example of the interrupt vectors available. See the device-specific data sheet for the complete interrupt vector list.

Interrupt Source	Interrupt Flag	System Interrupt	Word Address	Priority
Reset: power up, external reset watchdog, FRAM password	... WDTIFG FRCTLPW	... Reset	... 0FFFEh	... Highest
System NMI: JTAG Mailbox	JMBINIFG, JMBOUTIFG	(Non)maskable	0FFFCh	...
User NMI: NMI oscillator fault	... NMIIFG OFIFG	... (Non)maskable (Non)maskable	... 0FFFAh	... ...
Device specific			0FFF8h	...
...			...	...
Watchdog timer	WDTIFG	Maskable	...	...
...			...	...
Device specific			...	...
Reserved		Maskable	...	Lowest

Prioritization

- The interrupt priorities are fixed and defined by the arrangement of the modules in the connection chain as shown in below. Interrupt priorities determine what interrupt is taken when more than one interrupt is pending simultaneously.
- There are three types of interrupts:
  - System reset
  - (Non)maskable
  - Maskable



Prioritization

- Interrupt requests are considered ***simultaneous*** if they occur between the same two clock ticks
- Interrupts are prioritized to order the response simultaneous interrupt requests
  - Priorities of some interrupts are ***fixed by the hardware design***
    - Reset (highest priority)
    - NMI
    - DBC
    - Watchdog timer
    - Peripheral I/O
    - Single step
    - Address match
  - Priorities of other (peripheral) interrupts are ***defined within the hardware***

Saving and Restoring Context

- When an interrupt is requested from a peripheral and the peripheral interrupt enable bit and GIE bit are set, the interrupt service routine is requested. Only the individual enable bit must be set for (non)-maskable interrupts (NMI) to be requested.
- The interrupt latency is six cycles, starting with the acceptance of an interrupt request, and lasting until the start of execution of the first instruction of the interrupt service routine. The interrupt logic executes the following:
  - 1. Any currently executing instruction is completed.
  - 2. The PC, which points to the next instruction, is pushed onto the stack.
  - 3. The SR is pushed onto the stack.
  - 4. The interrupt with the highest priority is selected if multiple interrupts occurred during the last instruction and are pending for service.
  - 5. The interrupt request flag resets automatically on single-source flags. Multiple source flags remain set for servicing by software.
  - 6. All bits of SR are cleared except SCG0, thereby terminating any low-power mode. Because the GIE bit is cleared, further interrupts are disabled.
  - 7. The content of the interrupt vector is loaded into the PC; the program continues with the interrupt service routine at that address.

Enable and Disable Interrupt

- Due to the pipelined CPU architecture, the instruction following the enable interrupt instruction (EINT) is always executed, even if an interrupt service request is pending when the interrupts are enabled.
- If the enable interrupt instruction (EINT) is immediately followed by a disable interrupt instruction (DINT), a pending interrupt might not be serviced. Further instructions after DINT might execute incorrectly and result in unexpected CPU execution. It is recommended to always insert at least one instruction between EINT and DINT. Note that any alternative instruction use that sets and immediately clears the CPU status register GIE bit must be considered in the same fashion.

Return From Interrupt

- The interrupt handling routine terminates with the instruction:
- RETI //return from an interrupt service routine
- The return from the interrupt takes five cycles to execute the following actions and is illustrated below.
  - 1. The SR with all previous settings pops from the stack. All previous settings of GIE, CPUOFF, and so on are now in effect, regardless of the settings used during the interrupt service routine.
  - 2. The PC pops from the stack and begins execution where it was interrupted.

Checklist for Using Interrupts in C

- Read
  - MSP430 Software Manual, (*Interrupt*)
  - C Language Programming Manual, (*Interrupt Processing*)

- Configure MCU
  - Enable interrupts for system

- Write ISR `my_isr` and identify it as an ISR using **`#pragma vector = {Vector}`**

Write Interrupt Service Routine

```
#pragma vector = {Vector}
// do not forget the header
void my_isr(void) {
  /* do whatever must be done */
}
```

- Rules for writing ISRs
  - No arguments or return values – void is only valid type
  - Keep it short and simple
    - Much easier to debug
    - Improves system response time
- Tell compiler it's an ISR
  - ISR has a different stack frame compared with subroutine
    - Saves all registers
    - Flag register saved
  - Different return needed (REIT vs. EXITD)
  - So use **`#pragma vector = {Vector}`**

Building a Program – Start with the basics

- Program requirements:
  - Req1: When S1 is pressed, turn LCD Backlight off
  - Req2: When S2 is pressed, turn LCD Backlight on
  - Req3: Count the switch presses for S1 and S2 and display the counts
  - Req4: Use interrupts for switch press detection
- Both of the switches on our board are attached to port 4
  - S1 = p4_0 =>
    - P4IFG &= ~SW1; // P4.0 IFG SW1 cleared
    - P4IE |= SW1; // P4.0 SW1 interrupt Enabled
  - S2 = p4_1 =>
    - P4IFG &= ~SW2; // P4 IFG SW2 cleared
    - P4IE |= SW2; // P4.0 SW2 interrupt enabled
- Button presses are “negative logic” so the interrupt will be based on the negative edge (1 -> 0) of the press.
  - P4IES |= SW1; // P4.0 Hi/Lo edge interrupt
  - P4IES |= SW2; // P4.1 Hi/Lo edge interrupt

Building a Program – the ISR

```
#pragma vector=PORT4_VECTOR
__interrupt void switch_interrupt(void){
// Switch 1
  if (P4IFG & SW1) {
    P1OUT &= ~LCD_BACKLITE;
  }
// Switch 2
  if (P4IFG & SW2) {
    P1OUT |= LCD_BACKLITE;
  }
}
```

Start with the Basics, and Trust Nothing

- *IT IS ALWAYS BEST TO START SIMPLE!!!!!!!*
  - Verify that each part of the basic system runs as you build software.
  - Easier to debug – easier to exclude potential sources of problems, so bugs can be found faster
- Use an *incremental* approach to verify system works
  - Initialize ports for LCD_BACKLITE and switches
  - Test 1: Verify LCD_BACKLITE works
  - Test 2: Verify switches work – toggle LEDs based on polling switches
  - Test 3: Add interrupts and test they work
    - Can you control LCD_BACKLITE
    - Set breakpoints – do counts increase correctly?
- Use #defined symbols rather than “magic numbers” to help convey meaning of operation

Configure / Enable Interrupt

```
//-------------------------------------------------------------------
// Configure PORT 4
// Port 4 has only two pins
// SW1                  (0x01)     // Switch 1
// SW2                  (0x02)     // Switch 2
void Init_Port4(void){
  P4SEL0 = 0x00;                  // P4 set as I/O
  P4SEL1 = 0x00;                  // P4 set as I/O
  P4DIR = 0x00;                   // Set P4 direction to input

  P4SEL0 &= ~SW1;                 // SW1 set as I/O
  P4SEL1 &= ~SW1;                 // SW1 set as I/O
  P4DIR &= ~SW1;                  // SW1 Direction = input
  P4OUT |= SW1;                   // Configure pull-up resistor SW1
  P4REN |= SW1;                   // Enable pull-up resistor SW1
  P4IES |= SW1;                   // SW1 Hi/Lo edge interrupt
  P4IFG &= ~SW1;                  // IFG SW1 cleared
  P4IE |= SW1;                    // SW1 interrupt Enabled

  P4SEL0 &= ~SW2;                 // SW2 set as I/O
  P4SEL1 &= ~SW2;                 // SW2 set as I/O
  P4DIR &= ~SW2;                  // SW2 Direction = input
  P4OUT |= SW2;                   // Configure pull-up resistor SW2
  P4REN |= SW2;                   // Enable pull-up resistor SW2
  P4IES |= SW2;                   // SW2 Hi/Lo edge interrupt
  P4IFG &= ~SW2;                  // IFG SW2 cleared
  P4IE |= SW2;                    // SW2 interrupt enabled
}
//-------------------------------------------------------------------
```

Building a Program – the ISR

```
//-------------------------------------------------------------------------
// Port 4 interrupt. For switches, they are disabled for the duration
// of the debounce timer.  Flag is set that user space can check.

Include #pragma vector = [Assigned Vector]
#pragma vector=PORT4_VECTOR

Create Interrupt Service Routine Function with "__interrupt"
__interrupt void switch_interrupt(void) {

// Switch 1
  if (P4IFG & SW1) {
//  Set a variable to identify the switch has been pressed.
//  Set a variable to identify the switch is being debounced.
//  Reset the count required of the debounce.
//  Disable the Switch Interrupt.
//  Clear any current timer interrupt.
    P1OUT &= ~LCD_BACKLITE ;     // LCD_BACKLITE off to indicate boot ISR working
  }

// Switch 2
  if (P4IFG & SW2) {
//  Set a variable to identify the switch has been pressed.
//  Set a variable to identify the switch is being debounced.
//  Reset the count required of the debounce.
//  Disable the Switch Interrupt.
//  Clear any current timer interrupt.
    P1OUT |= LCD_BACKLITE ;        // LCD_BACKLITE on to indicate boot ISR working
  }

// Enable the Timer Interrupt for the debounce.
}
//-------------------------------------------------------------------------
```