UART in the Main Controller

Welcome back to Cypress Academy, PSoC 6 101.

In the last video I showed you how to use the UART and the *printf*. Now let's get started building the main controller project for the robot arm.

When you look in the PSoC 6 Creator workspace with all of the projects that you can get off of our website, the final version of the main robot controller will be called - get this- “MainController”. But I will also give you a project for each of thsee steps as I build through these lessons. Those projects will be called MC-2-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.

This is going to enable you to sort of figure out what's going on as we go step by step without having to look at the whole thing at the end.

Ok, so now I’m going to create the main project of our BLE controlled robot arm. First, do this by doing File->New Project just like before. Let’s name this “MainController”.

In the schematic I will start by dragging and dropping a UART component from the component library just like I did before. I'm going rename it UART. Then I need to setup the pins in the Design Wide Resources for P5[0] and P5[1] just like we did last time.

Now, let’s setup the FreeRTOS and Retarget I/O build settings like we did before in the previous lesson by going to the build settings and selecting the checkbox for FreeRTOS and the checkbox for 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 the standard I/O to be UART\_HW.

For the main controller, I am going to build up a bunch of tasks – one for the UART, one for the PWMs, etc. Each one of the tasks will go in their own files – a dot c and a dot h. For instance uartTask.h and uartTask.c Any communication between the tasks will be done with an RTOS primitive, like a queue, a semaphore or an 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 the includes for the project, FreeRTOS.h, the semaphore and standard I/O.

In the first UART example that I showed you, I polled the UART, but that's really 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 then send a message to the task – the UART task - using a semaphore when it needs to process the key strokes. The rest of the time it can just wait and not burn CPU.

So, I'll 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 then switch to the right task.

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

I don’t need any arguments so I’ll use void arg like before to get rid of the compiler warning. 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.

I also 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\_EnableIRQ to turn on interrupt handling.
3. Call the CY\_SCB\_SetRxInterruptMask to ask for interrupts when something has been put into the UART RX Buffer.

The last part of the UART task is the infinite loop. The way that this works is: I wait for the semaphore, then read the key press values from the UART until there aren’t any more. I process each key with a big switch statement.

First, wait for the UART semaphore, then while there are keys in the UART RX buffer, get a character 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 print out the 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 put all of the stuff in for the additional cases to handle the other commands.

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

Now let's 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 only included into your project one time, then a function prototype for the uartTask.

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

All right, let's 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 the framework to add more commands.

Now that 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.

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