Wireless Remote Oscilloscope Probe

Team Name: Scope-A

May 1, 2019

Joseph Chen	josephch@mtu.edu
Andrew Kirkum	adkirkum@mtu.edu
Samuel Solverson	sdsolver@mtu.edu

TABLE OF CONTENTS

1 INTRODUCTION

The Wireless Remote Oscilloscope Probe uses two MSP430 chips to allow for wireless analysis of an analog signal [1]. One of the MSP430 chips is connected to the waveform and transmits data describing that waveform to a receiving MSP430. The receiving MSP430 then sends this data to a Digital to Analog Converter (DAC) which reconstructs the signal. Both the original and reconstructed signals are displayed on an oscilloscope.

2 REQUIRED MATERIALS

Prior to beginning the development process, certain materials were procured. Some materials were not immediately necessary, but all were useful to test theories or chunks of code to reinforce/invalidate assumptions when available. Both Agilent products (the waveform generator and oscilloscope) were housed in a University Lab and not always available for use, so early testing and debugging was sometimes performed without them.

Item	Use	Resource(s)
(2) TI eZ430-RF2500 cards	Card which housed the MSP430 microcontroller and cc2500 transceiver chip (one for transmitting digital value acquired by ADC and one for receiving digital value and sending to DAC)	[2] [3] [4] [5]
(1) PC with IAR Workbench	Testing/debugging code on MSP430s[1]	[6]
(1) USB Debugger	Powering/downloading code on MSP430s	[4]
(2) Battery Packs	Powering MSP430s	[4]
(1) AD5320 Digital to Analog Converter (DAC)	Serial Peripheral Interface (SPI) Slave which interprets a digital signal and outputs a voltage (analog signal)	[7]

[1] Throughout the report, the board which houses both the cc2500 transceiver chip and the MSP430 microcontroller (the eZ430-RF2500) is referred to as an MSP430. It is understood that the eZ430-RF2500 card has more functionally than an isolated MSP430, but the device which is programmed by the IAR software is the MSP430 and therefore this abstraction is justified.

(1) Agilent 33220A 20 MHz Waveform Generator	Device used to produce an analog waveform which will be sensed by an MSP430's ADC and translated to a digital signal	[8]
(1) Agilent InfiniiVision 3000 X-Series Oscilloscope	Device used to view the original signal produced by the waveform generator and the signal produced by the DAC	[9]
(1) Protoboard	Interface to wire the DAC to the receiving MSP430	-

3 INITIAL STRATEGY

The Wireless Remote Oscilloscope Probe project was subdivided into sections to enable parallel development and to minimize debug time. The project was broken up in such a way so that only one functional piece of the project changed between any two steps. Prior to adding the next functional component, it was imperative that the current component was debugged thoroughly to minimize the likelihood of errors propagating to future steps.

The first major step was to convert an analog signal into a digital signal using the ADC on the MSP430 and reconstruct the signal using a DAC connected to the same MSP430. This abstraction removed any problems involving communication between two MSP430 boards. This also provided the team with an opportunity to view "ideal" behavior of an ADC wired to a DAC and allowed the team to understand the bandwidth of frequencies that could be translated with this setup. This was then tested with the MSP430 attached to a battery pack.

The next step was to split up the ADC and DAC code between two MSP430 boards. The boards would communicate using a Serial Peripheral Interface (SPI) communication protocol. Due to an equipment constraint, the team had only one USB debugger, so *at least* one of the two MSP430 boards would have to operate on battery power. The MSP430 using the ADC (the *transmitter*) operated on battery power while the MSP430 connected to the DAC (the *receiver*) was plugged into the computer. Upon completing this step, the team had a working project design *before* implementing the wireless functionality. Any issues encountered would (most likely) be due to problems with the wireless implementation.

Finally, wireless communication was implemented to complete the project requirements and to create a wireless oscilloscope probe. Again, the transmitting MSP430 operated on battery power while the receiving MSP430 board attached to the computer. When proper wireless functionality was observed, the receiving MSP430 was disconnected from the computer and attached to the battery pack.

4 OVERHEAD RESEARCH

4.1 ADC-DAC: Two Chips via SPI Alternative

In the team's "Initial Strategy," the plan was to setup an MSP430 to sample a value of the input signal using the ADC and act as a Master to send this value to a Slave MSP430 via SPI. The Slave MSP430 would then receive the data, become a Master itself, and transfer data to the Slave DAC. It was decided that this abstraction would likely introduce more problems than it would mitigate, so the step was de-scoped. Instead, thorough testing of the cc2500s and the wireless functionality would take place prior to implementing wireless into the system.

The testing that was performed was essentially re-doing Lab 8*: Wireless Comm via the cc2500* [10] and tweaking the code to establish a deeper understanding of how wireless works with the cc2500.

4.2 Component Functionality Rates

In order to reproduce waveforms with different geometries at a reasonable resolution, the sample rate of the ADC, baud rate of the cc2500 RF transceiver, and baud rate of the DAC must be fully understood to identify the bottleneck. The maximum sampling rate of *this* system is based on the Interrupt Service Routine (ISR) on an MSP430 whose functional speed is the slowest. In order to ensure that each ADC sample is not overrun by the next, each sample is taken and then immediately sent to the DAC via SPI or wireless depending on the configuration of the system.

4.2.1 ADC

The original analog waveform needs to be translated into a digital representation of the signal. The ADC on the MSP430 represents a voltage as a 10-bit integer: the ratio of the input voltage over the difference in reference voltages. In this project, the high reference voltage was V_{CC} and the low reference was ground.

The ADC clock source can be configured by the ADC Oscillator (ADC10OSC: @ ≈5 MHz), Auxiliary Clock (ACLK: defined by the "Very-Low-Power Low-Frequency Oscillator" (VLO) @ ≈12 KHz), Master Clock (MCLK: defined by the Digitally Controlled Oscillator (DCO) @ = 1.15 – 16 MHz), or the Sub-Master Clock (SMCLK: defined by the DCO = 1.15 – 16 MHz) [3]. In order to isolate the ADC clock from the SMCLK or the MCLK, it was decided to use the ADC10OSC. The ADC will capture a 10-bit value which represents the input analog signal sensed over the minimal "sample-and-hold" time period of four ADC10 clock cycles. It takes another thirteen ADC10 clock cycles to convert that sample to a usable value. Upon completing the conversion, the value will be sent to either the DAC immediately over SPI or over wireless to the other MSP430.

With an overhead of 17 cycles at the ADC10OSC's 5 MHz, the ADC the *theoretical* maximum frequency of samples produced by the ADC is:

$$f_{ADC} = \frac{1\ sample}{17\ cycles} * \frac{5 \times 10^6\ cycles}{1\ sec} = 294.12\ kHz\ or\ \tau_{ADC} = 3.4\mu s$$

4.2.2 UCIB0 SPI

Both the cc2500 and the DAC communicate via SPI. Per the specifications located on the data sheet of the AD5320 [7], the maximum SCLK frequency for the DAC is 30 MHz – well above the maximum value the DCO can output. The cc2500 lists that its maximum frequency it can operate at via SPI is 6.5 MHz with burst accesses to different registers on the chip. Due to the difficult nature of wireless on the MSP430s, it was decided that 500 kHz would be the rate which SPI will communicate at given the success of Lab 8 running at that frequency. With a data rate of 1 Mbps and the two byte word needed to be transferred between either the cc2500 or DAC, the *theoretical* maximum frequency of samples transmitted/received via SPI is:

$$f_{SPI} = \frac{1\ sample}{2\ bytes} * \frac{1\ byte}{8\ bits} * \frac{500 \times 10^3\ bits}{1\ sec} = 31.25\ kHz\ or\ \tau_{SPI} = 32\mu s$$

4.2.3 cc2500 Tx

If the data is sent via the cc2500 transceiver chip, much more overhead is required. The transceiver has the ability to configure its bit-rate up to 500 kbps; however, due to increased bit error rate (BER) at maximum bit rate, the transceiver will be configured at 250 kbps.

Based on Figure 11 in the cc2500 User's Guide [2], the wireless transceiver will transmit:

$$4\ bytes\ (Preamble) + 4\ bytes\ (Sync) + 1\ byte\ (Length) + 2\ bytes\ (Data) + 2\ bytes\ (CRC) = 13\ total\ bytes$$

The payload efficiency is then $\frac{2\ data\ bytes}{13\ total\ bytes} = 15.38\%$.

Preamble bits (MDMCFG1)	Sync word (MDMCFG2)	Length field (PKTLEN)	Address field (PKTCTRL1)	Data field	CRC (PKTCTRL0)
4 bytes	4 bytes	1 byte	0 bytes	2 bytes	2 bytes

Figure 1. cc2500 Packet Format [2] Based on Values from wireless_v2.h2

With a data rate of 250 kbps and the 10-bit (housed in two bytes) digital signal from the ADC, the *theoretical* maximum frequency of samples transmitted (not including SPI transfer times) by the cc2500 is:

$$f_{TX\_wireless} = \frac{1\ sample}{13\ bytes} * \frac{1\ byte}{8\ bits} * \frac{250\times10^3\ bits}{1\ sec} = 2.40\ kHz$$

Including the *minimum* SPI transfer time:

$$\tau_{TX\_wireless} = \tau_{TX\_SPI} + \frac{1}{2.40\ kHz} = 32\mu s + 417\mu s = 449\mu s\ or\ f_{TX\_wireless} = 2.23kHz$$

4.2.4 cc2500 Rx

The receiver can only receive the samples sent, so the theoretical maximum frequency of samples received by the cc2500 is also 2.40 kHz (not including SPI transfer times).

Including the *minimum* SPI transfer time:

$$\tau_{RX\_wireless} = \tau_{RX\_SPI} + \tau_{TX\_wireless} = 32\mu s + 449\mu s = 481\mu s\ or\ 2.08kHz$$

[2] This is a provided file that has predefined methods to interact with the cc2500 card as well as set it up properly.

4.2.5 AD5320 DAC

The AD5320 is a 12-bit DAC that receives a digital value and decodes this value into a voltage between ground and V_{CC}. The DAC is external to the MSP430s, unlike the ADC.

If the data is immediately sent to the DAC, the ISR will only transmit two bytes via SPI to the DAC which has a maximum sample transmission rate of 62.5 KHz [3].

5 PROCEDURE

5.1 Physical Interface

5.1.1 Wiring ADC-DAC: Single MSP430

The MSP430 requires the ability to receive an input for the ADC and to transmit the data via SPI to the DAC.

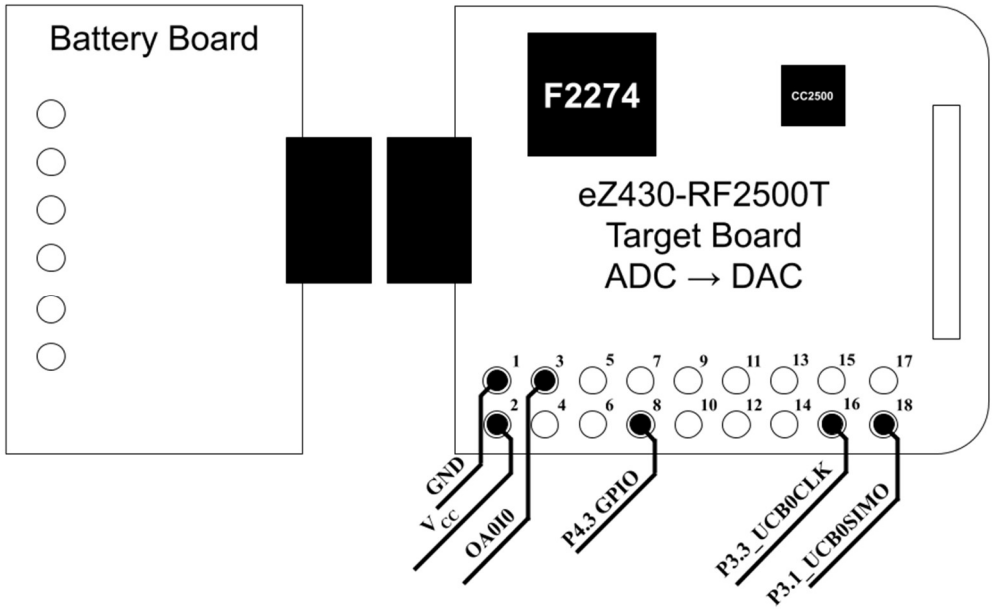

Figure 2. Wiring of the MSP430 for Single Chip use

Pin	Description
1: GND	Ground to Pin 2 of DAC
2: V_{CC}	Power to Pin 3 of DAC
3: 0A010	ADC Channel 0

8: P4.3	General Purpose I/O (GPIO) pin used for Slave Select (SS) - Pin 6 of DAC
16: UCB0CLK	SCLK to Pin 5 of DAC
18: UCB0SIMO	MOSI to Pin 4 of DAC

5.1.2 Wiring ADC-DAC: Two MSP430s via Wireless

The MSP430 transmitter requires the ability to receive an input for the ADC and to transmit the data via the cc2500 transceiver. Since the cc2500 transceiver is hard-wired to USCI_B0, the ADC shall be connected to other pins that shall not interfere with the USCI_B0 bus (P3.0 – P3.3).

MSP430 transmitter is wired as shown below:

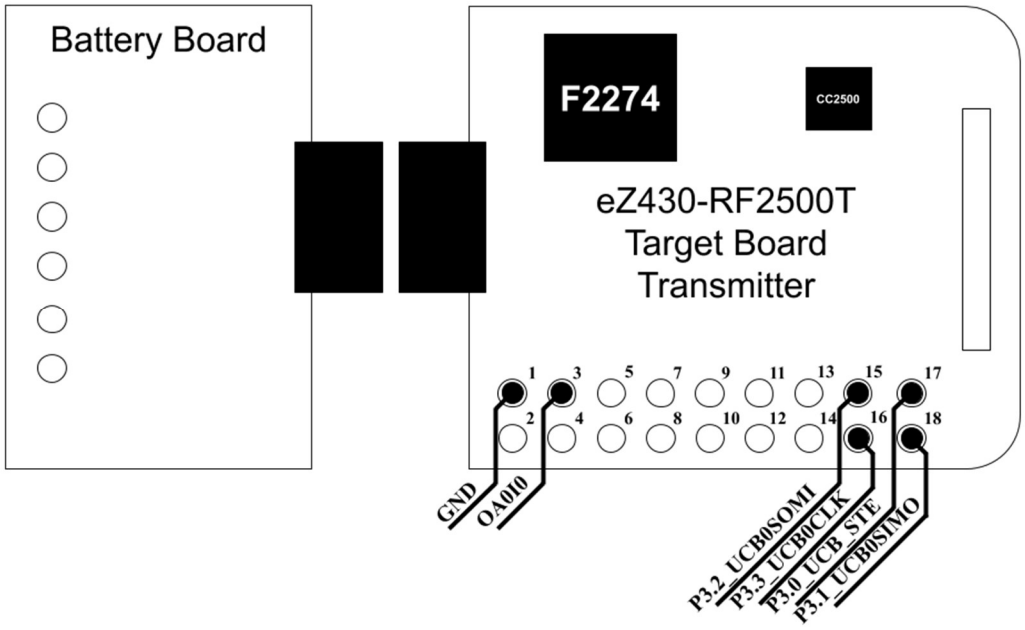

Figure 3. Wiring of the MSP430 Transmitter

Pin	**Description**
1: GND	Ground
3: 0A010	ADC Channel 0
15: UCB0SOMI	Internal connection to cc2500
16: UCB0CLK	Internal connection to cc2500

17: UCB_STE	Internal connection to cc2500
18: UCB0SIMO	Internal connection to cc2500

MSP430 receiver is wired as shown below:

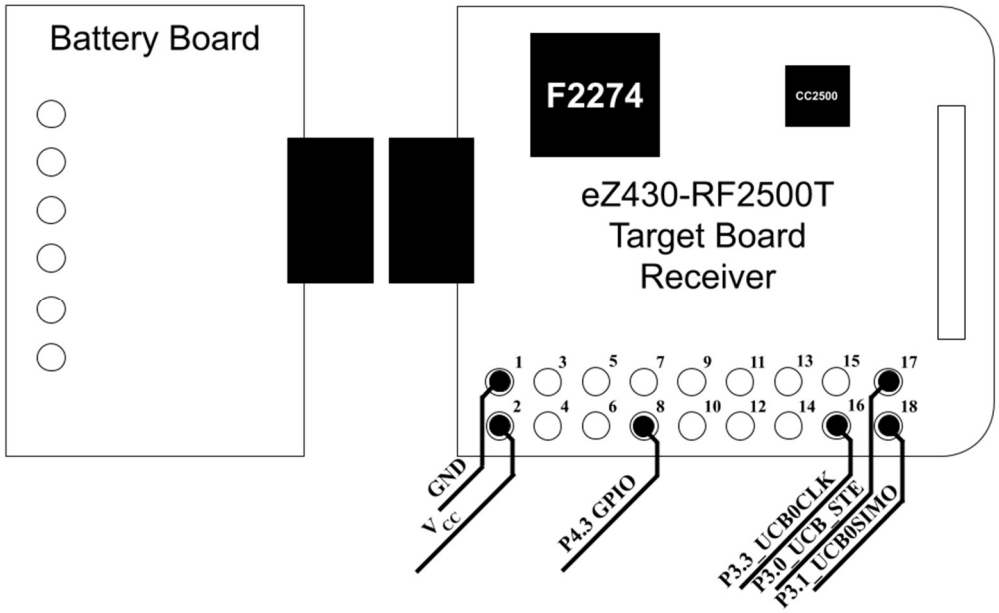

Figure 4. Wiring of the MSP430 Receiver

Pin	**Description**
1: GND	Ground to Pin 2 of DAC
2: V_{CC}	Power to Pin 3 of DAC
8: P4.3	General Purpose I/O (GPIO) pin used for Slave Select (SS) - Pin 6 of DAC
16: UCB0CLK	SCLK to Pin 5 of DAC and internal connection to cc2500
17: UCB_STE	Internal connection to cc2500
18: UCB0SIMO	MOSI to Pin 4 of DAC and internal connection to cc2500

5.2 Testing/Development of Code

Nearly all code used throughout the project is based on components used during the lab portion of EE4737: Embedded System Interfacing. The majority of debugging/development was understanding how to combine separate elements employed throughout the course.

5.2.1 ADC-DAC: Single Chip

The code behind the first step of the design process was based almost exclusively on the code developed from Lab 6: *MSP430 to DAC Chip via SPI* [11] and Lab 4: *MSP430 – ADC10 Converter* [12]. There were no surprises during the development and testing of this, however, the single chip configuration used P3.0 as the Slave Select for the DAC -- the same pin which is used as the Slave Select for the cc2500. No issues were encountered during this time because the cc2500 had not been implemented yet.

5.2.2 ADC-DAC: Two Chips via Wireless

The next step was to separate the DAC code from the ADC code, make sure all pins that were needed for communication were enabled, and implement wireless transmission on the transmitting chip and wireless reception on the receiving chip. The code behind the wireless communication implementation was based on code from Lab 8*: Wireless Comm via the cc2500* [10].

Through testing, it was discovered that it takes approximately 1.3 ms for the transmitter to complete its ISR. The transmitting MSP430's ISR is triggered when the ADC has completed its sampling and the value is ready to be read. This means that it takes nearly 1.3 ms just to send the data -- nearly three times longer than the minimum calculated in 4.2.3.

Once the message is received by the receiving MSP430, the ISR is triggered. It takes approximately 37.8μs to transfer the data from the MSP430 to the DAC. This is only 18% slower than the expected value calculated in 4.2.2.

6 RESULTS

Due to the hardware limitations in the MSP430s, the wireless oscilloscope probe has a bandwidth from 0 to 25 Hz before the fidelity of the reconstructed signal is considerably degraded. In Figures Figure 5 and Figure 6, the probe is capable of rendering the signal with high fidelity in reference of the original signal; however, at 25 Hz, there is a significant loss in fidelity due the bottleneck in

the cc2500 transceiver. This limitation can be seen in Figures Figure 7, Figure 8, and Figure 9. In Figure 7, the sampling rate of the waveform can be seen. The reconstructed waveform has voltage steps instead of the smooth curve of the original waveform. Figures Figure 8. Square Wave at 25 Hz, 0 – 2.4V, 80% Duty Cycle and Figure 9 show a delay for the conversion and transmission of the signal. The measured delay is 2.6173ms, which means that the reconstructed waveform is slightly out of phase with the original, posing a potential problem in applications which have timing constraints. Using only batteries to power both MSP430s, the probe has the ability to input and output from 0 to 2.7 V (given that the batteries are at maximum capacity). If the original waveform was greater than 2.7V or lower than 0V, the reconstructed wave would saturate to those bounds, creating flat areas on the signal. While the initial proposal only required for sinusoidal and square waveforms to be outputted, the probe exceeds this requirement by being capable of reconstructing both sawtooth and modulated PSK waveforms.

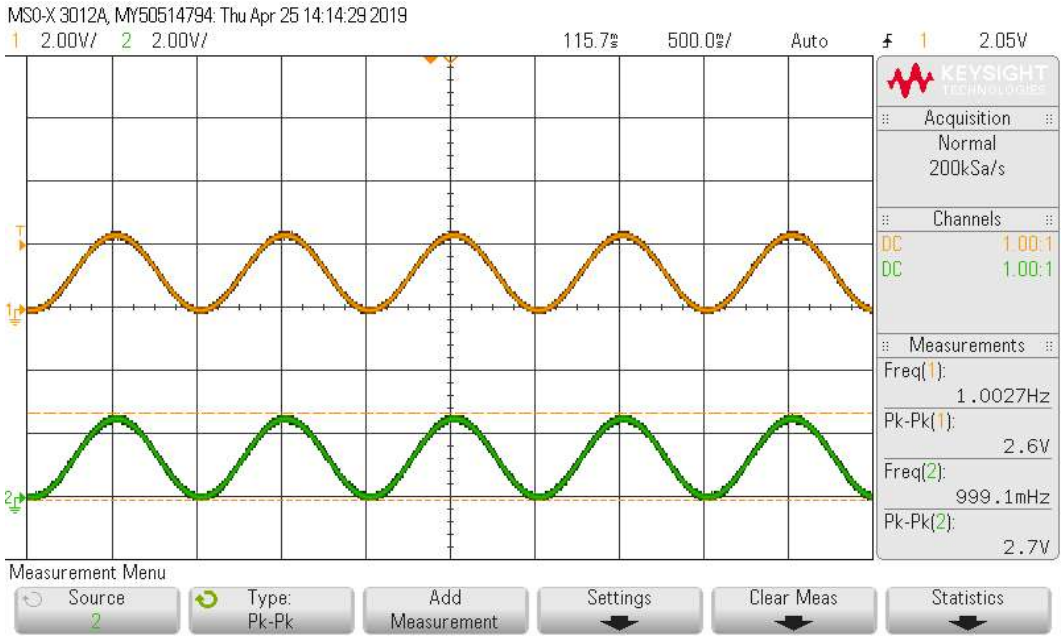

Figure 5. Sine Wave at 1 Hz, 0 – 2.4V, 80% Duty Cycle. Channel 2 is the Reconstructed Signal.

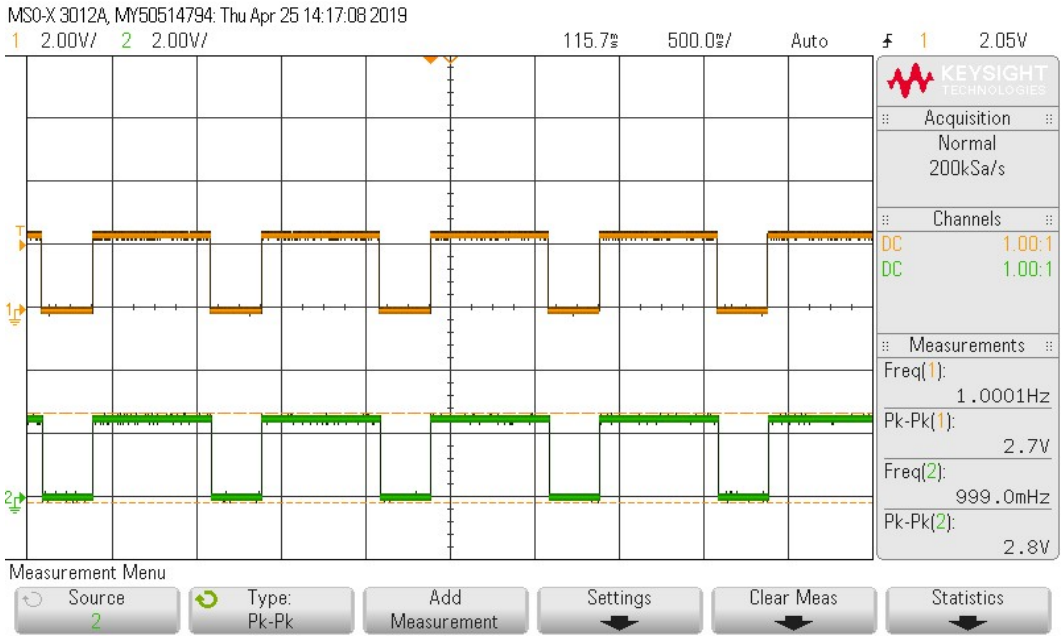

Figure 6. Square Wave at 1 Hz, 0 – 2.4V. Channel 2 is the Reconstructed Signal.

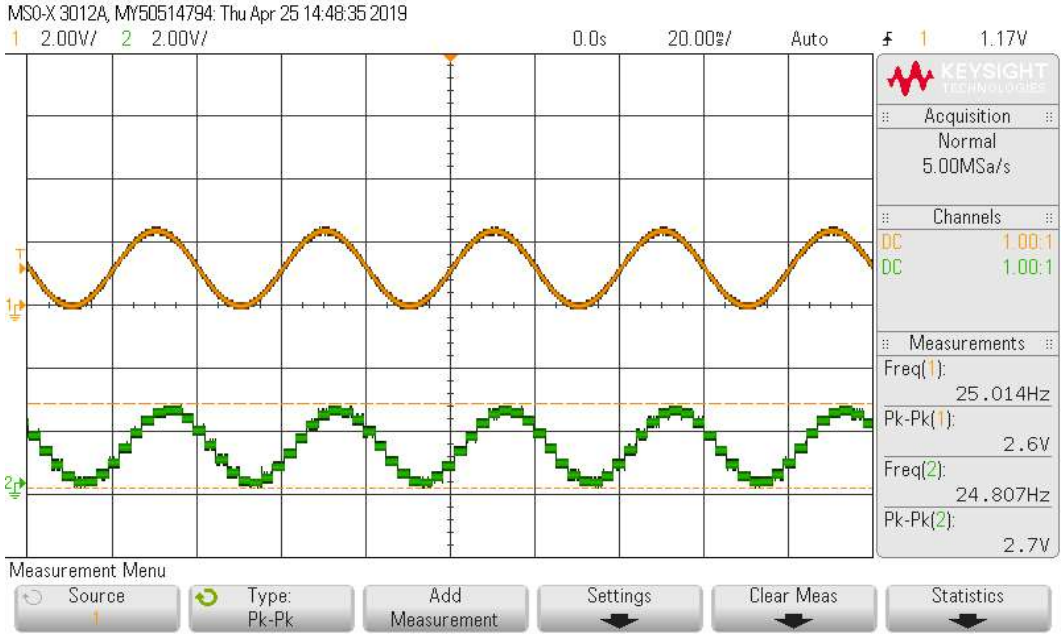

Figure 7. Sine Wave at 25 Hz, 0 – 2.4V. Channel 2 is the Reconstructed Signal.

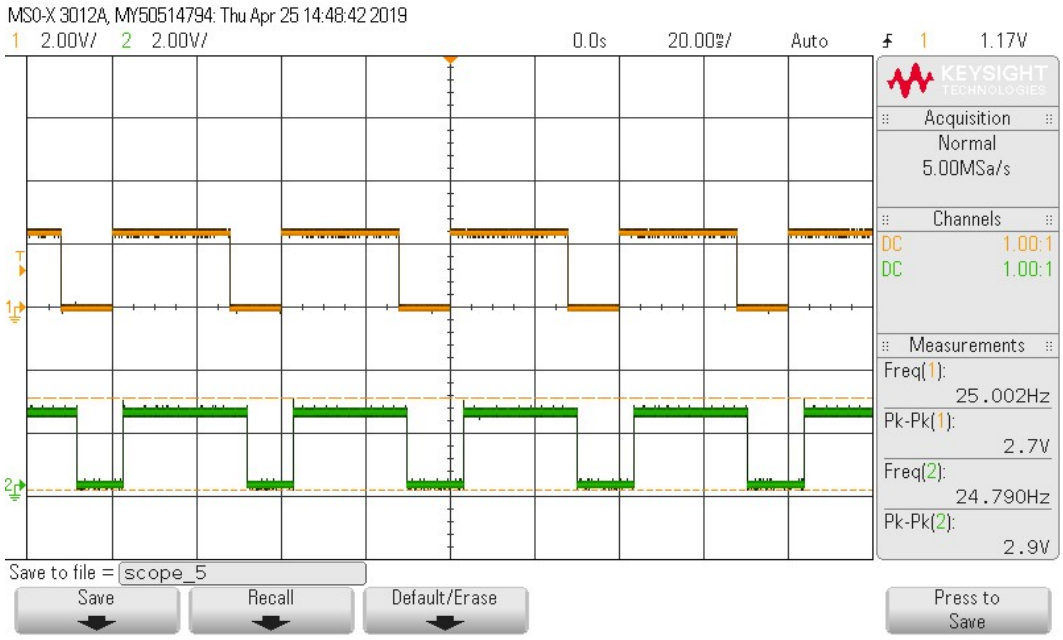

Figure 8. Square Wave at 25 Hz, 0 – 2.4V, 80% Duty Cycle. Channel 2 is the Reconstructed Signal

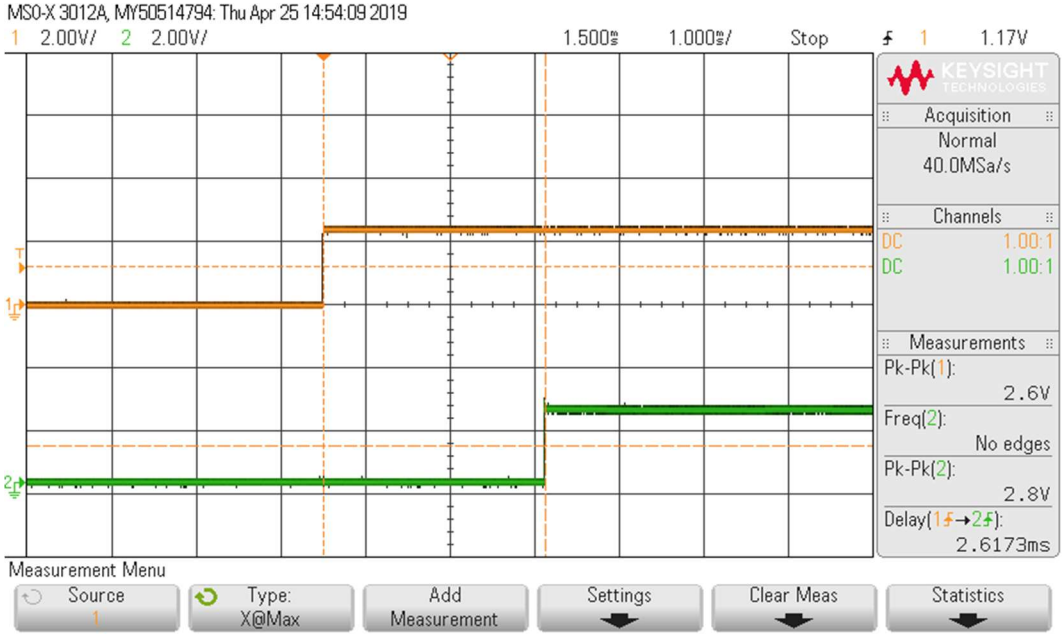

Figure 9. Delay Capture at 25 Hz, 0 – 2.4V. Channel 2 is the Reconstructed Signal.

7 LESSONS LEARNED

During the testing phase of the system, random "jumps" and other irregularities occurred as shown in Figure 10. Other irregularities included an automatic scaling of values which caused normal 12-bit values like 0x07FF to completely saturate the DAC and 0x0FFF to output a 0V signal.

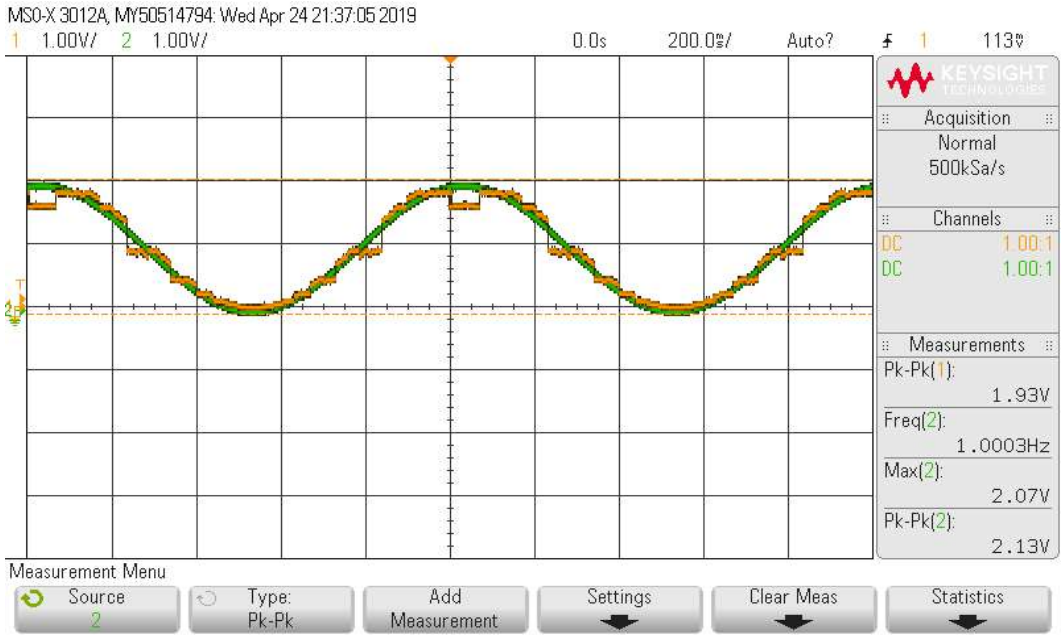

Figure 10. Inconsistent Sine Wave at 1 Hz, 0 – 2V. Channel 1 is the Reconstructed Signal.

After analysis of the symptoms, the cause of the scaling was discovered to involve inconsistencies of the phase and polarity between the devices on the USCI_B0 SPI bus. In the function TI_CC_SPISetup(void), located in *wireless-v2.h,* the USCI_B0 SPI bus is configured to have a clock phase and clock polarity of one and zero, respectively—the data is captured on the leading edge (rising edge), and the data is changed on the trailing edge (falling edge) [3]. While this setup supports the functionality of the cc2500, it contradicts the timing diagram in the DAC's datasheet, which has a phase and polarity of zero and zero—the DAC captures input values on the trailing edge (falling edge), and the data is changed on the leading edge (rising edge). This inconsistency did not affect the transmission by the MSP430, but affected how the data was read by the DAC. Before writing to the DAC, the UCB0CTL0 register must be modified to have a phase of zero to mitigate the differences of polarity between the cc2500 and the DAC. Once this change was made, the system was able to perform as described in the Results section.

REFERENCES

[1] R. M. Kieckhafer, "Final Project Specification," 2019. [Online]. [Accessed 27 March 2019].

[2] Texas Instruments, "CC2500 - Low-Cost Low-Power 2.4 GHz RF Transceiver," 2014. [Online]. Available: cc2500-DS-2014.pdf. [Accessed 27 March 2019].

[3] Texas Instruments, "MSP430x2xx Family User's Guide," 2013. [Online]. Available: MSP430x2xx-Fam-UG.pdf. [Accessed 27 March 2019].

[4] Texas Instruments, "eZ430-RF2500 Development Tool User's Guide," 2007. [Online]. Available: ez430-RF2500-UG.pdf. [Accessed 27 March 2019].

[5] Texas Instruments, "Mixed Signal Microcontroller - MSP430F22x4," 2011. [Online]. Available: MSP430f2274-DS.pdf. [Accessed 27 March 2019].

[6] R. M. Kieckhafer, A. Majdara and S. G. Ayalew, "Lab-01: Introduction to IAR and GPIO," 2019.

[7] Analog Devices, "2.7 V to 5.5 V, 140uA, Rail-to-Rail Output - 12-Bit DAC in an SOT-23 - AD5320," 2007. [Online]. Available: DAC-SPI_AD5320-DS.pdf. [Accessed 27 March 2019].

[8] Agilent Technologies, "Keysight 33220A 20 MHz Function/Arbitrary Waveform Generator," 2007. [Online]. Available: http://literature.cdn.keysight.com/litweb/pdf/5988-8544EN.pdf. [Accessed 27 March 2019].

[9] Keysight Technologies, "Keysight InfiniiVision 3000 X-Series Oscilloscopes User Guide," 2013. [Online]. Available: http://literature.cdn.keysight.com/litweb/pdf/75019-97073.pdf. [Accessed 27 March 2019].

[10] R. M. Kieckhafer and P. Ramesh, "Lab-08: Wireless Comm via the cc2500," 2019.

[11] R. M. Kieckhafer and P. Ramesh, "Lab-06: MSP430 to DAC Chip via SPI," 2019.

[12] R. M. Kieckhafer, S. G. Ayalew and J. Tan, "Lab-04: MSP430 - ADC10 Converter," 2019.

[13] R. M. Kieckhafer, "Written Document Style Guide," 2017. [Online]. [Accessed 27 March 2019].

[14] R. M. Kieckhafer, "Embedded Programming Style Guide," 2015. [Online]. [Accessed 27 March 2019].

APPENDIX

transmitter.c

```
// ========================================================================= BOF
#include "msp430x22x4.h"         // Large MSP header
#include "stdint.h"              // Standard Integers for MSP
#include "wireless-v2.h"         // Include the wireless header file
#include "stdio.h"

typedef union                    // Control & Data word type for DAC
{
  uint16_t u16;                  // 16-bits for easy arithmetic on full word
  uint8_t  u8[2];                // 08-bits for easy write to 8-bit TX-BUF reg.
} dacWord_t;

volatile dacWord_t dacWordOut;   // Global instance of union

#pragma vector=ADC10_VECTOR
__interrupt void IsrCaptureData (void)
// ------------------------------------------------------------------------------
// Func:    At ADC IRQ, capture the value from ADC10MEM, shift it over for the
//          DAC, and transfer it over to the cc2500 to send over wireless. Once
//          completed, the ISR will exit and be in Active Mode to then kick-off
//          the next ADC capture.
// Args:    None
// Retn:    None
// ------------------------------------------------------------------------------
{
  static uint8_t pktLen = 3;          // Fix Packet size

  // Save off ADC value (only the 10 bits)
  dacWordOut.u16 = ( 0x3FF & ADC10MEM );

  uint8_t pktData[3] = {0x02, dacWordOut.u8[0], dacWordOut.u8[1]};
   // 10b Number in 2 bytes
   // Contents:   pktData[3] = {Pyld leng, LSB, MSB}

  RFSendPacket( pktData, pktLen );    // Activate TX mode & transmit the pkt

  TI_CC_SPIStrobe(TI_CCxxx0_SIDLE);   // Set cc2500 to IDLE mode.
                                      // TX mode re-activates in RFSendPacket
                                      // w/ AutoCal @ IDLE to TX Transition

  __bic_SR_register_on_exit( LPM0_bits ); // Enter AM w/ IRQs enab.
}
```

```
void SetupAll (void)
// ------------------------------------------------------------------------
// Func:    Initialize Button & LED ports, and ports for I/O on TA1 Capture.
// Args:    None
// Retn:    None
// ------------------------------------------------------------------------
{
  dacWordOut.u16 = 0x0000;
  volatile uint16_t delay;             // Counter for delay timeout

  for( delay = 0; delay < 650; delay++ ) { }; // Empirical: cc2500 Power up

  // Set up clock system
  BCSCTL1 = CALBC1_8MHZ;               // DCO = 8 MHz due to battery power
  DCOCTL  = CALDCO_8MHZ;               // DCO = 8 MHz due to battery power
  // Wireless Initialization
  P2SEL = 0;                           // P2.6, P2.7 = GDO0, GDO2 (GPIO)
  TI_CC_SPISetup();                    // Initialize SPI port for both the
                                       // SPI communication b/w cc2500 and
                                       // DAC.
                                       // This method enables the chip select
                                       // Sets the SPI to have Polarity - 1
                                       // SMCLK as SCLK and Divided by /2

  UCB0BR0 = 0x10;                      // SCLK/16
  UCB0BR1 = 0;

  TI_CC_PowerupResetCCxxxx();          // Reset cc2500

  writeRFSettings();                   // Send RF settings to config regs

  TI_CC_SPIWriteReg( TI_CCxxx0_CHANNR,  8 );  // Set Your Own Channel Number
                                              // only AFTER writeRFSettings
  for( delay = 0; delay < 650; delay++ ) { }; // Let cc2500 finish setup

  // Configure ADC
  ADC10AE0  |= 0x01;                   // Enable chnl A0 = P2.0;

  ADC10CTL0 |= ADC10ON                 // Turn ON ADC10
            |  ADC10SHT_0              // samp-hold tim = 4 cyc;
            |  ADC10IE;                // Enable flag

  ADC10CTL1 |= ADC10SSEL_0             // ADC10CLK source = ADCOSC
            |  ADC10DIV_0              // ADC10CLK divider = 1
            |  INCH_0;                 // Select input = chnl A0 (default);
}
```

```
void main(void)
// -------------------------------------------------------------------------
// Func:    Init I/O ports, wireless, and ADC with SetupAll
// Args:    None
// Retn:    None
// -------------------------------------------------------------------------
{
  WDTCTL = WDTPW | WDTHOLD;        // Stop Watchdog Timer
  SetupAll();                      // Configure I/O pins, wireless, and ADC
  while ( 1 )
  {
    ADC10CTL0 |= ADC10SC | ENC;    // start next sample
    _BIS_SR ( LPM0_bits | GIE );   // Enter LPM1 w/ IRQs enab.
  }
}
// ========================================================================= EOF
```

receiver.c

```
// ========================================================================== BOF
#include "msp430x22x4.h"         // Large MSP header
#include "stdint.h"              // Standard Integers for MSP
#include "wireless-v2.h"         // Include the wireless header file

typedef union                    // Control & Data word type for DAC
{
  uint16_t u16;                  // 16-bits for easy arithmetic on full word
  uint8_t  u8[2];                // 08-bits for easy write to 8-bit TX-BUF reg.
} dacWord_t;

volatile dacWord_t dacWordOut;
uint8_t rxPkt[2] = {0x00, 0x00};

//-------------------------------------------------------------------------------
// Func:  Packet Received ISR:  triggered by falling edge of GDO0.
//        Parses pkt & transfers over SPI to DAC.
// Args:  None
// Retn:  None
//-------------------------------------------------------------------------------
#pragma vector=PORT2_VECTOR        // cc2500 pin GDO0 is hardwired to P2.6
__interrupt void PktRxedISR(void)
{
  static uint8_t len = 2;          // Packet Len = 2 bytes (data only)
  uint8_t status[2];               // Buffer to store pkt status bytes

  if( TI_CC_GDO0_PxIFG & TI_CC_GDO0_PIN )    // chk GDO0 bit of P2 IFG Reg
  {
    RFReceivePacket(rxPkt, &len, status);  // Get packet from cc2500
  }

  dacWordOut.u8[1] = rxPkt[1];  // Save the data from the transmission
  dacWordOut.u8[0] = rxPkt[0];  // LSB was first then MSB

  dacWordOut.u16 = dacWordOut.u16 << 2; // 10-bit to 12-bit scaling

  TI_CC_GDO0_PxIFG &= ~TI_CC_GDO0_PIN;  // Reset GDO0 IRQ flag
  TI_CC_SPIStrobe(TI_CCxxx0_SIDLE);  // Set cc2500 to idle mode.
  TI_CC_SPIStrobe(TI_CCxxx0_SRX);    // Set cc2500 to RX mode.
                                     // AutoCal @ IDLE to RX Transition

  UCB0CTL0 &= ~(UCCKPH);             // Change to be in the same phase
                                     // as the DAC (from being the phase
                                     // of the cc2500)

  while ( UCB0STAT & UCBUSY ){};     // Wait til SPI is idle (should be now)

  P4OUT &= ~0x08;                    // Assert SS (active low)

  while ( !(IFG2 & UCB0TXIFG) ){};   // Wait til TXBUF is ready (empty)
  UCB0TXBUF = dacWordOut.u8[1];      // Tx next byte (TXIFG auto-clr)
```

```
  while ( !(IFG2 & UCB0TXIFG) ){};   // Wait til TXBUF is ready (empty)
  UCB0TXBUF = dacWordOut.u8[0];      // Tx next byte (TXIFG auto-clr)

  while ( UCB0STAT & UCBUSY ){};     // Wait til ALL bits shift out SPI
  P4OUT |= 0x08;                     // De-assert SS => End of stream
  UCB0CTL0 |= UCCKPH;                // Change back to polarity of cc2500
}

void SetupAll (void)
// ------------------------------------------------------------------------
// Func:    Initialize Button & LED ports, and ports for I/O on TA1 Capture.
// Args:    None
// Retn:    None
// ------------------------------------------------------------------------
{
  volatile uint16_t delay;
  for( delay = 0; delay < 650; delay++ ) { }; // Empirical: cc2500 Power Up

  // Set up clock system
  BCSCTL1 = CALBC1_8MHZ;               // set DCO = 8MHz
  DCOCTL  = CALDCO_8MHZ;               // set DCO = 8MHz

  // Wireless Initialization
  P2SEL = 0;                           // P2.6, P2.7 = GDO0, GDO2 (GPIO)

  TI_CC_SPISetup();                    // Initialize SPI port for both the
                                       // SPI communication b/w cc2500 and
                                       // DAC.
                                       // This method enables the chip select
                                       // Sets the SPI to have Polarity - 1
                                       // SMCLK as SCLK and Divided by /2

  UCB0BR0 = 0x10;                      // UCLK/2
  UCB0BR1 = 0;

  TI_CC_PowerupResetCCxxxx();          // Reset cc2500
  writeRFSettings();                   // Send RF settings to config regs

  TI_CC_GDO0_PxIES |=  TI_CC_GDO0_PIN;  // IRQ on GDO0 fall. edge (end pkt)
  TI_CC_GDO0_PxIFG &= ~TI_CC_GDO0_PIN;  // Clear  GDO0 IRQ flag
  TI_CC_GDO0_PxIE  |=  TI_CC_GDO0_PIN;  // Enable GDO0 IRQ

  TI_CC_SPIStrobe(TI_CCxxx0_SRX);      // Init. cc2500 in RX mode.
  TI_CC_SPIWriteReg(TI_CCxxx0_CHANNR, 8);     // Set Your Own Channel Number
                                              // AFTER writeRFSettings

  for( delay = 0; delay < 650; delay++ ) { }; // Empirical: cc2500 Power Up

  // Slave Select for transmitting to DAC
  P4SEL &= ~0x08;              // P4.3 Select as GPIO Alt Function
  P4DIR |= 0x08;               // P4.3 Select as an output
  P4OUT |= 0x08;               // P4.3 Set to be High (SS is active low)
}
```

```
void main(void)
// -------------------------------------------------------------------------
// Func:    Init I/O ports & IRQs, Config. Timer TA Chnl 1, enter LowPwr Mode.
// Args:    None
// Retn:    None
// -------------------------------------------------------------------------
{
  WDTCTL = WDTPW | WDTHOLD;          // Stop Watchdog Timer
  SetupAll();                        // Configure I/O Pins

  _BIS_SR ( LPM0_bits | GIE );       // Enter LPM1 w/ IRQs enab.
}
// ========================================================================= EOF
```