```
//*******************************************************************************************
//  MSP430 ULP Advisor Demo - Worst Case Example
//
//  Description: This code is for demonstration purpose only.
//  The MSP430 stays in active mode all the time. Thus showing lots of things
//  one can do wrong in terms of low power consumption by not using low power
//  modes and intelligent peripherals.
//
//  Comment: Have a look at the Good Example to learn good
//
//               MSP430G2553
//            -----------------
//        /|\|              XIN|-
//         | |                 | 32kHz
//         --|RST          XOUT|-
//  Button_   |                 |
//       - -->|P1.3         P1.0|-->LED1
//            |             P1.6|-->LED2
//            |             P1.1|<--RXD    Serial UART
//            |             P1.2|-->TXD    Interface
//
//  This software has been placed into the Public Domain by Texas Instruments
//  Incorporated and is example code only  THIS SOFTWARE IS PROVIDED BY TEXAS
//  INSTRUMENTS INCORPORATED"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
//  INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY
//  AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL TEXAS
//  INSTRUMENTS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,  SPECIAL,
//  EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT  LIMITED TO,
//  PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
//  OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
//  WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
//  OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
//  ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
//
//  Wolfgang Lutsch
//  Texas Instruments, Inc
//  Mai 2012
//  Built with Code Composer Studio v5
//*******************************************************************************************

#include <string.h>
#include <msp430g2233.h>

// 2 application state
#define APPSTATE_IDLE    0
#define APPSTATE_MEASURE 1

// global variable holding the current application state
unsigned short app_state = APPSTATE_IDLE;

// defines for button and LEDs
#define BUTTON BIT3
#define LED1 BIT0
#define LED2 BIT6

// structure definition for measurement values
#define MEASUREMENT_INFO_SIZE 14
union MEASUREMENT_INFO {
       unsigned char b[MEASUREMENT_INFO_SIZE];
       struct {
              unsigned short A;
              unsigned short aA;
              unsigned short B;
              unsigned short aB;
              unsigned short F;
              unsigned short aF;
              unsigned short x;
       } rs;
};
typedef union MEASUREMENT_INFO MEASUREMENT_INFO_T;

// ringbuffer definition to hold values for average calculation
#define RINGBUFFERSIZE 16
unsigned short RingBuffer[RINGBUFFERSIZE] = {0};    // init with 0
unsigned char Idx = 0;

MEASUREMENT_INFO_T cr;           // current measurement record
MEASUREMENT_INFO_T tr;           // transmit measurement record

MEASUREMENT_INFO_T ProcessCurrentSample(unsigned short smpl, MEASUREMENT_INFO_T x)
{
       int i;

       RingBuffer[Idx++] = smpl;    // add new value to ring buffer, increment buffer index

       if(Idx >= RINGBUFFERSIZE)    // handle buffer overflow
       {
              Idx = 0;                 // roll over index when buffer is full
       }
```

```
	x.rs.A = smpl;
	// Temperature in Celsius
	// oC = ((A10/1024)*1500mV)-986mV)*1/3.55mV = A10*423/1024 - 278
	x.rs.B = ((smpl - 673) * 423) / 1024;

	// Temperature in Fahrenheit
	// oF = ((A10/1024)*1500mV)-923mV)*1/1.97mV = A10*761/1024 - 468
	x.rs.F = ((smpl - 630) * 761) / 1024;

	// calculate average value
	x.rs.aA = 0;				// set to 0

	for(i = 0; i < RINGBUFFERSIZE; i++)
	{
		x.rs.aA += RingBuffer[i];	// accumulate stored values
	}
	x.rs.aA /= RINGBUFFERSIZE;		// divide by number of values

	// Average temperature in Celsius
	x.rs.aB = ((x.rs.aA - 673) * 423) / 1024;
	// Average temperature in Fahrenheit
	x.rs.aF = ((x.rs.aA - 630) * 761) / 1024;

	x.rs.x++;				// increment sample counter

	// finally copy current record to transmit buffer
	memcpy(tr.b, x.b, MEASUREMENT_INFO_SIZE);

	return x;
}

void SerialCom(MEASUREMENT_INFO_T y)
{
	unsigned short i;

	// Send the measurement results
    for(i = 0; i < MEASUREMENT_INFO_SIZE; i++)
    {
       while (!(IFG2&UCA0TXIFG));	// USCI_A0 TX buffer ready?
        UCA0TXBUF = y.b[i];		// TX -> struct character
    }
}

int main(void) {

	// Configure Watchdog
	WDTCTL = WDTPW + WDTHOLD;	// Stop watchdog timer

	// Configure GPIO pins for User Buttons
//	P1OUT &= ~(LED1 + LED2);	// Init LED1 and LED2 to be off - only for visual testing
	P1DIR = ~BUTTON;		// Button (P1.3) is input for Button
	P1OUT = BUTTON;			// Select pullup for button
	P1REN = BUTTON;			// Enable pullup for button

	// Configure UART for Host Communication
	BCSCTL1 = CALBC1_1MHZ;			// Set DCO
	DCOCTL = CALDCO_1MHZ;
	P1SEL = BIT1 + BIT2 ;			// P1.1 = RXD, P1.2=TXD
	P1SEL2 = BIT1 + BIT2 ;			// P1.1 = RXD, P1.2=TXD
	UCA0CTL1 |= UCSSEL_2;			// SMCLK
	UCA0BR0 = 104;				// 1MHz 9600
	UCA0BR1 = 0;				// 1MHz 9600
	UCA0MCTL = UCBRS0;			// Modulation UCBRSx = 1
	UCA0CTL1 &= ~UCSWRST;			// **Initialize USCI state machine**

	// main loop
	while(1)
	{
		{ // Start of Button Handling
			if(!(P1IN & BUTTON))	// Check if BUTTON_1 is pressed (low input signal)
			{
				int i,j = 0;

				for(i = 0; i < 10; i++)	// Debounce delay loop....
				{
					__delay_cycles(25000);	// Delay
					if(!(P1IN & BUTTON))	// do a majority vote of the button-pressed time
					{
						j++;	// count up
					}
					else
					{
						j--;	// count down
					}
				}

				if(j > 0)	// Check if button was really pressed fro a certain time
				{
//					P1OUT ^= LED1;	// Toggle LED1 to visualize that button press was detected
					// if application is in IDLE state
```

```
                                if(app_state == APPSTATE_IDLE)
                                {
                                       // change to MEASURE state
                                       app_state = APPSTATE_MEASURE;

                                       cr.rs.x = 0;              // reset sample counter

                                       // Initialize ADC10
                                       ADC10CTL1 = INCH_10 + ADC10DIV_3;         // Temp Sensor ADC10CLK/4
                                       ADC10CTL0 = SREF_1 + ADC10SHT_3 + REFON + ADC10ON + ADC10IE;

                                       // and configure Timer for 1 second interval events
//                                     CCR0 = 0xFFFF;                // For testing without ACLK
//                                     TACTL = TASSEL_2 + MC_1;         // SMCLK, upmode
                                       CCR0 = 32768-1;
                                       TACTL = TASSEL_1 + MC_1;         // ACLK, upmode
                                }
                                else // otherwise, if application is in MEASURE state...
                                {
                                       P1OUT &= ~LED2;// Turn off LED2 to visualize no measurement activity
                                       // ...change state back to IDLE state
                                       app_state = APPSTATE_IDLE;

                                       // and de-activate Timer
                                       TACTL = 0;
                                }
                         }
                  }
            } // End of Button Handling

            // do some measurement if the application is in APPSTATE_MEASURE state
            if(app_state == APPSTATE_MEASURE)
            {
                  //while(TA0CCTL0 & CCIFG);         // a blocking empty loop <-- would be flagged by ULP Advisor
                  if(TA0CCTL0 & CCIFG)  // Check for Timer Event <-- take care!! if clause in main loop will not be flagged
                        // but is not energy efficient as CPu stays in active mode and constantly polls the flag
                  {
//                      P1OUT ^= LED2;              // Toggle LED2 to visualize measurement activity

                        TA0CCTL0 &= ~CCIFG;           // Reset Timer event

                     ADC10CTL0 |= ENC + ADC10SC;          // Sampling and conversion start

                        while(!(ADC10CTL0 & ADC10IFG));     // poll for conversion result ready

                        // pass current sample (ADC10MEM) to processing handler
                        // reading conversion result - automatically resets ADC10IFG
                        cr = ProcessCurrentSample(ADC10MEM, cr);

                        SerialCom(tr);         // transmit current measurement results via UART
                  } // if(TimerEvent)
            } // if(APPSTATE_MEASURE)
      } // end of while(1) main loop
} // main()
```