Week 2

How UART's Work

There are many ways to connect a microcontroller to another device, such as a sensor or a display. The simplest method is a direct parallel connection, where we connect each bit of some memory location in the microcontroller, usually called an I/O port, to a matching location on the other device. This results in a fast, reliable connection. But in a 32-bit device, it requires 32 I/O pins and wires. And anyone who has looked closely at a modern circuit board realizes that both are precious commodity. This is where serial communication comes in. Instead of moving 32 bits at once, we instead move one bit at a time using a shift register. This is, of course, much slower than a parallel interface, but it turns out that many forms of data transfer do not require super high speeds. A lot of data sources will talk about serial communication being done with a single wire, but you always have to have a ground connection between the two sources unless you're doing wireless transmission. So the minimum is really two wires.

There are many serial communication standards in use today. Some of the most common ones are I2C, SPI, ethernet, bluetooth, which is wireless, of course, and RS-232. They differ in their exact implementation, but all use a shift register.

A UART is a device designed to implement the ancient RS-232 serial data communication standard. It dates back to 1960.

The primary advantage of RS-232 is it just requires two wires in its simplest form, ground and one signal wire. This immediately presents an interesting problem. If data is shifted out on the signal wire one bit at a time, how does the receiving device know when each data bit is present? It will see only a series of high and low voltages.

You might imagine that we could add another line that would go high when the data is stable and ready to be sampled. Typically, this is done in other communication standards, and it's called a clock line, and that is what synchronous data communication is. There's a clock that tells the receiver when the data is present, but a UART doesn't have that. It saves out extra wire, so it really can operate with just two wires. But you have to somehow figure out when the data is stable and ready to be sampled when you're receiving it.

Both the I2C and SPI, and other types of serial data communication standards use a separate line, a clock line, that tells the receiving device when to sample the data. This is more accurate than the way the UART does it, because over time, the further and further you get from that leading edge that everything is based off of, the greater the chance is that you'll have a mismatch in where you sample the data. If you have a clock line, you don't have that problem because the clock line is constantly telling you when the next bit is

occurring, usually on the rising edge of the clock. The solution lies in timing each bit from the first falling edge of data transmission. The signal wire will sit high, and when it goes low, you start to do timing. In order to do this, of course, the receiving device will have to know in advance not only the pattern of the data, but also the rate of the data transmission. It has to know how long to time until the first bit and the subsequent bits. Mismatches in the agreed rate will result in garbled data, and you might have seen this when you're trying to set up a serial communication link with your PC. You'll always see a place to define the data rate when setting up a UART.

The UART does more than simply shift data out one bit at a time. It also formats the data. And by this, I mean the UART is responsible for automatically adding some extra bits to the serial data string. These are the parity and stop bits. Use of a parity bit is optional. It aids in error checking. If used, it is appended to the transmitted byte, and now we're up to nine bits total.

Use of the parity bit must be agreed upon in advance between the receiver and the transmitter. If the settings do not match, communication will not work. The polarity of the parity bit is chosen to make the total number of ones, including the parity bit, equal to either an odd or an even number. And yet, that's another setting that you have to agree on. There's something called odd parity and even parity. So first of all, you have to agree to use parity in the first place. And secondly, you have to decide whether it's odd or even. And it will be calculated on the sum of all eight data bits plus the parity bit. Here we have chosen to use even parity in the PSoC UART configuration dialog.

Remember, the receiving UART is responsible for recalculating parity on the nine bits received and declaring an error if the calculation is not even or odd, as was agreed to in advance. Of course, if two bits have flipped, the parity check will succeed anyway, and the error will go undetected. This is a limitation of this simple error checking scheme. More advanced communication formats use more bits to define checksums which can detect, and in some cases, even correct multiple bits in error.

The UART is also responsible for adding one or more so called stop bits to the serial data streams. These are added after the parity bit. Unlike parity, the standard requires at least one stop bit. There can be two, but no more.

Once again, if the number of stop bits sent by the transmitter does not match the number expected by the receiver, data errors can result.

Serial communication is complicated by the speed mismatch between how fast the CPU can write a single byte to the UART using a PutChar command, for instance, versus how long it takes the UART to actually move that data out to a receiver. In general, the CPU is much, much faster than the UART, so care must be taken to make sure the UART is ready for new data before writing to it. If you make this mistake, you will clobber the old data. This is known as an overrun error.

Calculating the exact time per instruction in any microprocessor can be surprisingly complex, but it is easy to measure.

A Basic Sensor Interface

But there are a few common issues to all of them. The first is what we call excitation, that is applying power to a sensor. Connecting a sensor to whatever power supply is available can introduce noise that corrupts the signal that you were trying to read. It may be necessary in some cases to filter the power supply noise before using it to power a sensor.

Some sensors, thermocouples come to mind, do not require any external power as they create their own voltages. Usually such voltages are too small to be directly applied to the input of an ADC, and thus require some amount of amplification.

Like all micro controllers, PSoC runs off a single-ended power supply, which means it can only interface to voltages between ground and VDD, which is usually 5 volts for our purposes. Of course, not all sensors are limited to this range and when this occurs you'll have to use external components to clamp the applied voltages to the range that PSoC can handle.

When you have two diodes of the same type in parallel, it's not clear how the current will be shared. Most of the current could go through the external diode, or it could go through the internal diode. It depends on exactly how the diodes were created. The internal diodes are limited to about 100 microamps and that's right here in the data sheet.

Remember that all GPIO pins, and pretty much anything for that matter, exhibit a certain amount of capacitance. This can be as much as 20 picoFarads for some pins, again according to the PSoC 5LP family data sheet.

But it's on the order of single digit to tens of picoFarads. That's not really a lot. Your typical oscilloscope probe has about that much. But it makes a difference when it comes to high frequency operation, because the input capacitance of the PSoC chip, in combination with that series resistor of 10K, forms a RC low-pass filter. And that limits how fast a signal you can apply externally and still see it in PSoC.

If you are dealing with frequencies in the audio frequency range, let's say 10 to 20 kilohertz, this is usually not a big deal. But if you get up into the hundreds of kilohertz or megahertz, then it starts to make a difference, and you can lose amplitude at the PSoC input.

Once we have satisfied input voltage swing protection, and it maybe that none is, the next step in a signal chain is usually amplification. And in PSoC that means a PGA component. If the input signal is small and varies from 0 volts to perhaps 0.2 volts, a simple PGA with a gain of 25 would bring the maximum input signal up to 5 volts. But we cannot configure a PGA for any gain we want, there are limited values available.

And if you want to use it in the simplest possible way you can reconfigure it and just say use internal VSS, not quite as accurate in some cases.

You can block all the DC by simply inserting a capacitor in series with the signal, that would have to be external component, because you don't have capacitors inside PSoC. But if you put a capacitor in series of the signal you create a high-pass filter. How high depends on how large a capacitor you use, and the resistance of whatever is connected to the capacitor. This works similarly to the low-pass filter we talked about. There's always some resistance and some capacitance, and they work together to create filters whether you want them to or not.

The second method works by essentially subtracting out the undesired offset before it can be amplified.

Besides applying gain and off et to an input signal is often useful to apply some degree of analog filtering before presenting it to an A to D converter. There are couple of reasons for this. First, the well known Shannon Nyquist sampling criteria states that frequencies exceeding half the ADC sample rate will corrupt the digital result. Since no analogue filter is perfect a common rule of thumb is to limit the input frequency to just one-tenth of the ADC sample rate. So if your input signal bandwidth is one kilohertz, you must sample at at least two kilohertz, and preferably 10 kilohertz or more.

Note, we are using the term hertz synonymously with samples per second. They're not quite the same thing but this approximation will suffice for now. Second, thermal and other types of noise are always present in electronic components. Most of these noise sources are wide band, so it pays to filter out as much of them as possible, up to the limit opposed by the bandwidth as the signal you want to pass.

Besides describe it in terms of amplitude, we also need to think about its bandwidth. Whatever that value is the bandwidth of the thermal noise is wider. And there's no point opening up the bandwidth wider than it needs to be to let the signal go through. Doing so will just let in more noise.

The decision of whether or not to implement an analog filter depends on how much noise is present in the signal bandwidth. It may be low enough to forgo this complication. But you should have a full understanding of the consequences. Once again, if this input signal contains frequencies that are approaching half the sample rate of this ADC, you'll get a corrupted output, it'll be wrong.

For many types of sensors, like temperature sensors, the rate of change is low and you can cut out any noise above a few hertz with a simple low-pass filter.
You can also block a DC offset with an external capacitor. Remember, when you do this, always think about the parasitic resistance and capacitances in the circuit. This input will have some resistance to ground and now you have a high-pass filter formed by this C and

whatever resistance exist over here. And if you're dealing with signals in the megahertz or hundreds of megahertz range that might become important.

Thermistor Lab, Part 1

Measuring temperature is one of the most common of all sensor applications. There are many ways to do it, but four most commonly used methods.

- The thermistor
- RTD: resistive temperature detector
- Thermocouples
- semiconductor devices like diodes.

Let's review the basic characteristics of thermistors. First of all, they're inexpensive. They can cost less than a dime. They're also highly sensitive. Much more so than many of the other sensor types. They work as a variable resistance, but they're non-linear and the range of temperatures over which this linear curve works is relatively limited. For instance, in the Ametherm catalog the table stops at 150 degrees C while other types of temperature sensors can go much higher.

RTD's are also variable resistors. Typically a 100 ohm chunk of platinum. They are not very sensitive, however. The amount of resistance change per degree C is much, much lower than a thermistor which makes them harder to interface to. On the other hand, the output is fairly linear and if you don't need the highest degree of accuracy, you can just assume a linear curve fit.Prices are more expensive than thermistors. A few dollars. However, they can work over a much wider temperature range. For instance, one of the devices I see from US Sensor is rated from minus 50 degrees C all the way up to 500 degrees C. And remember, the thermistor from Ametherm stopped at 150 degrees C.

Thermocouples, however, are a different animal. They are not variable resistors. They actually generate their own voltages. They consist of two dissimilar metals joined together, and the types of metals used determines the type of the thermocouple. The most common being the K type thermocouple. Interestingly, thermocouples always require a second temperature measuring device at the so called cold junction. However, the cold junction can be at room temperature while the other side can be very hot, like RTD's, into the many hundreds of degrees C. Four or five hundred degrees C is typical. So they operate over a very wide temperature range. But they're a little trickier to deal with in something as simple as our thermistor. For instance, it's quite easy to accidentally create a thermocouple on a printed circuit board, since the joining of any two dissimilar metals creates one. Usually the voltages produced by these parasitic thermocouples are small, but they can affect the accuracy of the circuit.

Diodes and over semiconductor devices can also be used to measure temperature. There'll be a voltage that is developed when the device is properly biased, and that will vary with temperature. Almost everything varies with temperature. The advantage of diodes is they're

easily included on an integrated circuit. However, they have a limited temperature range like any semiconductor device.

Here's an example of what that negative temperature coefficient curve looks like. At cold temperatures, the resistance is over 50,000 ohms. At hot temperatures, 90 or 100 degrees C, it's just a few hundred ohms, but it's a non-linear curve. So one of the problems we have to face in interfacing this to a microcontroller is to go from a non-linear curve to a linear output of temperature.

The curve you just saw can be modeled by something called a Steinhart-Hart equation. It involves three constants, a, b, and c, that are usually specified in the manufacturer's data sheets. The microcontroller needs only to measure the resistance and then calculate the temperature from this equation.

The temperature calculated by the Steinhart-Hart equation is not in degrees Farenheit, or even in degrees Celsius. It's in absolute degrees Kelvin. It's easy to use your microcontroller to add 273 to the result to get degrees Celsius from degrees Kelvin.

A little more difficult is evaluating the non-linear equation. That's best done if you include the math and floating point libraries that come with your compiler.

Thermistor Lab, Part 2

The ADC does not read resistance directly or inductance or capacitance, it only reads voltage. Thermister varies a lot with temperature. And so the voltage in between the two will change as the temperature increases or decreases.

Notice that by doing it this way, the ADC is never going to output a negative number because the input range is set up for plus or minus 1.024 volts. And since the minus terminal is at ground, the plus term was never going to get less than that and so the number read by the ADC is always going to be a positive number. So in some sense, you've lost half the resolution of your ADC, but let's not worry about that right now. In addition of that problem, this is still not the most accurate way to do the things, why?

Because well, there are several factors. First of all, the power supply is not a reference voltage. Which means that it's designed to provide a lot of electrons but it not necessarily very steady. So if the power supply varies at all, the current through this half-bridge will vary. Typical resistor precisions are 5% or 1%, but for not very much more money, you can get a 0.1% resistor.

And we can use that to determine how much current is flowing through the half-bridge. If we know the current in the half-bridge and we know the voltage across the thermister, then we know what the resistance of the thermister is. Just simply vehicles IR equation.

The second differential measurement will be made when we switch the analog multiplexer to channel 1, and that will be a measurement of the voltage across the thermister. And then by combining these two measurements, we will first determine the current through the 10K precision resistor and the thermister. Once we have the current, pi R equals V over I, we'll know what the resistance of the thermister is. Once we have the resistance of the thermister, we have to go back to that curve that shows what the temperature is versus resistance.

Which of the following files are stored in the Workplace Explorer Module of the Cypress PSoC Creator 4 workspace? All of the files related to your project

How do you keep track of external components on the schematic? Use the Off Chip tab in the component catalog

What code is stored in the PSoC file main.c when you first start your project? No executable code because you haven't created the project yet

Which of the following is not an off-chip component, available for selection from the Cypress Component Catalog? Motor controller

Which of the following is not an on-board component available for selection from the Cypress Component Catalog? 16 gigabyte memory

Which items are configurable for a digital pin in the PSoC system? Interrupt, Threshold, Sync Mode

How would you compare the resolution of the PSoC Delta Sigma ADC to that of the PSoC SAR ADC? The Delta Sigma ADC resolution can be configured between 8 and 20 bits in intervals of 1, while that of the SAR ADC can be configured to 8, 10, or 12 bits.

How do you connect a pin on the schematic to a physical pin on the PSoC development kit? Pin mapping is done in the pins tab (selected from Workspace Explorer). You click on the 'Port' and 'Pin' drop down, and select the appropriate port and pin number. Then the physical pins are connected to the pins in the schematic.

Which one of these items are commonly used LCD API in the PSoC system (assume the "instance name" has been configured to be just "LCD" on the schematic)? Use Cydelay () or other methods to make sure you don't write to the LCD too fast for it to respond, Write a string of characters (contained in double quotes) to the LCD using LCD_PrintString (), Insert the function LCD_Start () into main.c, Set where the cursor is on the LCD using the function LCD_Position (0,0)

What type of software protocol is implemented in a UART? RS-232, RS-485