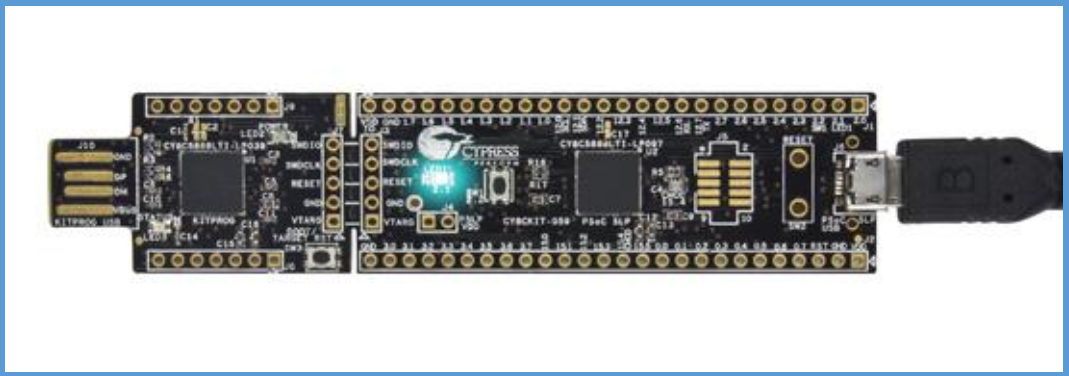
PSoC in the Advanced Laboratory v2.



Mark F. Masters, Ph.D.

IPFW Department of Physics

Introduction

This is a manual for using the Cypress Semiconductor PSoC 5 prototyping kit (CY8CKIT-059) in the advanced physics laboratory. This device is used in our instrumentation class and laboratory. Students use it on their own in the advanced laboratory. This is a fairly complex device and it is very capable. We have used it to create lock-in amplifiers. We have used it as a data acquisition tool. We have used it as controller for ovens. We have used it in coincidence counters.

This document will have several parts:

- An overview of how the PSoC functions
- An introduction to programming in c (or a brief refresher)
- Project 1: OpAmp
- Project 2: Programmable Gain Amplifer (PGA) & Trans-Impedance Amplifier (TIA)
- Project 3: Low-cost I^2C Character LCD
- Project 4: Digital to analog converters
- Project 5: Analog to digital converter
- Project 6: PWM's
- Project 7: Measuring Planck's Constant
- Project 8: Stock coincidence counter unit for single photons & Using an LED as a low cost and inefficient Single Photon Avalanche Detector. (Also measuring time between pulses).

PSoC Overview

PSoC stands for Programmable System on a Chip. It is a mixed signal microcontroller made by Cypress Semiconductor. Mixed signal means that it combines both digital and analog components. This makes the system especially useful to an experimentalist because we can do everything on a single chip.

While there are many different types of PSoC's: PSoC 1, 3, 4 and 5 – we are most interested in the 4 and 5. These are 32 bit ARM core processors. In particular, we will focus on the most powerful, the PSoC 5. This is made easier by the PSoC prototyping kit CY8CKIT-059 which costs $10 (http://www.mouser.com/ProductDetail/Cypress-Semiconductor/CY8CKIT-059/?qs=%2fha2pyFadujougpOLomFMYcjOwvAvII2mjozF0chtEmSM89%252b%252bi0EFg%3d%3d). This kit allows access to a 32 bit processor with plenty of onboard storage and many different analog and digital components in a convenient package that can be attached to a breadboard. The microcontroller used in the kit is the **CY8C5888LTI-LP097.**

What is interesting about the PSoC microcontrollers is that all of the peripherals can operate independently or mostly independent of the processor. In many instances, the processor can be completely shut down to save energy if necessary.

Key features of this processor are:

- A 32 bit ARM Cortex M3 processor,
- Four 8-bit digital to analog converters
- 4 op-amps
- 4 analog comparators
- 4 Digital to analog Converter

- 3 Analog to digital converters (A 700 Sample per second 20 bit analog to digital converter, two 1 MSample/second 12 bit analog to digital converters)
- Analog and digital multiplexers
- 24 universal digital blocks (UDB) - The UDB's consist of a matrix of uncommitted Programmable Logic Devices (PLD's) and a digital interconnect. The PLD may be configured to be any number of digital devices from simple logic elements to more complex digital devices such as counters, pulse width modulators and communication.
- Pulse Width Modulators, Counters and timers
- Basic logic (AND, OR, XOR, Flip Flops, etc.)
- A 24-bit digital filter system
- Built in communication blocks (I2C, USB, and others)
- Reconfigurable IO – Can assign almost any input and output to any pin.
- Ability to change the digital IO voltage (SIO and VDDIO)
- Built in ability for capacitive buttons
- Ability to program it like an FPGA using Verilog

All of this information may be found in the Datasheets.

One of the great advantages of the PSoC is that you can combine analog and digital circuitry on a single chip combined with a relatively powerful processor which allows you to collect data and then do some processing on the data. Figure 1 shows a diagram of the PSoC 5. Components are wired and routed on the IC. There is no external routing (unless using the op-amps and you are programming the gain). Finally, you can create your own components (Challenging).

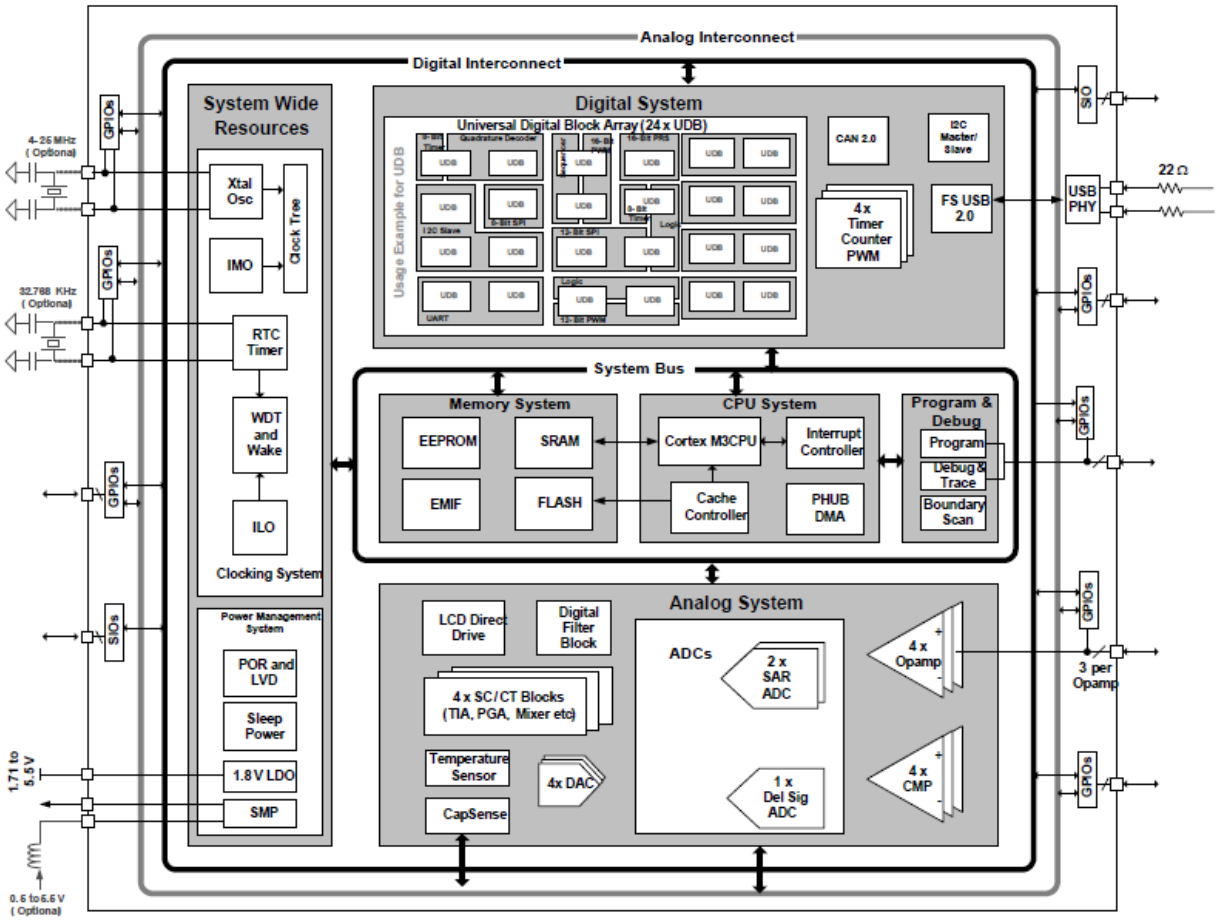


Figure 1

The CY8CKIT-059 prototyping kit has simplified the process of using the PSoC 5. It consists of a small circuit board with breakout pins to allow you to access most of the GPIO ports of the PSoC5 and its capabilities. It also comes with an integrated USB programmer which makes it very convenient to use. The programmer can also be used to communicate serially with the computer through a virtual serial port. The board also has a micro USB port that can be used for communication. Figure 2 shows an image of the CY8CKIT-059 prototyping kit. On the left side is the programmer which plugs directly into a USB port (**or preferably a USB extension cable**). This is also used for serial communications. Once your device is completely programmed, the programmer may be removed.

There are some drawbacks to using a prototyping kit. First, certain IO serve multiple purposes. In particular, pay close attention to the following IO pins when they are assigned:

- **P15.4 has a 2200pF cap to ground –** Cmod – it is used for the capacitive sensing.
- **P0.2, P0.3, P0.4 and P3.2** all have 1uF caps to ground for the SAR bypass capacitors

Other specific pins that might be of importance are the SIO pins (P12.0, P12.1, P12.2, P12.3, P12.4, P12.5, P12.6, and P12.7) and the op-amp input output ports (P0.1- opamp0 out, P0.2 – opamp 0 +, P0.3

– opamp 0- Notice that these coincide with caps that have a 1uF cap. Therefore that must be taken into account. P3.4 opamp 1+, P3.5 opamp 1-, P3.6 opamp 1 out. P0.4 opamp 2+, P0.5 opamp 2-, P0.0 opamp2 out. P3.2 opamp 3-, P3.3 opamp 3+, P3.7 opamp 3 out.).

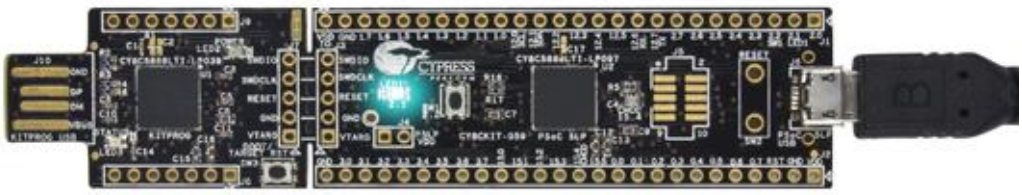


Figure 2

We have found it is really important to use a USB extension cable for the programming. We have found that most computers have sloppy USB ports and it is too easy to get frustrated with bad connections. Therefore we recommend the simple USB extension cable: http://www.monoprice.com/Product?c_id=103&cp_id=10303&cs_id=1030304&p_id=5431&seq=1&format=2 ($0.93).

Ports	Issue
15.4	CMod cap for capsense to ground (200 nF)
0.2	1 μF cap to ground for A/D bypass cap
0.3	1 μF cap to ground for A/D bypass cap
0.4	1 μF cap to ground for A/D bypass cap
3.2	1 μF cap to ground for A/D bypass cap

Ports	Special Capabilities
12.0	SIO – SIO ports allow you to change the
12.1	Threshold for digital transitions.
12.2	You can change the output voltage.
12.3	You can change the input voltage level.
12.4	This allows you to use low input voltage
12.5	Signals.
12.6	
12.7	

Getting Started

1. Purchase a CY8CKIT-059 http://www.cypress.com/documentation/development-kitsboards/cy8ckit-059-psoc-5lp-prototyping-kit - $10
2. Download PSoC Creator 4.1 : http://www.cypress.com/products/psoc-creator - requires creation of an account. Software is free. Windows only. Virtual Box does work.
3. Download the kit software: http://www.cypress.com/documentation/development-kitsboards/cy8ckit-059-psoc-5lp-prototyping-kit
4. Download the kit guide and work through some of the examples.

Further information and resources

Further information on the PSoC can be found at the cypress website. In particular, the following resources are very helpful:

The data sheet: http://www.cypress.com/documentation/datasheets/psoc-5lp-cy8c58lp-family-datasheet-programmable-system-chip-psoc

The Technical Reference manual: http://www.cypress.com/documentation/technical-reference-manuals/psoc-5lp-architecture-trm

The application notes (search) at cypress.com

The video resources (search) at cypress.com

The community: http://www.cypress.com/cdc

C programming guide: https://en.wikibooks.org/wiki/C_Programming

In the PSoC creator program (the IDE for the PSoC) there are many examples that can be downloaded and are very helpful.

Programming

Programming a PSoC is a combination of drop and drag components and wiring and programming in c-code. While this is incomplete, it gives some of the fundamentals to programming in c.

Fundamentals of c

c is a strongly typed programming language. That means that you have to declare variables in order to use them and the type of variable that you are using must also be declared. Types of variables are: **int**, **int8**, **int16**, **int32**, **uint8**, **uint16**, **uint32**, **long**, **char**, **float**, and **double**. Additionally, c is case sensitive. Finally, you must always remember the semicolon at the end of each sets of commands ';'. Sets of commands are placed between brackets.

The fundamental building blocks of any program are variables, statements, conditionals, and loops.

Variables

There is the scope of variables which depends upon where they are declared. If they are declared within a function, then they are local to that function (only usable in that function). If they are declared outside of the function, then they are global variables and may be used by all functions.

There is the **main** function. This is, as the name indicates, the main or primary function. Your program always starts in this function. You can declare variables inside of this function. These will be local to the main function.

Variable type	
uint8	8 bit unsigned integer
uint16	16 bit unsigned integer
uint32	32 bit unsigned integer
int8	8 bit signed integer
int16	16-bit signed integer
int32	32-bit signed integer
int	32 bit signed integer
long	32 bit signed integer
char	8-bit character
float	32-bit floating point
double	64-bit floating point

Conditionals

Conditional statements are statements that test for some condition in c they take the form of:

```
if (condition)
{
        Do this;
        And this;
}
else if (condition)
{
```

```
}
else
{

}
```

Loops

There are two types of loops in c: 'for' loops and 'while' loops. The form for the 'for' is given by:

```
For ( initial condition; end condition; increment)
{
        Do this;
}
```

Example:

```
main()
{
        int I;
        for (i=0;i<10;i++)
        {
                print i;
        }
}
```

If you just leave everything in the for loop arguments blank *for(;;)*, the loop will run forever.

The second type of loop is the while loop. The while loop is a bit more general in that it only checks a condition.

```
while (condition)
{
        Do this;
}
```

The condition can be any logical condition.

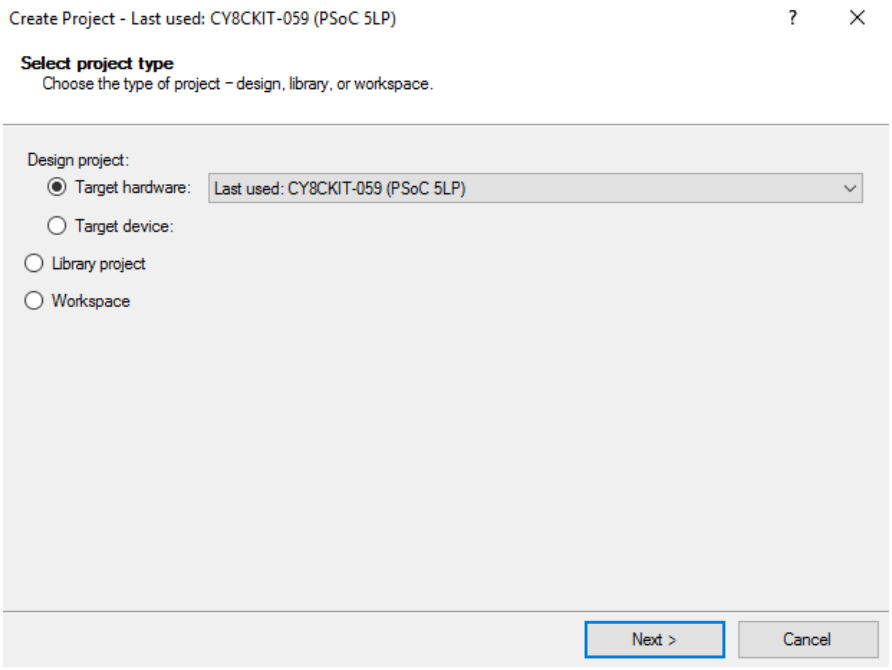
'c' programs have a specific structure. That structure is typically something like:

```
#include "something.h"
#include "somethingelse.h"

#define SOMETHING 10

Int function (int argument)
{
        int variable;
        char variable;
        instructions;
        return something;
}

Int main()
{
        Declartions

        for (;;)
        {
                X=function(y);
        }
}
```

Now that we have some basic programming, we can go through and start to design circuits!

Project 1: Operational Amplifier

Start PSoC Creator. Open a new PSoC5LP project. Choose either a target device (CY8C5888LTI-LP097) or alternatively, if you installed the CY8CKit-059 software you may choose that as a hardware option.



Name both the workspace and the project.

A single workspace can have multiple projects. However, one must be careful in this scenario to be certain to build and program the correct project as well as editing the correct project. PSoC Creator gives clues about whether you are working on the active project by shading the tabs of inactive projects. See Figure 3.

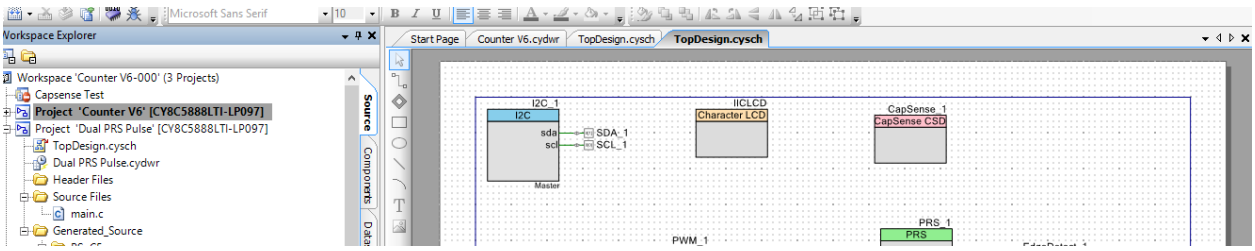


Figure 3 – See how the tab TopDesign.cysh is shaded. This is because it is NOT from the active project. The bolded project is active.

To make a project active in a multiple project workspace, you right click the project and choose "Set as Active Project".

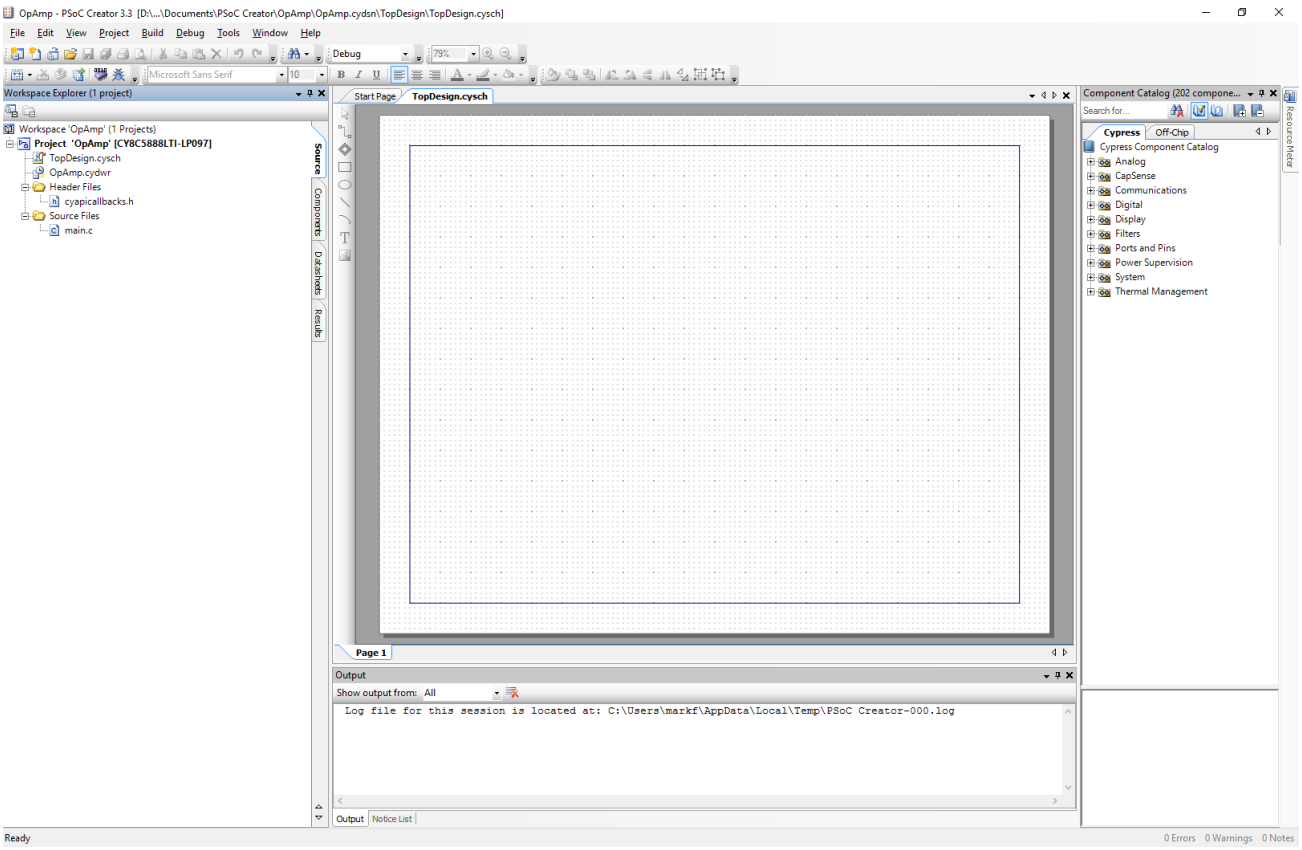


Figure 4 – Workspace. On the left is the project explorer. On the right is the component catalog and for the most part we simply drag components from here to the schematic in the center of the window.

We are going to start with an Op-Amp. On the left, choose the **Analog** folder, then open the **amplifiers** folder, and then drag an op-amp onto the workspace. Also drag on two analog pins from the **Ports and Pins** folder on the left (see Figure 5). Notice that the wires on the analog pins are red. We can rename any item by double clicking on that item.

Double click on the op-amp. You will see a menu come up (see Figure 6). This has many settings. For example, you could turn the op-amp into a **follower** by choosing that from the drop-down mode menu. It is also important to select the power setting. For better qualities, you want to use high power. You also will see a button for a datasheet. When in doubt refer to the data sheet.

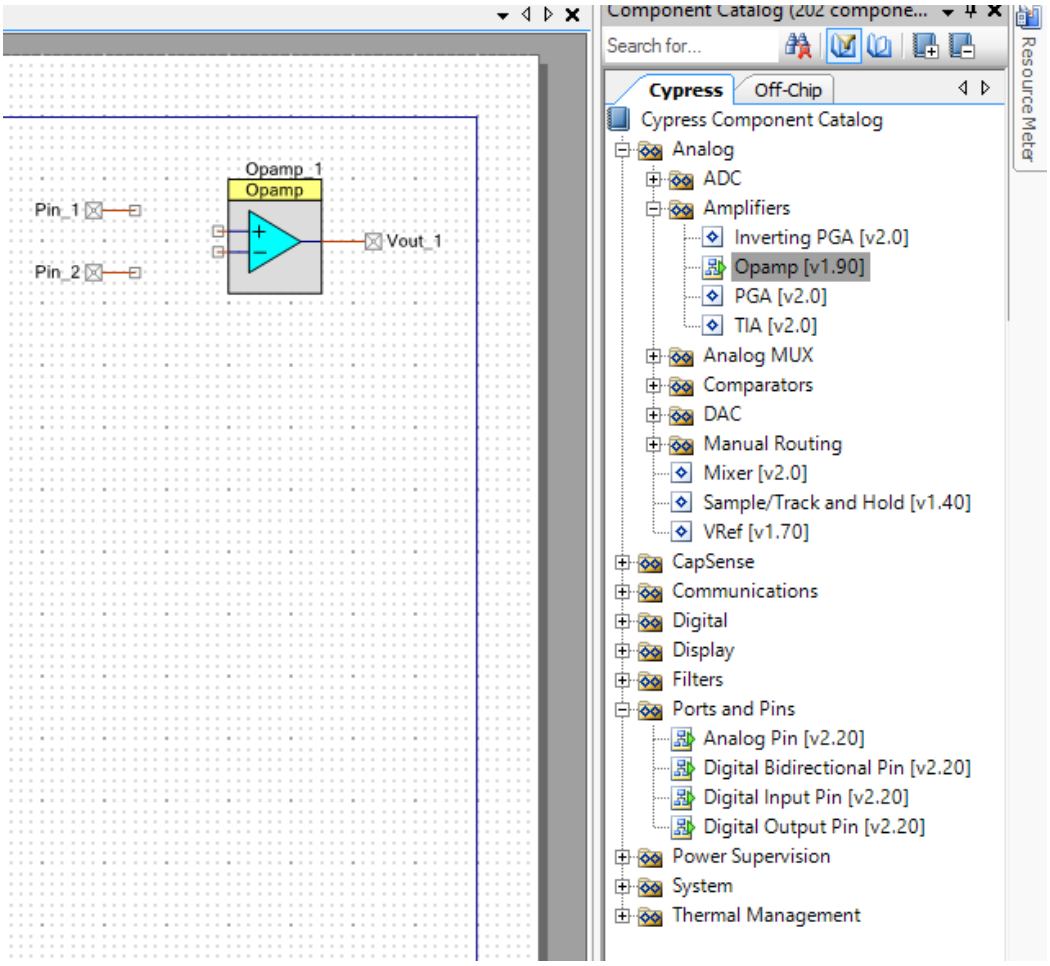
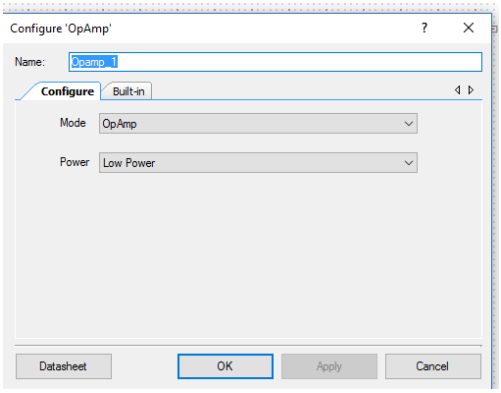


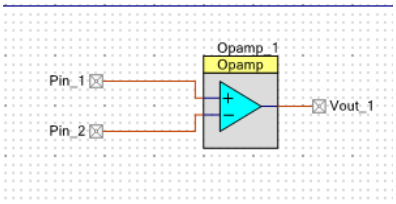
Figure 5 – Op-amp and analog pins on the work space.



If you look on the left side of the workspace, you will see a wiring tool. If you click on this, you can drag wires to make connections. Also, if you shift click, the wiring tool becomes "sticky" and you can make many connections without clicking on the wire tool each time. Finally, you can short cut and press the 'w' key.

Figure 6 – the menu when you click on an op-amp.

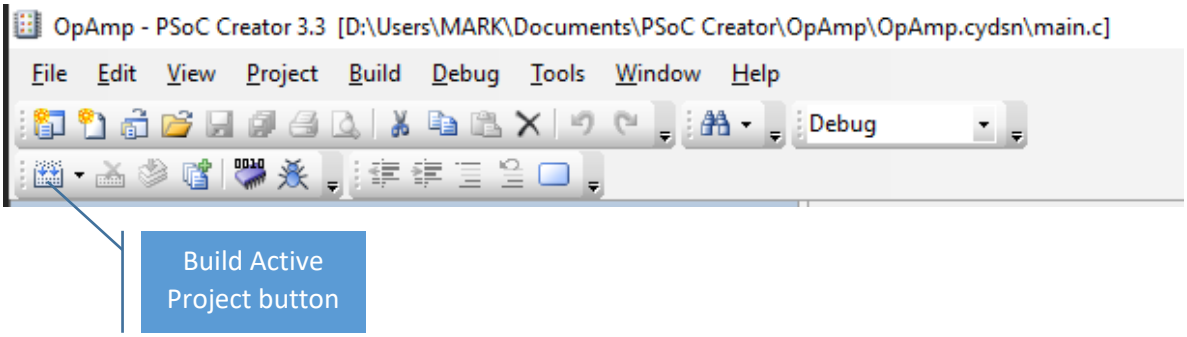
Wire the two pins to the opamp (see Figure 6). We can rename any component including pins by double clicking.



Now we can externally add resistors to program a gain just like any old operational amplifier. Before we can test this though, we have to create the firmware.

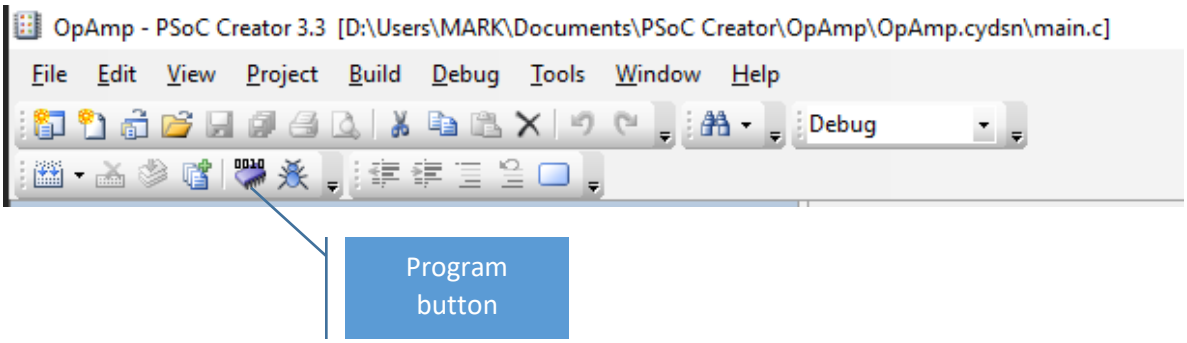
Prior to writing any code we should build the project. Go to the build menu at the top of the screen and select build project. We do this at this point just so that we can populate the auto-completion for coding. Building a project goes through and sets up the hardware. It also assigns pins. We will look at pin assignments before we test the program.

Once the build has completed, double click on main.c in the project explorer to open it for editing.



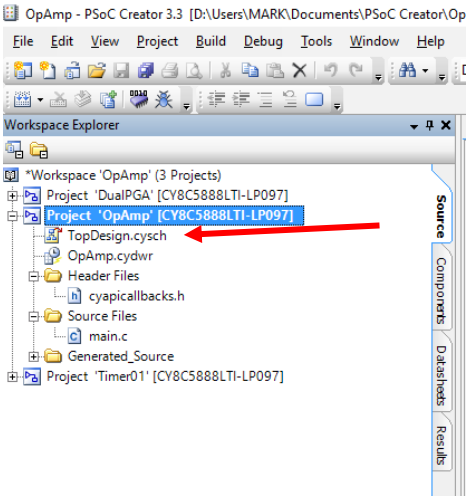
Start Page | TopDesign.cysch | main.c

```
/* ========================================
 *
 * Copyright YOUR COMPANY, THE YEAR
 * All Rights Reserved
 * UNPUBLISHED, LICENSED SOFTWARE.
 *
 * CONFIDENTIAL AND PROPRIETARY INFORMATION
 * WHICH IS THE PROPERTY OF your company.
 *
 * ========================================
*/
#include <project.h>

int main()
{
    CyGlobalIntEnable; /* Enable global interrupts. */

    /* Place your initialization/startup code here (e.g. MyInst_Start()) */
    Opamp_1_Start();
    for(;;)
    {
        /* Place your application code here. */
    }
}

/* [] END OF FILE */
```



After the *CyGlobalIntEnable* line, type *Opamp_1_Start()*; Having done this you are ready to build and program your device. Just plug the prototype kit into the USB port and choose the program button.

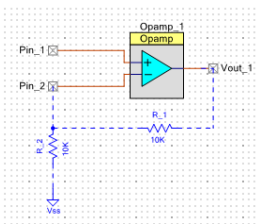
Now we can build our circuit and test how the op-amp behaves. However, what pins should we use?



Changing pin assignments: If you open the file which is named <your project name>.cydwr in the workspace explorer, you can see where the software assigned different pins.

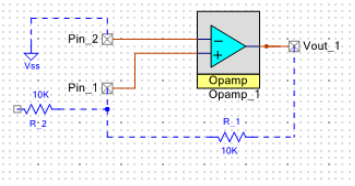
You see an image of the IC with the pin assignments. On the right, there are your specific pin assignments. These may be reassigned to other physical pins for your convenience. If you reassign pins you MUST rebuild.

This file is very important. If you look at the tabs below the IC image, you see the following tabs: Pins, Analog, Clocks, Interrupts, DMA, System, Directives, Flash Security and EEProm. The tabs that are of particular importance to us will be Pins, Clocks, and System. In the Pins tab you assign pins. In the Clocks, you set up all of the system clocks. In the System Tab, you can assign whether there are debugging pins, and how much space is reserved for the CPU, etc.



Back to the Op-Amp. We can build any type of Op-Amp circuit using these Op-Amps. Keeping things simply, build a non-inverting amplifier. Test how it responds to input signals that are centered around ground.

We can also build a non-inverting amplifer:

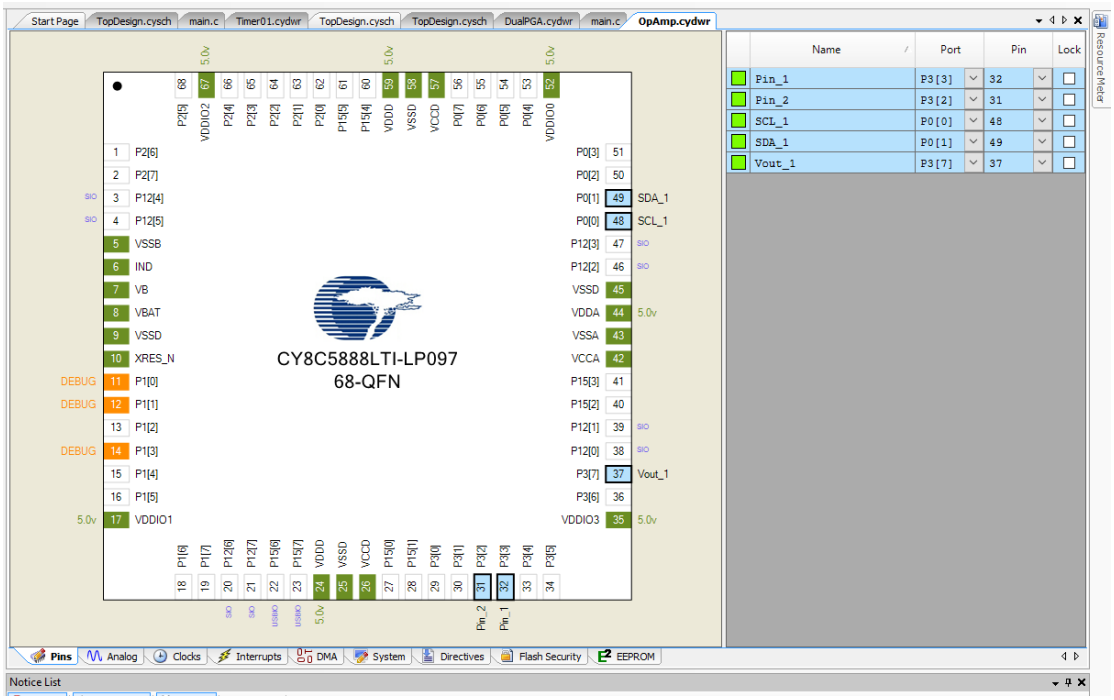


It is important to remember that this is a unipolar op-amp and it cannot produce a negative (below ground) output. Keep the input voltage below 5 V.

Thus far, you have built a simple op-amp. But remember this op-amp is integrated onto a microcontroller.

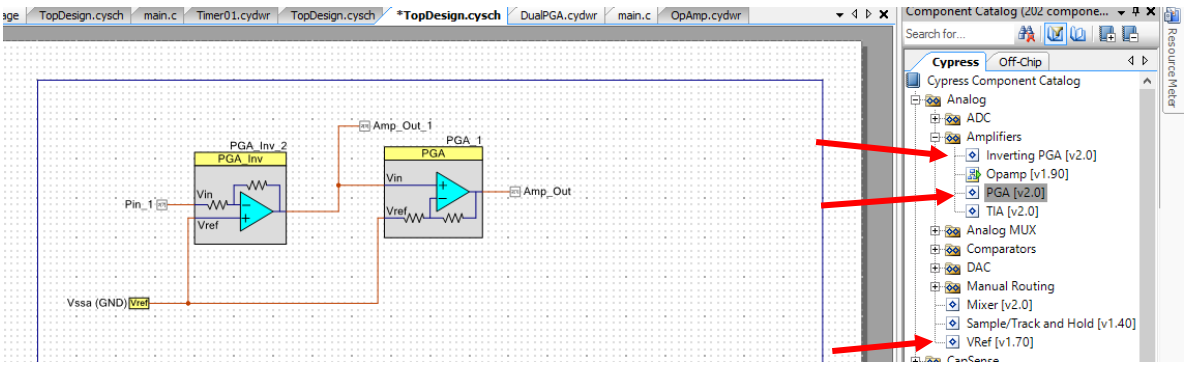
Using a function generator and an oscilloscope, look at the output of the opamp and examine its characteristics.

One thing that you will find that is interesting is that if you build an inverting amplifier referenced to ground, you can use inputs that are negative relative to ground, as long as your output is between 0 volts and the Power supply voltage (usually 5V).

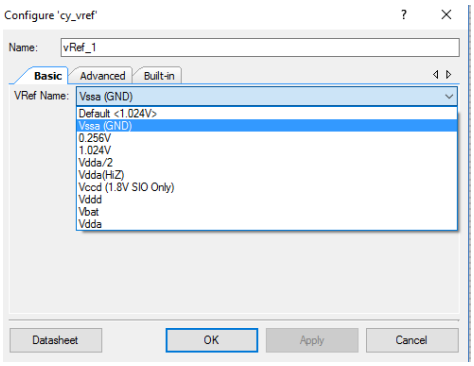


Project 2: PGA and TIA

Next try out the PGA (Programmable Gain Amplifier). It is really much the same as the previous example, but you can control the gain via software. There is also a TIA – Transimpedance amplifier which is very useful when working with photodiodes. These op-amps do not have as good specifications as the other op-amps. That is because they are created with switched capacitors – which are interesting. Regardless, they have their uses and it is very helpful to have an op-amp that has gain that can be controlled by software on the fly.

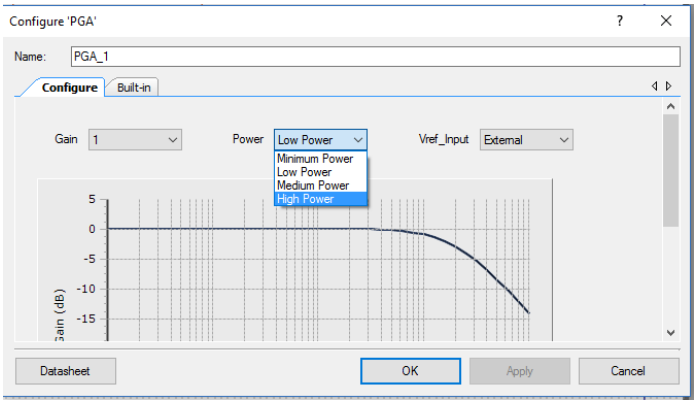


Drag an Inverting PGA, PGA and VRef onto the schematic. Then drag several analog pins onto the schematic. Notice that we have attached VRef to the reference junction of two PGA's. We can, of course, route the Vref junction to external pins. However, we can also route it internally to a reference point.

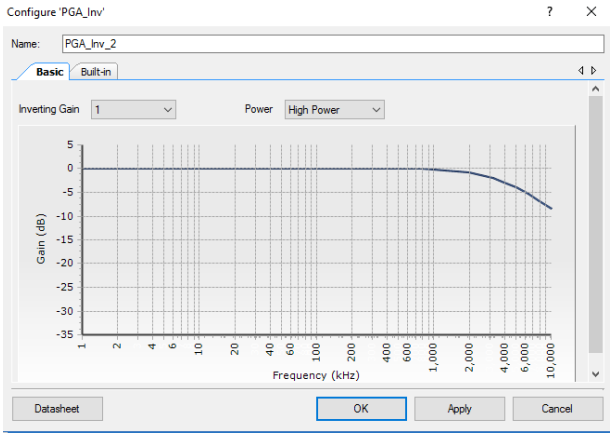


Double clicking on Vref brings up a number of options. There is the 1.024 V reference voltage, analog ground (VSSA), and other possibilities. We could also use a digital to analog converter (VDAC) as an adjustable reference. More on VDAC's later.

Double clicking on the PGA brings up the menu shown below. I recommend that you always choose High Power as it has a greater bandwidth. Likewise, we could have chosen a Vref_input of the Vssa rather than use the VRef component. You can also see the default Gain setting.



For the inverting PGA, the options are a little different. There is still the gain setting and the Power setting, however, there is no longer a reference junction setting. Also, notice the Gain v. frequency graph and pay attention to the units of the ordinate (kHz).



Setting up the PGA's is not difficult. Just as before, build the project. Then we can create the firmware. Double click on main.c. Then we can **start** each of the PGA's. We can also set the gains of each of the PGA's as shown.

```
#include <project.h>

int main()
{
    CyGlobalIntEnable; /* Enable global interrupts. */

    PGA_Inv_2_Start();
    PGA_1_Start();

    PGA_1_SetGain(PGA_1_GAIN_04); //set the PGA's gain to 4
    PGA_Inv_2_SetGain(PGA_Inv_2_GAIN_03); // set the inverting pga gain to 3.

    for(;;)
    {
        /* Place your application code here. */
    }
}
```

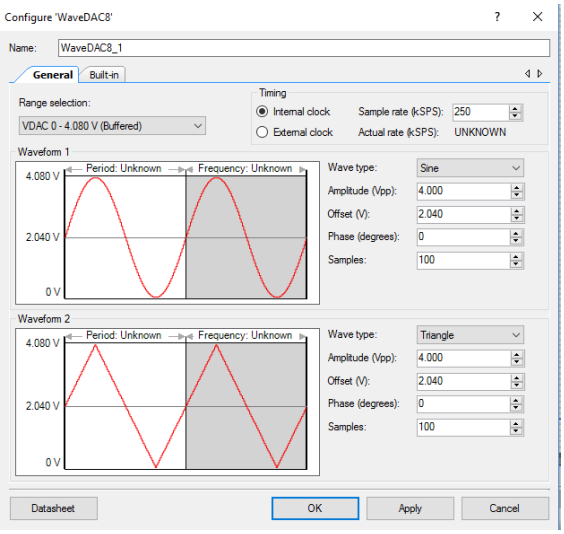
You have to be careful about the gains and the PGA's in use. First, the current circuit will provide gains for **negative** (relative to Vssa) input signals, but will not work for positive signals. Why?

What would be the maximum input signal we could use with this circuit?

We can change the circuit by changing the reference voltage. So if we set it to Vdd/2 (~2.5V), then we can have AC signals around that voltage! But our output will also be referenced to that voltage.

Project 3: Digital to Analog Converters (DAC)

There are four DAC's on the PSoC. They are a very useful device because they allow us to take digital values and create analog signals out. There are two basic DAC's on the PSoC: iDAC and VDAC. The iDAC is a current digital to analog converter. It produces a controllable current output. The second is a voltage digital to analog converter. This is really the same as the iDAC with an internal resistor. These two have some different properties such as maximum sample rate. Finally, there is the WaveDAC which allows simple output of various waveforms. The two types of DAC are 8-bit meaning you can have 256 values or outputs. There are several ways to create higher resolution such as the included 10-bit Dithered DAC. You can also combine two DAC together to create higher resolution (work with the iDAC!).

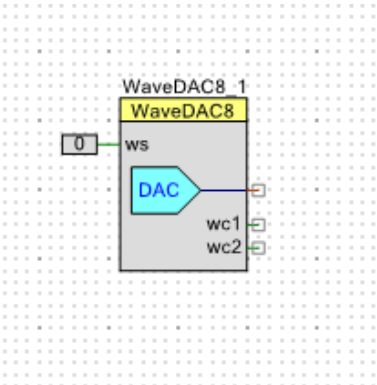


WaveDAC

Our first exploration is going to be the *waveform DAC*. Drag the *waveform DAC* component onto the schematic. Double click and examine the options. First, on the left you can choose between VDAC 0-1.020 V, VDAC 0-1.020 V (buffered), VDAC 0-4.080 V, VDAC 0-4.080 V (buffered), and then similar settings for IDAC (with current settings of course). Buffered means that there is a buffer on the output which can be helpful if you are drawing current from the VDAC.

We can also set the sample rate. Setting the number of samples per period determines the frequency (frequency = Sample rate/Samples). We can set an offset voltage and a phase.

The wave type allows one to select the waveform type. It is interesting that you can actually draw your own waveform by choosing arbitrary.



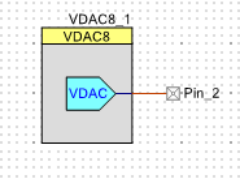
Wiring the WaveDAC is straight forward. You connect an analog pin to the output of the DAC (notice it is a red wire). You also must connect the ***ws*** port (wave select) on the waveform DAC. This is a logic control – low (or zero) chooses waveform 0. High chooses waveform 1. You could connect this to a PWM and switch between the two waveforms on the fly. Or you can set it to one or the other. We will do the latter. From the components in the digital logic portion of the catalog, choose either "logic high" or "logic low". Logic low selects the first waveform and logic high selects the second. Once you have done this (added the logic control), attach an analog pin to the output, and build the project. Then start the waveDAC in main.c, rebuild and program your device and observe the output on an oscilloscope.

```
#include <project.h>
int main()
{
    CyGlobalIntEnable; /* Enable global interrupts. */
```

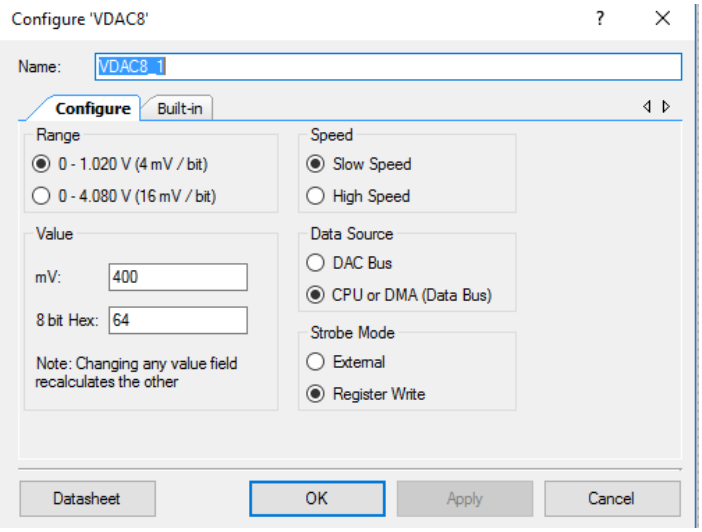
```
    WaveDAC8_1_Start(); //Start the WaveDAC.
    for(;;)
    {
        /* Place your application code here. */
    }
}
```

VDAC

The Voltage digital to analog converter or VDAC is a very useful tool. It can be used to set adjustable reference voltages (for example in a comparator or with an SIO pin).



Drag a VDAC on to the schematic and wire in an analog output. The parameters of the VDAC are found by double clicking on the component. You see a range setting, a speed setting and where the data is coming from. For the most part, we do not care about the where the Data comes from.



To get the VDAC to work, follow our usual procedure of build and then enter the code to start the VDAC. If you want to set the output value of the VDAC, then you use the command VDAC_SetValue(8bit #). As a challenge, create a for loop and have the VDAC change its output as a function of time. You may want to use a delay, CyDelay(ms), is the command for a millisecond delay.

```
#include <project.h>

int main()
{
    uint8 i;
    CyGlobalIntEnable; /* Enable global interrupts. */

    WaveDAC8_1_Start(); //Start the WaveDAC.
    VDAC8_1_Start(); // Start the VDAC

    for(;;)
    {
        CyDelay(10); // delay for 10 ms
        VDAC8_1_SetValue(i);  // write the value to the DAC.
        i++; // increment i.  Since i is an 8 bit variable, then once we get
over 255, the value will become zero.

    }
}
```

Project 4: Serial LCD

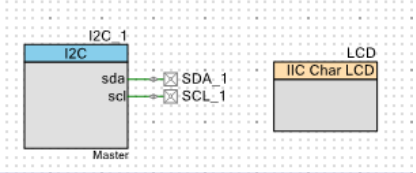
We are about to move on to using analog to digital converters. We would like to get some sort of readout of the values. Also, it is really handy to have an LCD just for debugging. We use a I^2C based LCD which is much simpler to wire up to the PSoC (only takes two wires!) than the way way too many in the usual parallel interface LCD. I like the low cost I2C, 4 line LCD screens available on ebay.
http://www.ebay.com/itm/Blue-Serial-IIC-I2C-TWI-2004-204-20X4-Character-LCD-Module-Display-For-Arduino-/181299099752?pt=LH_DefaultDomain_0&hash=item2a3644b868

I^2C is a great serial communication interface. You can communicate with multiple devices on the same serial line by choosing different addresses for those devices. You could have 3 LCD's and choose to write on each one separately – while keeping the same data and clock line. But, the I^2C interface requires pull up resistors (~5k to high). Now some LCD's have these built in. but others do not. Fortunately, the ports on the PSoC can be reconfigured to have active pull up simply with a software switch. Cool huh?

The physical wiring is easier, but you have to use a special library not available in the package. In particular, I modified a Cypress library to work with these LCD's. It is called MFM_LIB and is available on request by emailing masters@ipfw.edu.

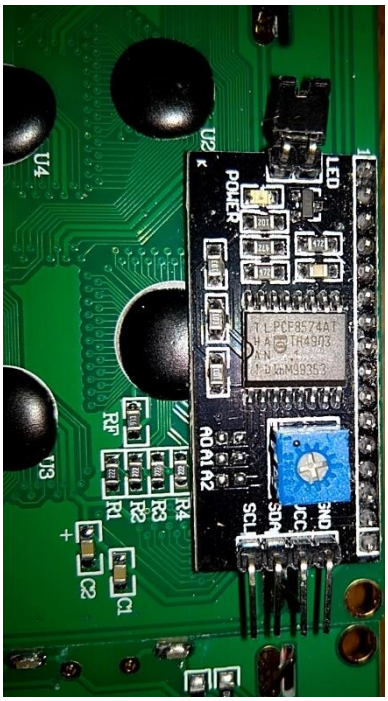
To include this library you must get a copy of the library, then go to the Project menu at the top of the screen. Select dependencies. In User dependencies, choose the folder and find the MFM_LIB. Then click ok. You will see a new tab appear in the component catalog called "Frustrated". If you go to this tab you can drag on the IIC 4bit character LCD device. Given the awkward name, IIC_4bit_Char_LCD of this component, I would recommend renaming it to something like LCD.

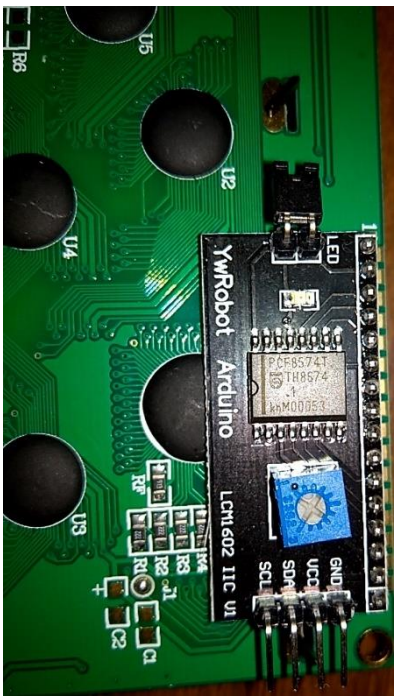
You also need to drag on the I2C_master communication package. Build the project, then identify the pins SDA and SCL for the I2C controller (remember to avoid the pins which have capacitors to ground!!!). Wire up the appropriate pins for the LCD (check the CYWDR file!) and you are ready to start.

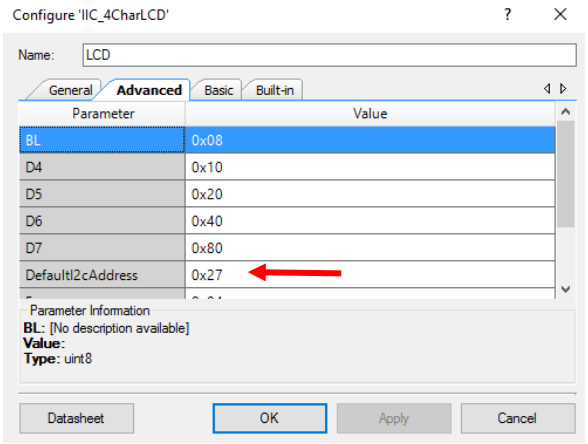


As usual, you build first. Then, in main() you need to start both the LCD device and the I2C master control. **Always start the I2C first!!!**

For these low cost serial LCD's there are two controllers. One is labeled PCF8754T and the other is labeled PCF8754AT. The two pictures shown at right are of two different LCD's. The one on the left has the PCF 8754AT controller and on the right has the PCF8754T controller. You have to look very closely at the controller IC's to see the part number. This is important because the two have different I2C addresses! The PCF8754T has an address of hexadecimal 0x27 while the PCF8754AT has an address of hexadecimal 0x3F! The LCD component defaults to the 0x27 value, so if you have an LCD with the PCF8754AT controller, then you have to change this value.







To change the value, double click on the LCD component. Choose the Advanced tab. You will find a variety of settings, the one you want to change is the DefaultI2cAddress to 0x3F for the PCF8754 AT LCD's.

Creating the Firmware:

The firm ware simply requires starting the I2C controller and the LCD itself. Then there are a variety of LCD commands.

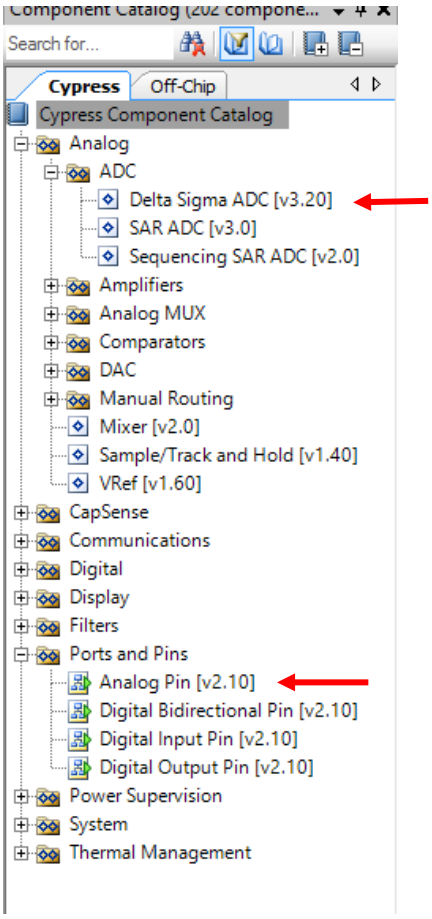
```
#include <project.h>

int main()
{
    uint8 i;
    CyGlobalIntEnable; /* Enable global interrupts. */
    I2C_1_Start();
    LCD_Start();
```

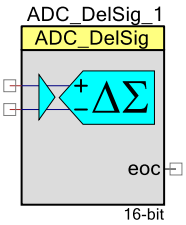
```
    LCD_PrintString ("Hello");

    for(;;)
    {
    }
}
```

Other useful LCD commands are things like LCD_Position(row, column) and LCD_PrintNumber(variable). It is often easiest to clear a screen just by printing x number of spaces.

Project 5: Analog to Digital Conversion

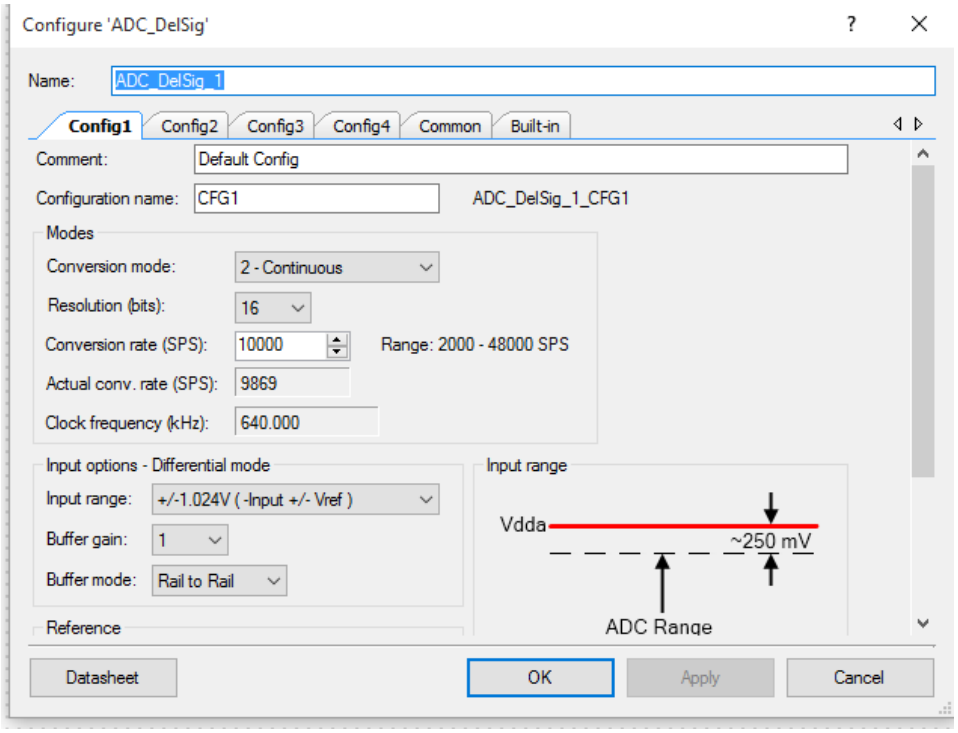
The analog to digital conversion is quite straight forward. You start in the same way as before, creating a blank project. Moving to the right you can open the analog folder and drag in a Delta Sigma ADC – which is up to 20bits.



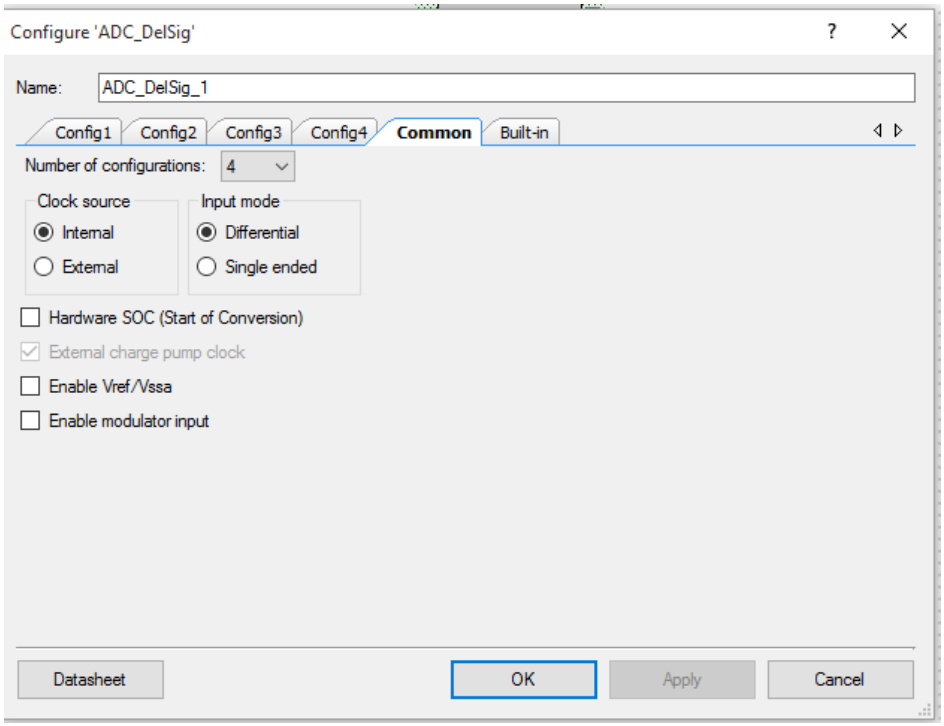


The ADC is by default differential. To change the resolution and make the ADC single ended, double click on the ADC.

On the "config" tabs you can set the resolution, conversion rate, conversion range and conversion mode. Conversion rate depends upon the resolution you choose and the buffer. The higher the resolution, the lower the conversion rate. Likewise, the higher the buffer gain, the lower the conversion rate. The delta sigma A/D can go up to 20-bit resolution. Something else that is worth noting is that the input range is also variable. You can have a conversion range from $\pm 6\ V \rightarrow \pm 64mV$.



Also worth noting are the multiple configurations. You can switch on the fly between different conversions so that one measurement may be made using one configuration and another made with a second configuration. While there is only one Delta Sigma A/D converter, there are also analog multiplexers. We will make use of this in the Planck's constant experiment.



In the common tab, you can set the input mode to single ended (amongst other features).

A simple voltmeter.

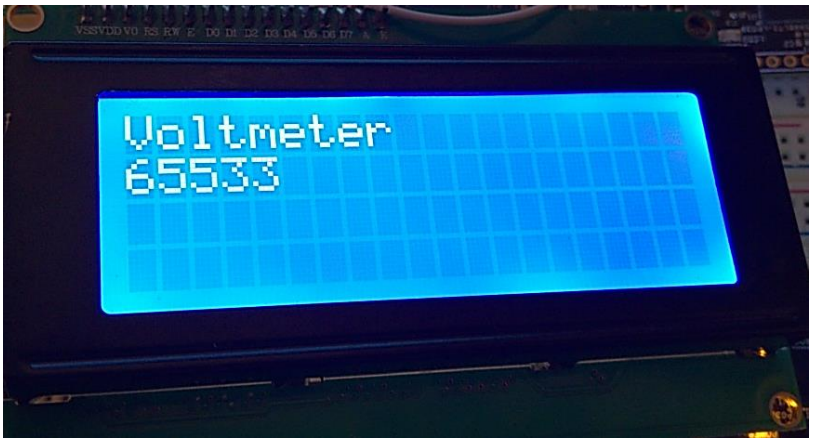
Set the Drag an analog pin onto the schematic and attach it to the ADC input. Finally, you need some type of output, we will use the same I2C LCD screen for a readout.

To develop the firmware it is useful to build the project first to load the necessary libraries. Then open main.c and add the following code:

```
int main()
{
    CyGlobalIntEnable; /* Enable global interrupts. */
    int Result;
    /* Place your initialization/startup code here (e.g. MyInst_Start()) */


    I2C_1_Start();
    LCD_Start();
    ADC_DelSig_1_Start();
    ADC_DelSig_1_StartConvert(); //start conversions
```

```
    for(;;)
    {
        ADC_DelSig_1_IsEndConversion(ADC_DelSig_1_WAIT_FOR_RESULT);//wait
until there is a conversion
        Result=ADC_DelSig_1_GetResult16();
        LCD_Position(1,0);
        LCD_PrintString("          "); //Clear a line
        LCD_Position(1,0);
        LCD_PrintNumber(Result);
        CyDelay(500);// wait 500ms
    }
}
```

So when you build this project and program the device, your output might look something like this:



And you are like "What???" But then it's ok because you realize that you had floating inputs so you then ground both of them. Guess what – it remains the same. What is going on?

So, first it is important to realize that what the A/D is returning is not a voltage but a number of counts. The counts represent the fraction of the range. The maximum number of counts would be indicative of maximum of the assigned range. We were using a 16-bit A/D so the maximum number is $2^{16} = 65536$. Right? Nope. We have set our A/D converter to bipolar – so the question is how does one represent negative values in a digital system? That is a longer discussion. First – with a 16-bit number you can only represent values from $-(2^{15}) \rightarrow (2^{15} - 1)$. To do this, for the positive half you use the least significant 15 bits. When the 16th bit is selected, then you have a negative number. But negative numbers count down from the maximum – so FFFF (in hexadecimal) represents (-1). FFFE represents (-2) at least if you have 16 bit data. So what we are seeing is a decimal representation of a signed integer and since it is close to the maximum value of 65536 we can see that it is actually reading -3 counts! If only we could display this! One way to do this would be to do the math – (65533 – 65536) – except that this will result in 65533 again because LCD_PrintNumber can only display unsigned integers!

Displaying more complicated numbers

To display more complicated numbers we are going to use the "stdio.h" library.

At the top of your code add in a line *#include "stdio.h"*. Within main() add in a new variable declaration: *char Buffer[20];* This last line adds a buffer to which we can write data and then display the data. To display the number we are going to use the *sprintf* function. The code for printing should look like *sprintf(Buffer, "%.6d", Result);*.

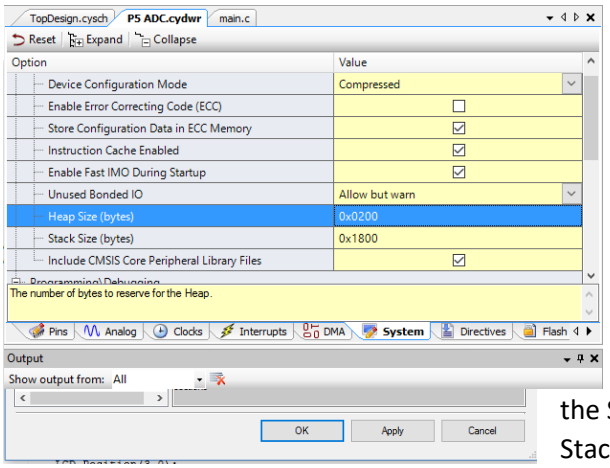
```
#include <project.h>
```

```
#include "stdio.h"

int main()
{
    int Result; //Since I am doing a 16 bit conversion.  Since I am doing
differential measurements, I have to use a signed integer.
    char Buffer[20];
    CyGlobalIntEnable; /* Enable global interrupts. */

    I2C_1_Start();
    LCD_Start();
    LCD_PrintString("Voltmeter");

    ADC_DelSig_1_Start(); // Start the A/D converter
    ADC_DelSig_1_StartConvert(); //Start the A/D conversions

    for(;;)
    {
        ADC_DelSig_1_IsEndConversion(ADC_DelSig_1_WAIT_FOR_RESULT);//wait
until there is a conversion
        Result=ADC_DelSig_1_GetResult16();
        LCD_Position(1,0);
        LCD_PrintString("                "); //Clear a line
        LCD_Position(1,0);
        LCD_PrintNumber(Result);
        LCD_Position(2,0);
        LCD_PrintString("                "); //Clear a line
        LCD_Position(2,0);
        sprintf(Buffer,"%.6d", Result);
        LCD_PrintString(Buffer);
        CyDelay(500);// wait 500ms

    }
}
```

We can use the sprintf statement to create more complex output including floating point numbers. However, to implement floating points requires a few extra steps. We often just present numbers in millivolts or microvolts so we can work with integer math (faster).

Back to our data, what does (-3) mean in this context? Well this is 3 counts. We had the A/D converter convert $\pm 1.024V$ into 2^{16} steps. So each step is equal to $3.125 \times 10^{-5}V$. This means that our A/D converter had an offset of about 90 microvolts.

To test our A/D converter, it would be handy to use a potentiometer. However, we can also use a DAC. To print the voltage out in millivolts, we have to convert the counts into volts. The measured potential difference is determined by $volts = \frac{Counts}{2^{bits}} \times range$ or in this case the values given $\Delta V(mV) = counts \times 2048/65536$. This allows us to do only integer math without resorting to floating points.



To do floating points is a bit more complicated. First, we have to enable the floating point library in the linker. Go to the project menu and choose build settings near the bottom of the menu. Choose the linker item and set the *Use newlib-nano Float Formatting to True*. But, we are not done yet. Next open the <project name>.cydwr file (this is the file with the pin settings) and choose the system tab (previously we were on the pins tab). There are two items that need to be changed: the Heap Size and the Stack Size. Increase the Heap Size to 0x0200 and the Stack Size to 0x1800. These values were found by trial and error. When they were smaller the program would lock up.

The ADC mode was changed to **single sample** from continuous sample and using the ADC_Read16() function which starts the data acquisition for a single sample is useful because I want a sample at a given time in the sequence. Also, a VDAC is put on the schematic and is incrementing within the loop. The code is below.

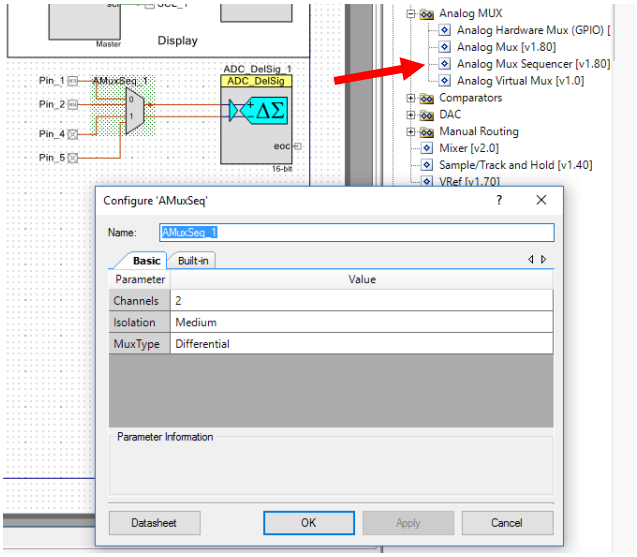
```
#include <project.h>
#include "stdio.h"

int main()
{
    int Result; //Since I am doing a 16 bit conversion.  Since I am doing
differential measurements, I have to use a signed integer.
    char Buffer[20];
    uint8   i;
    CyGlobalIntEnable; /* Enable global interrupts. */

    I2C_1_Start();
    LCD_Start();
    LCD_PrintString("Voltmeter");

    VDAC8_1_Start(); // Add the VDAC for a data source
    VDAC8_1_SetValue(0);


    ADC_DelSig_1_Start(); // Start the A/D converter
    //ADC_DelSig_1_StartConvert(); //Start the A/D conversions

    for(i=0;;i+=10)
    {

        VDAC8_1_SetValue(i); //128 * 4 mV is the voltage out
        Result = ADC_DelSig_1_Read16();
//        ADC_DelSig_1_IsEndConversion(ADC_DelSig_1_WAIT_FOR_RESULT);//wait
until there is a conversion
//        Result=ADC_DelSig_1_GetResult16();
        LCD_Position(0,0);
        LCD_PrintString("                    "); //Clear a line
        LCD_Position(0,0);
        LCD_PrintNumber(Result);
```

```
        sprintf(Buffer,"vDAC    %4d mV",i*4);   //4mV * VDAC value gives
voltage out
        LCD_PrintString(Buffer);
        LCD_Position(1,0);
        LCD_PrintString("                CNTs"); //Clear a line
        LCD_Position(1,0);
        LCD_Position(2,0);
        LCD_PrintString("                   "); //Clear a line
        LCD_Position(2,0);
        sprintf(Buffer,"A/D     %6.4f V", (Result+2)*2.048/65535.);
        LCD_PrintString(Buffer);
        LCD_Position(3,0);
        LCD_PrintString("                   "); //Clear a line
        LCD_Position(3,0);
        sprintf(Buffer,"A/D    %6.4f V", (Result+2)*2.048/65535.);
        LCD_PrintString(Buffer);

        CyDelay(2000);// wait 2000ms


    }
}
```

There are two other A/D converters on the PSoC. These are both 12 bit (max) Sequential Approximation Converters (SAR). One thing that is really nice about these is that they can do 1 million samples per second. However, to get 1MS/s, you actually have to make sure you use the correct clock frequency – a multiple of 18MHz.

Multiple inputs on a single A/D – Multiplexers

If we want to make measurements of different signals with a single A/D converter, then we have to use a multiplexer. Of course there are analog multiplexers available on the PSoC. In the analog section, drag the sequencing analog multiplexer onto the schematic. Double click on the multiplexer to bring up the configuration screen. Change the channels to 2, and type to Differential. Change isolation to Maximum. Now we have to start the multiplexer (remember to build first to update the libraries and autocomplete).



Since we are using a sequential multiplexer, we have to use software to switch it from one pair of channels to another. The command for this is AMUX_Next().

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

int main()
{
    int Result[2]; //Since I am doing a 16 bit conversion.  Since I am doing
differential measurements, I have to use a signed integer.
    char Buffer[20];
    uint8  i, j;
    CyGlobalIntEnable; /* Enable global interrupts. */

    I2C_1_Start();
    LCD_Start();
    LCD_PrintString("Voltmeter");

    VDAC8_1_Start(); // Add the VDAC for a data source
    VDAC8_1_SetValue(0);
    VDAC8_2_Start(); // Add the VDAC for a data source
    VDAC8_2_SetValue(0);
    AMuxSeq_1_Start();

    ADC_DelSig_1_Start(); // Start the A/D converter

    for(i=0;;i+=10)
    {
        VDAC8_1_SetValue(i); //128 * 4 mV is the voltage out
        VDAC8_2_SetValue(i); //128 * 16 mV is the voltage out
        for(j=0;j<2;j++)
        {
            AMuxSeq_1_Next();
            //ADC_DelSig_1_SelectConfiguration(j+1, 1);
            Result[j] = ADC_DelSig_1_Read16();//CyDelay(100);
        }
        LCD_ClearDisplay();
```

```
        LCD_Position(0,0);
        sprintf(Buffer,"%6d, %6d CNTs ", Result[0], Result[1]);
        LCD_PrintString(Buffer);

        LCD_Position(1,0);
        sprintf(Buffer,"DAC1 %6.4f V",i*0.004);  //4mV * VDAC value gives
voltage out
        LCD_PrintString(Buffer);

        LCD_Position(2,0);
        sprintf(Buffer,"DAC2 %6.4f V",i*0.016);  //4mV * VDAC value gives
voltage out
        LCD_PrintString(Buffer);

        LCD_Position(3,0);
        sprintf(Buffer,"0:%5.3fV, 1:%5.3fV", (Result[0]+2)*2.048/65535.,
(Result[1]+3)*2*6.144/65535.);
        LCD_PrintString(Buffer);

        CyDelay(2000);// wait 1000ms
            }
}
```

This works fine if you want both channels to have the same input parameters. However, if you have two signals which are very different in magnitude, you might want to have different input parameters. For example, you might want more resolution, or you might want to have a larger (or smaller) range. For this, you switch configuration.

To switch configuration you simply add the command (commented out in the code above) ADC_SelectConfiguration(Config Number, (0 or 1)). The config number is a value of 1 – 4 and corresponds to the each of the configurations. The 0 or 1 tells the ADC to either restart (1), or wait to be told to restart (0).

Rudimentary IIR Filter

Often times signal is noisy and we would like to perform averaging or filtering. There are many ways to do this, but digital filters fall into two classes: Finite Impulse Response Filters (FIR) and Infinite Impulse Response Filters (IIR). We will develop an IIR averaging routine which is simple to implement.

Consider a windowing average in which you store 10 data points and average them, and then average the next 10, etc. This is an FIR filter because the average only effects 10 points. One problem with this method is that it requires storage, something of which a microcontroller has little. If we want 100 points, then we need to store 10x the data.

If we write out the expression for averaging N points: $\overline{x_N} = \frac{\sum_N x_i}{N}$, where x_i are the individual measurements. This means that the sum can be represented by $\sum_N x_i = N\overline{x_N}$. Now imagine that we want to average over N+1 data points in which the new data point is x_{new}. The average would be given by: $\frac{\sum_N x_i + x_{N+1}}{N+1} = \overline{x_{N+1}}$. Therefore, $\overline{x_{N+1}} = \frac{x_{N+1} + N\overline{x_N}}{N+1}$. This is easy to implement. It says that the average of the last N+1 points is equal to the most recent measured value (x_{N+1}) + N multiplied by the old average divided by N+1. If you model this with an impulse, what you will find is that it is exponential in nature. Therefore, if you start at zero, you need to average more points to get an accurate average.

It turns out that this method of averaging is a model of a passive low pass filter if analyzed using finite differences and N/N+1 is equivalent to 1/RC.

Communicating with computer

If we are going to acquire data, it is important that we do something with the data besides simply look at it with a small LCD screen. What we want to do is acquire the data to computer. To do this, we will use the programming port rather than the USB mini connector. Bitter experience has shown that the mini connector is remarkable easy to break off. It could be used, but it is a better connection to the programming port. This port communicates as a serial port. You must identify the serial port and then can use anything from LabView to access the serial port to Python. In python you need PySerial and can use that to open the serial port. Then it is a simple matter of reading the data.

We need a communication port on the PSoC. To add one, drag on a UART from the component catalog (UARTS are in communications folder).

The only three things you need to do are:

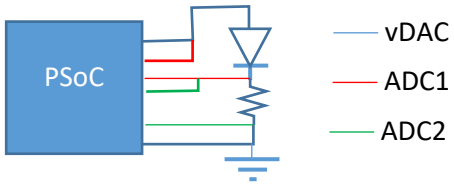
1) Make sure the RX and TX pins are tied to P12.6 and 12.7 respectively.
2) Make certain to turn on the UART in the software.
3) Set the Baud rate of the UART

A sample of python code for windows to read the UART is given below.

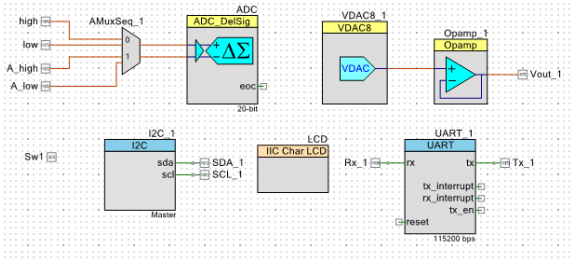
```
import serial
signal = []
ser=serial.Serial(port=”COM3:”)
ser.baudrate=115200
print ser

t=0
reading = "Wait"
while (reading <> "Start\n"):
   print reading
   reading = (ser.readline())

while (reading <>"End"):

   reading = (ser.readline())
   data = [float(val) for val in reading.split()]

   signal.append(data)
   t=t+1

print signal
ser.close()
```

Measuring Planck's Constant

A classic investigation is one of using many different LED's, determining their turn on voltage by plotting their I-V curves. Plotting the turn on voltage as a function of emitted frequency of light yields Planck's constant. For this project we will collect I-V curves from LED's. The basic idea is to read data for the voltage across the LED, read data across a resistor in series with the LED, and then increment the VDAC and then start over. The circuit looks like:



The PSoC schematic is shown below:



The follower is added to the VDAC in order to boost the current. The VDAC can only supply 0.5mA in the High Speed mode (less in the slow mode). Therefore, we can use an Op-Amp which will allow up to 10mA current. It is important to allow the system to choose the output of the Op-Amp.

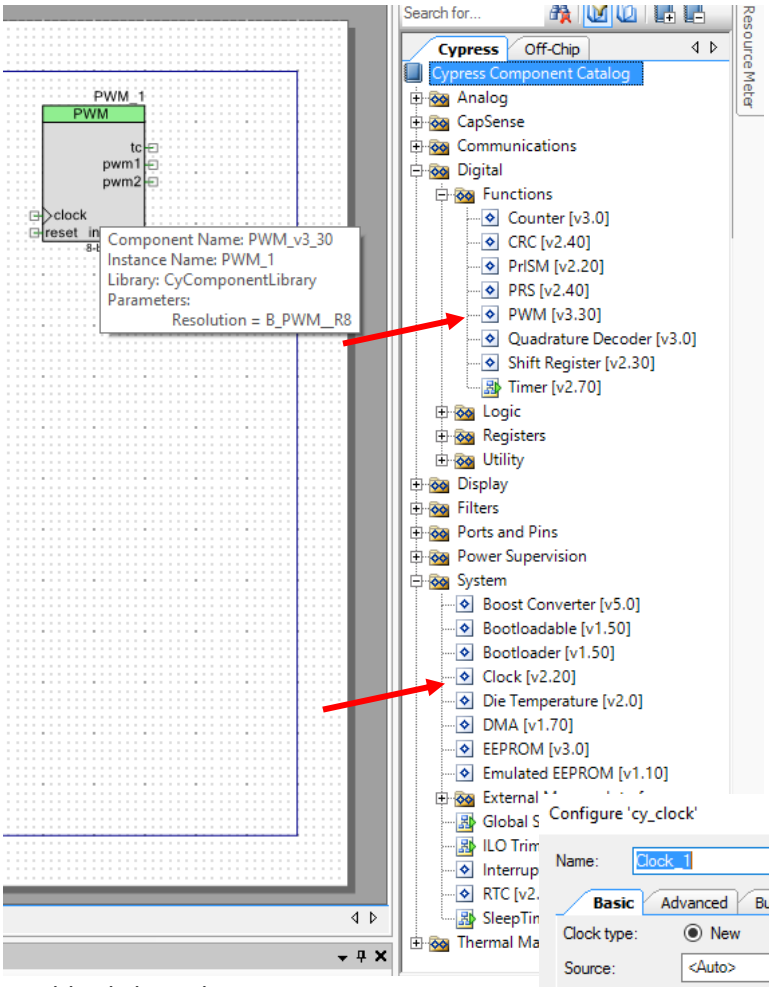
```
void PrintToUART(char *Buffer);

#include <project.h>
#include "stdio.h"

int button_press = 0; // global variable that is used to note the state of
the button press.
void button()
{
    if ((button_press == 0) && (Sw1_Read()==0))
        button_press = 1;
    else
        button_press = 0;

}
int main()
{
    int32 Vresult, Aresult;
    int N, i;
    uint8 k=1;

    float32 Volt = 0, Amp = 0;
    char strg[33];
    CyGlobalIntEnable; /* Enable global interrupts. */
    ADC_Start();
    I2C_1_Start();
    LCD_Start();
    AMuxSeq_1_Start();
    Opamp_1_Start();
    VDAC8_1_Start();
    //USBUART_1_Start(0u, USBUART_1_5V_OPERATION);
    UART_1_Start();
    UART_1_PutString("COM Port Open");

    AMuxSeq_1_Next();
```

```
N=10;


ADC_StartConvert();
for(;;)
{
    LCD_Position(0,0);
    LCD_PrintString("                    ");
    LCD_Position(1,0);
    LCD_PrintString("                    ");
    LCD_Position(2,0);
    LCD_PrintString("                    ");
    LCD_Position(3,0);
    LCD_PrintString("                    ");
    LCD_Position(0,0);
    sprintf(strg, "VoltiAmmeter AVG=%d",N);
    LCD_PrintString(strg);

    while (button_press == 0) // wait for button press to start
    {
        button();
        CyDelay(10);
    }
    //PrintToUART("Start\n");
    LCD_Position(1,0);
    LCD_PrintString("Start");
    VDAC8_1_SetValue(0);
    CyDelay(1000);
    for(k=0; k < 255 ;k++)
    {
        VDAC8_1_SetValue(k); //set the VDAC value

        for (i=0;i<N;i++) // averaging
        {


            ADC_SelectConfiguration(ADC_Voltmeter, 1);
            Vresult = ADC_Read32();

            ADC_StopConvert();
            AMuxSeq_1_Next();


            ADC_SelectConfiguration(ADC_Ammeter, 1);
            Aresult = ADC_Read32();
            ADC_StopConvert();

            AMuxSeq_1_Next();

            Volt = ((N-1)*Volt + Vresult)/N;
            Amp = ((N-1)*Amp + Aresult)/N;
        }

        LCD_Position(1,0);
        sprintf(strg, " Volt = %1.5f ", Volt/85333.33);
```

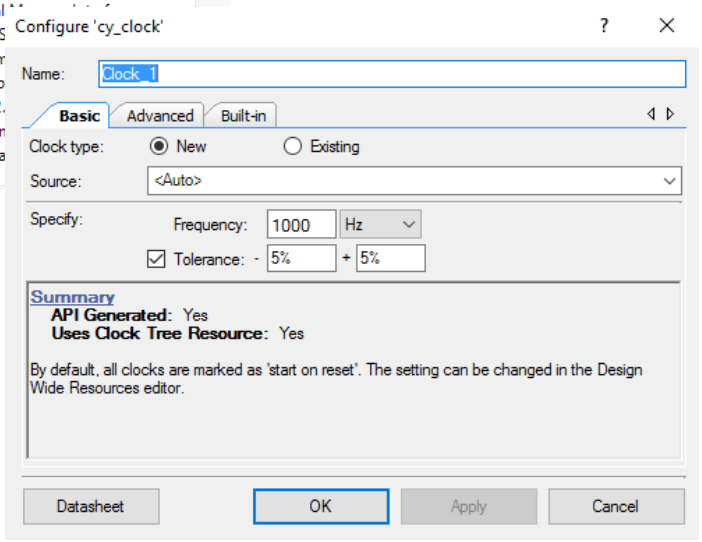
```
            LCD_PrintString(strg);

            LCD_Position(2,1);
            sprintf(strg, "mAmp = %1.5f ", Amp/51200.0);
            LCD_PrintString(strg);
            sprintf(strg," %1.5f %1.5f \n",Volt/85333.33, Amp/51200.0);//100
ohm resistor and current in mA
            PrintToUART(strg);
            LCD_Position(3,1);
            sprintf (strg, "k = %d", k);
            LCD_PrintString(strg);



        }

        LCD_Position(3,0);
        LCD_PrintString("End");
        PrintToUART("End\n");
        button();
    }
}

void PrintToUART(char *Buffer)
{
    //while(USBUART_1_CDCIsReady() == 0u);
    UART_1_PutString(Buffer);
}
```

Project 6 – Pulse Width Modulator

Pulse width modulators are wonderful tools for producing pulses. As input they take some clock source. They have two settings: period and duty cycle. The period essentially sets the number of clock "ticks" between outputs. The duty cycle varies from 0 to the period and determines how much of the pulse is "on" (or "off" depending on the settings).



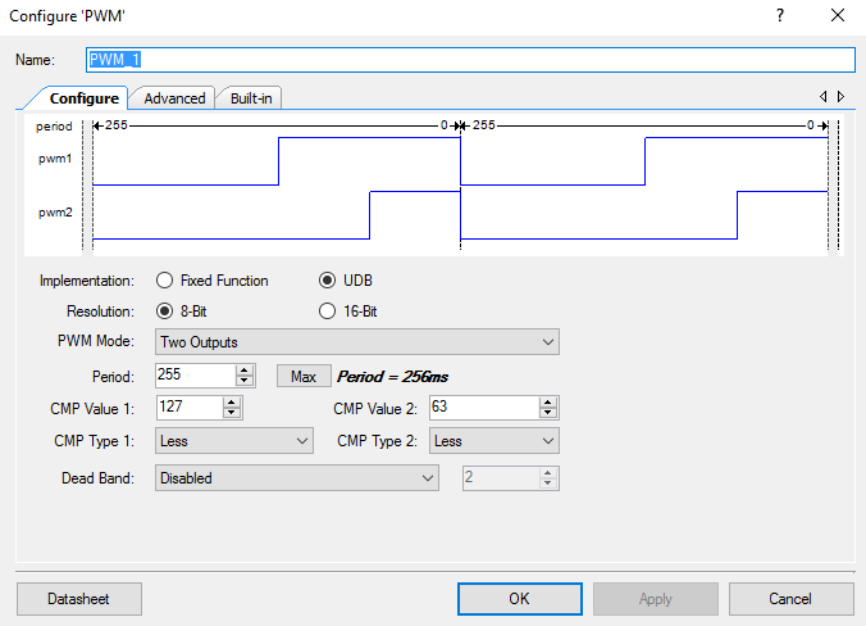
Drag the PWM onto the schematic from the digital/functions folder. Drag a clock on from the system folder. Attach the clock to the clock input. Then double click on the clock for the settings. You can change the frequency of the clock over a wide range of values. Be aware that this frequency is determined by dividing the system clock. The system clock can be set in the clock tab of the <my project>.cywdr file. Once you build the project you might wish to check this tab to make certain your clock is set as you desire. Because the clock is determined by division, you can only divide by integers. So if you want an 18MHz clock, but have a 24 MHz system clock, it is impossible. But a 12 MHz is possible as is a 8 and 6 MHz.



Double click on the PWM to open its settings. There are a variety of different settings that need to be discussed. Implementation: Fixed function or UDB – There are fixed function PWM's on the PSoC, these are dedicated. On the other hand, the UDB (Universal Digital Block) is part of the reconfigurable digital circuitry on the PSoC – the FPGA like part. There are 24 UDB's so you can only use so much of this.

Resolution: determines how large a number you can count for the period and duty cycle (CMP value).

PWM Mode – defaults to two outputs for UDB PWM's. The two outputs have the same period and different pulse widths. Other options are single output, dual edge – which allows you to use both PWM outputs to define a pulse, center align, dither, and hardware select. For the moment we will just leave it as dual output. Finally, we get to the explicit settings of the PWM. The graphic shows what each of the two pulses are doing. CMP type changes the response of the PWM, this should be explored.



Add two digital outputs to the PWM, and then build the project. Open the main.c file and start the PWM. Then rebuild and program the PSoC. That's it! We have a working PWM that we can look at the output on the oscilloscope, or if we set a slow period we can make an LED flash (YAY!).

If you wish to change the period or the compare value, you should stop the PWM, and then change the values, and then restart it.

Synchronous Counter

A synchronous counter is a counter that uses an external clock. Pulses ultimately must align with that clock.

We will start out by setting up the window for counting. This is done using a pulse width modulator. On the schematic page, drag in a PWM from the Digital->Functions component in the library on the right. Drag in a clock from the System folder. Then also drag in a Digital Output Pin from the Ports and Pins folder. You can wire the clock to the clock input of the PWM. You can wire the pin to the pwm port of the PWM component.

Double click on the clock and set it to 1 kHz.

Double click on the PWM, set it to one output and set the period to 999 and the compare value (CMP) to 500. This will create a 50% duty cycle 1 Hz pulse.

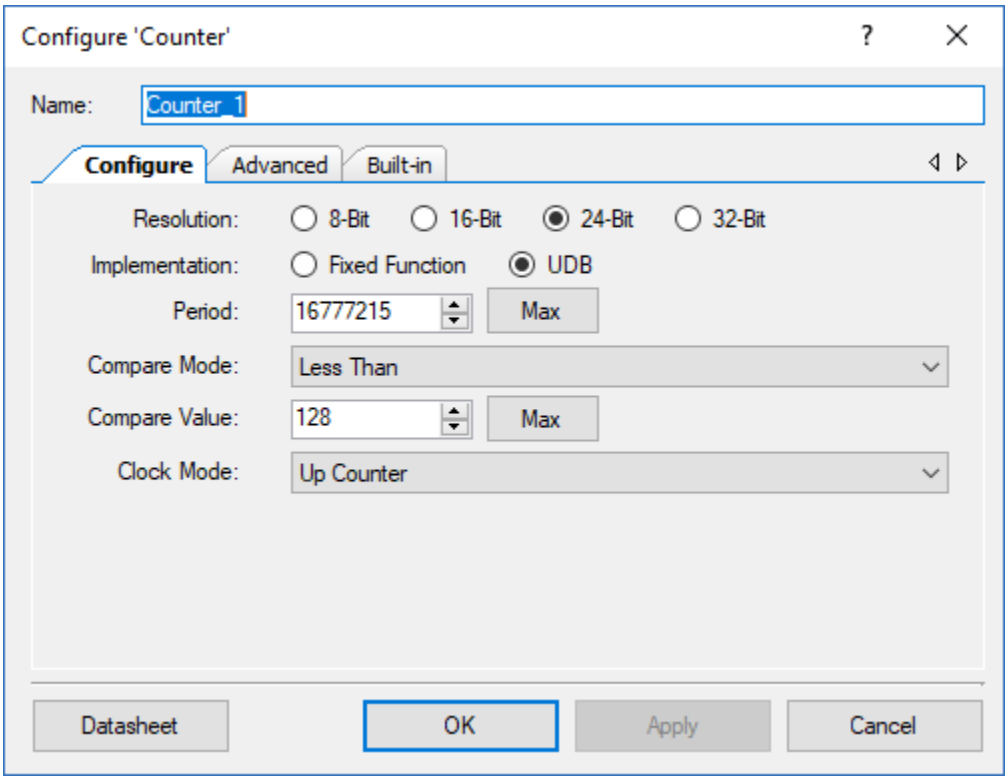
From the menu at the top, choose Build->Project (whatever you named your project). This serves the purpose of loading all the libraries and headers necessary.

Double click on the main.c file in the left panel. Type in PWM_1_Start(); before the for loop start. This turns on the PWM.

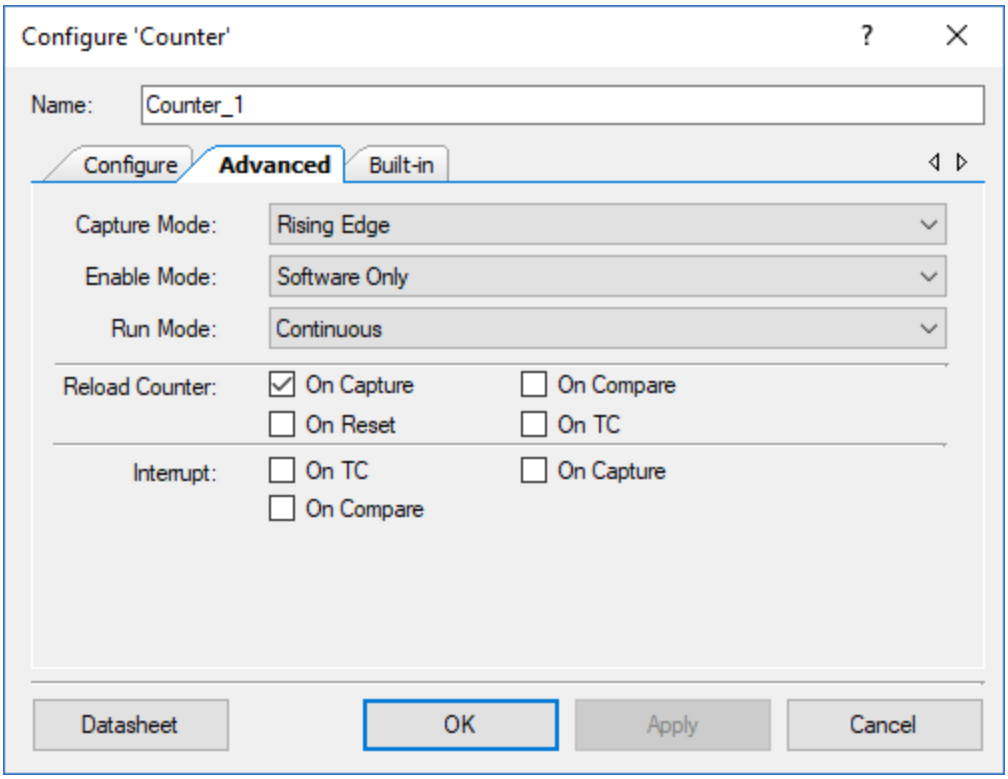
Double click on the .cydwr file which will open a diagram of the IC. This will show to which port Pin_1 was assigned. It can be reassigned on this page.

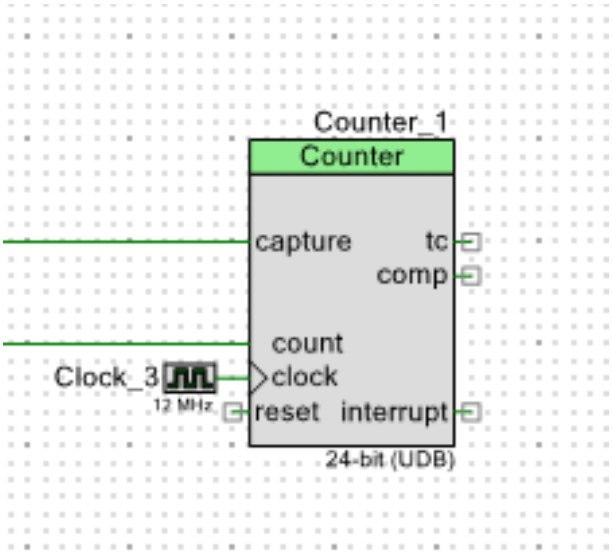
Now you can build and program a device. Attach an LED to the pin if you like so you can see the window pulse.

We still need to count. Drag in a counter.



In the advanced tab set the counter so that it has a capture on rising edge and the reload counter is set to “on capture”.





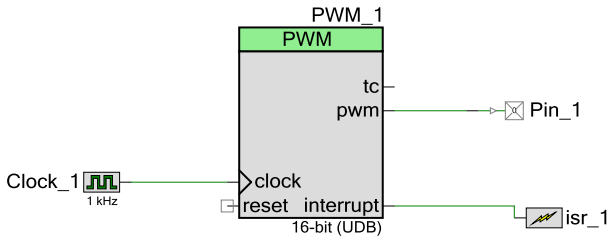
Attach the PWM's output to the capture input of the counter. The way the counter works is that when capture goes high (or low depending on how we choose to set it). The counter stores the value of the counter in the capture buffer. This value can be read by the CPU. We need to tell the CPU that there is data ready to read. This is the role of the interrupt.

The signal goes into the **count** input.

Setting up the interrupt (this is the hardest part).

We need to read the counters when the window completes a cycle. This will require creating an interrupt.

Drag an interrupt from the system component and attach it to the interrupt output of the PWM.



Double click on isr_1 and set the interrupt to rising edge. Choose advanced and set the interrupt to be on compare.

An interrupt is a method by which some device can "poke" the cpu and tell it to "do something".

Once the device sends an interrupt, you must have a routine to deal with the interrupt. This is shown below. It is placed in main.c as a new function.

```
volatile uint8 flag=0;
CY_ISR(isr_1_Interrupt)
{
    /*  Place your Interrupt code here. */
    /* `#START isr_1_Interrupt` */
      flag = 1; /* this sets the flag for to say that the interrupt had
                 * occurred */
      PWM_1_ReadStatusRegister(); /*this clears the interrupt so that
the
                                   * next one can occur */
    /* `#END` */
}
```

flag is used to communicate within the software .

In the main() routine, turn on the isr with **isr_1_StartEx(`isr_1_Interrupt`);** and in the for loop add the lines:

This code simply checks to see if the flag is set, then it resets the flag to zero. Within this if clause, we will read the counter values.

```
if (flag ==1)
{
        flag = 0;
```

```
        //do something here.
        Reading = Counter_1_ReadCapture(); //this reads the capture buffer.
}
```

The PWM sends the interrupt once every period! It sends the compare signal high, once very period. This means the capture occurs once every period. **Therefore to set the capture window, we need to set only the period of the PWM which is acting as the window of the counter.**

Input. Drag two analog pins onto the schematic (we are going to be using low voltages. If we were using a detector that put out digital pulses we would use a digital input).

Drag a second PWM into the schematic and then use this as the source for the signal of the counter to test the counter. You may wish to use an LCD as a readout while you are testing your counter.

Once you are satisfied with the results, try using a signal from an external source. See how the counting works? You may find that the readings are irregular and seemingly random. This happens because we are using a synchronous counter and it expects signals to be aligned with the system clock. To do this, drag a Sync Block out of the digital->utilities folder in the catalog. This requires a clock – which should be the same as the Counter's clock. **We can then have the signal go to the sync block BEFORE going to the counter.**

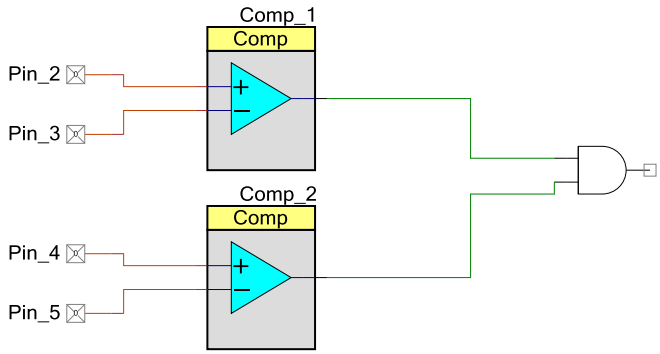
Counting Analog signals and Coincidences.

Drag a comparator on the schematic (in the analog components folder). Double click on the comparator and set it to fast and bypass sync. Attach the pins to the two inputs.

Click and drag a rectangle around the pins and comparator, copy (ctrl-c) and paste to duplicate the pins and comparator.

We now have the input configured for two channels. If we do nothing to the output of the comparators, then the output width will be determined by the input signal. This will be several microseconds. If we want shorter, we will have to do something to this signal. We can, for instance add a pulse converter or edge detector – both of which will limit the output to the duration of about 1 clock cycle.

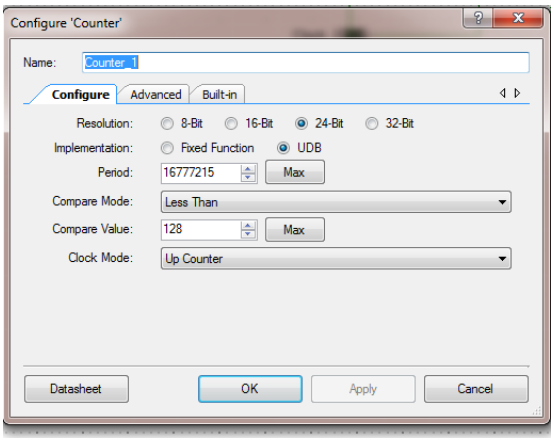
Because we only have two inputs, we only need to have coincidence detection between the two channels. This is performed by an "and" gate. Drag an AND gate from the digital logic components.



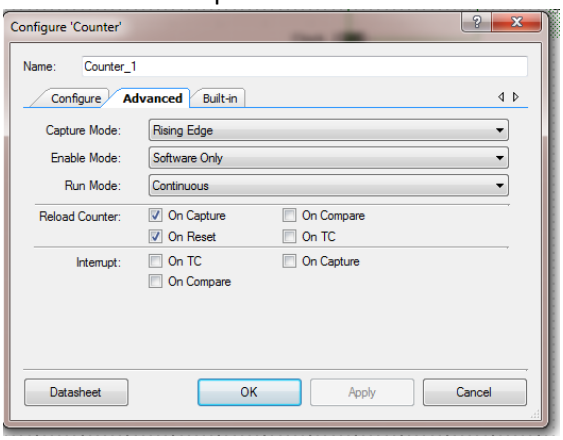
Notice that we can get the coincidence

In the main() program, we need to turn on the comparators (this is critical!). The AND gate does not need to be started.

Counters. Drag in a counter from the digital functions component catalog. Double click on the counter. Change the type to UDB (we will need the higher resolution). Choose 24-bit for the counter. For the period choose Max.



Choose the advanced tab of the counter and set it to have the capture mode be rising edge and reload the counter on capture and reset.

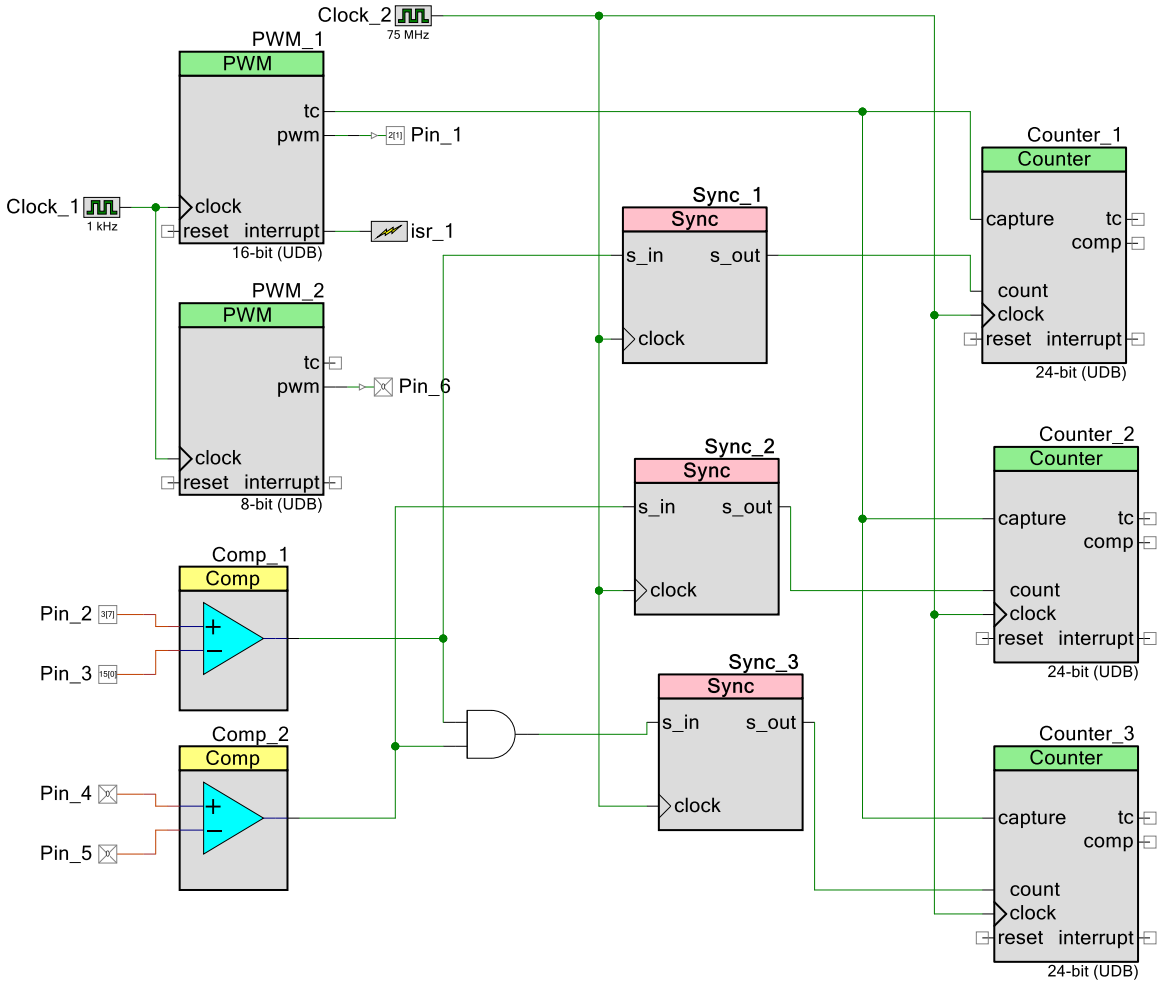


Wire the terminal labeled tc from the PWM to capture of the counter. Wire the output from the first comparator to the count terminal. Add a clock from the system catalog. Default maximum is 24 MHz. We can raise this safely to 75 MHz. To do this you must go to the chip view again, then select clock from the bottom tabs. Double click on the bus_clk item, then set the PLL to the desired frequency. Set the frequency of the counter's clock to 75 MHz.

Copy and past this counter so that you have a total of three counters. Wire all the clocks and all of the capture terminals together. Then wire the second comparator to the second counter and the AND gate output to the third comparator.

Finally, we have to turn on the counters with software, and then read them during an interrupt. To read the value of the counter we use the command Counter_ReadCapture()

At this point we have a working counter without a readout. All we have to do is attach the LCD and print the counter value to the screen. It would be painful if we had to copy down values once very second (turns out this is very hard to do for long). So it is worthwhile for counting statistics to save the values to computer.



This is a 2-channel coincidence counter able to take inputs from analog signal. We are limited here by the number of comparators available on the PSoC 5 [the number is 4]. This is pretty good. However, the speed of the comparators is a limiting factor to the counting rate.

SIO

If we are doing single photon work with detectors that already put out a TTL pulse, we can remove the comparators and go directly to digital inputs. But what if our detectors do not put out true TTL pulses. What can we do then? How do we deal with that problem? You can create these

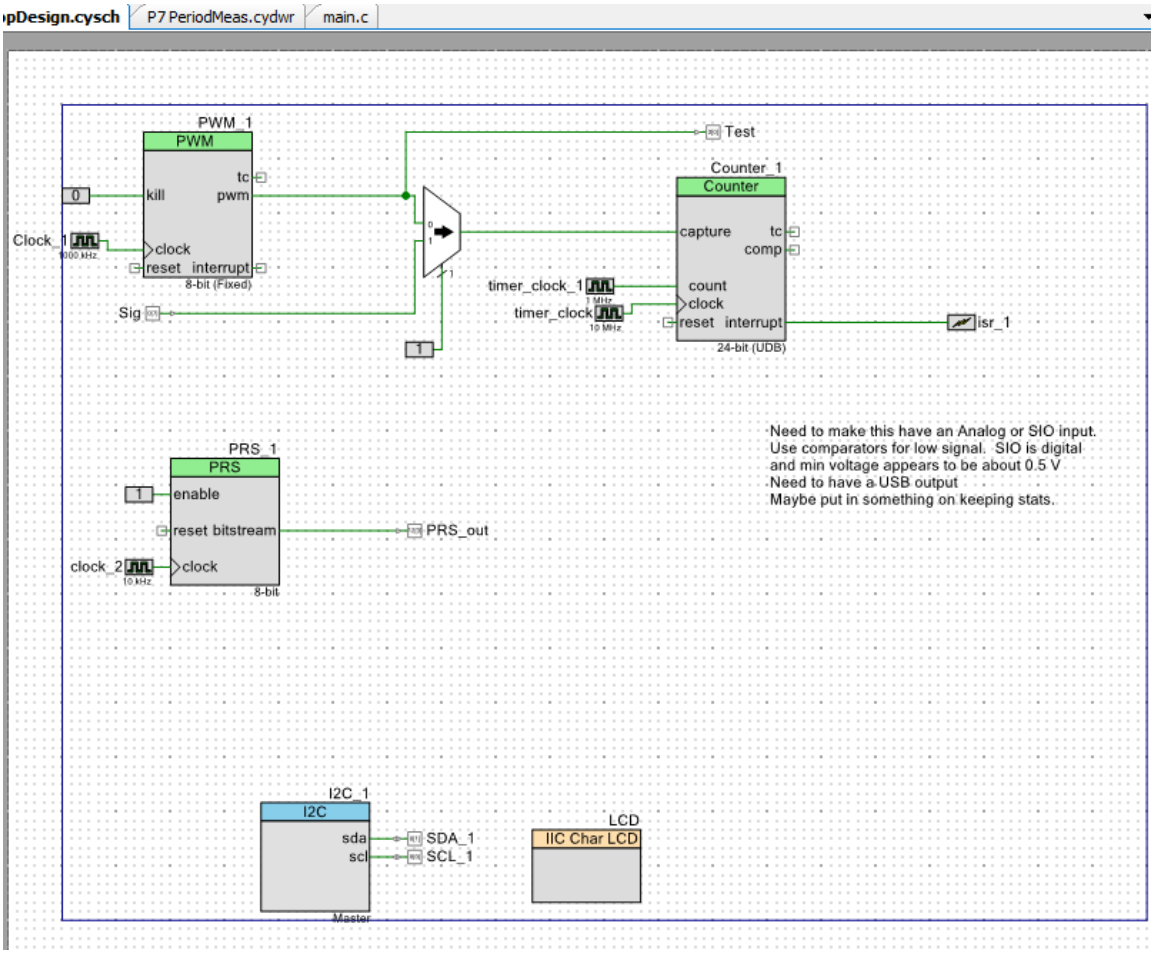
It turns out that there is a set of ports on the PSoC 5 which are called **SIO** ports. These ports allow level shifting. You can control the level that these ports trigger on with an external voltage (somewhat like a comparator) or with a DAC. You can get down to 0.5V transition voltages. You can turn on SIO for output by double clicking on an output pin/port and then selecting the output tab. Then select the **vref** from the drive level menu. For an SIO input port – select the input tab and then select vref from the threshold drop down menu.

So that is one problem down. What if we want to count more things, especially if we have four input channels. What we discover is that our six-channel counter (using these synchronous counters) will use up all the digital capacity (all the Universal Digital Blocks) of the chip. What if we want a truly asynchronous counter? These are different challenges that require significant efforts to address.

Measuring Periods

Another investigation that can use the LED's as single photon detectors is looking at the period measurement.

To measure the period, one needs a known clock and a counter. The counter is started to capture on the rising edge of the capture (i.e. the signal that you are measuring). Then it keeps counting until it gets another pulse at which point it reloads the counter buffer (sets it to 0). We need an interrupt when the counter is ready to reload. This lets us read the value that the counter reached. The time between pulses will be the number of counts multiplied by the period of the clock.



In this schematic, there is a digital multiplexer that allows you to switch from the test PWM (PWM_1) to a signal input pin (Sig) by changing the logic value. This could be a control register to control via software. The PRS is also for testing the period measurement, but with a pseudo random pulse generator.

The counter counts a 1MHz clock so it has a resolution of 1 microsecond. If we want higher resolution, we need a higher frequency clock.

The interrupt is set for when the counter captures. This tells the CPU to collect the count value.

Improvements would be to build in some type of binning on the PSoC, or perhaps simply acquire 1000 points and send that to the computer for binning there.