UARTPrintf (2-1 reshoot)

Welcome back to Cypress Academy, PSoC 6 101. In next few videos I will show you the basic building blocks to create our BLE-Controlled robotic arm; this will include a UART terminal interface, PWMs to control the servo motors, EZ-I2C dashboard interface, digital logic-based kill switch, capacitive sensing controls and an advanced technique for debugging RTOS applications. As I go through these videos, I will first create a bare metal implementation so you understand the basics for each peripheral function and then we’ll integrate those functions with an RTOS into the BLE-controlled robotic arm project.

Let’s start with the UART interface to a PC terminal client. The UART interface is great as a rudimentary debug interface when developing an application. It can also be used as a basic communications peripheral for other system ICs.

So, let’s start by adding a new project to our previously used workspace. Let’s call this “BasicUART”.

As with our other projects, let’s drag and drop the UART component on to the schematic. Double click again to edit. Let’s call it “UART”. All the other settings look good, so click okay.

Double click on the pins file under the design wide resources and assign the RX and TX to P5[0] and P5[1].

On the back of the PSoC 6 BLE Pioneer board, you can actually see the silkscreen notes regarding the I/Os. This is a nice quick reference as you build out these projects.

[CUT FROM PREVIOUSLY EDITED VIDEO AND RESHOOT – ~2:04]

So, everybody likes to printf. How do I make that work? To do that I need to retarget stdout to the UART. We’ve built a library that will allow you to retarget which we call Retarget I/O. To include this library into your project, open build settings, click on PDL, scroll to down and click the check box next to Retarget I/O.

Now let’s generate the application. Once that’s done, you’ll see a file called “StandardIO\_user.h” that’s been generated. This is where you configure which UART you want stdout and stdin to point to. Scroll down to the first few lines of code. We need to insert a #include project.h so we can then reference the appropriate UART below. Next, go to the #define IO\_STDOUT\_UART and put in the name of the UART component, in this case, UART\_HW. Then do the same for the IO\_STDIN\_UART #define. And now our retarget I/O library is setup.

We’re going to control the UART with the Cortex-M4, so let’s open the M4 main application file. Start the UART using the API call. Standard IN is typically buffered, which means characters go into a buffer but you don’t know they’re there until you read. I want to turn that off for this program so we can handle each character as it comes in. To do this, we’ll write a line of text, setvbuf( stdin, NULL, \_IONBF, 0).

For this basic project I just want to echo the characters the PSoC 6 receives back to the terminal client. So, let’s create a character variable called c. Let’s show that printf works by printing out “Started UART example”. Now, in our main loop, let’s get a character, see if anything was returned, then print that character.

And that’s it, now time to build, program and test it.

First I’m going to do is open Windows’ device manager to see which COM port the KitProg is attached to. You can see which COM port it’s attached to under Ports and labeled KitProg2 USB-UART.

Now open up your favorite terminal client and attach it to the correct COM port at 115200 baud 8-n-1. I know the baud rate and 8-n-1 setting because they are in the component configuration dialog we saw earlier.

Now in the terminal client whatever I type is now echoed back to me on the screen…in this case, PSoC 6 is awesome! Which I completely agree with!

Lets start building the main controller project for the robot arm.

When you look in the PSoC Creator workspace with all of the projects. The final version of the main robot controller is called “MainController”. But I will also give you a project for each of the steps as I build through these lessons, those projects will be called MC-2-1-something or the other. For instance, in this case you will find MC-2-1-uart because this is the uart part of the project.

by doing ”

In the schematic start by gingping

Now, Let’s setup the FreeRTOS and Retarget I/O build settings like we did before in a previous lesson by going to build settings and selecting the checkbox for FreeRTOS and Retarget I/O. Now run Generate application.

Then fix the FreeRTOSConfig.h to get rid of the warning,

turn on semaphores,

and change the heapsize to 48K

Next configure the stdio\_user.h for happy-print-f-ing by adding the include for the project and updating the SCB that we will use for standard i/o to be the UART\_HW

For the main controller, I am going to build up a bunch of tasks – one for the UART, One for the PWM etc. Each of the tasks will go in their own files – a dot h and a dot c for instance uartTask.c and uartTask.h Any communication between the tasks will be done with RTOS primitives, like a queue, semaphore or event group etc. Those primitives will be shared in the file “global.h”

Start by creating the file uartTask.c using right click add new item. Then selecting C-file and giving it the name uartTask.c. In that file, first add includes for the project, FreeRTOS, the semaphore and standard io.

In the first UART example that I showed you, I polled the UART, but that is a silly waste of CPU time. For this project, I will use an interrupt so that the task can sleep until a key is pressed. The interrupt service handler will send a message to the task using a semaphore when it needs to process key strokes. The rest of the time it can just wait.

So, ill declare a handle to hold the semaphore called uartSemaphore

Now I need to create the interrupt service routine function called UART\_ISr. That function

will turn off the UART RX interrupts,

clear the RX interrupt, and clear the pending interrupt.

Once that is done I will give the semaphore, and switch to the right task.

Finally, I will create the UART task like we did with the blinking LED task before in the FreeRTOS intro lesson.

I don’t need any arguments so I’ll use void arg like before. Now, let’s start the UART, turn off the standard in buffer, just like before and print a message saying that the task has started.

Now I need to initialize the semaphore.

To configure the UART interrupt to happen when a key is pressed there are three steps..

1. Call Cy\_SysInt\_Init to install the ISR into the CM4 vector table
2. Call the CMSIS function NVIC\_Enable to turn on the interrupt handling
3. Call CY\_SCB\_SetRxInterruptMask to ask for interrupts when something has been put into the UART RX Buffer

The last part of the main task is the infinite loop. The way that this works is I wait for the semaphore, then read the keys from the UART until there aren’t any more while processing them with a command processor implemented with a big switch statement.

First, wait for the uart semaphore, then while there are keys in the UART RX buffer, get C from the UART. then use a switch to process the different commands. For now the only command will be the question mark, which will just printout the a help message for each of the commands. As we add new commands to the command processor switch I will add more print-f’s to this case as well as additional cases to handle the other commands.

Once all of this is done, you turn the UART RX interrupt back on and loop back to the top.

Now lets add the header information for the uartTask. To do this, create uartTask.h by right clicking the “header files” folder, selecting add new item. Then pick Header file and finally name the file uartTask.h.

In this file, add a #prama once so that this file is included into the project only one time, then a function prototype for the uartTask.

Now that we have built the task, it is time to get it started in the main program. To do this, you need to edit the main CM4 application. Add the include for uartTask.h, and FreeRTOS.h, then launch the uart task and then startup the scheduler.

All right let it rip by pressing the Build, program button…

To test it go to the Terminal again and press ?. OK… good, the help function works and we have a framework to add more commands.

Now we have our first UART interface working. In the next video, I will walk you through adding and configuring the PWM peripherals to control the servos in the robotic arm.

[RE-RECORD OR REUSE FROM PREVIOUS RECORDING]

You can post your comments and questions in our PSoC 6 community or as always you are welcome to email me at alan\_hawse@cypress.com or tweet me at @askioexpert with your comments, suggestions, criticisms and questions.