Lab3_ADC_PWM

Generated by Doxygen 1.8.13

Contents

Chapter 1

README

In this laboratory work, a program was developed to control the brightness of the LED using a potentiometer on Wi↩
Fire board and using external voltage regilator - microphone. Turning the potentiometer knob changes the voltage that is applied to the analog port. Under the influence of sound signals and noise, the microphone changes the voltage value. The choice of the desired voltage regulator is made by changing the value of the MODE variable in the file user.c. Voltage is measured by the ADC. The timer generates a PWM signal whose duty cycle varies depending on the measured voltage. Thus, the brightness of the LED is changed by turning the potentiometer knob or by making noise near microphone. The display of the brightness level with three LEDs was realized: a low level - one LED is on, an average level is 2, high is 3. Indication occurs only when the BTN1 button is pressed and held.

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

File Documentation

3.1 D:/GIT/TheConnectedMCU_Labs/bkarachok/lab3-ADC_PWM/ADC.c File Reference

```
#include <stdint.h>
#include <stdbool.h>
#include "user.h"
```

Functions

- void initWiFIREadc (void)
- int convertWiFIREadc (uint8_t channelNumber)
- int ReadPotentiometerWithADC (void)

3.1.1 Function Documentation

3.1.1.1 convertWiFIREadc()

```
int convertWiFIREadc (
            uint8_t channelNumber )
```

```
#if defined(__32MZ2048ECG100__)
    // EC uses old ADC, requires work-around to eliminate noise
    // Samples ADC 17 times, discarding first 16 results.
    // Uses DMA to reduce management overhead.

    int i, k = 0;
    uint8_t vcn = channelNumber;
    int analogValue = 0;
    volatile unsigned td;

#define KVA_2_PA(v)             (((uint32_t) (v)) & 0x1fffffff)
    static uint16_t __attribute__((coherent)) ovsampValue;

    // set the channel trigger for GSWTRG source triggering
    if (channelNumber == 43 || channelNumber == 44 || channelNumber >= 50) {
        return (0);
    } else if (channelNumber >= 45) {
```

```
        vcn = channelNumber - 45;
        AD1IMOD |= 1 << ((vcn * 2) + 16); // say use the alt; set SHxALT
    }

    AD1CON3bits.ADINSEL = vcn; // manually trigger the conversion
    AD1CON1bits.FILTRDLY = AD1CON2bits.SAMC + 5; // strictly not needed, but set the timing anyway

    // set up for 16x oversample
    AD1FLTR6 = 0; // clear oversampling
    AD1FLTR6bits.OVRSAM = 1; // say 16x oversampling
    AD1FLTR6bits.CHNLID = vcn; // the ANx channel

    // setup DMA
    IEC4bits.DMA7IE = 0; // disable DMA channel 4 interrupts
    IFS4bits.DMA7IF = 0; // clear existing DMA channel 4 interrupt flag

    // setup DMA channel x
    DMACONbits.ON = 1; // make sure the DMA is ON
    DCH7CON = 0; // clear DMA channel CON
    DCH7ECON = 0; // clear DMA ECON
    DCH7ECONbits.CABORT = 1; // reset the DMA channel
    while (DCH7CONbits.CHEN == 1); // make sure DMA is not enabled
    while (DCH7CONbits.CHBUSY == 1); // make sure DMA is not busy
    DCH7CONbits.CHPRI = 3; // use highest priority
    DCH7ECONbits.CHSIRQ = _ADC1_DF6_VECTOR; // say the ADC filter complete triggers the DMA
    DCH7ECONbits.SIRQEN = 1; // enable the IRQ event for triggering
    DCH7SSA = KVA_2_PA(&AD1FLTR6); // address of the source
    DCH7SSIZ = 2; // source size is 2 bytes
    DCH7CSIZ = 2; // cell size transfer
    DCH7DSA = KVA_2_PA(&ovsampValue); // destination address
    DCH7DSIZ = sizeof (ovsampValue); // destination size

    // must throw out first 16 samples
    // keep the 17th. We can not access any ADC registers
    // however we can look at the DMA to see when things complete
    for (i = 0; i < 17; i++) {
        do {

            DCH7ECONbits.CABORT = 1; // reset the DMA channel
            AD1FLTR6bits.AFEN = 0; // make sure oversampling is OFF
            while (DCH7CONbits.CHEN == 1); // wait for DMA to stop
            while (DCH7CONbits.CHBUSY == 1); // wait for DMA not to be busy
            DCH7INT = 0; // clear all interrupts
            DCH7CONbits.CHEN = 1; // enable the DMA channel
            AD1FLTR6bits.AFEN = 1; // enable oversampling

            AD1CON3bits.RQCONVRT = 1; // start conversion

            // we have noticed problems that the DMA channel is not always
            // triggered, so after a time out value, just try again.
            // fundamentally the problem is that AD1FLTR6bits.AFEN must
            // be reset for each oversample conversion, disable and reenabled.

            // we know a conversion is going to take 8000ns * 16, or 128 us
            // we don't want to check too often so DMA and ADC can work
            // but we don't want to wait too long and hold things up

            for (k = 0; k < 20; k++) // this is more than enough time for the conversion,
            { // really 8 should be good enough
                for (td = 0; td < 16 * 200; td++) // Time delay
                    ;
                if (DCH7INTbits.CHBCIF == 1) {
                    break; // DMA completed and copied the oversampled result
                }
            }
        } while (DCH7INTbits.CHBCIF == 0); // if the conversion did not finish, try again
    }
    analogValue = ovsampValue >> 2; // 16 oversample gets you 2 extra bits

    // we are done, clean up the DMA, oversampling filter, and ADC
    DCH7CON = 0;
    while (DCH7CONbits.CHBUSY == 1);
    AD1CON3bits.ADINSEL = 0;
    AD1FLTR6 = 0;

    if (channelNumber >= 45) {
        AD1IMOD &= ~(0b11 << ((vcn * 2) + 16)); // don't use alt
    }

    return (analogValue);
#elif defined(__32MZ2048EFG100__)
    uint32_t mask;

    if (channelNumber >= 32) // this code only supports channels 0 to 31
        return 0;

    // Select channel to convert
```

```
    ADCCON3bits.ADINSEL = channelNumber;
    // Manually trigger conversion
    ADCCON3bits.RQCNVRT = 1;

    // wait for completion of the conversion
    mask = 0x1 << channelNumber;
    while ((ADCDSTAT1 & mask) == 0);

    // return the converted data
    return ((int) ((uint32_t *) & ADCDATA0)[channelNumber]);
#else
#error  Unsupported processor type, please add to ADC.c.
#endif
```

3.1.1.2 initWiFIREadc()

```
void initWiFIREadc (
            void  )
```

Files to Include User Functions

```
#if defined(__32MZ2048ECG100__)
    // Configure AD1CON1
    AD1CON1 = 0; // No AD1CON1 features are enabled including: Stop-in-Idle, early
    // interrupt, filter delay Fractional mode and scan trigger source.

    // Configure AD1CON2
    AD1CON2 = 0; // Boost, Low-power mode off, SAMC set to min, set up the ADC Clock
    AD1CON2bits.ADCSEL = 1; // 1 = SYSCLK, 2 REFCLK03, 3 FRC
    AD1CON2bits.ADCDIV = 10; // DIV_20 TQ = 1/200 MHz; Tad = 10 * (TQ * 2) = 100 ns; 10 MHz ADC clock; the
       sweet spot
    AD1CON2bits.SAMC = 75; // settling time is 76 TADs  ((samc +1) + 4) * TAD <= 8000    (125Kbps is max,
       time 1/125000 = 8000 ns) => ((75  + 1) + 4) * 100 ns = 8000

    // Configure AD1CON3
    AD1CON3 = 0; // ADINSEL is not configured for this example. VREFSEL of ?0?
    // selects AVDD and AVSS as the voltage reference.

    // AD1CON3bits.VREFSEL = 0b011; // set external VRef+/-

    // Configure AD1GIRGENx
    AD1GIRQEN1 = 0; // No global interrupts are used.
    AD1GIRQEN2 = 0;

    // Configure AD1CSSx
    AD1CSS1 = 0; // No channel scanning is used.
    AD1CSS2 = 0;

    // Configure AD1CMPCONx
    AD1CMPCON1 = 0; // No digital comparators are used. Setting the AD1CMPCONx
    AD1CMPCON2 = 0; // register to ?0? ensures that the comparator is disabled. Other
    AD1CMPCON3 = 0; // registers are ?don?t care?.
    AD1CMPCON4 = 0;
    AD1CMPCON5 = 0;
    AD1CMPCON6 = 0;

    // Configure AD1FLTRx
    AD1FLTR1 = 0; // No oversampling filters are used.
    AD1FLTR2 = 0;
    AD1FLTR3 = 0;
    AD1FLTR4 = 0;
    AD1FLTR5 = 0;
    AD1FLTR6 = 0;

    // Set up the trigger sources
    AD1TRG1 = 0; // Initialize all sources to no trigger.
    AD1TRG2 = 0;
    AD1TRG3 = 0;

    // Set up the CAL registers
    // AD1CAL1 = DEVADC1;          // Copy the configuration data to the
    // AD1CAL2 = DEVADC2;          // AD1CALx special function registers.
    // AD1CAL3 = DEVADC3;
    // AD1CAL4 = DEVADC4;
    // AD1CAL5 = DEVADC5;
```

```
    // comply to the errata
    AD1CAL1 = 0xF8894530;
    AD1CAL2 = 0x01E4AF69;
    AD1CAL3 = 0x0FBBBBB8;
    AD1CAL4 = 0x000004AC;
    AD1CAL5 = 0x02000002;

    // Turn the ADC on, start calibration
    AD1IMODbits.SH0MOD = 2; // put in differiential mode for self calibration
    AD1IMODbits.SH1MOD = 2; // put in differiential mode for self calibration
    AD1IMODbits.SH2MOD = 2; // put in differiential mode for self calibration
    AD1IMODbits.SH3MOD = 2; // put in differiential mode for self calibration
    AD1IMODbits.SH4MOD = 2; // put in differiential mode for self calibration
    AD1IMODbits.SH5MOD = 2; // put in differiential mode for self calibration
    AD1CON1bits.ADCEN = 1; // enable, start calibration
    while (AD1CON2bits.ADCRDY == 0); // wait for calibration to complete
    AD1IMODbits.SH0MOD = 0; // put in unipolar encoding
    AD1IMODbits.SH1MOD = 0; // put in unipolar encoding
    AD1IMODbits.SH2MOD = 0; // put in unipolar encoding
    AD1IMODbits.SH3MOD = 0; // put in unipolar encoding
    AD1IMODbits.SH4MOD = 0; // put in unipolar encoding
    AD1IMODbits.SH5MOD = 0; // put in unipolar encoding
#elif defined(__32MZ2048EFG100__)
    // initialize configuration registers
    ADCCON1 = ADCCON2 = ADCCON3 = ADCANCON = 0;
    // resolution 0 - 6bits, 1 - 8bits, 2 - 10bits, 3 - 12bits
    ADCCON1bits.SELRES = 3; // shared ADC, 12 bits resolution (bits+2 TADs, 12bit resolution = 14 TAD).
    // 0 - no trigger, 1 - clearing software trigger, 2 - not clearing software trigger, the rest see
       datasheet
    ADCCON1bits.STRGSRC = 1; //Global software trigger / self clearing.
    // 0 - internal 3.3, 1 - use external VRef+, 2 - use external VRef-
    ADCCON3bits.VREFSEL = 0; // use internal 3.3 reference
    // set up the TQ and TAD and S&H times
    // TCLK: 00- pbClk3, 01 - SysClk, 10 - External Clk3, 11 - interal 8 MHz clk
    ADCCON3bits.ADCSEL = 0b01; // TCLK clk == Sys Clock == F_CPU
    // Global ADC TQ Clock: Global ADC prescaler 0 - 63; Divide by (CONCLKDIV*2) However, the value 0 means
       divide by 1
    ADCCON3bits.CONCLKDIV = 0; // Divide by 1 == TCLK == SYSCLK == F_CPU
    ADCCON2bits.ADCDIV = 8;
    ADCCON2bits.SAMC = 10;
    // the warm up count is 2^^X where X = 0 -15
    ADCANCONbits.WKUPCLKCNT = 9; // Wakeup exponent = 2^^15 * TADx
    // Configure ADCIRQENx
    ADCCMPEN1 = ADCCMPEN2 = 0; // No interrupts are used
    // Configure ADCCSSx
    ADCCSS1 = ADCCSS2 = 0; // No scanning is used
    // Configure ADCCMPxCON
    ADCCMP1 = ADCCMP2 = ADCCMP3 = 0; // No digital comparators are used. Setting the ADCCMPxCON
    ADCCMP4 = ADCCMP5 = ADCCMP6 = 0; // register to '0' ensures that the comparator is disabled.
    // Configure ADCFLTRx
    ADCFLTR1 = ADCFLTR2 = ADCFLTR3 = 0; // Clear all bits
    ADCFLTR4 = ADCFLTR5 = ADCFLTR6 = 0; // Clear all bits
    // disable all global interrupts
    ADCGIRQEN1 = ADCGIRQEN2 = 0;
    // Early interrupt
    ADCEIEN1 = ADCEIEN2 = 0; // No early interrupt
    // no dedicated trigger sources
    ADCTRGMODE = 0;
    // put everything in single ended unsigned mode
    ADCIMCON1 = ADCIMCON2 = ADCIMCON3 = 0;
    // triggers are all edge trigger
    ADCTRGSNS = 0;
    // turn on the ADC
    ADCCON1bits.ON = 1;
    // Wait for voltage reference to be stable
    while (!ADCCON2bits.BGVRRDY); // Wait until the reference voltage is ready
    while (ADCCON2bits.REFFLT); // Wait if there is a fault with the reference voltage
    // Enable clock to analog circuit
    ADCANCONbits.ANEN7 = 1; // Enable the clock to analog bias and digital control
    // Wait for ADC to be ready
    while (!ADCANCONbits.WKRDY7); // Wait until ADC0 is ready
    // Enable the ADC module
    ADCCON3bits.DIGEN7 = 1; // Enable shared ADC
#else
#error  Unsupported processor type, please add to ADC.c.
#endif
```

3.1.1.3 ReadPotentiometerWithADC()

```
int ReadPotentiometerWithADC (
```

```
            void )
```

```
//read voltage from ponentiometer divider
#if defined(__32MZ2048ECG100__)
    // EC uses old ADC, requires work-around to eliminate noise
    return convertWiFIREadc(VR1_AN_CHAN_NUM);
#elif defined(__32MZ2048EFG100__)
    // Select channel to convert
    ADCCON3bits.ADINSEL = VR1_AN_CHAN_NUM;
    // Manually trigger conversion
    ADCCON3bits.RQCNVRT = 1;
    // wait for completion of the conversion
    while (ADCDSTAT1bits.ARDY8 == 0)
        ;
    // return the converted data
    return ADCDATA8;
#else
#error  Unsupported processor type, please add to ADC.c.
#endif
}

int ReadExternalRegulatorWithADC(void) { //read voltage from micropfone (external voltage regulator)
#if defined(__32MZ2048ECG100__)
    // EC uses old ADC, requires work-around to eliminate noise
    return convertWiFIREadc(VR2_AN_CHAN_NUM);
#elif defined(__32MZ2048EFG100__)
    // Select channel to convert
    ADCCON3bits.ADINSEL = VR2_AN_CHAN_NUM;
    // Manually trigger conversion
    ADCCON3bits.RQCNVRT = 1;
    // wait for completion of the conversion
    while (ADCDSTAT1bits.ARDY8 == 0)
        ;
    // return the converted data
    return ADCDATA8;
#else
#error  Unsupported processor type, please add to ADC.c.
#endif
```

3.2 D:/GIT/TheConnectedMCU_Labs/bkarachok/lab3-ADC_PWM/ADC.h File Reference

Functions

- void initWiFIREadc (void)
- int convertWiFIREadc (uint8_t channelNumber)

3.2.1 Function Documentation

3.2.1.1 convertWiFIREadc()

```
int convertWiFIREadc (
            uint8_t channelNumber )
```

```
#if defined(__32MZ2048ECG100__)
    // EC uses old ADC, requires work-around to eliminate noise
    // Samples ADC 17 times, discarding first 16 results.
    // Uses DMA to reduce management overhead.

    int i, k = 0;
    uint8_t vcn = channelNumber;
    int analogValue = 0;
    volatile unsigned td;

#define KVA_2_PA(v)             (((uint32_t) (v)) & 0x1fffffff)
```

```
    static uint16_t __attribute__((coherent)) ovsampValue;

    // set the channel trigger for GSWTRG source triggering
    if (channelNumber == 43 || channelNumber == 44 || channelNumber >= 50) {
        return (0);
    } else if (channelNumber >= 45) {
        vcn = channelNumber - 45;
        AD1IMOD |= 1 << ((vcn * 2) + 16); // say use the alt; set SHxALT
    }

    AD1CON3bits.ADINSEL = vcn; // manually trigger the conversion
    AD1CON1bits.FILTRDLY = AD1CON2bits.SAMC + 5; // strictly not needed, but set the timing anyway

    // set up for 16x oversample
    AD1FLTR6 = 0; // clear oversampling
    AD1FLTR6bits.OVRSAM = 1; // say 16x oversampling
    AD1FLTR6bits.CHNLID = vcn; // the ANx channel

    // setup DMA
    IEC4bits.DMA7IE = 0; // disable DMA channel 4 interrupts
    IFS4bits.DMA7IF = 0; // clear existing DMA channel 4 interrupt flag

    // setup DMA channel x
    DMACONbits.ON = 1; // make sure the DMA is ON
    DCH7CON = 0; // clear DMA channel CON
    DCH7ECON = 0; // clear DMA ECON
    DCH7ECONbits.CABORT = 1; // reset the DMA channel
    while (DCH7CONbits.CHEN == 1); // make sure DMA is not enabled
    while (DCH7CONbits.CHBUSY == 1); // make sure DMA is not busy
    DCH7CONbits.CHPRI = 3; // use highest priority
    DCH7ECONbits.CHSIRQ = _ADC1_DF6_VECTOR; // say the ADC filter complete triggers the DMA
    DCH7ECONbits.SIRQEN = 1; // enable the IRQ event for triggering
    DCH7SSA = KVA_2_PA(&AD1FLTR6); // address of the source
    DCH7SSIZ = 2; // source size is 2 bytes
    DCH7CSIZ = 2; // cell size transfer
    DCH7DSA = KVA_2_PA(&ovsampValue); // destination address
    DCH7DSIZ = sizeof (ovsampValue); // destination size

    // must throw out first 16 samples
    // keep the 17th. We can not access any ADC registers
    // however we can look at the DMA to see when things complete
    for (i = 0; i < 17; i++) {
        do {

            DCH7ECONbits.CABORT = 1; // reset the DMA channel
            AD1FLTR6bits.AFEN = 0; // make sure oversampling is OFF
            while (DCH7CONbits.CHEN == 1); // wait for DMA to stop
            while (DCH7CONbits.CHBUSY == 1); // wait for DMA not to be busy
            DCH7INT = 0; // clear all interrupts
            DCH7CONbits.CHEN = 1; // enable the DMA channel
            AD1FLTR6bits.AFEN = 1; // enable oversampling

            AD1CON3bits.RQCONVRT = 1; // start conversion

            // we have noticed problems that the DMA channel is not always
            // triggered, so after a time out value, just try again.
            // fundamentally the problem is that AD1FLTR6bits.AFEN must
            // be reset for each oversample conversion, disable and reenabled.

            // we know a conversion is going to take 8000ns * 16, or 128 us
            // we don't want to check too often so DMA and ADC can work
            // but we don't want to wait too long and hold things up

            for (k = 0; k < 20; k++) // this is more than enough time for the conversion,
            { // really 8 should be good enough
                for (td = 0; td < 16 * 200; td++) // Time delay
                    ;
                if (DCH7INTbits.CHBCIF == 1) {
                    break; // DMA completed and copied the oversampled result
                }
            }
        } while (DCH7INTbits.CHBCIF == 0); // if the conversion did not finish, try again
    }
    analogValue = ovsampValue >> 2; // 16 oversample gets you 2 extra bits

    // we are done, clean up the DMA, oversampling filter, and ADC
    DCH7CON = 0;
    while (DCH7CONbits.CHBUSY == 1);
    AD1CON3bits.ADINSEL = 0;
    AD1FLTR6 = 0;

    if (channelNumber >= 45) {
        AD1IMOD &= ~(0b11 << ((vcn * 2) + 16)); // don't use alt
    }

    return (analogValue);
#elif defined(__32MZ2048EFG100__)
```

```
    uint32_t mask;

    if (channelNumber >= 32) // this code only supports channels 0 to 31
        return 0;

    // Select channel to convert
    ADCCON3bits.ADINSEL = channelNumber;
    // Manually trigger conversion
    ADCCON3bits.RQCNVRT = 1;

    // wait for completion of the conversion
    mask = 0x1 << channelNumber;
    while ((ADCDSTAT1 & mask) == 0);

    // return the converted data
    return ((int) ((uint32_t *) & ADCDATA0)[channelNumber]);
#else
#error  Unsupported processor type, please add to ADC.c.
#endif
```

3.2.1.2 initWiFIREadc()

```
void initWiFIREadc (
              void  )
```

Files to Include User Functions

```
#if defined(__32MZ2048ECG100__)
    // Configure AD1CON1
    AD1CON1 = 0; // No AD1CON1 features are enabled including: Stop-in-Idle, early
    // interrupt, filter delay Fractional mode and scan trigger source.

    // Configure AD1CON2
    AD1CON2 = 0; // Boost, Low-power mode off, SAMC set to min, set up the ADC Clock
    AD1CON2bits.ADCSEL = 1; // 1 = SYSCLK, 2 REFCLK03, 3 FRC
    AD1CON2bits.ADCDIV = 10; // DIV_20 TQ = 1/200 MHz; Tad = 10 * (TQ * 2) = 100 ns; 10 MHz ADC clock; the
       sweet spot
    AD1CON2bits.SAMC = 75; // settling time is 76 TADs  ((samc +1) + 4) * TAD <= 8000    (125Kbps is max,
       time 1/125000 = 8000 ns) => ((75  + 1) + 4) * 100 ns = 8000

    // Configure AD1CON3
    AD1CON3 = 0; // ADINSEL is not configured for this example. VREFSEL of ?0?
    // selects AVDD and AVSS as the voltage reference.

    // AD1CON3bits.VREFSEL = 0b011; // set external VRef+/-

    // Configure AD1GIRGENx
    AD1GIRQEN1 = 0; // No global interrupts are used.
    AD1GIRQEN2 = 0;

    // Configure AD1CSSx
    AD1CSS1 = 0; // No channel scanning is used.
    AD1CSS2 = 0;

    // Configure AD1CMPCONx
    AD1CMPCON1 = 0; // No digital comparators are used. Setting the AD1CMPCONx
    AD1CMPCON2 = 0; // register to ?0? ensures that the comparator is disabled. Other
    AD1CMPCON3 = 0; // registers are ?don?t care?.
    AD1CMPCON4 = 0;
    AD1CMPCON5 = 0;
    AD1CMPCON6 = 0;

    // Configure AD1FLTRx
    AD1FLTR1 = 0; // No oversampling filters are used.
    AD1FLTR2 = 0;
    AD1FLTR3 = 0;
    AD1FLTR4 = 0;
    AD1FLTR5 = 0;
    AD1FLTR6 = 0;

    // Set up the trigger sources
    AD1TRG1 = 0; // Initialize all sources to no trigger.
    AD1TRG2 = 0;
    AD1TRG3 = 0;
```

```
    // Set up the CAL registers
    // AD1CAL1 = DEVADC1;          // Copy the configuration data to the
    // AD1CAL2 = DEVADC2;          // AD1CALx special function registers.
    // AD1CAL3 = DEVADC3;
    // AD1CAL4 = DEVADC4;
    // AD1CAL5 = DEVADC5;

    // comply to the errata
    AD1CAL1 = 0xF8894530;
    AD1CAL2 = 0x01E4AF69;
    AD1CAL3 = 0x0FBBBBB8;
    AD1CAL4 = 0x000004AC;
    AD1CAL5 = 0x02000002;

    // Turn the ADC on, start calibration
    AD1IMODbits.SH0MOD = 2; // put in differiential mode for self calibration
    AD1IMODbits.SH1MOD = 2; // put in differiential mode for self calibration
    AD1IMODbits.SH2MOD = 2; // put in differiential mode for self calibration
    AD1IMODbits.SH3MOD = 2; // put in differiential mode for self calibration
    AD1IMODbits.SH4MOD = 2; // put in differiential mode for self calibration
    AD1IMODbits.SH5MOD = 2; // put in differiential mode for self calibration
    AD1CON1bits.ADCEN = 1; // enable, start calibration
    while (AD1CON2bits.ADCRDY == 0); // wait for calibration to complete
    AD1IMODbits.SH0MOD = 0; // put in unipolar encoding
    AD1IMODbits.SH1MOD = 0; // put in unipolar encoding
    AD1IMODbits.SH2MOD = 0; // put in unipolar encoding
    AD1IMODbits.SH3MOD = 0; // put in unipolar encoding
    AD1IMODbits.SH4MOD = 0; // put in unipolar encoding
    AD1IMODbits.SH5MOD = 0; // put in unipolar encoding
#elif defined(__32MZ2048EFG100__)
    // initialize configuration registers
    ADCCON1 = ADCCON2 = ADCCON3 = ADCANCON = 0;
    // resolution 0 - 6bits, 1 - 8bits, 2 - 10bits, 3 - 12bits
    ADCCON1bits.SELRES = 3; // shared ADC, 12 bits resolution (bits+2 TADs, 12bit resolution = 14 TAD).
    // 0 - no trigger, 1 - clearing software trigger, 2 - not clearing software trigger, the rest see
       datasheet
    ADCCON1bits.STRGSRC = 1; //Global software trigger / self clearing.
    // 0 - internal 3.3, 1 - use external VRef+, 2 - use external VRef-
    ADCCON3bits.VREFSEL = 0; // use internal 3.3 reference
    // set up the TQ and TAD and S&H times
    // TCLK: 00- pbClk3, 01 - SysClk, 10 - External Clk3, 11 - interal 8 MHz clk
    ADCCON3bits.ADCSEL = 0b01; // TCLK clk == Sys Clock == F_CPU
    // Global ADC TQ Clock: Global ADC prescaler 0 - 63; Divide by (CONCLKDIV*2) However, the value 0 means
       divide by 1
    ADCCON3bits.CONCLKDIV = 0; // Divide by 1 == TCLK == SYSCLK == F_CPU
    ADCCON2bits.ADCDIV = 8;
    ADCCON2bits.SAMC = 10;
    // the warm up count is 2^^X where X = 0 -15
    ADCANCONbits.WKUPCLKCNT = 9; // Wakeup exponent = 2^^15 * TADx
    // Configure ADCIRQENx
    ADCCMPEN1 = ADCCMPEN2 = 0; // No interrupts are used
    // Configure ADCCSSx
    ADCCSS1 = ADCCSS2 = 0; // No scanning is used
    // Configure ADCCMPxCON
    ADCCMP1 = ADCCMP2 = ADCCMP3 = 0; // No digital comparators are used. Setting the ADCCMPxCON
    ADCCMP4 = ADCCMP5 = ADCCMP6 = 0; // register to '0' ensures that the comparator is disabled.
    // Configure ADCFLTRx
    ADCFLTR1 = ADCFLTR2 = ADCFLTR3 = 0; // Clear all bits
    ADCFLTR4 = ADCFLTR5 = ADCFLTR6 = 0; // Clear all bits
    // disable all global interrupts
    ADCGIRQEN1 = ADCGIRQEN2 = 0;
    // Early interrupt
    ADCEIEN1 = ADCEIEN2 = 0; // No early interrupt
    // no dedicated trigger sources
    ADCTRGMODE = 0;
    // put everything in single ended unsigned mode
    ADCIMCON1 = ADCIMCON2 = ADCIMCON3 = 0;
    // triggers are all edge trigger
    ADCTRGSNS = 0;
    // turn on the ADC
    ADCCON1bits.ON = 1;
    // Wait for voltage reference to be stable
    while (!ADCCON2bits.BGVRRDY); // Wait until the reference voltage is ready
    while (ADCCON2bits.REFFLT); // Wait if there is a fault with the reference voltage
    // Enable clock to analog circuit
    ADCANCONbits.ANEN7 = 1; // Enable the clock to analog bias and digital control
    // Wait for ADC to be ready
    while (!ADCANCONbits.WKRDY7); // Wait until ADC0 is ready
    // Enable the ADC module
    ADCCON3bits.DIGEN7 = 1; // Enable shared ADC
#else
#error  Unsupported processor type, please add to ADC.c.
#endif
```

3.3 D:/GIT/TheConnectedMCU_Labs/bkarachok/lab3-ADC_PWM/configuration_bits.c File Reference

3.4 D:/GIT/TheConnectedMCU_Labs/bkarachok/lab3-ADC_PWM/main.c File Reference

```
#include <stdint.h>
#include <stdbool.h>
#include "user.h"
```

Functions

- int32_t main (void)

3.4.1 Function Documentation

3.4.1.1 main()

```
int32_t main (
            void )
```

```
// Initialize I/O and Peripherals for application
InitApp();
while (1) {
    AdjustLED1Brightness();
}
```

3.5 D:/GIT/TheConnectedMCU_Labs/bkarachok/lab3-ADC_PWM/README.md File Reference

3.6 D:/GIT/TheConnectedMCU_Labs/bkarachok/lab3-ADC_PWM/user.c File Reference

```
#include <stdint.h>
#include <stdbool.h>
#include "user.h"
#include <sys/attribs.h>
#include "ADC.h"
```

Macros

- #define MODE 2

Functions

- void InitTimer2AndOC5 (void)
- void AdjustLED1Brightness (void)
- void InitGPIO (void)
- void InitApp (void)

3.6.1 Macro Definition Documentation

3.6.1.1 MODE

```
#define MODE 2
```

3.6.2 Function Documentation

3.6.2.1 AdjustLED1Brightness()

```
void AdjustLED1Brightness (
            void )
```

```
unsigned int ext_level = 0, counts=0;
 if(BTN2_PORT_BIT) { // if button2 pressed, then LEDs indicate bright level
     if (ReadExternalRegulatorWithADC() <= 1024)  // indicating bright level using 3 another LEDs
    {
    LD2_PORT_BIT = 0;
    LD3_PORT_BIT = 0;
    LD4_PORT_BIT = 1;
    }
    else if (ReadExternalRegulatorWithADC() <= 2048)
    {
    LD2_PORT_BIT = 0;
    LD3_PORT_BIT = 1;
    LD4_PORT_BIT = 1;
    }
    else
    {
    LD2_PORT_BIT = 1;
    LD3_PORT_BIT = 1;
    LD4_PORT_BIT = 1;
    }
}
           // Read ADC
     ext_level = ReadExternalRegulatorWithADC();
     // Convert value to on-time (counts).
    counts = (ext_level*PWM_PERIOD_COUNTS)/MAX_ADC_VALUE;
    // Update OC5
    OC5RS = counts;
```

3.6.2.2 InitApp()

```
void InitApp (
              void  )
```

```
// Initialize peripherals
InitGPIO();
initWiFIREadc();
InitTimer2AndOC5();
```

3.6.2.3 InitGPIO()

```
void InitGPIO (
              void  )
```

```
   // Setup functionality and port direction
   // LED output
// Disable analog mode if present
   ANSELG &= ~((1 << 6) | (1 << 15));
   ANSELB &= ~(1 << 11);
   // Set direction to output
   TRISG &= ~((1 << 6) | (1 << 15));
   TRISB &= ~(1 << 11);
   TRISD &= ~(1 << 4);
   // Turn off LEDs for initialization
   LD1_PORT_BIT = 0;
   LD2_PORT_BIT = 0;
   LD3_PORT_BIT = 0;
   LD4_PORT_BIT = 0;
   // Button inputs
   // Disable analog mode
   ANSELA &= ~(1 << 5);
   // Set directions to input
   TRISA |= (1 << 5)|(1 << 4);
```

3.6.2.4 InitTimer2AndOC5()

```
void InitTimer2AndOC5 (
              void  )
```

```
T2CON = 0; // clear timer settings to defaults
T2CONbits.TCKPS = 7; // divide clock by 256 with prescaler
TMR2 = 0;
// Set period for timer
PR2 = PWM_PERIOD_COUNTS-1;
// Set initial duty cycle to 50%
OC5R = PWM_PERIOD_COUNTS/2;
// Set reload duty cycle to 50%
OC5RS = PWM_PERIOD_COUNTS/2;
// Configure OC5 control register
OC5CONbits.ON = 1;
OC5CONbits.OC32 = 0;     // 16 bit mode
OC5CONbits.OCTSEL = 0; // Select timer 2
OC5CONbits.OCM = 6; // Select PWM mode without fault pin
// Map OC5 signal to pin G6
RPG6R = 11; // Select OC5
// Start Timer 2
T2CONbits.ON = 1;
```

3.7 D:/GIT/TheConnectedMCU_Labs/bkarachok/lab3-ADC_PWM/user.h File Reference

Macros

- #define LD1_PORT_BIT LATGbits.LATG6
- #define LD2_PORT_BIT LATDbits.LATD4
- #define LD3_PORT_BIT LATBbits.LATB11
- #define LD4_PORT_BIT LATGbits.LATG15
- #define BTN1_PORT_BIT PORTAbits.RA5
- #define BTN2_PORT_BIT PORTAbits.RA4
- #define PWM_FREQ_HZ (1000)
- #define PWM_PERIOD_COUNTS (100000000/(256*PWM_FREQ_HZ))
- #define MAX_ADC_VALUE (4095)
- #define VR1_AN_CHAN_NUM (8)
- #define VR2_AN_CHAN_NUM (45)

Functions

- void InitApp (void)
- void AdjustLED1Brightness (void)

3.7.1 Macro Definition Documentation

3.7.1.1 BTN1_PORT_BIT

```
#define BTN1_PORT_BIT PORTAbits.RA5
```

3.7.1.2 BTN2_PORT_BIT

```
#define BTN2_PORT_BIT PORTAbits.RA4
```

3.7.1.3 LD1_PORT_BIT

```
#define LD1_PORT_BIT LATGbits.LATG6
```

3.7.1.4 LD2_PORT_BIT

```
#define LD2_PORT_BIT LATDbits.LATD4
```

3.7.1.5 LD3_PORT_BIT

```
#define LD3_PORT_BIT LATBbits.LATB11
```

3.7.1.6 LD4_PORT_BIT

```
#define LD4_PORT_BIT LATGbits.LATG15
```

3.7.1.7 MAX_ADC_VALUE

```
#define MAX_ADC_VALUE (4095)
```

3.7.1.8 PWM_FREQ_HZ

```
#define PWM_FREQ_HZ (1000)
```

3.7.1.9 PWM_PERIOD_COUNTS

```
#define PWM_PERIOD_COUNTS (100000000/(256*PWM_FREQ_HZ))
```

3.7.1.10 VR1_AN_CHAN_NUM

```
#define VR1_AN_CHAN_NUM (8)
```

3.7.1.11 VR2_AN_CHAN_NUM

```
#define VR2_AN_CHAN_NUM (45)
```

3.7.2 Function Documentation

3.7.2.1 AdjustLED1Brightness()

```
void AdjustLED1Brightness (
            void )
```

```
unsigned int ext_level = 0, counts=0;
 if(BTN2_PORT_BIT) { // if button2 pressed, then LEDs indicate bright level
     if (ReadExternalRegulatorWithADC() <= 1024)  // indicating bright level using 3 another LEDs
    {
    LD2_PORT_BIT = 0;
    LD3_PORT_BIT = 0;
    LD4_PORT_BIT = 1;
    }
    else if (ReadExternalRegulatorWithADC() <= 2048)
    {
    LD2_PORT_BIT = 0;
    LD3_PORT_BIT = 1;
    LD4_PORT_BIT = 1;
    }
    else
    {
    LD2_PORT_BIT = 1;
    LD3_PORT_BIT = 1;
    LD4_PORT_BIT = 1;
    }
}
           // Read ADC
     ext_level = ReadExternalRegulatorWithADC();
     // Convert value to on-time (counts).
    counts = (ext_level*PWM_PERIOD_COUNTS)/MAX_ADC_VALUE;
    // Update OC5
    OC5RS = counts;
```

3.7.2.2 InitApp()

```
void InitApp (
            void )
```

```
// Initialize peripherals
InitGPIO();
initWiFIREadc();
InitTimer2AndOC5();
```

Index