PANDORA PRODUCTS

ARDUINO to ARM
NXP tools

Jim Schimpf

Document Number: PAN-201406001
Revision Number: 1.0
10 February 2018

Pandora Products.
215 Uschak Road
Derry, PA 15627

Pandora Products.

215 Uschak Road

Derry, PA 15627

Phone: 724-539.1276

Email: jim.schimpf@gmail.com

Pandora Products. has carefully checked the information in this document and believes it to be accurate. However, Pandora Products assumes no responsibility for any inaccuracies that this document may contain. In no event will Pandora Products. be liable for direct, indirect, special, exemplary, incidental, or consequential damages resulting from any defect or omission in this document, even if advised of the possibility of such damages.

In the interest of product development, Pandora Products reserves the right to make improvements to the information in this document and the products that it describes at any time, without notice or obligation.

Document Revision History

Version	Author	Description	Date
0.1	js	Initial Version	20-Jun-2014
0.2	js	Editing and add contributors	21-Jun-2014
0.3	js	Fix circuit design	21-Jun-2014
0.4	js	Fix Link 2 cable picture	22-Jun-2014
0.5	js	Add information to activate LPCXpresso	22-Jun-2014
0.6	js	Picking the processor	22-Jun-2014
0.7	js	Startup sequence	27-Jun-2014
0.8	js	Edits to make first pgm run clearer	17-Jul-2014
0.9	js	Creating a new project	15-Sep-2014
1,0	js	Better debug connection	10-Feb-2018

Contents

List of Figures

Thanks

I want to thank the denizens of the NPX LPC11xx forum for their patient and useful answers to my rather naive questions. I would have never gotten this far without them. **LabRat**,**lxpresso-support** and **serge** thanks.

1   Introduction

I've used the Arduino[1] for a long time and being a lazy person I really like the system. All you need is the board and the Arduino IDE to be up and going. It's very low friction to get into and use. You don't have to make a deep study of the ATMEL part or how to set up a serial port baud rate, the Arduino code has that done for you. You can just write the higher levels of your idea and zip it's running.

But I also program on PC's and like the idea of a real debugging system with single step, stack unwinds and variable viewers. I've been wanting to use ARM's as they have the power to do this and have become really really cheap. The problem up till now has been that they are kind of high friction systems. That is unless you go for something like a Raspberry Pi or Beagle Board where you are running Linux just getting to main() in a program requires rather deep study of the chip, clocking and hardware. Not that this is a bad thing, you have to master a lot of this to do a complicated program but is a real pain to do your first "Hello World" or blinky program. In addition getting the ARM tool chain is not simple although going to http://www.linaro.org/downloads/ will get you a prebuilt tool chain which saves hours of work. Even with it you do still need the startup code to get you to main(). I found the tool chain link in http://www.meatandnetworking.com/ tutorials/arm-cortex-mx-quickstart/ which is a good article if you are interested in bare metal programming.

NPX has introduced the LPC1114 ARM M0 in a 28 Pin DIP package for around $3. This chip has 8K of RAM, 40K of Flash and runs at 48 MHz. It is in the running for an Arduino like niche. The 28 pin DIP means you (the average hobbyist) can use this on a standard breadboard. Over and above that it comes with software packages that bring it almost to the Arduino level of simplicity and abstracts the hardware enough that you won't be bogged down in that just to start.

There are debugging pods available for ~$20 that give you JTAG type debugging where you can breakpoint,single step and view stacks or variables of your running code. This is the part I like, running code with crossed fingers hoping it works always scares me a lot.

And lastly freeRTOS comes with the development code, it gives you a real time kernel where you have a real time OS with threads,mutex's and semaphore's. You can do true multitasking like you can in Linux.

The rest of this paper will detail how to set this up with the NXP tools so you can develop code and use the really neat little machine.

2   Materials

2.1   Hardware

LPC1114FN28/102 This is the part, it is available from DigiKey,Mouser and Newark

OM13054UL - Link-2 Debugging Pod, available from DigiKey,Mouser and Newark

Breakout-Board - SWD available from Adafruit

Breadboard Solder less Breadboard available from the same or Radio Shack

3.3VDC Power Supply, adjustable supply or see below for a design

LED Any color used in sample programs

330ohm Resistor Current limiter for LED

15Kohm Resistor (2) pull-ups for nRESET and PO0_1

2.2   Software

LPCXpresso Available from NXP http://www.lpcware.com/lpcxpresso/download
(Windows/Mac/Linux)

lpcopen_v2_00a_lpcxpresso_nxp_lpcxpresso_11c24 Available from NXP http://www.lpcware.
com/content/nxpfile/lpcopen-software-development-platform-lpc11xx-packages-

2.2.1   Activating LPCXpresso

After you install LPCXpresso it will be in Evaluation mode and will have a number of things disabled. To fix this you must activate the FREE version.

- Start the program and it will come up with a Welcome page that describes the activation process.

- Pick Help -> Activate and you will be taken to an NXP page that says you have to login or create a new account to complete process. If you don't have an account create one.

- Also it will show your program serial number. Cut and paste this into an editor for later.

- After you have created the account you will get an e-mail with your login link to the new account. Log in and set your password. Then log into the account.

- On the right you will see My Activations, pick this and you will go to a dialog where you enter your program serial number. When entered it will give you the activation #, cut and paste this into an editor.

- Now Pick Help->Activate->Activate(Free Edition) and you can enter the activation number here.

- Program will re-start and you are in business.

2.3   Power Supply

If you only have a 5V supply then you can create a 3.3V supply on the breadboard with it plus a 3 terminal regulator. Your system when running will draw less than 20 ma, only a small supply will be needed.
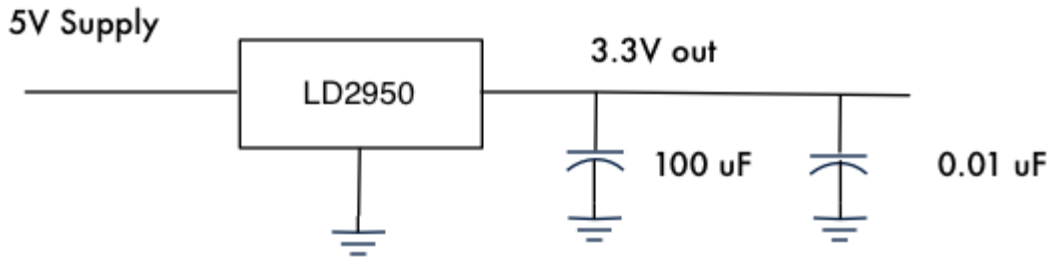
Figure 1: 3.3 V supply from 5V voltage

2.4 Link 2 Setup

After you receive the Link 2 pod move the enclosed jumper to J-2, J-1 does not need a jumper. If you don't do this the pod will not connect to the processor.

3 Building the board

The basic circuit for the processor will be quite simple as we will only be blinking an LED for the initial test program. We will not need a crystal as the internal RC oscillator is quite good enough.
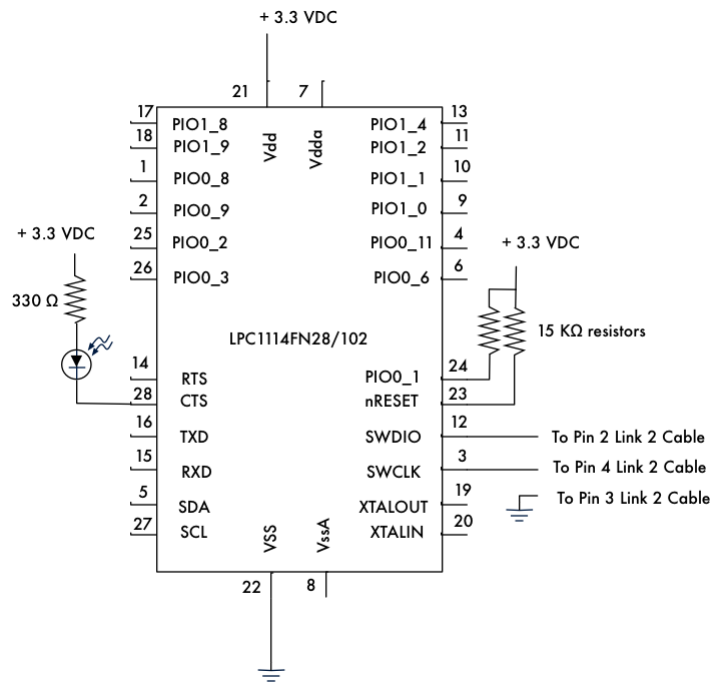

Figure 2: LPC1114 circuit

Note the debugging lines SWDIO and SWCLK, these are the connections to the Link-2 Pod. Also the LED is hooked to pin 28, this will be used in our test program.

There is a small 10 pin ribbon cable supplied with the pod and that we use to connect the pod to the breadboard. Shown here is the end of the cable looking into the 10 pin connector with the notes on the three wires you need.

You can obrail the W5110-A breakout board from Adafruit `https://www.adafruit.com/?q=SWD`. For the other end of the cable. Then plug the 10 pin cable from the Link-2 to the SWD breakout board plugged into the bread board.
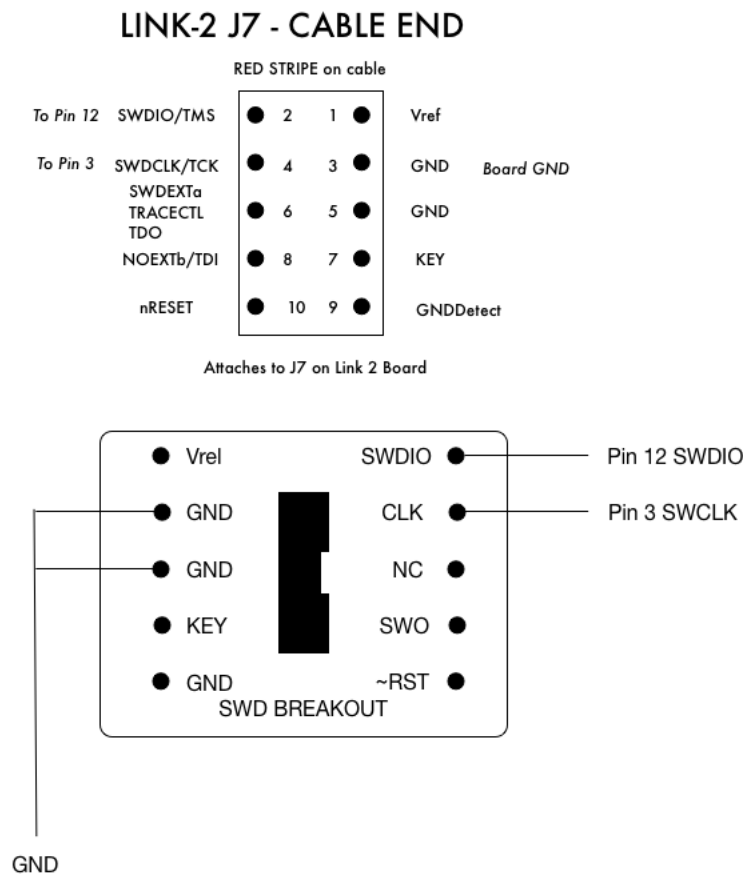
Figure 3: Link-2 Connector

4   Software

After you have actived LPXpesso (see 2.2.1 on page 3) then you can begin using it. To do this you need to create a workspace and install the library **lpcopen_v2_00a_lpcxpresso_nxp_lpcxpresso_11c24.**

The zip file can be left anywhere as we will browse to it when needed.

Start LPCXpresso and decide where to put your workspace (it can be anywhere you as a user can read and write files) and don't worry you can always create new workspace if needed. When you create the workspace it will ask you to find **lpcopen_v2_00a_lpcxpresso_nxp_lpcxpresso_11c24,** point at it andLPXpresso will expand this in your workspace. This gives you a bunch of example projects plus the board and chip support code.

You can read the **LPCXpresso_Getting_Started_User_Guide[2]** for a detailed guide to it's operation. Also it has notes on the various windows that will be discussed below.

5   First Program

5.1   Initial build

Look at the projects in the left side Project window of LPCXpresso and pick nxp_lpcxpresso_11c24_periph_blinky.
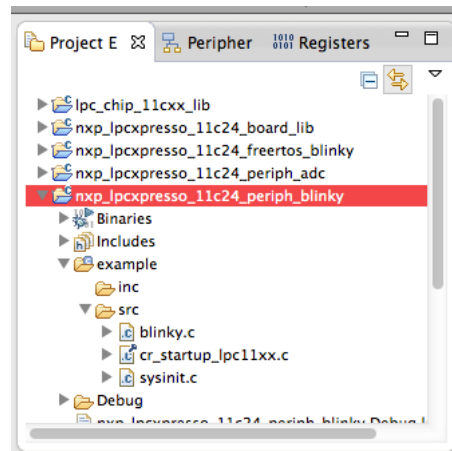

Figure 4: Pick project

Right click the selected line and pick Properties at the bottom of the list. Pick the MCU item and then LPC11xx list. Finally pick LPC1114FN/102 in the list. This should set the correct processor for your testing.
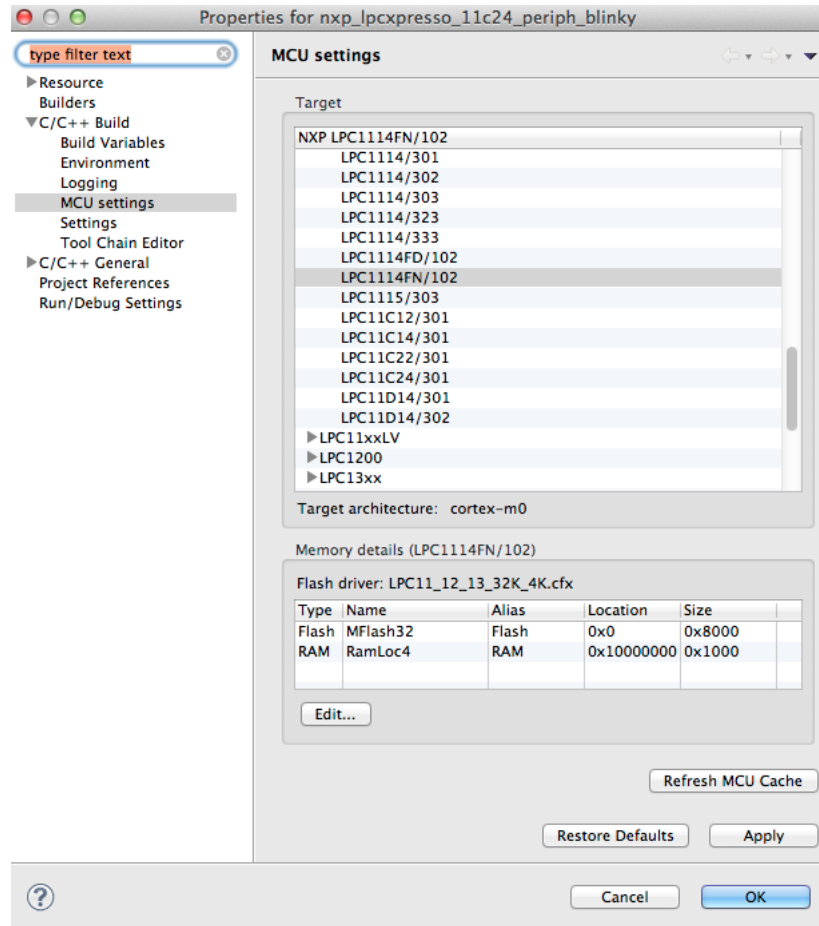
Figure 5: Picking the processor

In the this window pick build the project and it should build without error.
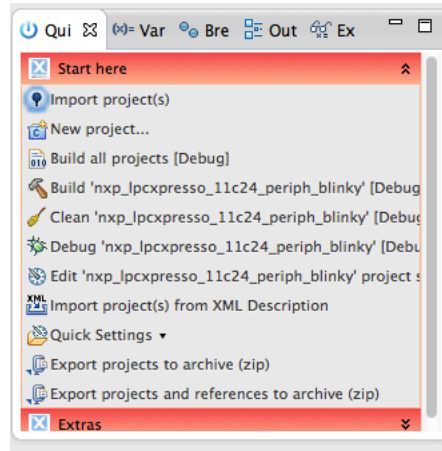
Figure 6: First Build

If all that worked without error we are almost ready, we have to modify the board support slightly to get it working with our single chip.

5.2 Modifications

Move up in the project list and open the nxp_lpcxpresso_11c24_board_lib project and open the board_sysinit.c file.
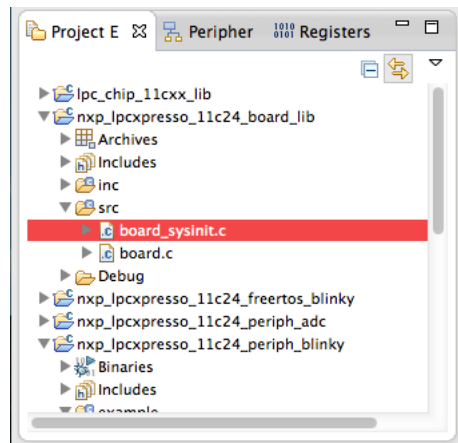

Figure 7: board_sysinit.c

We now have to edit this file to tell it our chip is using the RC oscillator and not a crystal to run.

```
/* Setup system clocking */
STATIC void SystemSetupClocking(void)
```

```
{
volatile int i;
/* Powerup main oscillator */
// Don't start MAIN
//Chip_SYSCTL_PowerUp(SYSCTL_POWERDOWN_SYSOSC_PD);
/* Wait 200us for OSC to be stablized, no status
   indication, dummy wait. */
for (i = 0; i < 0x100; i++) {}
/* Set system PLL input to main oscillator */
// USE RC OSC
Chip_Clock_SetSystemPLLSource(SYSCTL_PLLCLKSRC_IRC);
```

Two lines are changed, the first is to comment out the power up of the main oscillator as it's not used. The second is to change the original **SYSCTL_PLLCLKSRC_MAINOSC** to **SYSCTL_PLLCLKSRC_IRC.** This sets up the system to use the RC oscillator as we have built on our breadboard. Note: All the projects in this workspace will now use this board_sysinit and assume using the RC oscillator. This is what I meant about having different workspaces, you could have another where your chips did use a crystal and would not modify this code.

After this is done go back to the blinky project, CLEAN the project the build again, this gets the board_sysinit.c changes. Now you are ready to run.

6   First Run

After all this you can now press the Debug button either the bug at the top of the screen or the debug button shown in Figure 7 on the previous page. If all goes correctly and you have your Link-2 connected and plugged into USB it should then run the code and stop just after main() in blinky.c. You can single step at this point and if you hit Go (green arrow) it should run and the LED should blink.

If it fails to connect, a fix is to unplug the USB to the debug pod, go to debug again and it will bring up a dialog saying it doesn't have a connection. Then plugin the pod and in the dialog, tell it to search for the pod. This usually get's it going. If this still doesn't work check your debug pod connections, circuit connections and power supply.

7   Next Steps

All the demo projects should now build and run on the chip. In particular look at the freeRTOS project as that shows you how to use this RTOS and is the gateway to multitasking projects. You will also see in the various projects how they have routines to handle chores like setting up ring buffers for serial I/O, calls to setup hardware like I2C, SPI and GPIO. While not quite as simple as Arduino the examples make it quite clear how to use them and the supplied code has routines that make it easy to set up the hardware. You can check in the processor user manual for further details.[3]

8 Creating a new project

8.1 Introduction

So far we have used previously created projects to build and work with. This next section will show you how to create a brand new freeRTOS project. In doing this I have created the code for the project in OS X Xcode. I have written a simulator for freeRTOS and the board enviroment. This lets me write the code in XCode and debug a lot of it there. While LPCXpresso is a great enviroment, initially building the code involves a LOT of trial and error and XCode can build and debug MUCH faster since you don't to re-program FLASH each time.

8.2 Simulation

The simulation is in 5 parts:

- LPC1114 - This simulates the chip and board library

- freeRTOS - This simulates the freeRTOS system

- Devices - This simulates the i2c and Serial subsystems (that are used in projects I have done)

- Timer - This is used to simulate a assembly code fast timer

- TTY - This is used to supply a tgkbhit() function for stdio

8.2.1 Board and Chip Library

These are routines that usually do nothing but supply the linking calls for routines used in programs. Things like set a bit or read a bit and have the same for as the call. The chip/board .h files have defines for the objects used and structures. These are written as needed in developing programs. Also some of the calls have some functionality for testing the program.

8.2.2 freeRTOS

This library uses the pthreads library to simulate the threading and mutex's of the freeRTOS system. Also calls, structures and h files are duplicated so you can write code in the same way a real system would work.

8.2.3 Devices

These are simulations of routines written for the hardware. In this case I2C and serial I/0. Hardware based routines were witten to set up these devices and have convenienent calls into them. In the case of the serial device, calls similar to Arduino serial calls. These simulation routines duplicate the calls. The serial routine uses STDIO instead of a serial port but the calls are the same.

8.2.4 Timer

This is a C code version of the Assembly language fast timer routine.

8.2.5 TTY

This whole section was added to have a tgkbhit() function.

8.3 Preparation

After building and running the simulated project the code is ready to be added to an LPCXpresso project. Before we begin we need a number of bits and pieces. First get the LPC1114 devices code, in this case:

- fastimer.s/h - Fast timer ARM asm code and H file

- Menu.c/h used in this project

- Serial.c/h the real LPC1114 code not the simulator code

We also need the FreeRTOS sources and include files. Go to the workspace that LPCXpresso uses and find the freeRTOS project. In there you will find a directory called freertos with an inc and src directory. Also in the Example folder for the FreeRTOS project in the inc folder is a file called FreeRTOS_Config.h

Build a folder called Support and then it it create the a Support subfolder with src & inc folders holding the timer,menu and serial code mentioned above.

Then just copy the freertos folder from the freertos_blinky project into the top level Support.

Finally copy the FreeRTOSConfig.h from the example/inc folder in this project to the Support/Support/inc folder.

```
511 USB_PowerLink> ls -R Support/
Support
freertos          Support - My files, freertos - from Sample code
Support//Support:
inc src
Support//Support/inc:          My Code H files + FreeRTOSConfig.h
FreeRTOSConfig.h Serial.h
Menu.h fastTimer.h
Support//Support/src:          My Code C/ASM files
Menu.c Serial.c fastTimer.s
Support//freertos:             This is just a copy of FreeRTOS folder
inc src
```

```
Support//freertos/inc:
FreeRTOS.h mpu_wrappers.h semphr.h
FreeRTOSCommonHooks.h portable.h task.h
StackMacros.h portmacro.h timers.h
croutine.h projdefs.h
list.h queue.h
Support//freertos/src:
FreeRTOSCommonHooks.c list.c readme.txt
croutine.c port.c tasks.c
heap_1.c queue.c timers.c
```

8.4 Making the project.

8.4.1 Create a directory

Make a new directory for your project. By default these are created inside the LPCXpresso workarea but you can create this anywhere you wish. In this case we are creating PC05Example (PC05 is an X10 interface). Inside this directory create two folders, FreeRtos and Support.

8.4.2 LPCXpresso steps

8.4.2.1 New Project Pick File->New->Project and get the following window
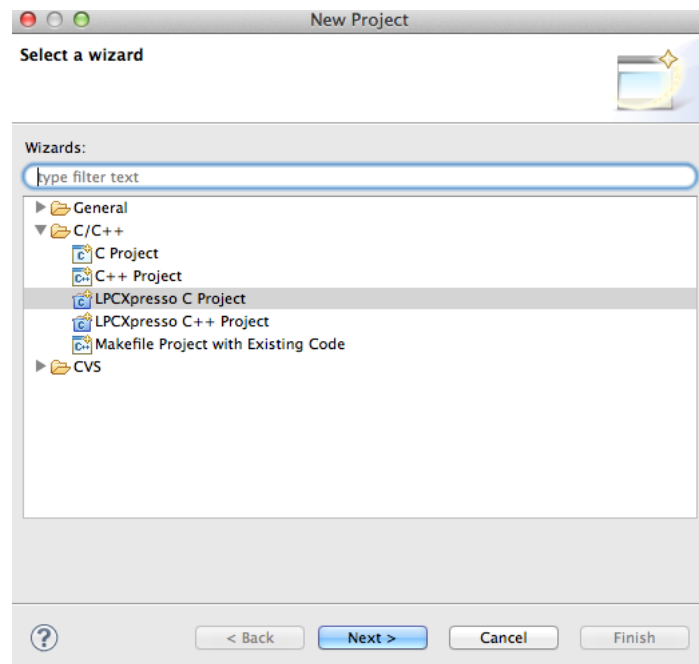

Figure 8: New Project

Pick the LPCXpresso C project item. Then on the next pick the correct project type:
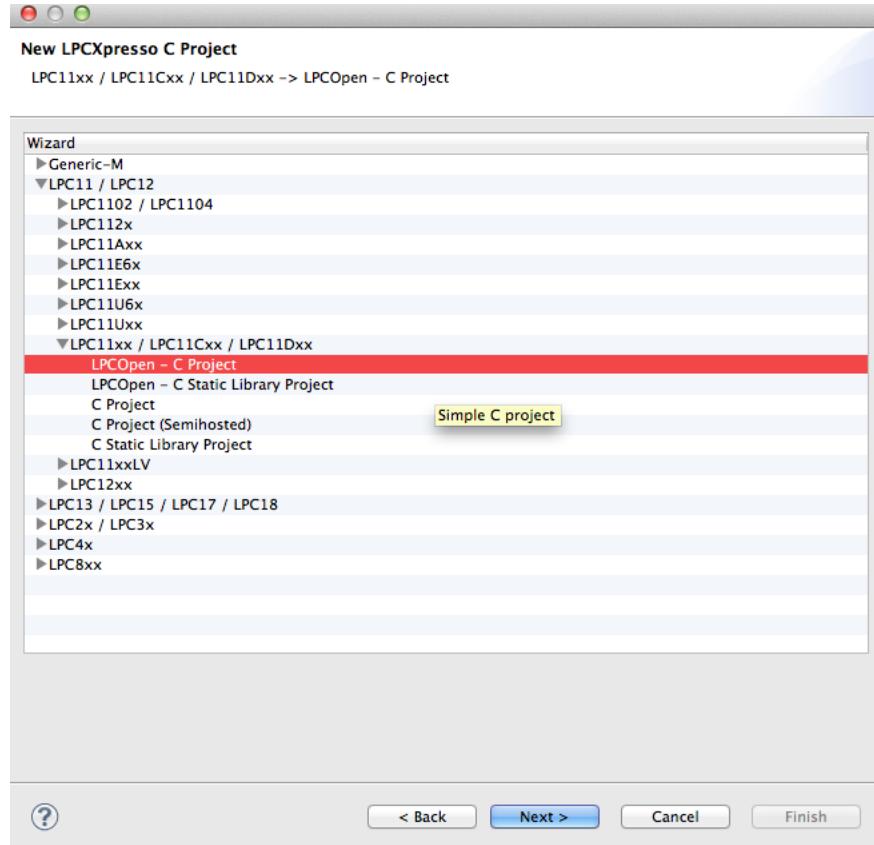

Figure 9: Project Type

Next we pick where to put it. In this case we over-ride the default and pick the folder we created also fill in a name for the project:
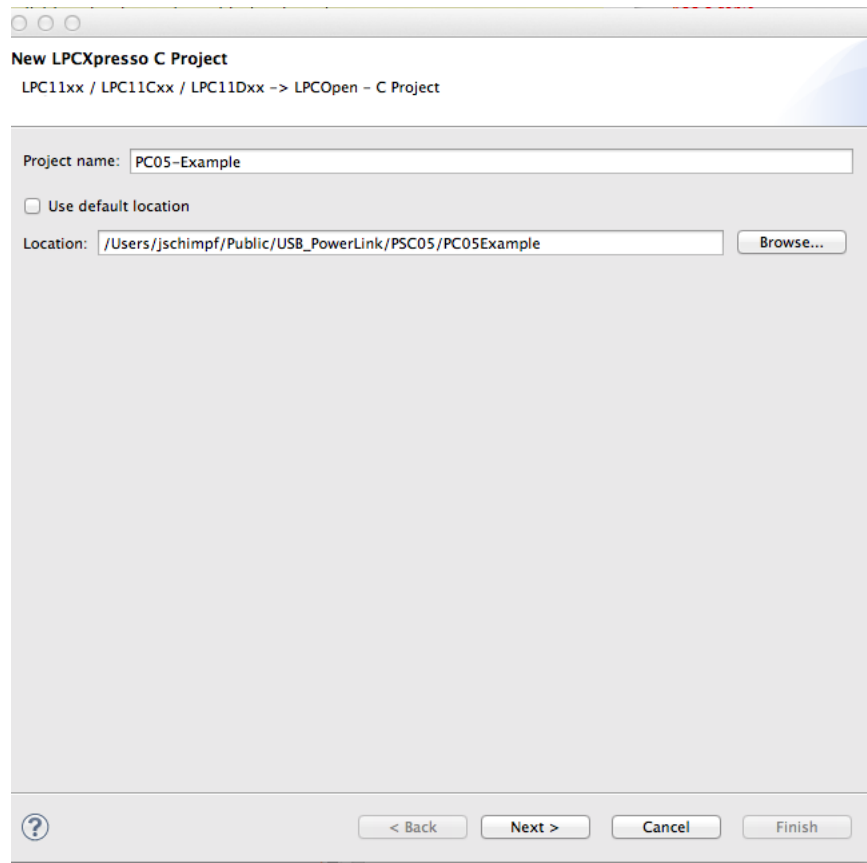
Figure 10: Project Folder & Name

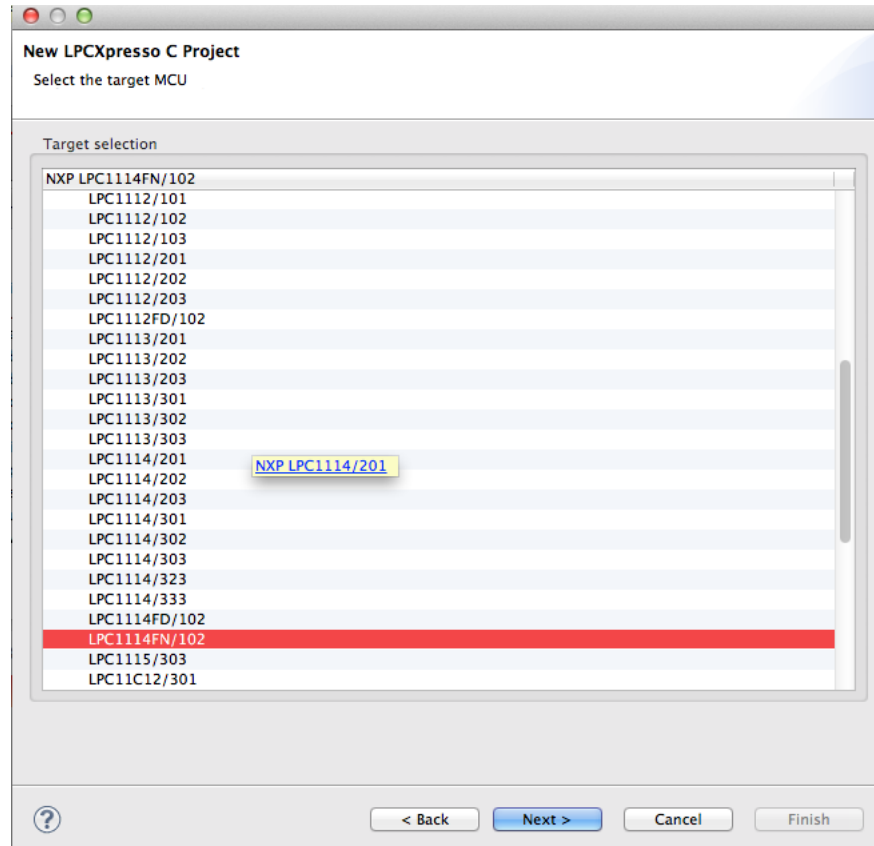Then we have to select the correct MCU for this project:

Figure 11: MCU Setting

Next you have to select the Board and Chip library for the project. These are in the Workspace already:
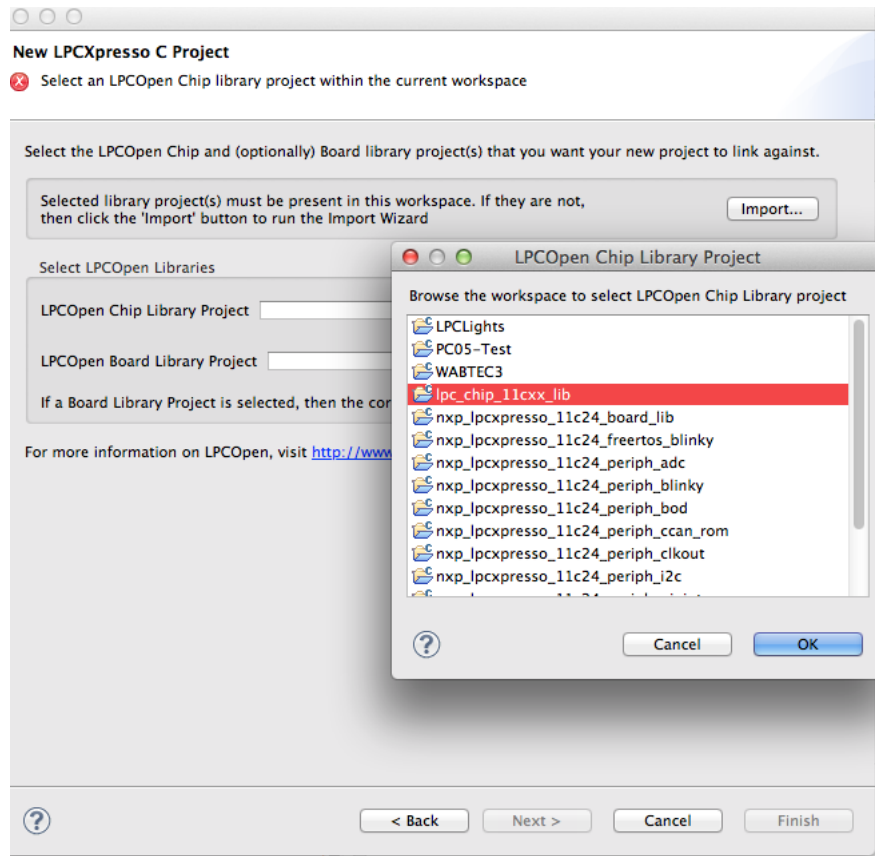
Figure 12: Selecting the Chip library

Selecting the board library is done the same way.

The next screen is for the CMSIS library and you can skip over this screen, no need to set it.

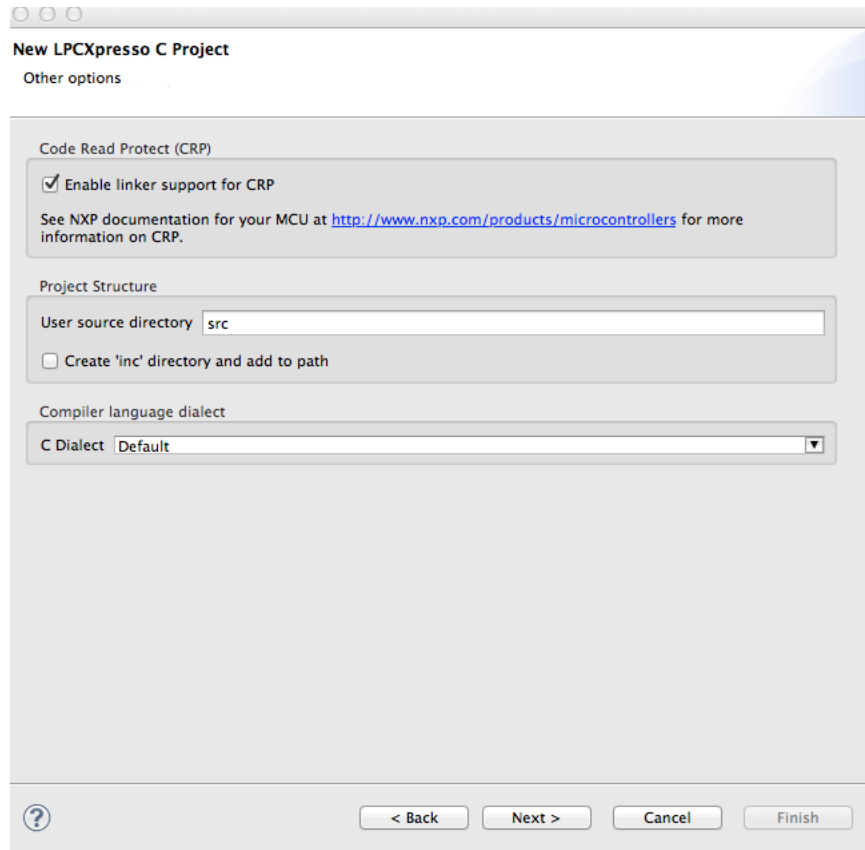The other options you can just take the defaults:

Figure 13: Other Options

Printf options. In this case we choose neither. The LPC1114 has very limited memory so we cannot affort it. We use the serial function instead of printf.
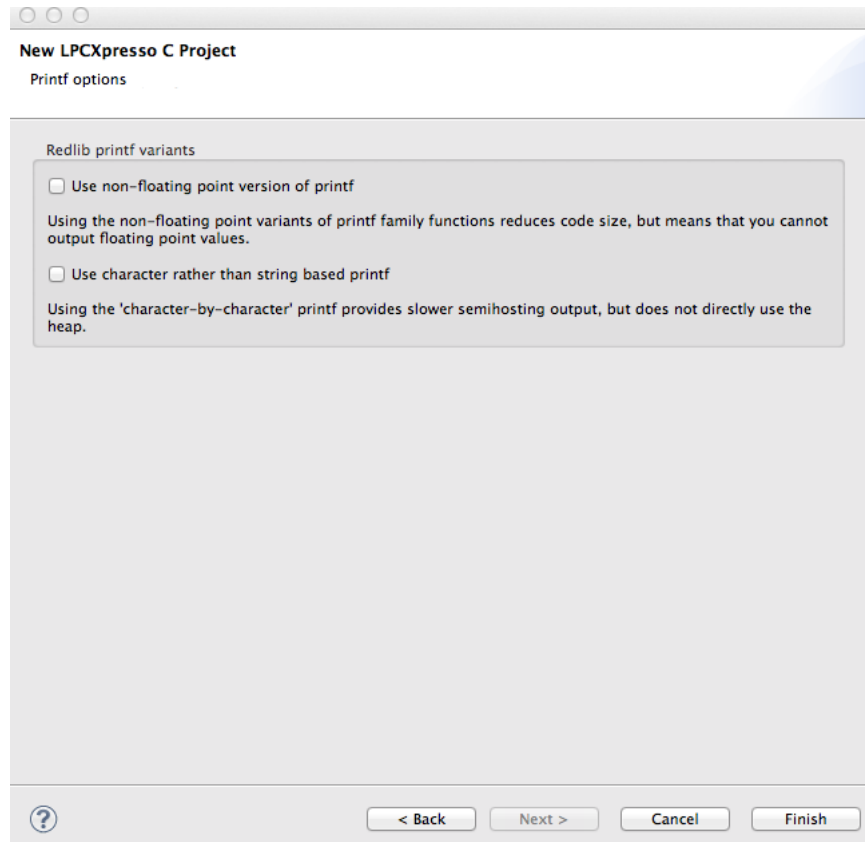
Figure 14: Printf Options

After this you now have your project with the src, FreeRTOS and Support directories present.
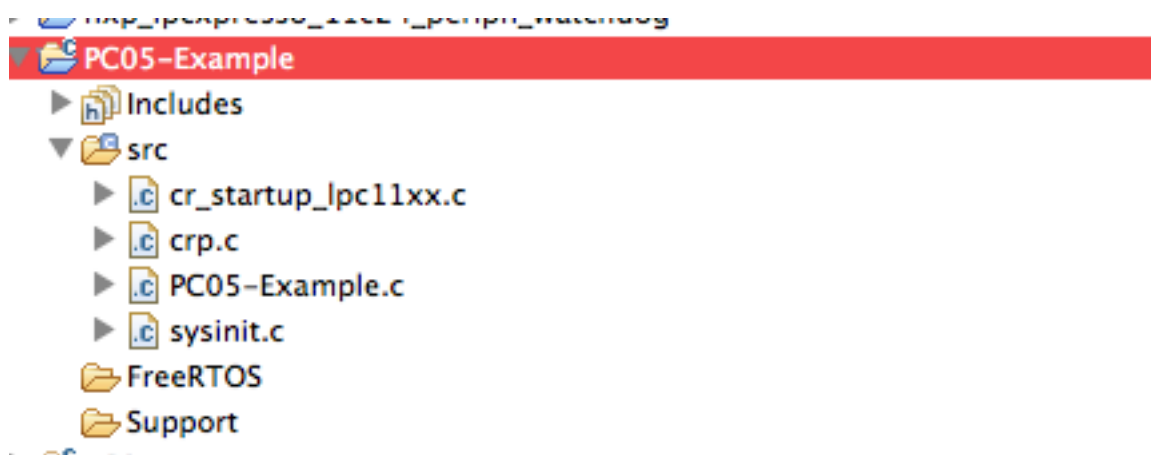

Figure 15: Created Project

8.4.2.2 **Adding Files** Now you can add the support files, this is done by copying the Support directory you made in 8.3 into the src direcory of the project (PCO5-Example/src). This will automatically add these files to the project.

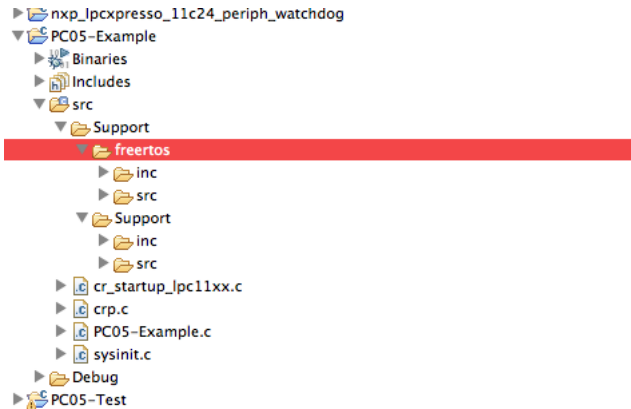If you do a build at this point the Support items will be added into the project:

Figure 16: Support Added

Finally you can import the files from the XCode project with File->Import->File System
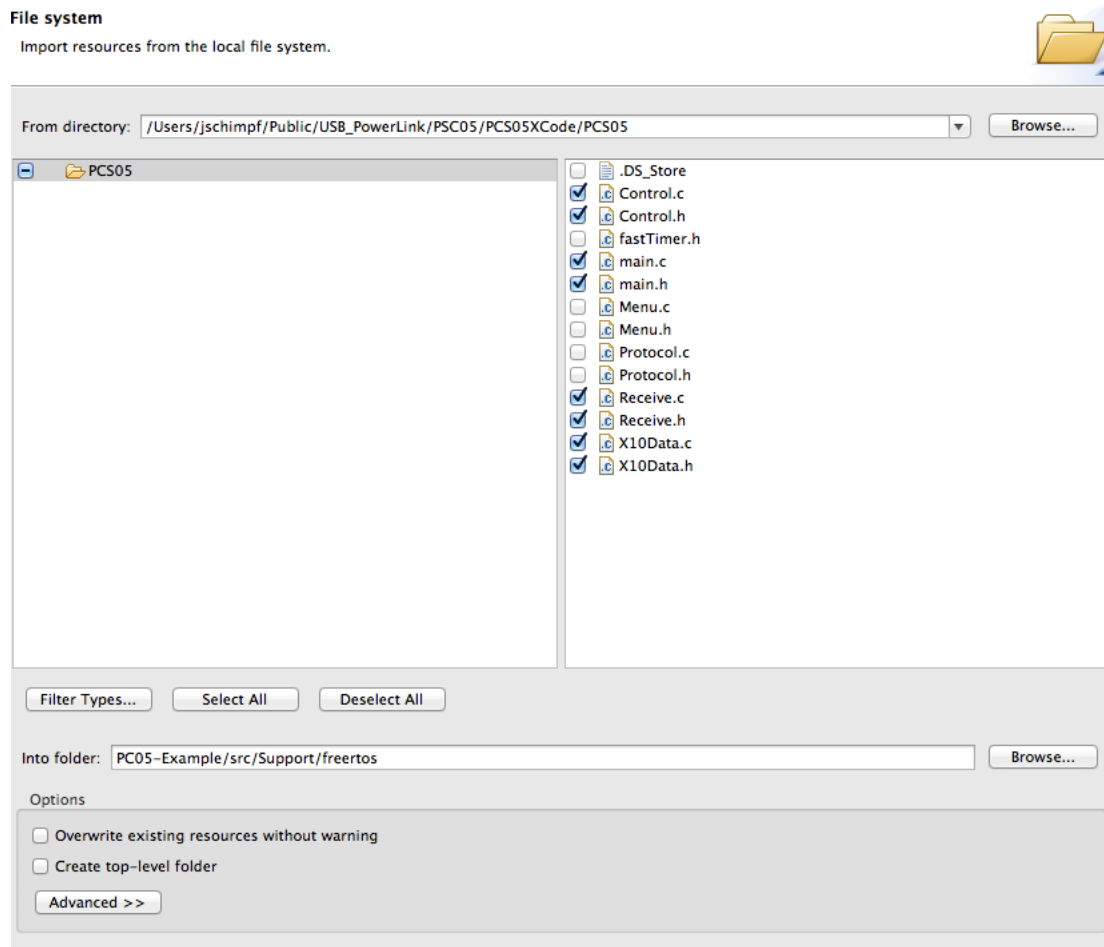
Figure 17: Import XCode Project Files

8.4.2.3   Deleting the project created MAIN   Before going on it will be necessary to delete the file **PC05-Example.c** as it was created when the project is made and has a main but in the XCode files there is already a main.c so this must be deleted, right click the file then pick **Delete**.

8.4.2.4   Setting Paths   The very last step is to set the paths to the include files by right clicking on the project and picking properties. Then Properties->Settings->Includes and use the tool to set the paths to **Support/Support/inc** and **Support/freertos/inc**.
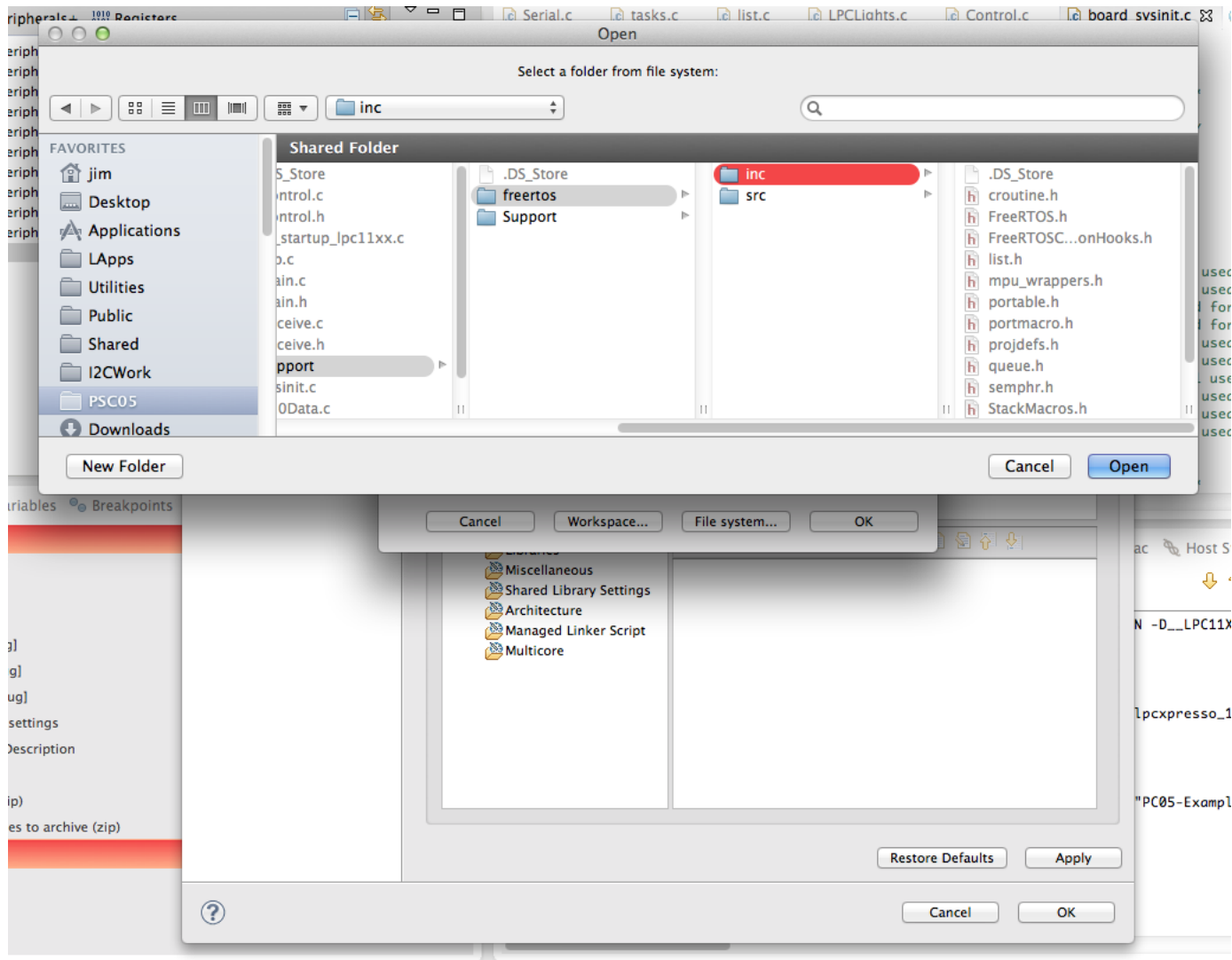
Figure 18: Setting Paths

8.4.2.5 Building and Running At this point the project should build and debugging with the real processor can be done.

9 Startup code - LPC support library

9.1 Introduction

As mentioned before there is a lot of stuff going on before we get to main() in our code. The NXP examples and support code does all of this for you. This next part will show you how that is done and how you can modify it so if you need specialized operations in your code it will be possible.

It's a little complicated but you don't have to do it all, just know where to put in modifications if you need then.

9.2   RESET - cr_startup_lpc11xx.c

The whole process begins with the RESET vector, when the chip is started. If you look in the file cr_startup_lpc11xx.c for ResetISR(). Each project has a cr_startup_lpc11xx.c file and in it is the vector table for the processor:

```
__attribute__ ((section(".isr_vector")))
void (* const g_pfnVectors[])(void) = {
    &_vStackTop,       // The initial stack pointer
    ResetISR,                          // The reset handler
    NMI_Handler,                       // The NMI handler
    HardFault_Handler,                 // The hard fault handler
```

The hardware in the ARM jumps to ResetISR when the RESET pin is pulled low or the processor is powered up.

The first part of ResetISR() copies the program data (i.e. variables whose value is set) from FLASH to RAM and sets up the RAM area. Next it calls the routine SystemInit() (more on that below) and when that is done it jumps to __main() which is part of the NXP preparation for main(). Finally if __main() ever returns (it shouldn't), there is a forever loop to catch that case, so the processor doesn't go wandering off in space.

9.3   SystemInit() - systinit.c

Right after ResetISR() has preped RAM this routine is called. It is in the file sysinit.c which is also a local program file. The file itself is very simple it is just a shell to call the routine Board_SystemInit().

9.4   Board_SystemInit() - board_sysinit.c

This call contains two routines SystemSetupClocking() and SystemSetupMuxing().

9.4.1   SystemSetupClocking()

This routine sets up clocking for the run. That is it determines if the crystal, external clock or internal clock will run the chip. It also sets up the division ratio that determines was speed the system will run. See 5.2 on page 8 section for how this is changed when we run with the internal RC clock.

9.4.2   SystemSetupMuxing()

This code sets up the pins on the chip for a default setup. This can be edited in a structure here. By default pins are GPIO and in here they can be set to specialized things. You don't have to set up all the pins here, you can move that part of the muxing into specialized routines like Serial.c or I2C.c but this is the default setup.

9.5   main() - <your code>

At this point it enters your code at main() and in there you may place the initialization for the parts of the chip you need

References

[1] Arduino. Arduino home page.

[2] NXP. *Getting started with NXP LPCXpresso*. NXP B V, 17 april 2013 edition, 2013.

[3] NXP. *UM10398 LPC111x/LPC11Cxx User manual*. NXP BV, rev. 12.3 edition, June 2014.