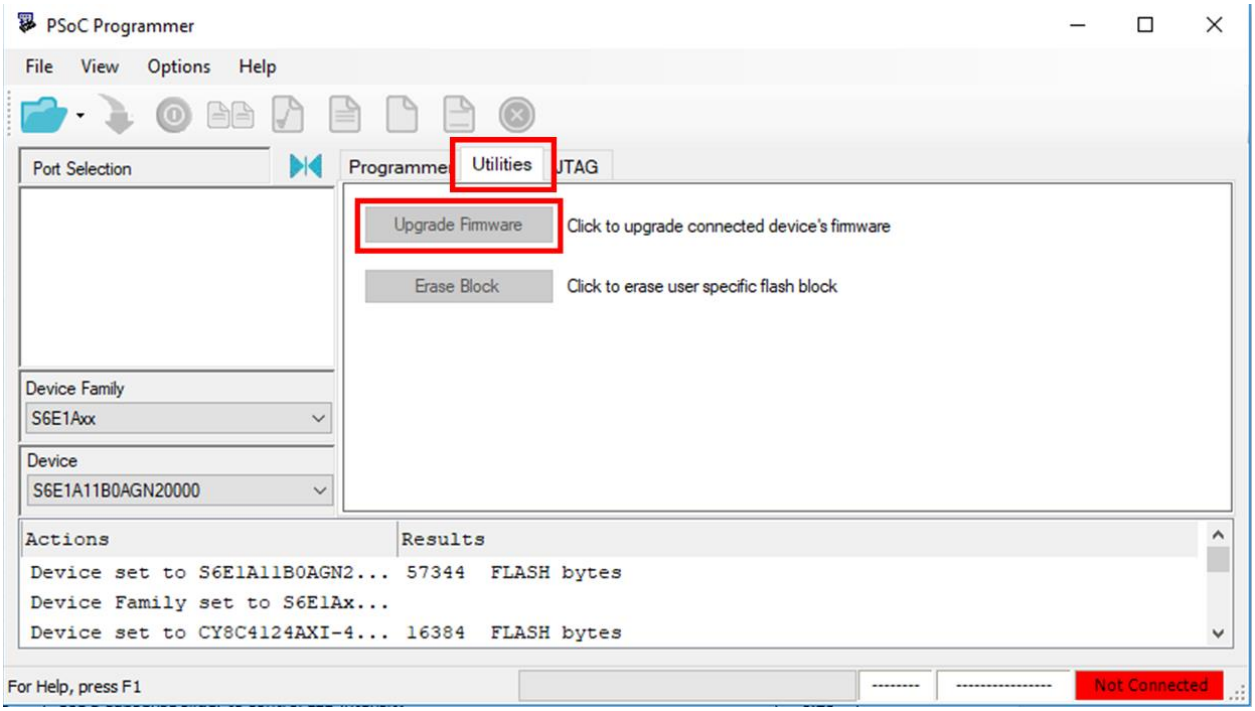
Lab Guide

Update KitProg Firmware (Example L0)

Description: Use PSoC Programmer to update the firmware on the KitProg device to the latest version. This will ensure that program/debug will work with the latest versions of the tools.

Process:

1. Plug the kit into a USB port on your computer using the cable supplied with the kit.
2. Open PSoC Programmer, navigate to the “Utilities” tab, and click “Upgrade Firmware”.
3. Follow the instructions on the screen to complete the firmware upgrade.



Measure an Analog Voltage and Display the Voltage on an LCD (Example L1)

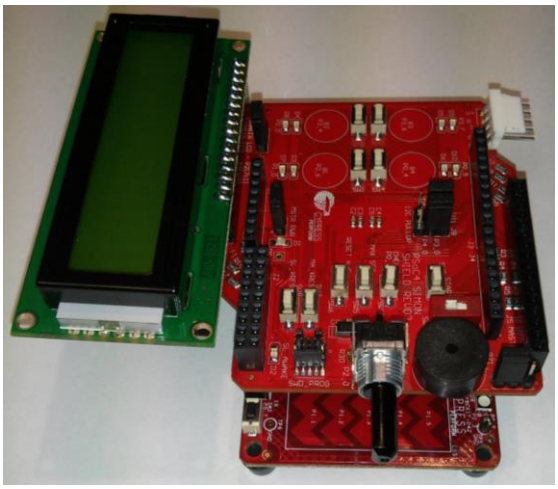
Description: Use the SAR ADC to measure an analog voltage input from a potentiometer and display the voltage value on a character LCD.

Objective:

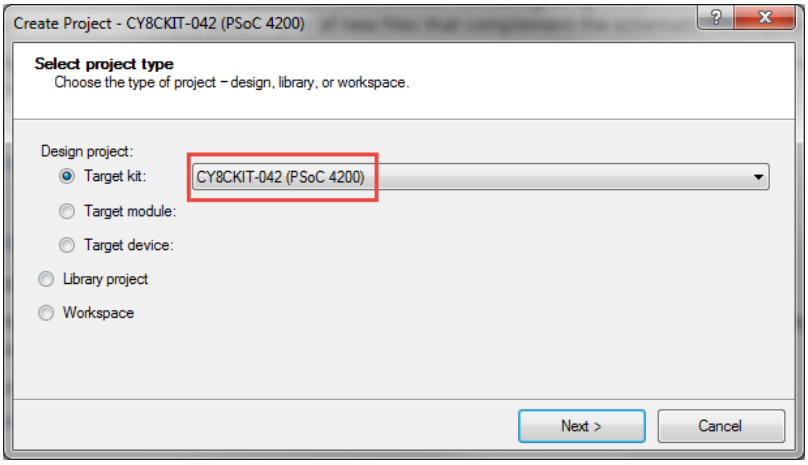
- Use a CY8CKIT-042 PSoC 4 Pioneer Devkit (Pioneer)
- Demonstrate using the ADC to measure an analog voltage
- Demonstrate using the character LCD component

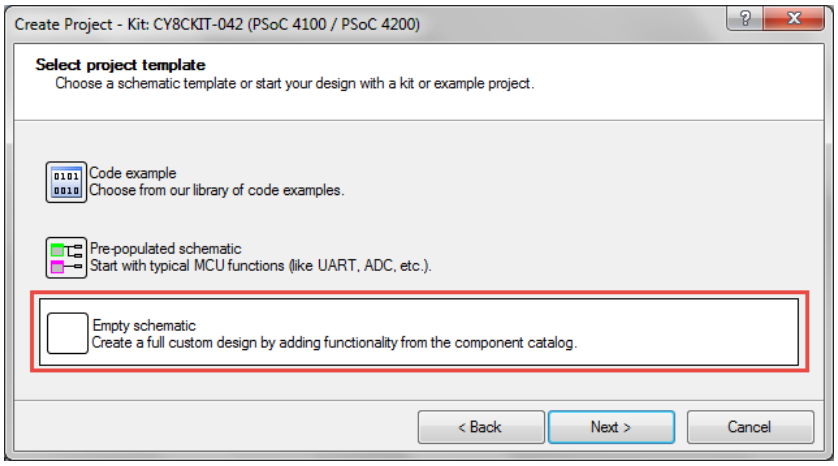
Process:

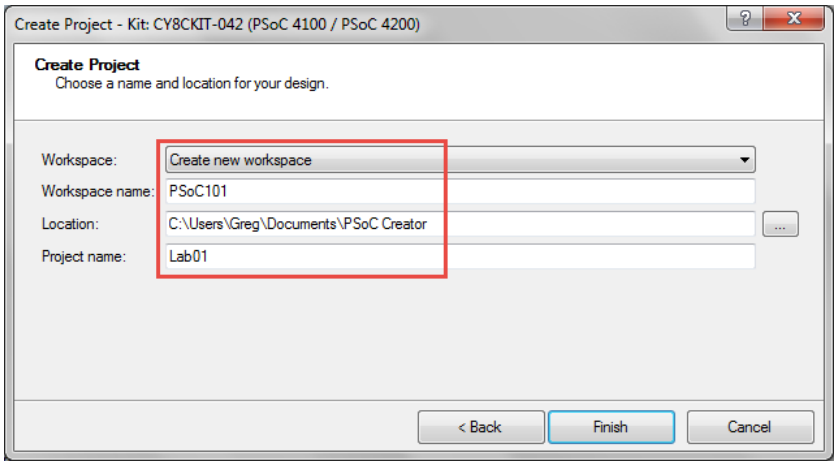
4. Assemble the Pioneer, Shield board and LCD as shown below. Plug the kit into a USB port on your computer using the cable supplied with the kit.



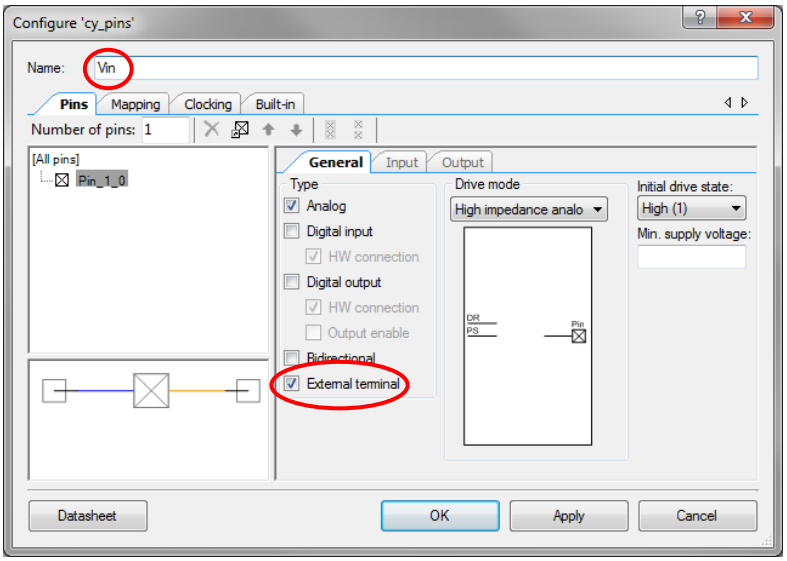
5. Create a new PSoC 4100 / PSoC 4200 design project



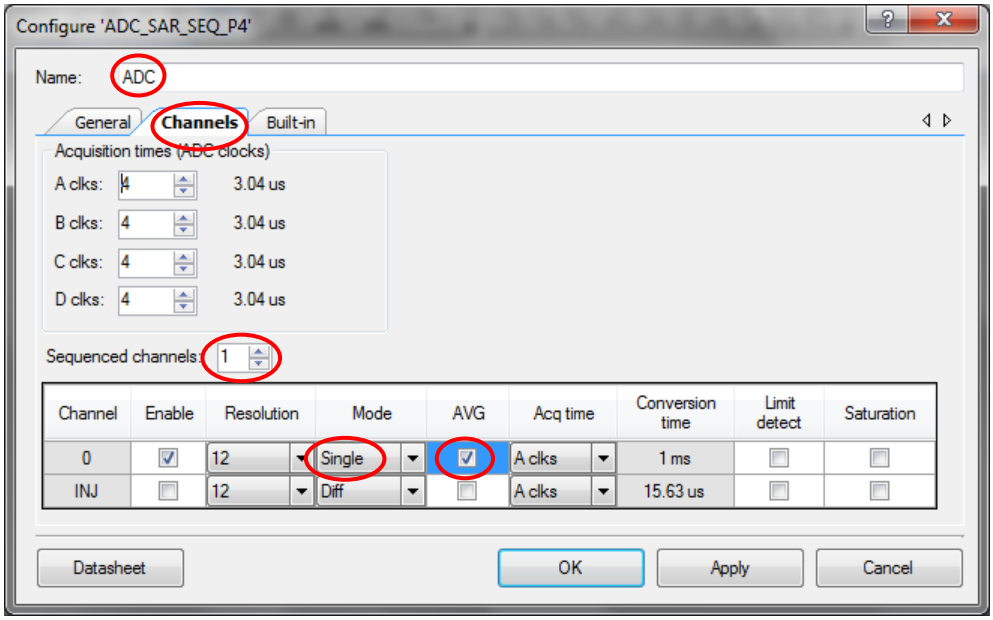


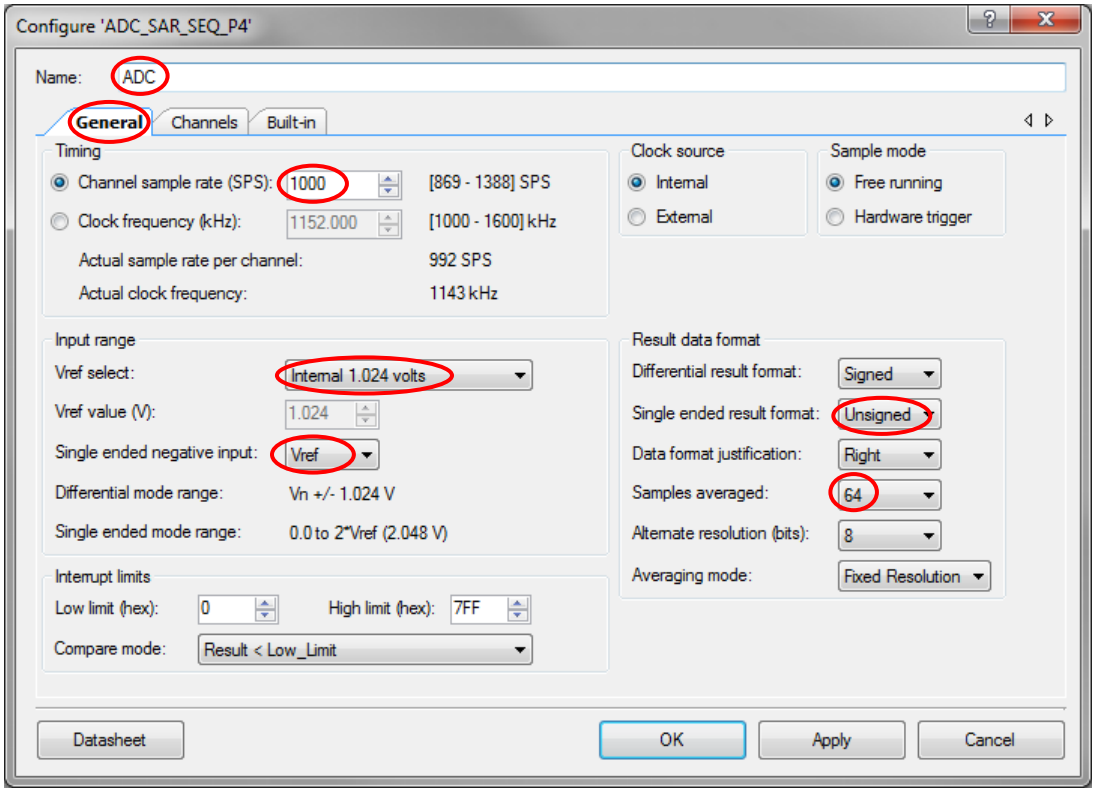


6. Add an analog pin to the schematic and name it “Vin” (see PIN description in the “Internal Components” chapter of this manual - chapter 5 - and the figure below). Enable the “External Terminal” connection.



7. Add a SAR ADC to the schematic and name it “ADC” (see the ADC description in the “Internal Components” chapter of this manual - chapter 5)
8. Configure the SAR (see the figures below)

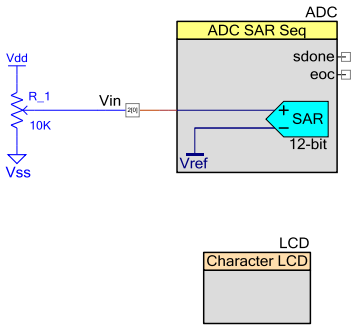




As seen in the figure above, the ADC is set for a reference of “Internal 1.024V” instead of the default value of “Internal 1.024V, bypassed”. Using an external bypass capacitor on the

reference allows the SAR to run faster since it provides additional stability on the reference voltage. In this case we are sampling slowly so the external bypass capacitor is not needed. If the SAR clock frequency is larger than 3MHz, then the bypass capacitor must be used. For frequencies lower than 3MHz it is not required but may still benefit the system if it is sensitive to switching noise. See the SAR datasheet for additional information.

9. Connect the analog pin "Vin" to the SAR input
10. Add Off-Chip components for a potentiometer (POT), power, and ground – see the POT description in the "External (Off-Chip) Components" chapter of this manual - chapter 4. Connect them to the external terminal of the pin as shown on the schematic below.
11. Add a Character LCD to the schematic and name it "LCD". Leave all other configuration as-is.
12. Once you schematic is done, it should look similar to the figure below.



13. Assign the pins as follows – double click on "Pins" under the Design Wide Resources in the Workspace Explorer in the left pane (if you don't see the Workspace Explorer window, use the menu item "Window->Reset Layout":
    a. Analog input voltage Vin -> P2[0]
    b. LCD:LCDPort[6:0] -> P2[7:1]
14. Do Build->Generate Application – this will create the APIs for you
15. Edit the firmware (shown a few pages below) – double click on "main.c" in the workspace explorer in the left pane to get to the firmware:
    a. Start the components in the initialization section
    b. In the infinite loop:
        i. Read the ADC's value
        ii. Display the Voltage to the LCD including the units of the measured value
16. Program the device.
17. Turn the POT and observe the voltage on the LCD change. Verify the voltage with a digital volt meter.

Firmware:

```
#include <project.h>

/* ADC channel 0 */
#define POT_CHAN (0)

int main()
{
    int16 counts = 0;    /* ADC result in counts */
    int16 mVolts = 0;    /* ADC result in mVolts */

    CyGlobalIntEnable;

    ADC_Start();
    ADC_StartConvert();
    LCD_Start();

    for(;;)
    {
        /* Get ADC result and convert to mV */
        counts = ADC_GetResult16(POT_CHAN);
        mVolts = ADC_CountsTo_mVolts(POT_CHAN,counts);

        /* Display value to LCD */
        LCD_Position(0,0);
        LCD_PrintNumber(mVolts);
        LCD_PrintString("mV   ");
    }
}
```

Alternates:

1. (L1.2) Add a hardware timer to only update the LCD once per second so that the least significant digit of the result does not change rapidly.

Blink an LED with Firmware (Example L2)

Description: Use a C program (a.k.a. firmware) to blink an LED at 1 Hz (a.k.a. 1/second)

Objective:

- Use a Devkit
- Learn simple firmware development
    - Learn about software delays
    - Learn about firmware control of pins
- Use PIN

Process:

1. Create a project
2. Add an output pin to the schematic
3. Configure the pin (see LED in the "External (Off-Chip) Components" chapter of this manual-chapter 4)
    a. Turn off the hardware (HW) connection since we will use firmware to drive the pin
4. Edit firmware (see the pin datasheet for the component API function to write a value to the pin – i.e. the <pin>_Write function) where <pin> is the name that you used for the component.
    a. In the infinite loop:
        i. Write a 1
        ii. Delay (hint: "CyDelay")
        iii. Write a 0
        iv. Delay

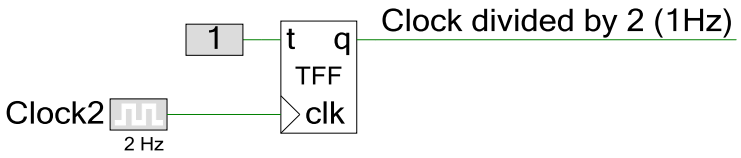
Alternates:

2. (L2.2) Blink faster or slower
3. (L2.3) Blink 2 LEDs at different rates
4. (L2.4) Alter the "duty cycle" (have the led on longer than it is off… but blink at the same frequency)

Blink an LED using a Clock (Example L3)

Description: Use the built-in PSoC Clock to blink an LED at 1 Hz (a.k.a. 1/second)

In PSoC 4, clocks can only drive component clock ("clk") inputs – they cannot directly connect to other digital input terminals. We will use a toggle flip flop with its "t" input tied high in order to convert the clock to a digital signal that is ½ the frequency of the input clock.



Objective:

- Demonstrate a hardware only project
- Use a Clock
- Use a PIN

Process:

1. Create a project
2. Add an output pin to the schematic
3. Configure the pin (see LED in the "External (Off-Chip) Components" chapter of this manual - chapter 4)
    a. Turn on hardware (HW) connection on the pin to drive it from the schematic
4. Add a clock to the schematic
5. Configure the Clock
6. Connect the clock to the pin through a toggle flip flop with "t" tied to logic High. Note that this will divide the clock by 2 so the input clock must be double the desired output frequency.
7. In this case, no firmware is required (the infinite loop will be empty)

Alternates:

5. (L3.2) Repeat by using the pin feature that allows a clock to drive out of PSoC 4 directly. (See the "Clock" description in the "Internal Components" chapter in this manual - chapter 5 - or read the pin component datasheet.)
6. (L3.3) Blink 2 LEDs at different rates

Use an external switch to control an LED (Example L4)

Description: Connect a mechanical switch through the PSoC to an LED

Objective:

- Use a Devkit
- Learn how to use a mechanical switch
- Learn how to use an LED

Process:

1. Create a project
2. Add a digital output pin to the schematic to drive the LED
3. Configure the digital output pin (see LED in the "External (Off-Chip) Components" chapter of this manual - chapter 4). The pin should have the HW connection enabled.
4. Add a digital input pin to the schematic to monitor the switch state
5. Configure the digital input pin (see mechanical switch in the "External (Off-Chip) Components" chapter of this manual - chapter 4). Our switches are active low so the pins need resistive pull up mode to pull the pin high when the switch is not pressed. The pin should have the HW connection enabled.
6. Connect the HW terminal from the input pin to the output pin with a wire
7. Assign the pins
8. In this case, no firmware is required (the infinite loop will be empty)

Alternates:

7. (L4.2) Have the light turn on/off in the opposite way. That is, have the LED turn on when the switch is pressed and turn off when the switch is not pressed (look at the "not" component).
8. (L4.3) Turn on/off two LEDs opposite from each other when the button is pressed.
9. (L4.4) Have an LED blink rate change when the switch is pushed (look at the clock, toggle flip flop and multiplexer components).

Control the intensity of an LED using a PWM (Example L5)

Description: Use a PWM to modify the 50% duty cycle of a clock to reduce the total current into an LED (makes it dimmer)

Objective:

- Use a Devkit
- Learn how to use PWM
- Learn how to use a Clock
- Learn how to use an LED

Process:

1. Create a project
2. Add a digital output pin to the schematic
3. Configure the pin (see LED in the "External (Off-Chip) Components" chapter of this manual - chapter 4)
4. Add a clock. Use a high enough frequency so that you don't observe the LED flashing.
    a. The human eye can't see switching faster than about 50Hz
    b. The PWM period will divide the clock so your clock should be at least 50Hz multiplied by the period
5. Add a PWM
    a. Configure the PWM to a 20% duty cycle. This will make the LED appear dimmer.
6. Edit firmware
    a. Start the PWM component in the initialization section
    b. The infinite loop will be empty

Alternates:

- (L5.2) View the output of the clock and PWM on an oscilloscope
- (L5.3) Set the PWM duty cycle using a POT (remember the ADC project)
    - You will need to write to the PWM's compare value
- (L5.4) Display the intensity (PWM duty cycle) on an LCD as a bar graph

Play sounds using buzzer (Example L6)

Description: Use a buzzer to play a “C” note

Objective:

- Use a Devkit
- Learn how to use a Clock
- Learn how to use a Buzzer

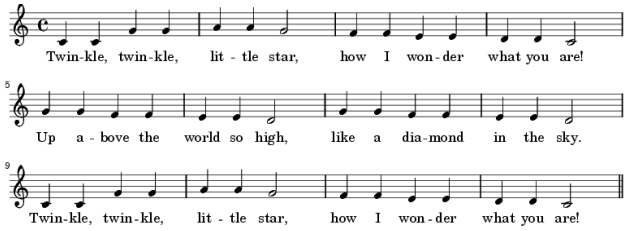
Process:

1. Create a project
2. Add a digital output pin to the schematic
3. Configure the pin (see Buzzer)
4. Add a clock
    a. A table of note frequencies can be found in the “Buzzer” section in the “External (Off-Chip) Components” chapter in this manual - chapter 4.
5. Connect the clock to the pin using a toggle flip flop, a PWM, or by configuring the pin to accept a clock directly (review previous labs if you need a refresher)
6. Edit firmware
    a. Start the PWM component (if you chose to connect the clock to a PWM)
    b. In the infinite loop:
        i. Turn on clock or PWM
        ii. Delay for time
        iii. Turn off clock or PWM
        iv. Delay for a time

NOTE: You can turn on/off the buzzer by using the slider switch on the shield board next to the buzzer.

Alternates:

- (L6.2) Play the note when a mechanical button is pressed
- (L6.3) Play the note when a CapSense button is pressed
- (L6.4) Play two notes using a digital multiplexer to switch between notes with a button press
- (L6.5) Use a CapSense slider to play a scale
- (L6.6) Play a song (Twinkle, twinkle little star)
    - The notes for the first line are: CC GG AA G FF EE DD C.



Use a Control Register (Example L7)

Description: Use a control register to select between sounds

Control registers are the means by which firmware can drive digital signals on the PSoC Creator schematic. They can provide access for 1 to 8 signals. See the “Internal Components” chapter of this manual - chapter 5 - for more details.

Objective:

- Learn how to use a control register
- Learn how to use a multiplexer

Process:

1. Create a project
2. Add a control register and configure it for 1-bit
3. Add a 2:1 multiplexer and connect the select input of the multiplexer to the control register
4. Connect two different clocks to the multiplexer inputs (using toggle flip-flops) to generate two different sounds
5. Connect the multiplexer output to the buzzer via an output pin
6. Edit firmware
    a. In the infinite loop:
        i. Write to the control register to select input 0
        ii. Delay for time
        iii. Write to the control register to select input 1
        iv. Delay for a time

Alternates:

- (L7.2) Add two pins and assign them to LEDs. Turn on one LED for each of the two buzzer sounds.
    - Hint: the control register output can connect to more than just the multiplexer input.

Use a Status Register (Example L8)

Description: Use a status register to monitor buttons

Status registers are the means by which digital signals on the PSoC Creator schematic can be monitored by firmware. They can provide access for 1 to 8 signals. See the "Internal Components" chapter of this manual - chapter 5 - for more details.

Objective:

- Learn how to use a status register to efficiently read the state of 4 buttons simultaneously

Process:

1. Create a project
2. Add a status register and configure it for 4-bits
3. Add four digital input pins and configure them to connect to buttons 1 – 4
4. Connect the digital input pins to the status register inputs
5. Connect a clock to the status register clock input.
    a. You can use an existing clock such as HFClk
6. Add a character LCD component
7. Edit firmware
    a. Start the LCD component
    b. In the infinite loop:
        i. Position the LCD cursor to (0,0)
        ii. Read the status register and display the value in hex on the LCD (use the character LCD PrintInt8 function)
            1. Notice how this allows you to read the state of all four buttons at once instead of reading each pin individually.

Alternates:

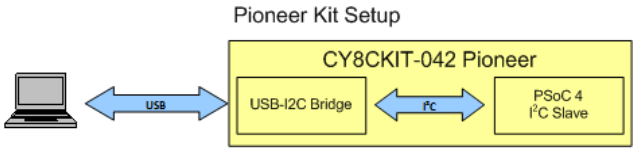
- (L8.2) Invert the button values in the firmware so that a pressed button returns a value of 1
- (L8.3) Invert the button values in the hardware so that a pressed button returns a value of 1

Plot an Analog Voltage using EZI2C and the Bridge Control Panel (Example L9)

Description: Add an EZI2C interface to the earlier project that measures an analog input voltage so that the value can be read and displayed by a PC.

I2C is a two wire (clock and data) communication bus used for communication between integrated circuits. I2C uses a single-master, multiple slave protocol. EZI2C adds a protocol on top of I2C to allow random access to a buffer. (More details to follow).

Cypress's Bridge Control Panel Software (included free and by default with PSoC Creator) along with the built-in bridge on the CY8CKIT-042 Pioneer kit can be used together as an I²C master with powerful I2C debugging capability. The setup is as shown here:

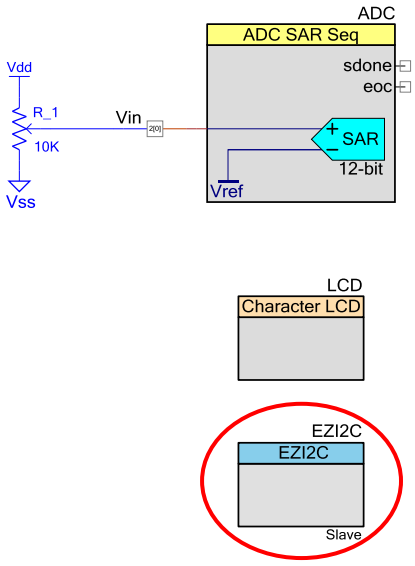


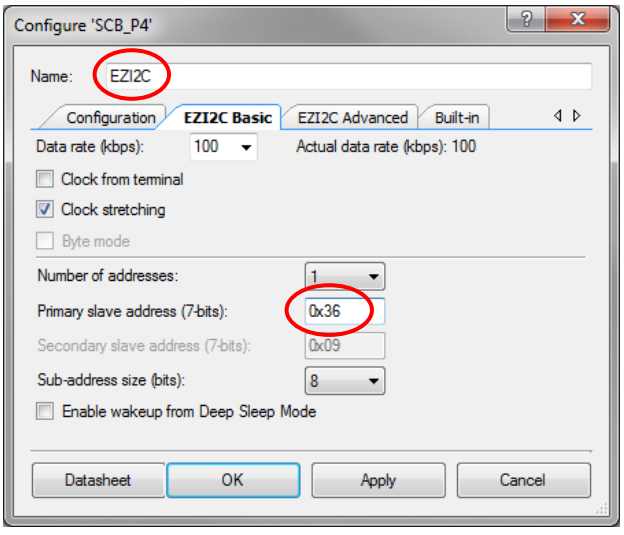
Objective:

- Demonstrate reading a value over the I2C interface
- Demonstrate plotting the value using the Bridge Control Panel on a PC

Process:

1. Start with the project which measures the POT and displays its value to the LCD (L1)
2. Add an EZI2C component to the schematic, name it "EZI2C", and configure it as shown below)





3. Modify the firmware. Start with the firmware from lab L1. Additions from L1 FW are shown in bold below:

```
#include <project.h>

/* ADC channel 0 */
#define POT_CHAN (0)
#define WRITABLE (0)

int main()
{
    int16 counts;      /* ADC result in counts */
    int16 mVolts;      /* ADC result in mVolts */
    uint8  i2cReg[2];  /* 2 byte I2C register */

    ADC_Start();
    ADC_StartConvert();
    LCD_Start();
    EZI2C_Start();

    /* 2 byte I2C buffer, 0 bytes writeable by the host, name is I2CReg */
    EZI2C_EzI2CSetBuffer1(sizeof(i2cReg), WRITABLE, i2cReg);

    CyGlobalIntEnable;

    for(;;)
    {
        /* Get ADC result and convert to mV */
        counts = ADC_GetResult16(POT_CHAN);
        mVolts = ADC_CountsTo_mVolts(POT_CHAN,counts);

        /* Display value to LCD */
        LCD_Position(0,0);
        LCD_PrintNumber(mVolts);
        LCD_PrintString("mV   ");

        /* Store ADC value in I2C buffer */
        i2cReg[0] = HI8(mVolts); /* upper byte of result */
        i2cReg[1] = LO8(mVolts); /* lower byte of result */
    }
}
```

The changes from the prior project to get EZI2C working are minimal. We only need to:

A. Define a buffer to hold I2C data (in this case a 2-byte array called I2CReg).
B. Start the EZI2C component.
C. Tell the I2C component about the buffer (EZI2C_EzI2CSetBuffer1). This function takes three arguments:
    i. Size of the I2C buffer in bytes
    ii. Number of bytes in the buffer that the master can write starting from the beginning. We used a macro called WRITABLE to set this to 0 since the master will only be reading data.
    iii. The location of the buffer
D. Copy the ADC data to the I2C buffer so that the master can access it.

Everything else is handled internally by the EZI2C component. When the master sends data to the slave it will appear in the buffer and can then be accessed by the slave (not done in this example). When the master requests data from the slave, it will get whatever is currently in the buffer.

4. Assign the EZI2C pins as follows:
    a. EZI2C:scl = P3[0]
    b. EZI2C:sda = P3[1]

    Note that these pins are connected on the Pioneer board's built-in I2C to USB Bridge. You do not need to add any wires.

5. Program the device.
6. Follow the instructions below to plot the values on your PC using the Bridge Control Panel.

Plotting Values using the Bridge Control Panel (refer to the following figures for these steps):

A. Open the Bridge Control Panel program (Start -> All Programs -> Cypress->Bridge Control Panel -> Bridge Control Panel).
B. Connect to the KitProg in the Bridge Control Panel and make sure the device is powered (Figure 1).
C. If the kit does not connect and you see a message that the kit firmware is out of date then follow these steps:
    i. Open PSoC Programmer (Start -> All Programs -> Cypress -> PSoC Programmer -> PSoC Programmer)
    ii. Connect to the KitProg if not done automatically
    iii. Select the "Utilities" tab
    iv. Click "Upgrade Firmware"
    v. If you don't see a success message, click on "Upgrade Firmware" a second time.
    vi. Close PSoC Programmer, and reconnect to the KitProg in the Bridge Control Panel
D. Go to "Tools -> Protocol Configuration -> I2C" and verify that 100KHz is selected (Figure 2).
E. Go to "Chart -> Variable Settings" and rename one of the variables to "v" with the type "int", and check the "Active" box (Figure 3). Note that the "v" is lowercase.
F. Click the "List" button and verify that a device with 7-bit address 36 shows up (Figure 1).
G. Enter the write and read commands as shown below (use Ctrl-Enter to add a new line) (Figure 1).
H. Execute the write command and then the read command (to execute a command, click on the line containing the command and press Enter). The read command will read the ADC value for your project (Figure 1).
    i. Note the "+"after each byte. This indicates an ACK, meaning that the slave acknowledged the request. If you see a "-", that indicates a NAK, which means that the slave did not acknowledge the request for some reason (i.e. the transaction did not succeed).
I. Go to the "Chart" Tab (Figure 4).

J. Click “Repeat” to see the voltage plotted real time (Figure 4).
K. Turn the knob on the POT to see the voltage value change.
L. Click “Stop” when done (Figure 4).
M. Note: you must disconnect from the bridge to re-program the PSoC (Figure 1).

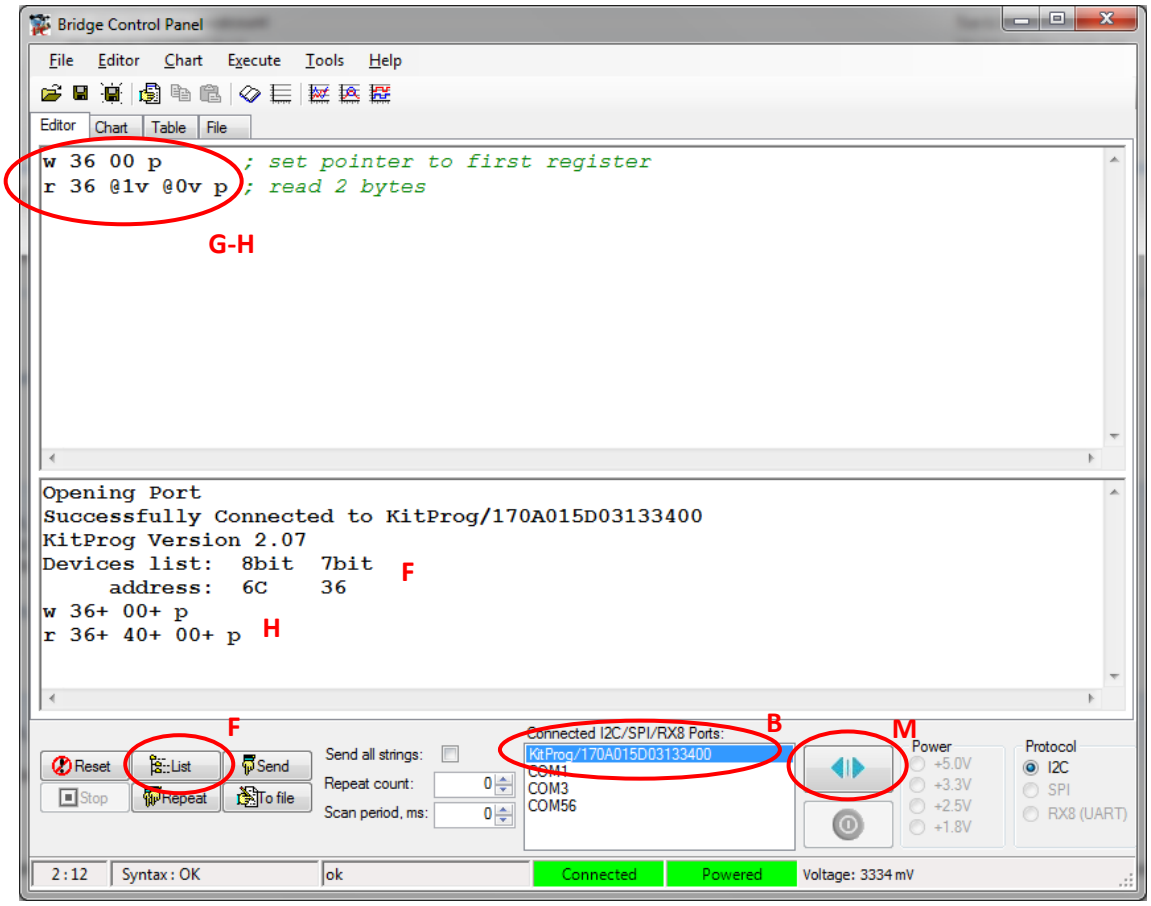


Figure 1: Bridge Control Panel Main Window

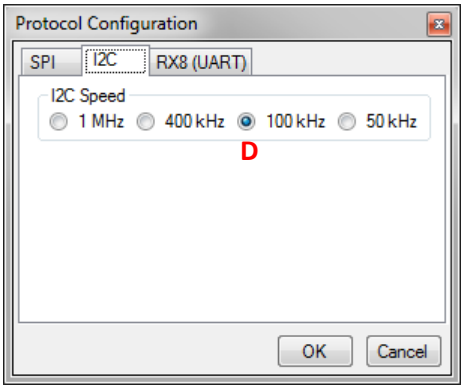


Figure 2: Bridge Control Protocol Configuration

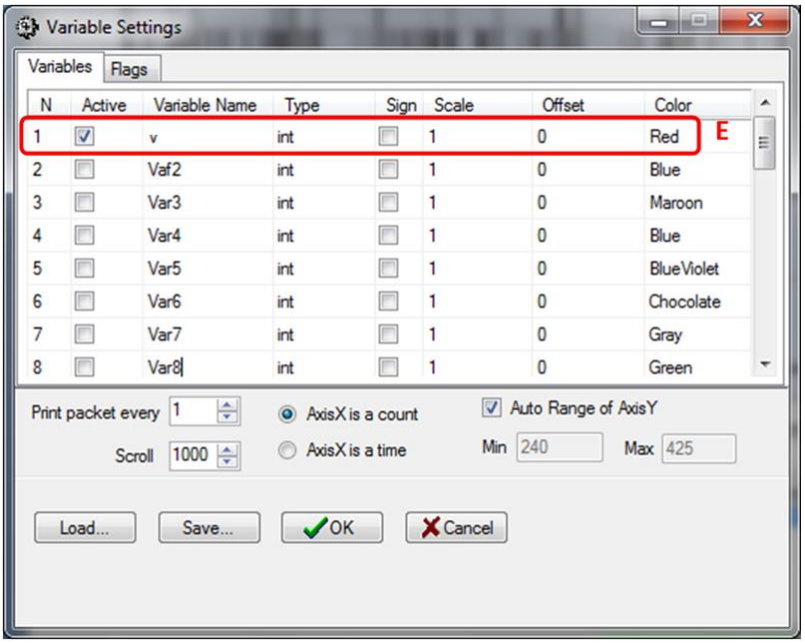


Figure 3: Bridge Control Panel Variable Setup

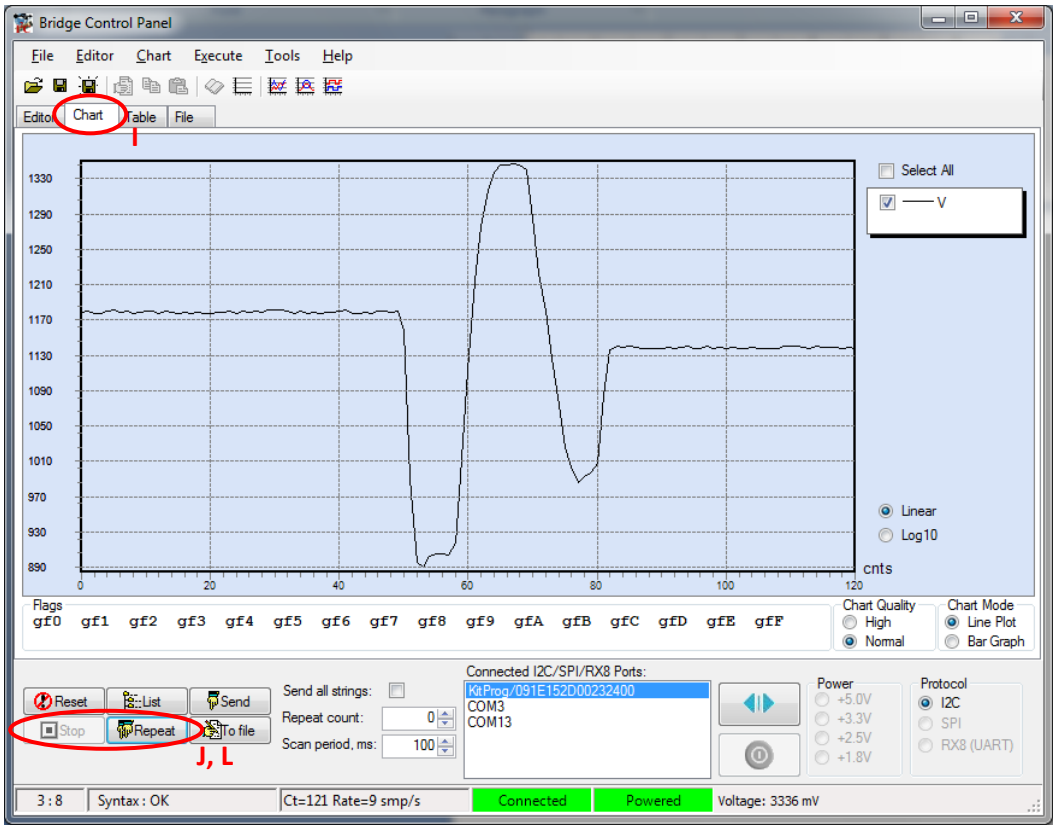


Figure 4: Bridge Control Panel Chart Window

EZI2C Protocol Explanation

As mentioned previously, EZI2C adds a protocol on top of I2C which allows the master to have random access to the data in the I2C buffer. The EZI2C component can be configured to have either 1 or 2 bytes of address offset. The default is 1 byte which means the buffer can be up to 256 bytes.

The first byte (or first two bytes if configured for a 2 byte offset) sent by the master in a write sequence is an offset which specifies which location in the buffer to start from. This offset will also be used in any following read sequences. The best way to understand the offset is with a few examples. Let's assume we have a 1 byte offset and have set up a 4 byte buffer with the initial values as shown here:

Location	Value
0	0x10
1	0x20
2	0x30
3	0x40

Now, let's execute a series of commands:

w 36 01 55 AA p ; Set the offset to location 0x01 in the buffer and write values of 0x55 and 0xAA. We now have:

Location	Value
0	0x10
1	**0x55**
2	**0xAA**
3	0x40

r 36 x x x p ; Read back 3 bytes. Since the offset was set to 0x01 by the previous write, we will get: 0x55, 0xAA, 0x40

r 36 x x x p ; Read back 3 bytes. The offset remains at 0x01 so we will get the same values again: 0x55, 0xAA, 0x40

w 36 00 p ; Set the offset to the 1st location in the buffer

r 36 x x x p ; Read back 3 bytes. Since the offset is now set to 0x00, we will get: 0x10, 0x55, 0xAA

CapSense (Example L10)

Description: Use CapSense buttons and sliders to control an LED

Objective:

- Learn how to use CapSense buttons
- Learn how to use CapSense sliders
- Use the CapSense tuner

Process:

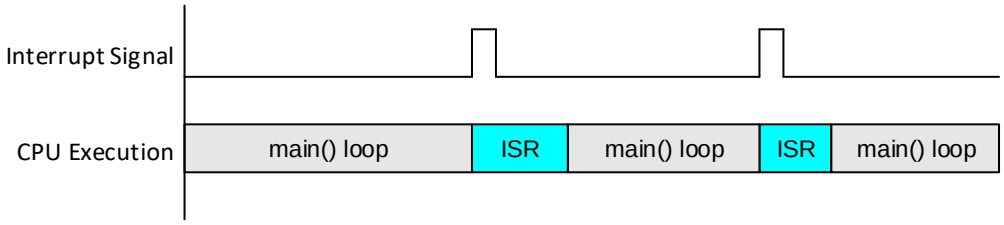
1. Create a project
2. Add a CapSense component to the project – use “CapSense” not “CapSense CSD”.
    a. “CapSense CSD” is an older version that supports self-cap mode only and is not recommended for new designs. The new version supports both self-cap and mutual cap modes.
3. Configure CapSense for 1 button
    a. See the CapSense Button section of the “Internal Components” chapter of this manual - chapter 5. Additional background information on CapSense can be found in the CapSense Fundamentals section of that chapter.
4. Assign the button and Cmod to the appropriate pins.
5. Edit firmware
    a. Turn on CapSense, setup the widget, and start an initial scan
    b. In the loop:
        i. Process the button when a scan is done and then start a new scan
        ii. Light the LED if the button is pressed

Alternates:

- (L10.2) Use a CapSense slider to change an LED’s intensity (see the CapSense sliders section of the “Internal Components” chapter of this manual - chapter 5)
- (L10.3) Use the CapSense tuner to plot CapSense signals (see the CapSense tuning section of the “Internal Components” chapter of this manual - chapter 5)

Use an Interrupt to Blink an LED (Example L11)

Interrupts are often an important part of any embedded application. They free the CPU from having to continuously poll for the occurrence of a specific event and, instead, notify the CPU only when that event occurs. Thus the CPU of the embedded system can continue its work as normal and divert its attention to a new task when it is notified.



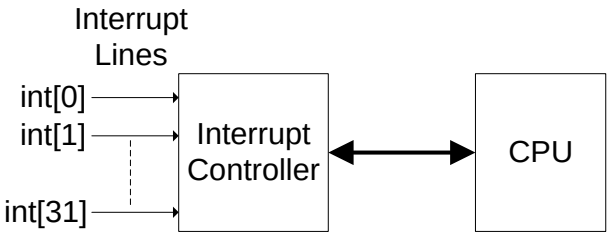
In PSoC, interrupts are highly flexible and are often used to communicate the status of on-chip peripherals to the CPU. Here are some of the features of interrupts for PSoC 3, PSoC 4, and PSoC 5LP:

- Up to 32 interrupts into the CPU.
- 8 programmable interrupt priority levels for PSoC 3 and PSoC 5LP, 4 levels for PSoC 4.
- Programmable interrupt vectors for each interrupt.

Basic PSoC Interrupt Architecture

PSoC contains an interrupt controller, which acts as the interface between the interrupt lines and the CPU. In traditional microcontrollers there is no interrupt controller (or minimal at best) - the interrupt source is hard-wired to each interrupt line. PSoC, on the other hand, gives you the flexibility to choose the interrupt source for each interrupt line. This flexible architecture enables any digital signal to be configured as an interrupt source. Interrupt lines can be connected to any number of sources (such as a UART, I/O pins, etc).

There are 32 interrupt lines – int[0] to int[31] – in PSoC. For PSoC 4, each interrupt line can be assigned one of four priority levels (0 to 3), where 0 is the highest priority. The concept of priority enables higher priority interrupts to interrupt lower priority interrupts. Interrupts with lower or equal priority will wait until higher priority interrupts complete first.



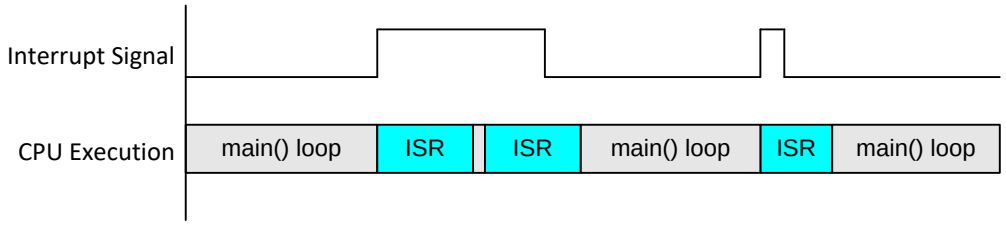
Each interrupt line is associated to an interrupt vector address, which refers to the starting address of the interrupt executable code. This is typically referred to as the Interrupt Service Routine (ISR) which is really just a function that is executed whenever an interrupt is received. The interrupt controller sends the interrupt vector address of an interrupt line to the CPU along with the interrupt request signal. The CPU receives this information and branches to this vector address after receiving an interrupt request.

In a traditional microcontroller, the interrupt vector address is fixed for each interrupt line. Typically, a "JUMP" instruction is placed in that fixed address to branch the CPU execution to the actual ISR. With PSoC you can instead dynamically configure the interrupt vector address. The CPU execution can be directly branched to any ISR code when the interrupt occurs. Thus with PSoC, interrupt execution latency is reduced.

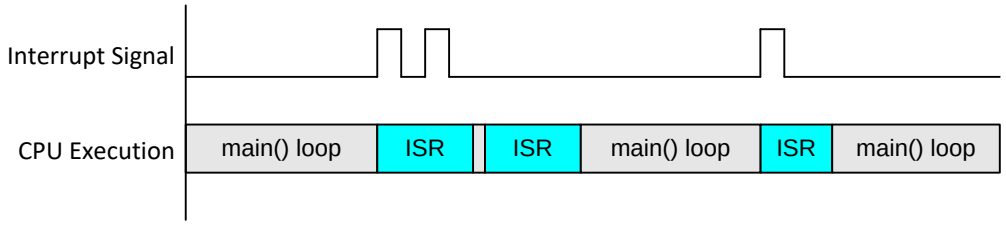
Process Flow with Interrupts

Assuming that the interrupt line is initially inactive (logic low), the following sequence of events explains how level-sensitive and edge-triggered interrupts are handled:

1. On a rising edge event on the interrupt line, the interrupt controller registers the interrupt request. The interrupt is now in the pending state, which refers to interrupts whose requests have not yet been serviced by the CPU.
2. The interrupt controller then sends the interrupt vector address along with the interrupt request signal to the CPU. When the CPU starts executing the ISR for the interrupt request, the pending state of the interrupt is cleared.
3. What happens next depends on how the interrupt triggering is set:
    a. Level-sensitive Interrupt: After completing the ISR, if the interrupt line is still high, the ISR is executed again. The ISR is continually executed as long as the interrupt line remains high.



    b. Edge-Triggered Interrupt: As the name implies, an edge is required to cause an interrupt to occur. Therefore, no matter how long the interrupt line remains high, the ISR will execute only a single time for a single interrupt line pulse. If one or more rising edges on the interrupt line occur while the ISR is being executed, they are logged as a single pending request. The pending interrupt is serviced after the current ISR execution is complete.



Interrupt Sources

There are several interrupt sources in PSoC which are described below:

Fixed Function Interrupt Sources: These are a predefined set of interrupt sources from the on-chip peripherals. Examples of these include the interrupt signals from the fixed function timers, counters, and so on.

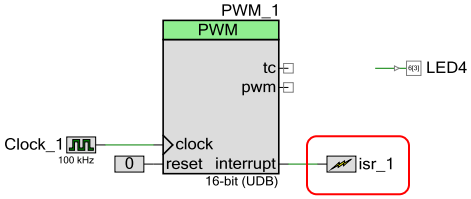
UDB Interrupt Sources: Any digital signal can be configured as an interrupt source by routing it through the Digital System Interconnect (DSI). These sources are broadly referred to as UDB interrupt sources because most of these interrupt sources are from the universal digital blocks (UDBs) in PSoC. The UDB is the basic building block for configurable digital peripherals, such as UART, SPI, I2C, timers, counters, and PWMs.

Pins: input pins can trigger an interrupt on rising, falling, or both edges.

Interrupts and PSoC Creator

Interrupt Component

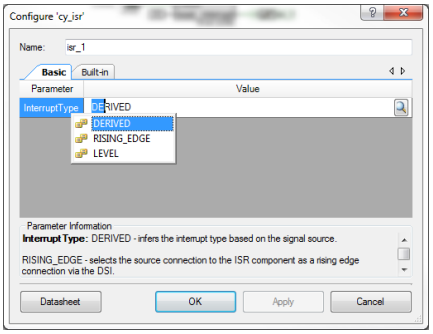
An Interrupt Service Routine (ISR) is created when an Interrupt component is placed in a PSoC Creator schematic and the project is built. Many components have an interrupt signal for attaching an interrupt component to, but this is not a strict requirement. The interrupt component is a resource that can be attached to most digital logic signals.



Interrupt Sensitivity

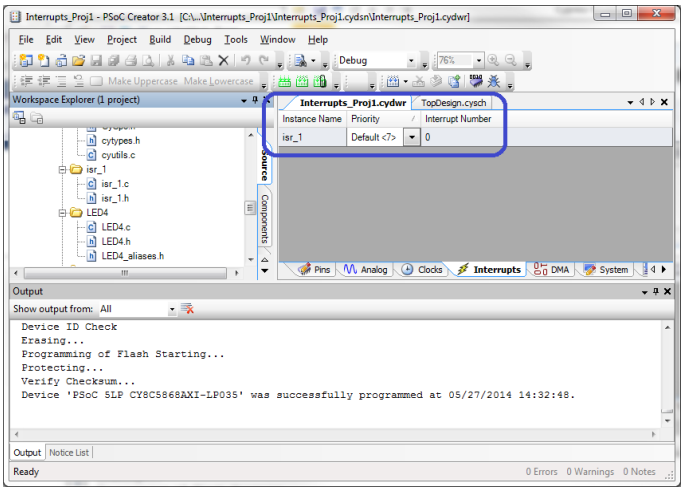
An interrupt's response depends on the signal source. Typically the desired response can be derived from the source the interrupt component is connected to.

- RISING_EDGE – Triggers the interrupt on the rising edge of the source signal. If this option is selected, a rising edge on the input is converted into a pulse of BUS_CLK period and is sent to the interrupt controller.
- LEVEL – Selects the source connected to the interrupt as a level-sensitive connection through the DSI. The source signal input is directly passed to the interrupt controller.
- DERIVED – Inspects the driver of the interrupt signal and, when connected to a fixed-function block (I2C, USB, CAN, and so on), derives the interrupt type based on what it is connected to.



Interrupt Priority

Interrupt priority is selectable through Design Wide Resources > Interrupts. Any interrupts that are available within a project are shown in the Interrupts tab. Again, the higher the interrupt number the lower the priority. Thus an interrupt with priority 0 has the highest priority and can interrupt any lower priority interrupt.



Interrupt Service Routine (ISR)

The Interrupt Service Routine is a function that you write to contain the code that is executed each time a particular interrupt is called. This function can be placed anywhere in your project such as main.c or in a separate file if you choose. It can take no arguments and must return no value.

There are two ways to indicate where the ISR for a given interrupt is located. You can use an automatically generated callback function name for the ISR or you can use any function name that you prefer and specify the name of the function when you start the interrupt. In the first case, you must define a macro to enable the callback function (#define) and declare the function in the file cyapicallbacks.h. In the second case, the interrupt must be started using "StartEx" instead of "Start" so that you can specify the function name.

We will use the second method here. If you prefer the first method, look in the generated .c file for your interrupt to find the name of the function required and the macro that must be defined to enable it (search for the word "CALLBACK" in the .c file). You can also search in the PSoC Creator help for "Macro Callbacks".

As mentioned above, in order to use you own interrupt function name, you start the interrupt using the "StartEx(addr)" function instead of the "Start()" function. The argument to StartEx is the address of the ISR function. If the ISR function is located below main then the ISR function must be declared using "CY_ISR_PROTO(addr)". The ISR itself is defined using "CY_ISR(addr)". An example is shown below for an ISR called "my_ISR".

```
/*******************************************************************
* ISR example using StartEx
********************************************************************
*
* Summary:
*   Each time an interrupt occurs, a count value is incremented
*
*******************************************************************/
uint8 count = 0; /* Count number of interrupts */

CY_ISR_PROTO(my_ISR); /* Declare the ISR */

/*******************************************************************
* Main function
*******************************************************************/
int main()
{
        CyGlobalIntEnable;

        isr_1_StartEx(my_ISR); /* Start interrupt */

        for(;;)
        {
                        /* Do nothing */
        }
}

/*******************************************************************
* ISR function
*******************************************************************/
CY_ISR(my_ISR)
{
        count++;  /* Increment counter on an interrupt */
}
```

Interrupt Macros and APIs

There are several APIs automatically generated for Interrupt components. The APIs that are likely most useful are briefly noted here:

- void ISR_Start(void) – Starts up the interrupt to function.
- void ISR_StartEx(addr) – Starts up the interrupt with the ISR address set to “addr”.
- void ISR_Enable(void) – Enables the interrupt to the interrupt controller.
- void ISR_Disable(void) – Disables the interrupt.
- void ISR_SetPriority(uint8 priority) – Sets the priority of the interrupt.

Refer to the CyInterrupt component datasheet for more information.

There are several macros and APIs that are a part of Creator. The following is a selection of macros and APIs that are particularly useful for interrupts:

- uint8 CyEnterCriticalSection(void) – This disables interrupts and returns a value indicating whether interrupts were previously enabled (the actual value depends on the device architecture).
- void CyExitCriticalSection(uint8 savedIntrStatus) – This re-enables interrupts if they were enabled before CyEnterCriticalSection was called. The argument should be the value returned from CyEnterCriticalSection.
- CyGlobalIntEnable – Macro statement that enables interrupts using the global interrupt mask.
- CyGlobalIntDisable – Macro statement that disables interrupts using the global interrupt mask.

Refer to the CyBoot component datasheet for more information (this is also known as the System Reference Guide). The “Interrupts” and “System Functions” sections contain information on the functions listed above.

Additional Information

Clearing Interrupts

In some cases, you need to clear the interrupt in your code or else it will just re-pend right away. For example, the pin component has an interrupt terminal called “irq”. If you connect a pin’s irq terminal to an interrupt component you will need to call the GPIO’s “ClearInterrupt” function. Typically, this is done inside the ISR itself so that once the interrupt is serviced it is also cleared at the same time before the ISR exits. Refer to each component’s datasheet to see if it requires an interrupt clear API call.

The interrupt component itself also has an API to clear a pending interrupt called “ClearPending”. Typically this is not needed since entering the ISR automatically clears the pending interrupt. However, this API may be useful to clear interrupts that pend while interrupts are disabled (assuming that you want to ignore those particular interrupts).

Latency

One reason for enabling an interrupt is to get a reduced latency benefit. In the event of an interrupt the core stops whatever it is doing to go work on the ISR; therefore, the latency is generally very low for a high priority or single interrupt. Note that this is a singularity. When multiple interrupts are enabled concurrently, latency cannot be guaranteed. For example, if multiple interrupts of the same priority are pending, all but one of the interrupts will remain pending until the first ISR is finished. Latency is no longer guaranteed since there is some waiting for other services to finish.

Synchronization

Interrupts can be a major source of frustration when an error is made in code. For example, a large memory structure may be operated on in an interrupt. In a similar manner a second interrupt may also desire to operate on the same memory structure. If one ISR is interrupted by another and they both are operating on the same memory structure, how does one ISR know it is operating on good data? The answer is it cannot possibly know without some forced synchronization to maintain data coherency.

General Guidance

- Minimize the use of interrupts. If there is no latency requirement or the requirement is so long that it does not matter, then it is probably best to eliminate the interrupt altogether.
- When using interrupts, keep the executable code within the interrupt as short as possible. This insures that processing time will not be overly consumed at the expense of other processes. In general, very long interrupts can degrade the 'real-time' character of an embedded real-time system. One technique is to only set a flag in the ISR. The flag is tested in the main code loop with an "if" statement and any further operations required are performed within the "if" statement block (and the flag is reset).
- With the above in mind, you should NEVER use CyDelay inside an interrupt service routine.
- Interrupt latency is never zero. There is always overhead on the CPU to save the current context of the core. Although the latency is low, there is some code and execution time expense by using an interrupt.
- Avoid calling functions from within interrupts. Reentrancy will be required for functions that are used in multiple places, and this adds code and degrades execution performance.

References

Much of the material found in this section is taken from application note AN54460 "PSoC® 3, PSoC 4, and PSoC 5LP Interrupts". See that document for additional information:

http://www.cypress.com/go/AN54460

Lab Exercise

Description: Create a project to blink an LED. The LED must be driven via firmware within an interrupt.

Objective:

- Learn how to use interrupts on PSoC using PSoC Creator.

Process:

1. Connect your computer to the USB port of the kit provided.
2. Create a new project and place:
    - PWM component.
    - Clock component.
    - Interrupt component.
3. Connect the Interrupt component to the PWM's interrupt signal.
4. Connect the clock to the PWM's clock input.
5. Configure the clock and the PWM to generate a total signal period of 0.5 sec. There are multiple ways to achieve this. Make sure you enable the appropriate interrupt from the PWM.
6. Place a Digital Output Pin and configure it for firmware control ("HW Connection" option unchecked). Connect the pin to one of the LEDs.
7. Generate the application. This generates the templates for the ISR as well all the code.
8. Write the firmware:
    - Start the PWM.
    - Start the ISR (remember to use StartEx if you want to place your ISR in main.c).
    - Use the CY_ISR_PROTO(MyISR) and CY_ISR(MyISR) macros to define your ISR function.
    - Write the ISR code to toggle the LED every time the interrupt is triggered.
    - Make sure to clear the PWM's interrupt inside the ISR.
        - If you use a UDB PWM, clear the interrupt by reading the status register.
        - If you use a TCPWM, clear the interrupt by calling the ClearInterrupt API function with the appropriate mask.
    - Enable global interrupts.
    - The main super-loop can be completely empty in this case (even better, you could put the chip into sleep mode inside the super loop so that the CPU shuts down until an interrupt occurs).
9. Compile and program.
10. Verify that the LED blinks at 1Hz.

Alternates:

- (L11.2) Use a pin interrupt to toggle an LED
    1. Open your project from the previous lab and add:
        - Digital output pin (firmware controlled)
        - Digital input pin (firmware controlled).

    - Interrupt Component
2. Assign and configure the output pin to drive an LED.
3. Assign and configure the input pin to the mechanical button.
    - Hint: you will need to enable the pull-up resistor on the pin.
4. Configure the input to generate an interrupt on a falling edge.
5. Generate the application.
6. Write the firmware:
    - Start the ISR. Use the StartEx function to point to an ISR function that is contained in main.c
    - Use the CY_ISR_PROTO(MyISR) and CY_ISR(MyISR) macros to define your ISR function.
    - Write the ISR to toggle the state of the LED.
    - Enable global interrupts.
    - The main super-loop should be completely empty.
7. Compile and program.
8. Push the button and notice that the LED toggles each time you press the button.

Using the Debugger (Example L12)

Most PSoC devices contain an on-chip debugging block. This block uses SWD and SWV (serial wire debug and serial wire viewer) to allow you to use PSoC Creator's debug features to peek into the run-time execution of your project. Some of the things you can do include:

1. Stop code execution at a point of your choosing (breakpoint).
2. Stop code execution on a specific event (conditional breakpoint).
3. View and even modify the values of variables, memory, and registers.
4. Single step through the code line by line.
5. Globally enable or disable interrupts.

Build Configurations

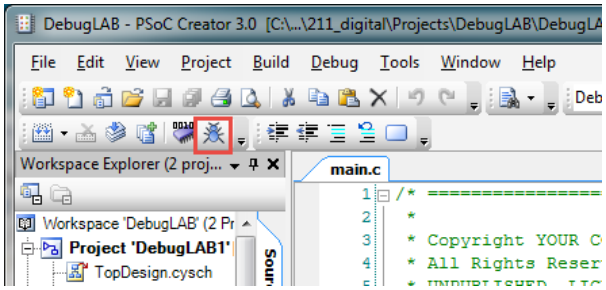
Each PSoC Creator project has two build configurations: Debug and Release. Each of these configurations can be independently customized but the main default difference is that the Debug configuration has compiler optimization disabled. The result is that it is possible to single step through code more closely since the compiler has not removed or re-arranged anything in the code. However, it is still possible to run the debugger on the Release configuration.

Hardware Vs. Firmware

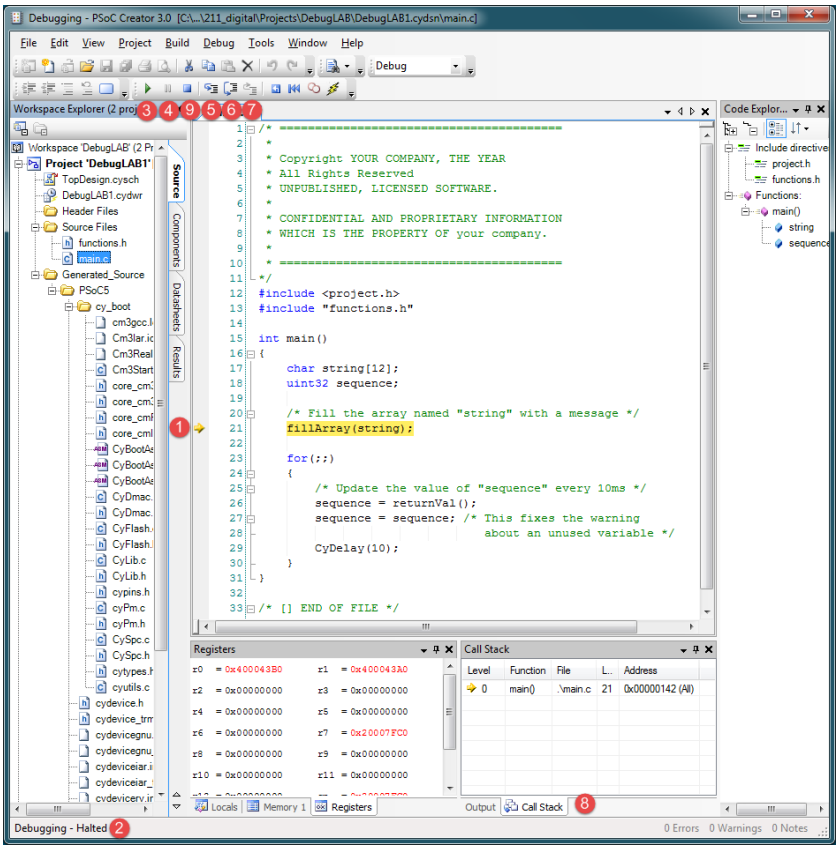
By adding various breakpoints, the debug block halts execution of the CPU when certain conditions are met. However, clocks are not stopped and none of the hardware is stopped. So, if you have hardware in your design it will continue to operate even when the CPU is halted by the debugger. You will need to take this into account when debugging.

Running the Debugger

To start the debugger, choose "Debug->Debug" from the PSoC Creator menu, press "F5", or click on the bug icon in the toolbar (circled in the figure). The debugger will then build the firmware (if necessary), program the PSoC, and halt execution at the start of main.



A yellow arrow will indicate the instruction that the debugger has stopped at (#1) (the line that the yellow arrow points to has not yet been executed). The lower left corner of the window shows the current state of the debugger – in this case, it says “Halted” (#2).



If you use the menu item “Debug->Resume Execution”, or hit “F5”, or press the green arrow icon (#3), the firmware start to execute and we will see “Running” in the lower left corner. We can use “Debug->Halt Execution”, or hit “Ctrl-Alt-Break” or press the Pause icon (#4) to stop execution at whatever point the firmware is currently at. This may take you to some location outside of main.

Once the CPU is halted, you can again restart execution or you can single step through the code. The three features available are “Step Into” (#5), “Step Over” (#6), and “Step Out” (#7). There descriptions are as follows:

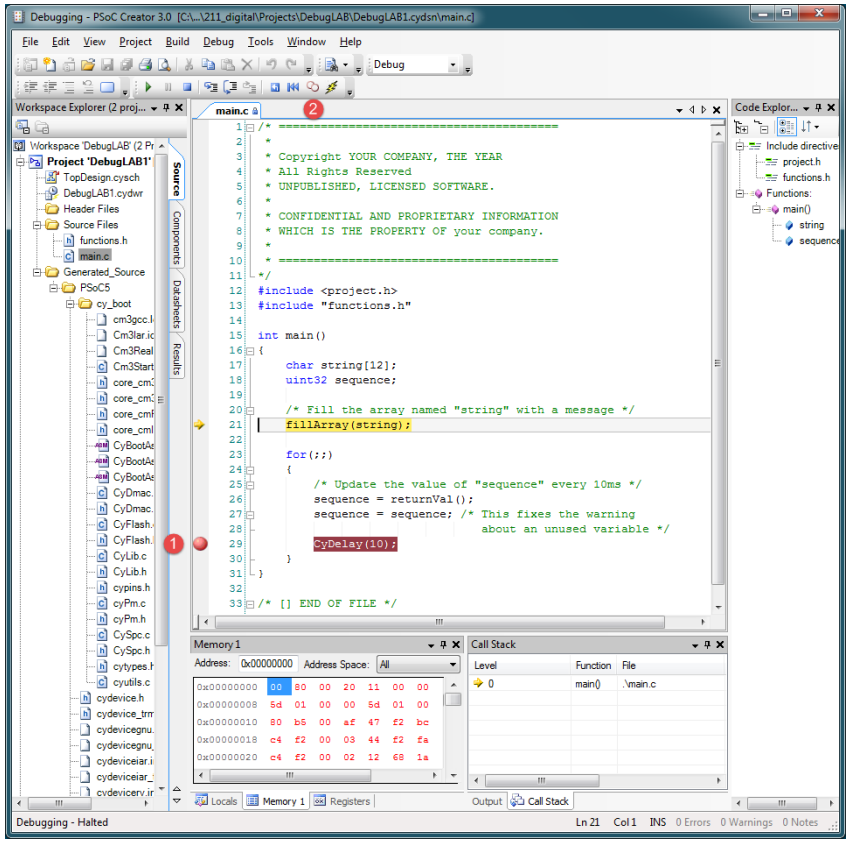
Step Into	Executes a single line of code. If the line is a function call, the debugger will break at the first instruction in the function. If the line is not a function call, the debugger will break at the following line of code.
Step Over	Executes a single line of code. The debugger will break at the following line of code. If the current line of code is a function call, it will execute the function without stopping inside the function. The debugger will break at the line of code after the function.
Step Out	Finishes executing the current function. The processor is allowed to run until the current function has finished. It will halt again at the first instruction after the function call.

The Call Stack tab lists the functions that have been called to get to the point that the CPU is stopped in the code (#8). This can be useful if the program ends up in an unexpected location.

Once done debugging, use “Debug->Stop Debugging”, hit “Shift-F5”, or press the blue square (#9).

Breakpoints

Instead of using the “Halt Execution” command, you can stop execution at a predictable place by using breakpoints. To do this, click the left mouse button in the vertical grey area to the left of the code. A red circle will appear for any breakpoints that you set (#1). Click on top of an existing breakpoint to remove it. Once the CPU reaches an instruction with the breakpoint, code execution will be halted and the yellow arrow will appear over the breakpoint. Again, note that the line that the arrow points to has NOT been executed. If you “Resume Execution” the code will continue running until another breakpoint is reached (or the same breakpoint on the next loop through the firmware).

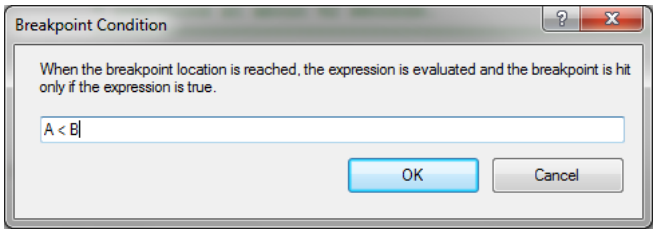


You can add/remove breakpoints to your project even when the debugger is not running. The breakpoints are saved with your project.

You can temporarily disable a breakpoint without removing it by right-clicking on it and choosing “Disable”. The red circle will then change to a hollow circle.

You can disable/enable all breakpoints in a project by using “Debug->Disable All Breakpoints” and “Debug->Enable All Breakpoints” or by clicking the button (#2).

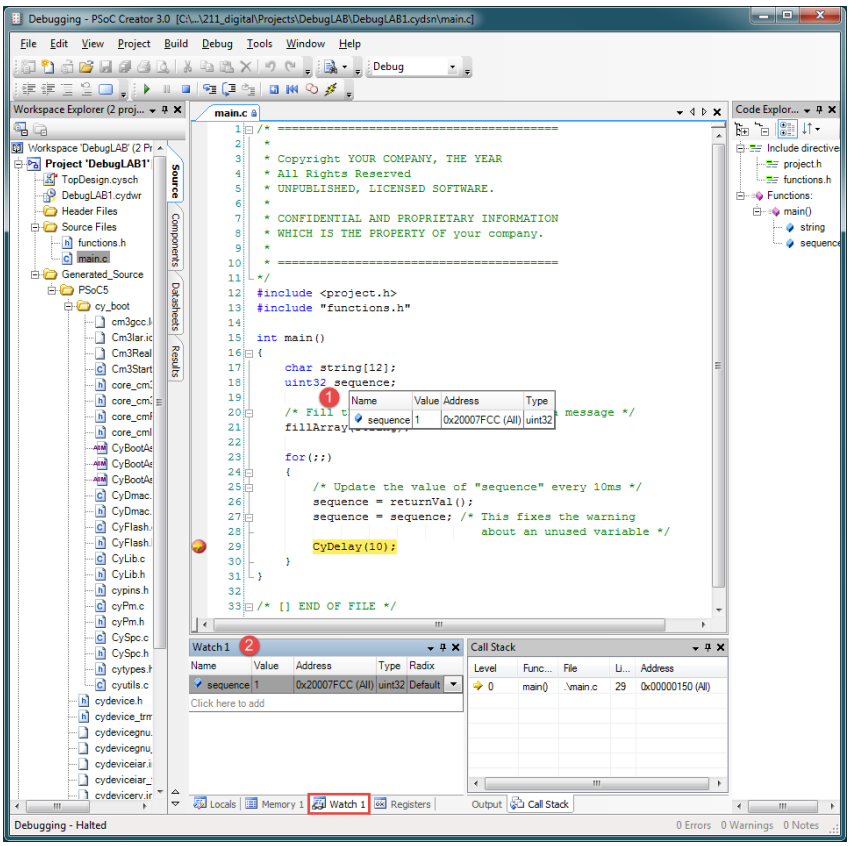
You can add conditions to breakpoints so that they only cause the CPU to halt when the condition is met. The condition can be any valid C expression. For example, you could have a condition to stop when variable A has a value that is less than the value of variable B.



To add a condition to a breakpoint, right-click on the breakpoint, select “Condition…” and add the desired expression in the box.

Viewing/Modifying Variables

Whenever the CPU is halted, you can examine and in some cases modify values. This includes variables in your code as well as direct access to the memory and CPU registers.

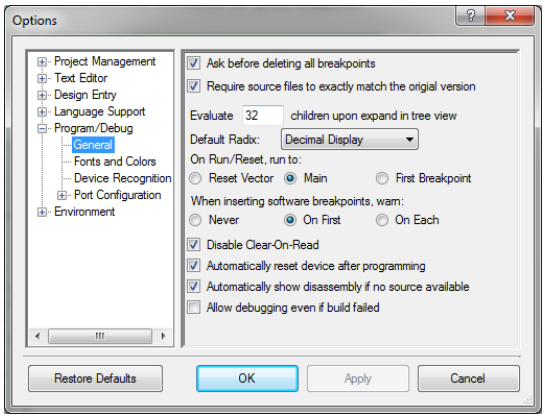


To view the value of a variable you can hover over it in the code window (#1). You can also right click on the variable and select "Add Watch". This will add the variable to the Watch window (#2) so that you can easily see it any time the CPU is halted. As with most Creator windows, the Watch window can be moved around in different locations or can be undocked, depending on your preference. In addition to the value of the variable, you can see its location in memory and its type.

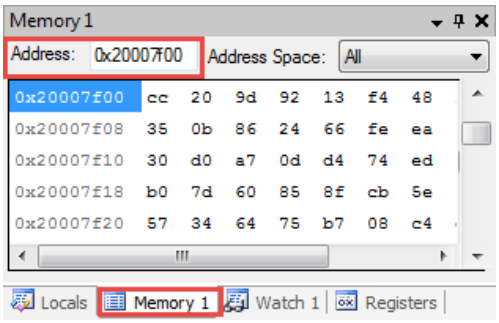
Variables whose values have changed since the last time the CPU was halted will show up in red text in the watch window.

To modify the value of a variable, click on the value, enter the new value, and press Enter. This can be done either when hovering over the value in the code or from the Watch window. The new value will be seen by the processor when it re-starts execution. This can be useful for testing program conditions that are otherwise difficult to reach.

In the watch window you can change the radix used to display the value (binary, octal, decimal, hex). You can change the default radix in "Tools->Options->Program/Debug->General->Default Radix". For values in the printable ASCII range (i.e. 0x20 – 0x7E) the ASCII value will automatically be shown in the value box.



The memory window allows you to look at values directly. Note that the memory space contains Flash, SRAM, and peripheral registers so not all values are editable. You can look at a specific location in memory by entering the starting address in the “Address” box.

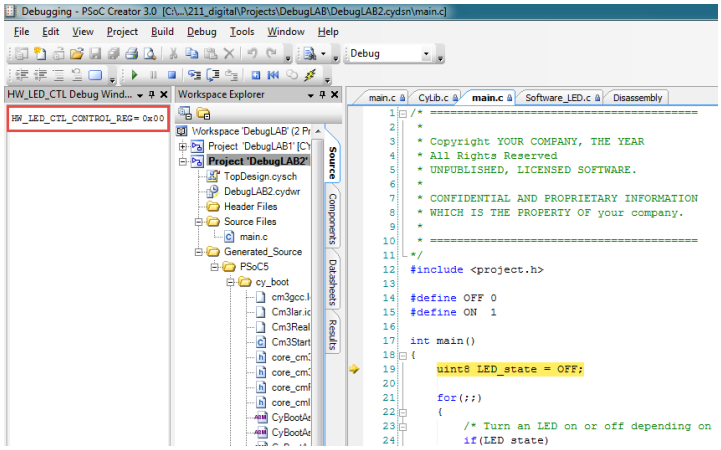


Likewise, you can look at and modify the CPU’s registers and all of the local variables associated with the currently executing function by clicking on the appropriate tab.

In addition to stopping at specific lines of code, you can stop when a particular variable or memory location is accessed (the condition can be read, write, or any access). To do this, right click on the variable name in the code, the variable name in a Watch window, or the memory location in the Memory window. Hover over “Add Watchpoint” and select the type that you want.

Viewing/Modifying Component Registers

The debugger does not halt any of the hardware besides the CPU, but you can still view and in many cases modify registers associated with components. To see component registers, use “Debug->Windows->Components…” and select the component(s) that you want to see. Once you click on “OK” a new window will open (by default it will be on the left side, but it can be moved if desired).



Whenever the CPU is halted, the component register values will be updated with their current values. In some cases you can click on the value, type in a new value, and press Enter to change the register’s value. Note that this will take effect immediately even while the CPU is halted.

References

The debugger has many additional capabilities and features that have not been covered here. The PSoC Creator help system has information on these additional features. Go to “Help->Topics->Using the Debugger” for additional information.

Lab Exercise 12

Description: Use the debugger to examine the firmware.

Objective:

- Learn how to run the debugger, set different types of breakpoints, and examine variables.

Process:

1. Open the workspace “DebugLAB.cywrk” from “Template/Debug_1” in the Class Files folder.

2. Set “DebugLAB1” as the active project and start the debugger. This project has an empty schematic. The code is shown here:

```
#include <project.h>
#include "functions.h"

int main()
{
      char string[12];
      uint32 sequence;

      /* Fill the array named "string" with a message */
      fillArray(string);

      for(;;)
      {
            /* Update the value of "sequence" every 10ms */
            sequence = returnVal();
            sequence = sequence; /* This fixes the warning
                                    about an unused variable */
            CyDelay(10);
      }
}
```

   The project contains an array called “string” and a variable called “sequence” which will be modified by the code. The fillArray and returnVal functions are provided only as object code so you will use the debugger to examine their values.

3. Add a breakpoint in the main loop so that you can monitor the value of the variable “sequence”.

4. Run the code to the breakpoint and record the first 10 values that are returned for “sequence” in the table below. Record the values in decimal.
    - You can either hover over “sequence” when you reach the breakpoint, or add a “Watch” for it.

Pass 1	Pass 2	Pass 3	Pass 4	Pass 5	Pass 6	Pass 7	Pass 8	Pass 9	Pass 10

Can you name or describe the sequence that is being generated?

5. Add a conditional breakpoint to find the first value (in decimal) of “sequence” that is greater than 10,000. Record its value here:

6. Identify the value that is stored in the array “string”. Write down its ASCII equivalent here:

7. Use the address provided for the array to find it in the memory window. Notice that the ASCII text is shown along the right side of the memory window so that the text is easier to read.

   Note: Instead of entering the address, you can type the name of the array or variable that you want to find and the debugger will take you to its location in memory.

Lab Exercise 12.2

Description: Use the debugger to modify values.

Objective:

- Learn how to use the debugger to change both firmware and hardware (register) values.

Process:

1. Open the workspace “DebugLAB.cywrk”.

2. Set “DebugLAB2” as the active project and start the debugger. The schematic and code for the project are shown here:


```
#include <project.h>

#define OFF 0
#define ON  1

int main()
{
    volatile uint8 LED_state = OFF;

    for(;;)
    {
        /* Turn an LED on or off depending on LED_state */
        if(LED_state)
        {
            Software_LED_Write(ON);
        }
        else
        {
            Software_LED_Write(OFF);
        }
    }
}
```

   The schematic has one output pin which controls an LED. Note that the default behavior is that the LED will never turn on. We will use the debugger to turn the LED on/off.

3. Add breakpoints on the two Software_LED_Write function calls.

4. Run the code until it stops at a breakpoint. Is it the breakpoint that you expected?

5. Change the value of “LED_state” to 0x01 and re-start code execution (make sure you delete the entire contents of LED_state including \000). Does the code now stop at the other breakpoint? Note that you have to re-start code execution twice before the LED turns on. Why?