Module 10. Operating Modes and More

Topics

- Revisiting interrupts
- Operating modes

ADC12 Interrupts

Our examples so far have used the `ADC12SC` bit to start conversions, and then we poll the `ADC12BUSY` bit to see when a conversion is complete:

```
ADC12CTL0 |= ADC12SC;


while (ADC12CTL1 & ADC12BUSY) {
    __no_operation();       // Could just leave body of loop empty
}


in_value = ADC12MEM0 & 0x0FFF;
```

This is another form of *busy-waiting*, which is like `swDelay`:

- The CPU isn't doing any useful work–it's just sitting in a loop!
- While ADC conversions happen very quickly (<< 1 ms), the CPU executes faster, so we do need to wait for a result

The main purpose of on-chip peripherals like the Timer and ADC is to remove burdens from or provide services to the CPU.

The solution to this is to use an **Interrupt**, an external signal sent by a peripheral requesting that the CPU do something. Interrupts are hard-wired into certain peripherals. Fortunately for us, the ADC12 can trigger interrupts.

On the ADC12, an interrupt can signal the end of a conversion, **meaning that it is ready for the CPU to read data from its memory registers**.

Here is an example configuration with two channels:

```
ADC12CTL0 = ADC12SHT0_9 | ADC12REFON | ADC12ON | ADC12MSC;
ADC12CTL1 = ADC12SHP | ADC12CONSEQ_1;

// Here, we are performing conversions for two channels
ADC12MCTL0 = ADC12SREF_0 + ADC12INCH_5;
ADC12MCTL1 = ADC12SREF_1 + ADC12INCH_6 + ADC12EOS;

// Because we are converting for two channels, we want the interrupt
// to occur after BOTH conversions are complete, so we enable the
// interrupt for MEM1.
ADC12IE = BIT1;

__enable_interrupt(); // Globally enable interrupts

ADC12CTL0 |= ADC12SC + ADC12ENC; // Enable ADC and start conversion
```

What should the ISR do? It's triggered when a conversion is finished, so it just needs to read the memory registers!

```
// Global variables for storing data
// (could also store into an array!)
volatile unsigned int in_value1, in_value2;

#pragma vector=ADC12_VECTOR
__interrupt void ADC12_ISR(void)
{

    // Move the results for both channels into global variables
    in_value1 = ADC12MEM0 & 0x0FFF;
    in_value2 = ADC12MEM1 & 0x0FFF;

}
```

"Scheduling" ADC measurements

It's also possible to have the ADC perform conversions automatically. An easier option is to trigger ADC conversions from a timer and use ADC interrupts to read the results.

```
// NOTE:  this example assumes the timer and ADC have already been configured.



volatile unsigned int in_value1, in_value2;


// Timer A2 ISR
#pragma vector=TIMER2_A0_VECTOR
__interrupt void Timer_A2_ISR(void)
{
    timer++;


    ADC12CTL0 |= ADC12SC;
}

// ADC 12 ISR
#pragma vector=ADC12_VECTOR
__interrupt void ADC12_ISR(void)
{
    in_value1 = ADC12MEM0 & 0x0FFF;
    // . . .
}

void main(void)
{
    setup_everything();
    _enable_interrupt();

    while(1){
        do_something_with_adc_values(in_value1, in_value2);
    }
}
```

Here, we now have two ISRs, one for the ADC, and one for the timer. Thus, our main program can simply use `in_value1` and `in_value2` without needing to explicitly start conversions.

Polling versus Interrupts Revisited

So far most of our code has relied heavily on *Polling*. While we have been using the timer to keep track of fixed time intervals, our main functions are still busy-waiting in some form of loop:

```
while (1)
{
   if (global_time_cnt > last_time)
   {
      take_ADC_meas();
      last_time = global_time_count;
   }
   button = checkButtons();
     . . .

   // Other task(s) can occur at different intervals
   if ((global_time_cnt % cnt_per_second) == 0)
   {
       toggleLED(LED2)
        displayHHMMSS(global_time_cnt);
    }
      . . .
}
```

This is not efficient: while it is waiting, the CPU is using precious energy to check if it needs to do something!

As an alternative, we can organize our code to *schedule* tasks to occur at specific "real" times, and even assign priorities to tasks.

--> We do this by implementing the Scheduler INSIDE Timer ISR!

```
// A simple task scheduler for the MSP430F5529
#pragma vector=TIMER2_A0_VECTOR
__interrupt void Timer_A2(void)
{
      // First priority:  maintain time base
      global_time_cnt++;


      // Some of the app task(s) may execute every time slice
      take_ADC_meas();
      button = checkButtons();

     . . .

     // Other application task(s) at different intervals
      if ((global_time_cnt % cnt_per_second) == 0)
      {
         toggleLED(LED2)
          displayHHMMSS(global_time_cnt);
     }
       . . . .

}
```

Here, main() would consist of initialization followed by an "empty" loop:

```
void main()
{
  WDTCTL = WDTPW + WDTHOLD;    // Stop watchdog timer
  init_sys();                  // Initialize the MSP430
  setupButton();              // configure buttons
  setupADC();                 // configure ADC

    . . .

  _enable_interrupt();       // Global Interrupt enable
  runTimerA2();              // Start scheduler

  // An empty forever loop!
  // All application tasks are scheduled and dispatched
  // from within TimerA2 ISR
  while(1)
  {
     __no_operation();
  }
```

What are we assuming by organizing or scheduling our application's tasks like this?

There some very important assumptions here:

Remember the rule of interrupts!

- We can have multiple interrupt sources for different peripherals—we want to maintain this behavior when scheduling tasks.

- By default, interrupts are disabled inside an ISR. Therefore, one interrupt cannot normally interrupt another one, but we can change this behavior when we need it.

Interrupt Priorities

Interrupt Vector Addresses

The interrupt vectors and the power-up start address are located in the address range 0FFFFh to 0FF80h. The vector contains the 16-bit address of the appropriate interrupt-handler instruction sequence.

Table 4. Interrupt Sources, Flags, and Vectors

INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
System Reset Power-Up External Reset Watchdog Timeout, Password Violation Flash Memory Password Violation	WDTIFG, KEYV (SYSRSTIV)(1)(2)	Reset	0FFFEh	63, highest
System NMI PMM Vacant Memory Access JTAG Mailbox	SVMLIFG, SVMHIFG, DLYLIFG, DLYHIFG, VLRLIFG, VLRHIFG, VMAIFG, JMBNIFG, JMBOUTIFG (SYSSNIV)(1)	(Non)maskable	0FFFCh	62
User NMI NMI Oscillator Fault Flash Memory Access Violation	NMIIFG, OFIFG, ACCVIFG, BUSIFG (SYSUNIV)(1)(2)	(Non)maskable	0FFFAh	61
Comp_B	Comparator B interrupt flags (CBIV)(1)(3)	Maskable	0FFF8h	60
TB0	TB0CCR0 CCIFG0(3)	Maskable	0FFF6h	59
TB0	TB0CCR1 CCIFG1 to TB0CCR6 CCIFG6, TB0IFG (TB0IV)(1)(3)	Maskable	0FFF4h	58
Watchdog Timer_A Interval Timer Mode	WDTIFG	Maskable	0FFF2h	57
USCI_A0 Receive or Transmit	UCA0RXIFG, UCA0TXIFG (UCA0IV)(1)(3)	Maskable	0FFF0h	56
USCI_B0 Receive or Transmit	UCB0RXIFG, UCB0TXIFG (UCB0IV)(1)(3)	Maskable	0FFEEh	55
ADC12_A	ADC12IFG0 to ADC12IFG15 (ADC12IV)(1)(3)(4)	Maskable	0FFECh	54
TA0	TA0CCR0 CCIFG0(3)	Maskable	0FFEAh	53
TA0	TA0CCR1 CCIFG1 to TA0CCR4 CCIFG4, TA0IFG (TA0IV)(1)(3)	Maskable	0FFE8h	52
USB_UBM	USB interrupts (USBIV)(1)(3)	Maskable	0FFE6h	51
DMA	DMA0IFG, DMA1IFG, DMA2IFG (DMAIV)(1)(3)	Maskable	0FFE4h	50
TA1	TA1CCR0 CCIFG0(3)	Maskable	0FFE2h	49
TA1	TA1CCR1 CCIFG1 to TA1CCR2 CCIFG2, TA1IFG (TA1IV)(1)(3)	Maskable	0FFE0h	48
I/O Port P1	P1IFG.0 to P1IFG.7 (P1IV)(1)(3)	Maskable	0FFDEh	47
USCI_A1 Receive or Transmit	UCA1RXIFG, UCA1TXIFG (UCA1IV)(1)(3)	Maskable	0FFDCh	46
USCI_B1 Receive or Transmit	UCB1RXIFG, UCB1TXIFG (UCB1IV)(1)(3)	Maskable	0FFDAh	45
TA2	TA2CCR0 CCIFG0(3)	Maskable	0FFD8h	44
TA2	TA2CCR1 CCIFG1 to TA2CCR2 CCIFG2, TA2IFG (TA2IV)(1)(3)	Maskable	0FFD6h	43
I/O Port P2	P2IFG.0 to P2IFG.7 (P2IV)(1)(3)	Maskable	0FFD4h	42
RTC_A	RTCRDYIFG, RTCTEVIFG, RTCAIFG, RT0PSIFG, RT1PSIFG (RTCIV)(1)(3)	Maskable	0FFD2h	41
Reserved	Reserved(5)		0FFD0h	40
			⋮	⋮
			0FF80h	0, lowest

(1) Multiple source flags
(2) A reset is generated if the CPU tries to fetch instructions from within peripheral space or vacant memory space. (Non)maskable: the individual interrupt-enable bit can disable an interrupt event, but the general-interrupt enable cannot disable it.
(3) Interrupt flags are located in the module.
(4) Only on devices with ADC, otherwise reserved.
(5) Reserved interrupt vectors at addresses are not used in this device and can be used for regular program code if necessary. To maintain compatibility with other devices, it is recommended to reserve these locations.

Operating Modes

What's the advantage of scheduling tasks?

>>Our CPU doesn't need to be running all the time!!

This means we can use the MSP430's operating modes to save energy when not performing tasks.

The MSP430F5529 has 6 Operating Modes:

Active Mode => "Normally Active" = CPU is active, all enabled clocks are active

Low Power Mode 0 => CPU, MCLK are disabled , SMCLK , ACLK are active

Low Power Mode 1 => CPU, MCLK, DCO osc. are disabled , DC generator is disabled if the DCO is not used for MCLK or SMCLK in active mode, SMCLK, ACLK are active

Low Power Mode 2 => CPU, MCLK, SMCLK, DCO osc. are disabled , DC generator remains enabled, ACLK is active

Low Power Mode 3 => CPU, MCLK, SMCLK, DCO osc. are disabled , DC generator disabled, ACLK is active

Low Power Mode 4 => CPU and all clocks disabled (RAM retention mode)

Low Power Mode 4.5 => CPU and all clocks disabled (no RAM retention), PWR management off, Digital IO pin configuration retained

Operating Mode Basics

>> Most embedded systems run on batteries

-- Enter Low Power Mode to conserve batteries

On MSP430F5529 Low Power Modes are indicated/selected thru the CPUOFF, OSCOFF, SCG0, and SCG1 bits in the *Status Register (SR)*

Mode	***SCG0***	***SCG0***	***OCSOFF***	***CPUOFF***
Active	0	0	0	0
LPM0	0	0	0	1
LPM1	0	1	0	1
LPM2	1	0	0	1
LPM3	1	1	0	1
LPM4	1	1	1	1
LPM4.5 *	1	1	1	1

*PMMREGOFF bit = 1, as well

Status Register Layout:

Reserved	SCG1	SCG0	OSCOFF	CPUOFF	GIE	N	Z	C

How do you enter a Low Power Mode?
Can set with the `_BIS_SR()` function:

Ex:
```
_BIS_SR(LPM3_bits) ;       // set bits to enter LMP3

_BIS_SR(LPM3_bits|GIE) ;  // set bits to enter LMP3
                          // and enable interrupts
```

When to enter LPM and how do you exit?

--> Ideally want to enter LPM whenever not executing tasks

--> This is made simple if program is organized as a "Scheduler"

--> Enter LPM after starting timer in `main()`

```
// A simple task scheduler for the MSP430F5529
#pragma vector=TIMER2_A0_VECTOR
__interrupt void Timer_A2(void)
{
      // First priority = maintain the time base
      global_time_cnt++;

      // Some app task(s) may execute every time slice
      take_ADC_meas();
      button  =checkButtons();
     . . .

     // Other application task(s) execute less frequently
      if ((global_time_cnt % cnt_per_second) == 0)
          displayHHMMSS();
      . . . .

}
```

How do you exit LPM if CPU is OFF?

>> **INTERRUPTS!**

When an interrupt is received from a certain source, the CPU automatically does the following:

1. Finishes its current (assembly) instruction
2. Saves Status Register (SR) and Program Counter (PC) to stack
3. Clears Status Register (set to 0)
4. Loads address of ISR that was triggered from Interrupt Vector Table (IVT), loads it into PC
5. Execution continues in ISR
6.

Returning from an ISR restores the Status Register to its previous values meaning that your program will <u>automatically return to Low Power Mode</u> when exiting the ISR!

```
void main(void)
{
   WDTCTL = WDTPW | WDTHOLD;

   init_sys();     // initialize system
   _BIS_SR(GIE);

      . . .
   runtimerA2();   // task scheduler runs in Timer A2 ISR
      . . .
   _BIS_SR(LPM0_bits|GIE);  // Enter low power mode
    // Arrival of timer interrupt will cause MSP430 to exit low
    // power mode and enable it to execute all tasks within the
    // Timer A2 ISR

    while(1) {
      // Main program loop does nothing!
    }

}
```

Which LPM you enter depends on which clocks you are using for your scheduling Timer.

-- Also, on how quickly you need other clocks

>> Low Power Modes are great in practice, but can make debugging painful!

-- It's simple so add it last when debugging is complete

-- Then test to see if '430 is waking and executing tasks properly

Computing Power Usage: Example

An MSP430F5529 is being powered by a Duracell 2/3A $LiMnO_2$ battery, which has a nominal voltage of 3V and a listed capacity of 1550 mAh.

The MSP430F5529 is running an application that is in Active Mode for 5.6 % of the time, in LPM0 for 16.7% of the time, and in LPM4 for the rest of the time.
How long can the application run using this battery?