Getting Started with the LPC810

Created by Kevin Townsend

https://learn.adafruit.com/getting-started-with-the-lpc810

Last updated on 2025-05-21 02:34:14 PM EDT

Table of Contents

Introduction 3

Setting up an ARM Toolchain 4

- Downloading the LPCXpresso IDE
- Installing LPCXpresso

Importing a Project Into LPCXpresso 7

Blinky! 10

- Building Firmware in LPCXpresso

Programming the LPC810 with Flash Magic 13

- Programming the LPC810
- Hooking Everything Up
- Using Flash Magic
- Get Your .HEX File Ready
- Configuring Flash Magic for the LPC810
- Checking your Connection
- Programming the Device
- Testing Your Firmware
- Sample Hex Files

OK ... But why the LPC810 at Adafruit? 20

- 'Small is Beautiful'

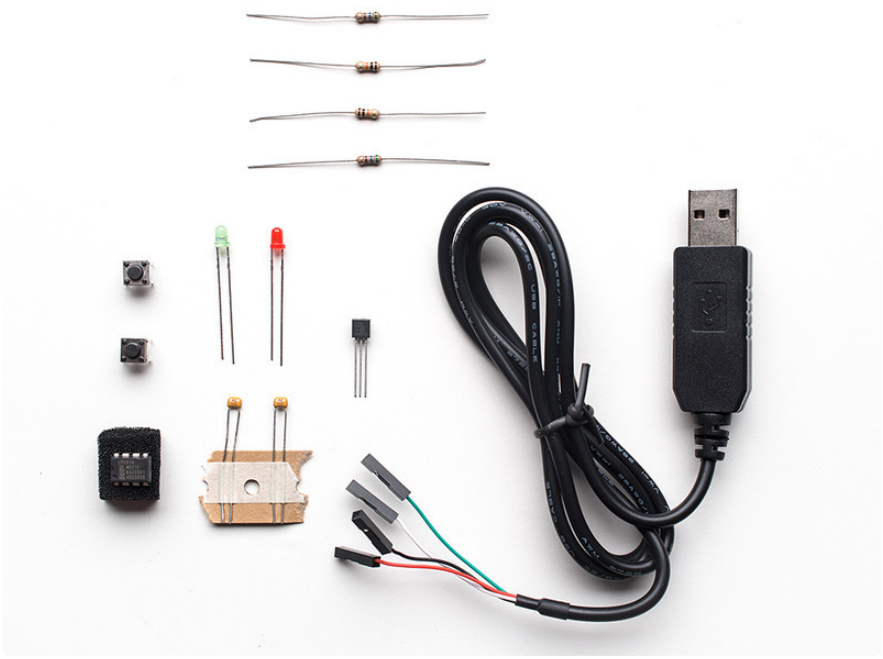
Introduction

The LPC810 starter pack from Adafruit has been discontinued. This guide is here for reference only and may contain conflicts with modern software or have other issues.

The LPC810 is a new microcontroller from NXP, a super fast little controller has only 8 pins but packs a lot of punch with a high speed processor and 32 bit instructions. This learning guide will show you everything you need to know to get started with the ARM Cortex M0+ based LPC810 microcontroller. It will cover:

- Setting up a cross-compiling toolchain for ARM
- Creating and compiling your first blinky program
- Programming the LPC810 using free and open source tools

This tutorial is designed to be followed using the Adafruit LPC810 Mini Starter Pack (http://adafru.it/1336)



Setting up an ARM Toolchain

While there are literally dozens of choices out there for compilers and IDEs for ARM, we're partial to GCC, which tends to have particularly good support for ARM chips in general.

While most seasoned veterans probably prefer a good old makefile, if you're new to ARM you'll appreciate the hand-holding an IDE can provide. We'll be using NXP's free LPCXpresso IDE in this tutorial, which is based on Eclipse and GCC, and works with all of their recent chips.

If you don't wish to use LPCXpresso, most of these steps also apply to a vanilla Eclipse + GCC installation, but you will have to do a bit more work yourself since LPCXpresso allows you to cut a lot of corners, generating the makefile and linker script on the go, etc. If there's enough interest, we'll also put together a tutorial on creating a makefile and building from the command line for this chip using only vanilla GCC.

Downloading the LPCXpresso IDE

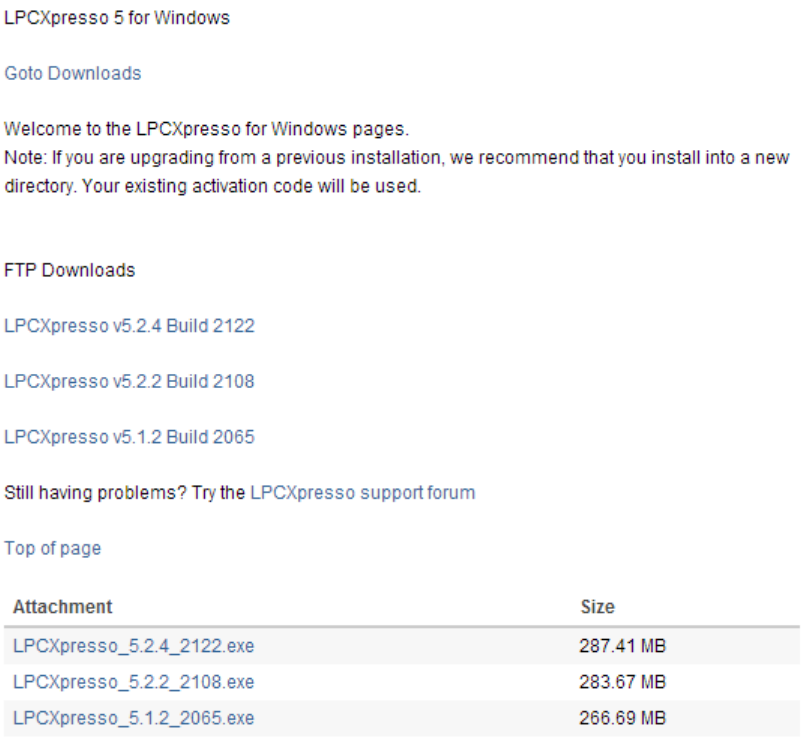
In order to download NXP's free LPCXpresso IDE, you do need to create an account on Code Red's website (who are now owned by NXP), at [https://www.lpcware.com/lpcxpresso/download](https://adafru.it/j0d) (https://adafru.it/j0d)

While registering sucks, LPCXpresso is still the easiest free tool using an open source toolchain, and you can easily break out of it once you're a bit more comfortable with

ARM and GCC, creating your own makefile and linker script, and LPCXpresso is just a bit of icing on top of the Eclipse + GCC cake.

An advantage of LPCXpresso is that it includes a relatively easy to use installer for Linux, Windows and Mac OS X, so you're free to use the toolchain and IDE on any major platform.

After registering, you should see a download list similar to the following:


LPCXpresso 5 for Windows

Goto Downloads

Welcome to the LPCXpresso for Windows pages.
Note: If you are upgrading from a previous installation, we recommend that you install into a new directory. Your existing activation code will be used.

FTP Downloads

LPCXpresso v5.2.4 Build 2122

LPCXpresso v5.2.2 Build 2108

LPCXpresso v5.1.2 Build 2065

Still having problems? Try the LPCXpresso support forum

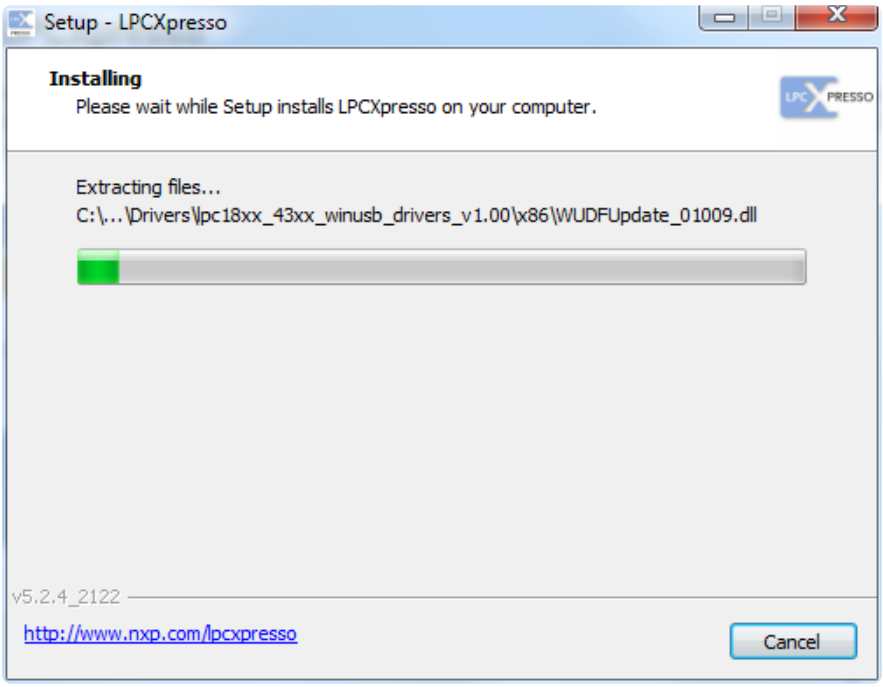
Top of page

Attachment	Size
LPCXpresso_5.2.4_2122.exe	287.41 MB
LPCXpresso_5.2.2_2108.exe	283.67 MB
LPCXpresso_5.1.2_2065.exe	266.69 MB

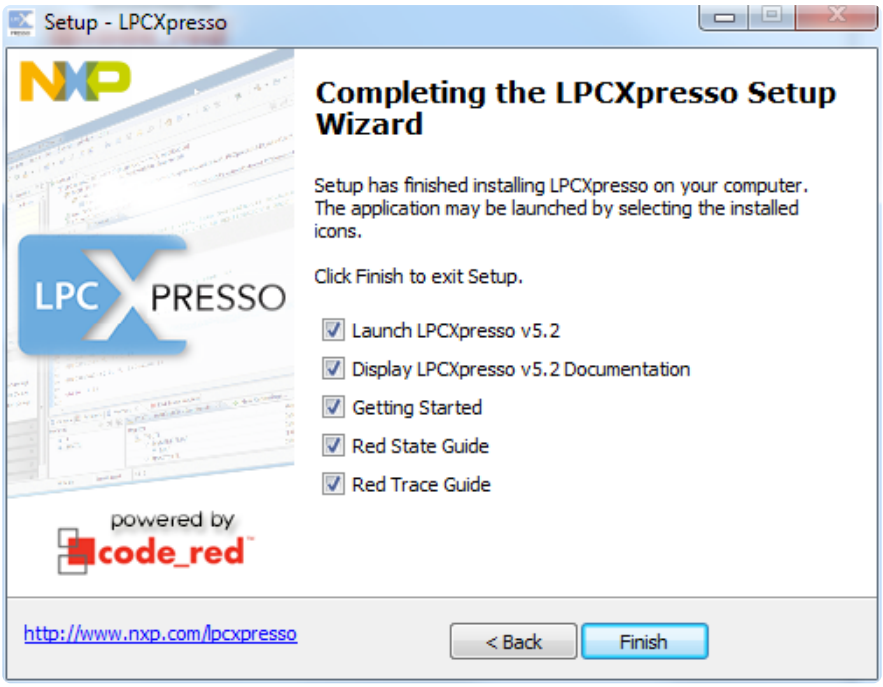
These are the latest downloads for Windows, but LPCXpresso is also available for LInux and MAC OS X.

Installing LPCXpresso

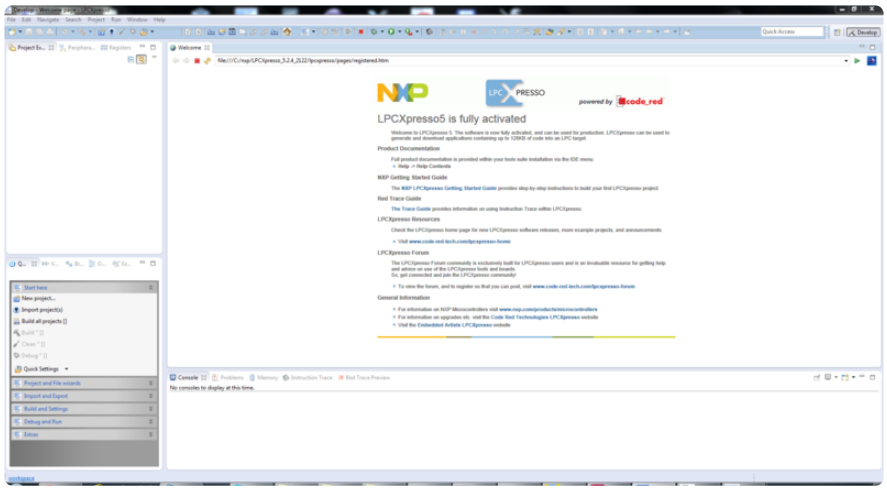
The installation process is relatively straight forward, and it should be fairly foolproof. Starting with the welcome screen, just run through the installer, and LPCXpresso will automatically setup all the tools, including the cross-compiling toolchain, Eclipse, etc.



At the end of the installation process you can optionally look at some documentation on how to use LPCXpresso by selecting or deselecting the checkboxes, but at this point you've installed everything you need to start writing programs for the LPC810 (or any other LPC chip from NXP)!



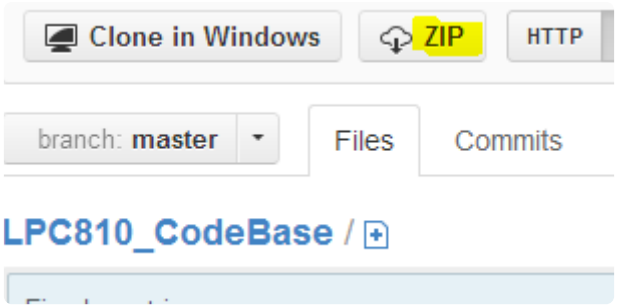
Next you can start the LPCXpresso IDE (you may need to activate it if this is the first time using it), and you'll get a screen similar to the following:



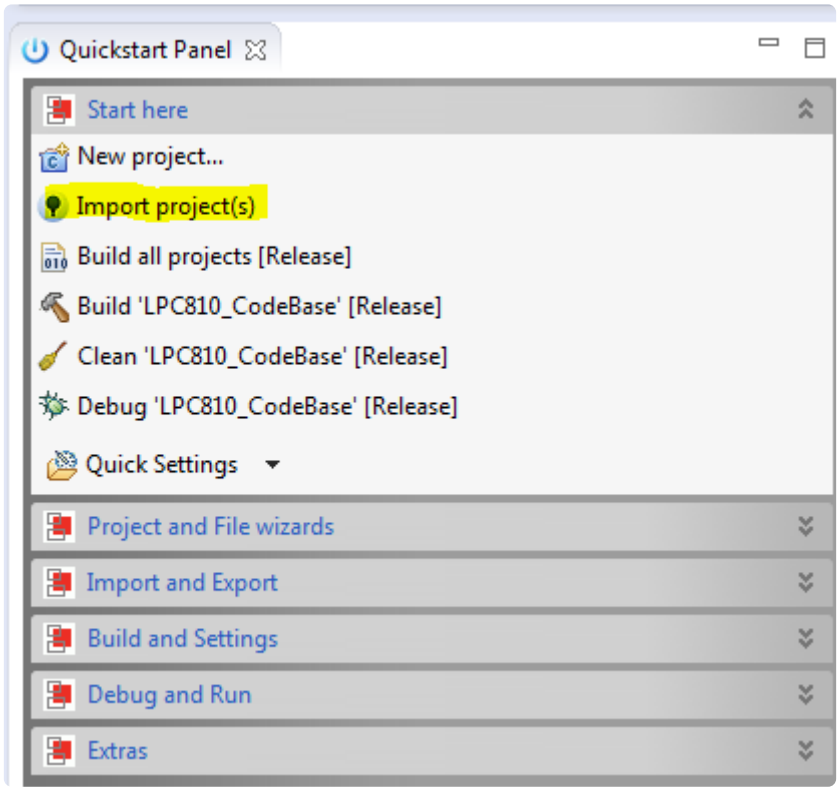
Importing a Project Into LPCXpresso

The easiest way to get started with the LPC810 is to use an existing project, where everything has already been setup for you in Eclipse. We'll use the code base available available here for this tutorial: https://github.com/microbuilder/LPC810_CodeBase (https://adafru.it/cbc)

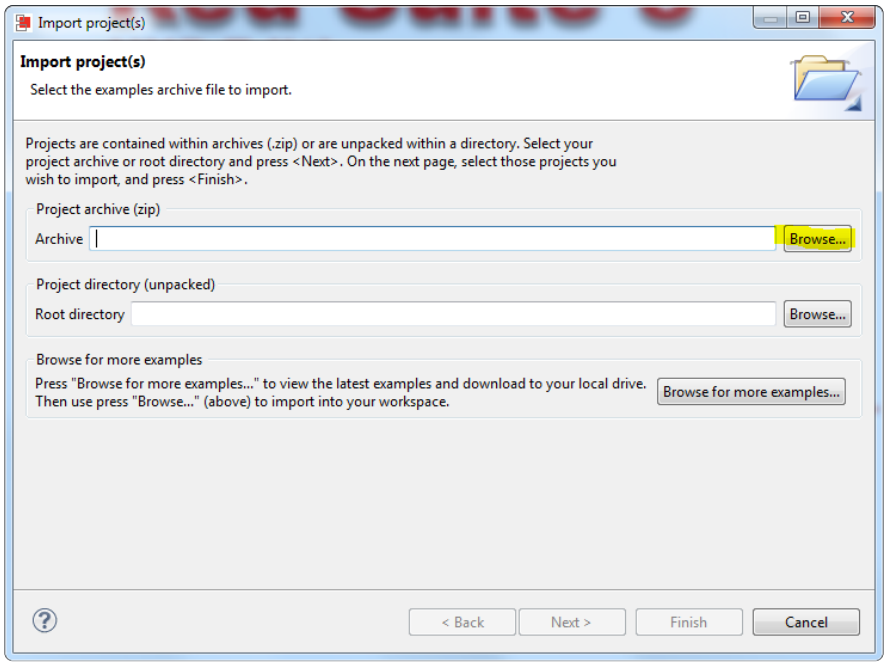
The first thing to do is download a .zip file containing all of the files in the project. You can do this by going to the github project page (https://adafru.it/cbc) and clicking the 'ZIP' button shown below:



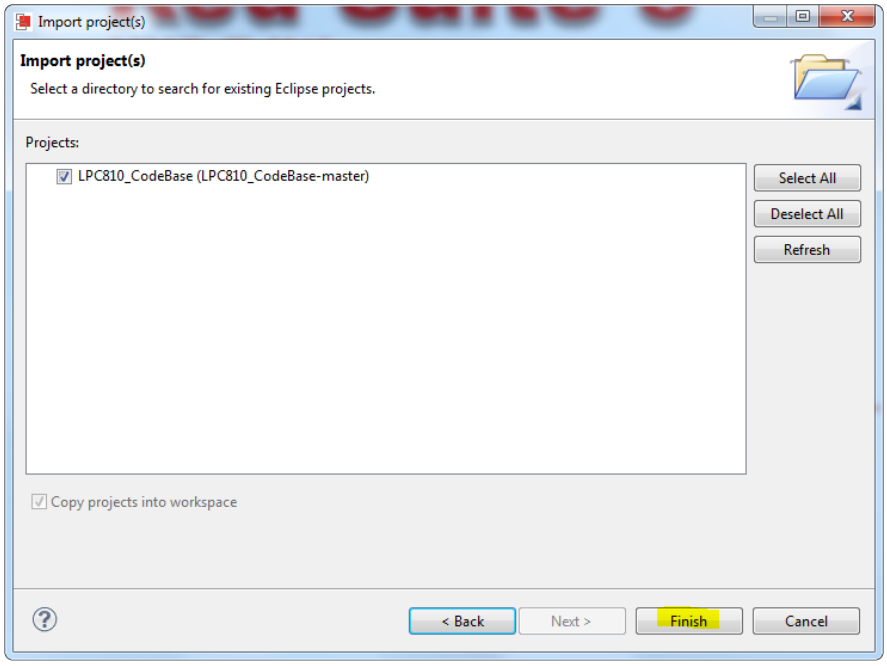
Once you have this .zip file available somewhere on your computer, open up the LPCXpresso IDE, and click the "Import Project(s)" button in the bottom left-hand side:



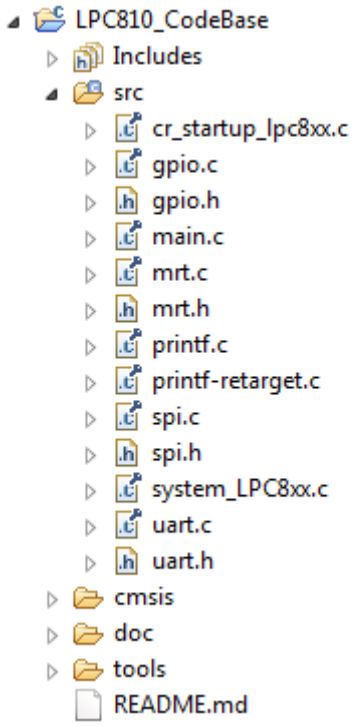
This will bring up a dialogue box, and you need to click the '**Browse...**' button under 'Project Archive (zip)', highlighted below:



Select your .zip file, and then click the **Next >** button, which will bring up the following dialogue:



Just click **Finish**, and your project will be imported into your workspace. You should see an entry in the project explorer on the left hand side named 'LPC810_CodeBase', shown below:



That's everything we need to know to start writing our own program with this chip!

Blinky!

Now that we have our project imported, we can get start creating our own projects.

By default, the LPC810_CodeBase you downloaded is setup to run a blinky example on pin 0.2, and it spits out 'Hello, LPC810!' on the default UART pins, which we can see in the 'while(1)' super-loop in main.c, shown below:

```
while(1)
{
  #if !defined(USE_SWD)
    /* Turn LED Off by setting the GPIO pin high */
    LPC_GPIO_PORT-&gt;SET0 = 1 &lt;&lt; LED_LOCATION;
    mrtDelay(500);

    /* Turn LED On by setting the GPIO pin low */
    LPC_GPIO_PORT-&gt;CLR0 = 1 &lt;&lt; LED_LOCATION;
    mrtDelay(500);
  #else
    /* Just insert a 1 second delay */
    mrtDelay(1000);
  #endif

  /* Send some text (printf is redirected to UART0) */
  printf("Hello, LPC810!\n");
}
```

"LPC_GPIO_PORT->CLR0 = 1 << LED_LOCATION" will clear whatever pin is at the 'LED_LOCATION' bit on GPIO bank 0. 'LPC_GPIO_PORT->SET0' does the opposite. This is how you turn the LED on or off, but for those changes to be visible, we need to

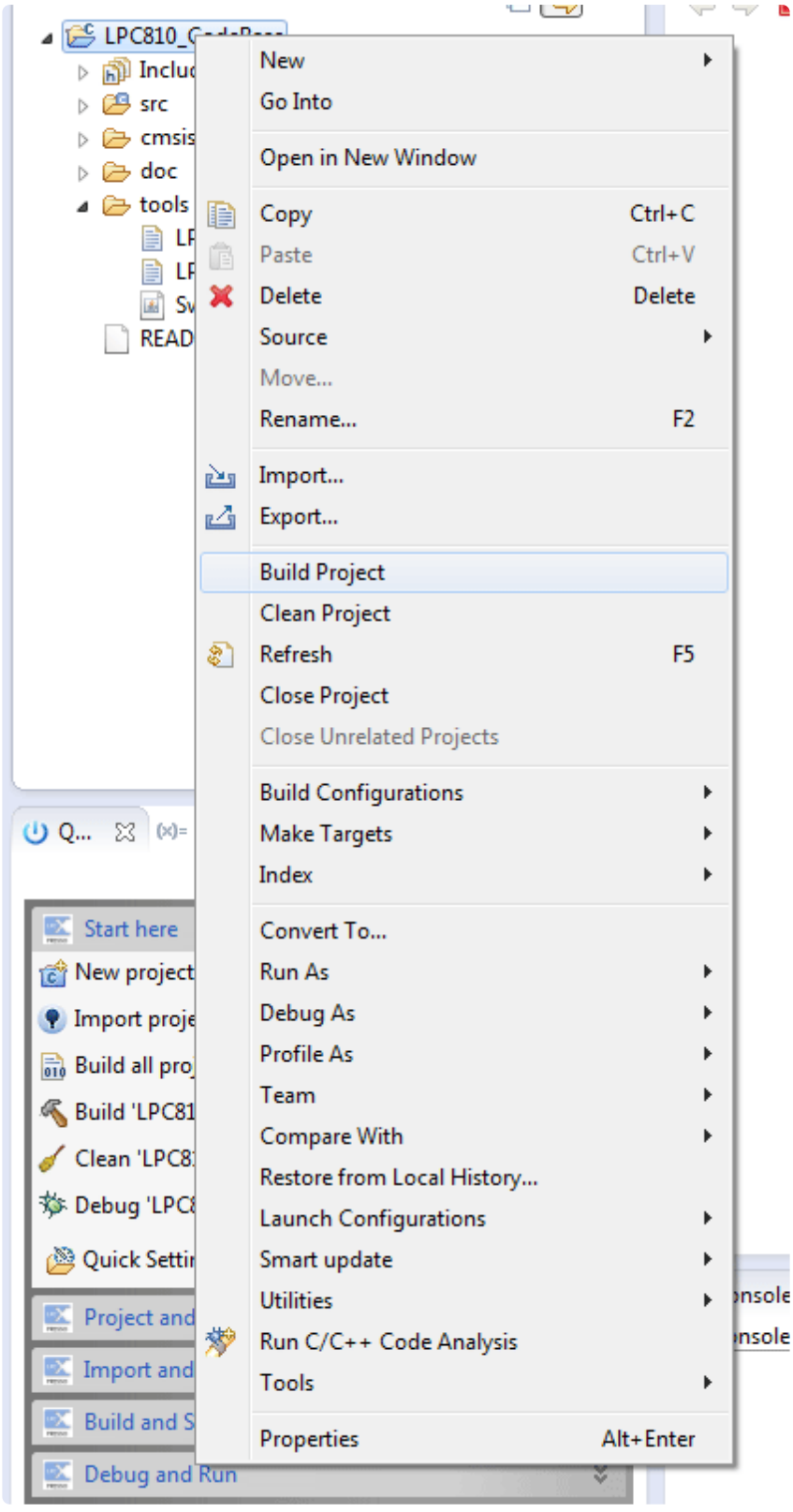
insert a delay between state changes, which is accomplished with the mrtDelay(500) lines, which causes a 500ms delay.

Blinky is following by our printf() statement, and by default all printf output is redirect to UART0 on the LPC810.

Before worrying about writing and programs, though, lets build the firmware as-is!

Building Firmware in LPCXpresso

To build a project in LPCXpresso (or Eclipse), you need to right-click on the current project in the project explorer (located on the left-hand side of the IDE by default), and select the '**Build Project**' menu item.



If everything goes well (and there are no errors in our code), you should get something like this in the output window at the bottom of LPCXpresso to indicate that the build process is complete:

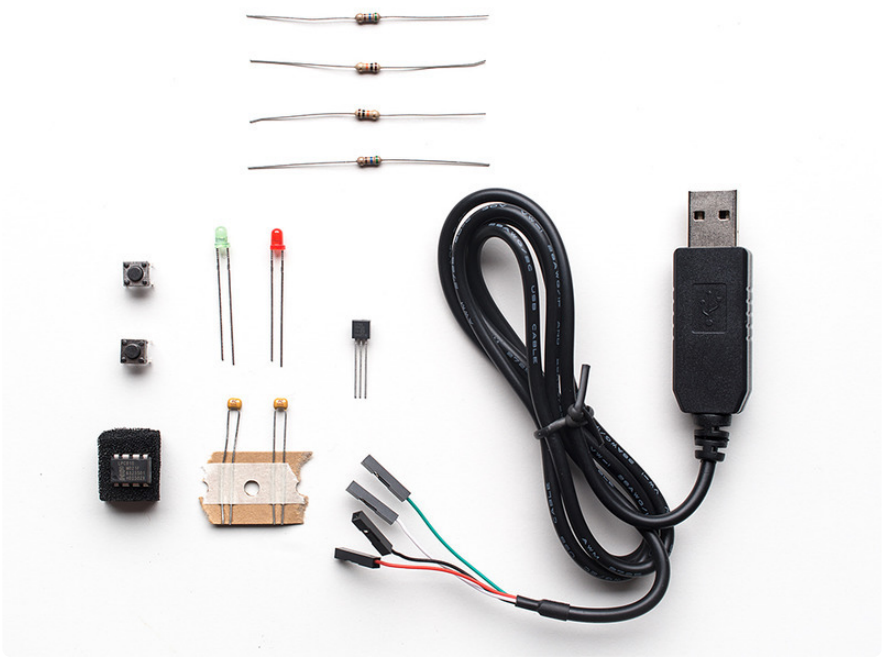
```
Console  Problems  Memory  Instruction Trace  Red Trace Preview
CDT Build Console [LPC810_CodeBase]
make --no-print-directory post-build
Performing post-build steps
arm-none-eabi-size "LPC810_CodeBase.axf"; arm-none-eabi-objcopy -O binary "LPC81
"LPC810_CodeBase.bin";
   text    data     bss     dec     hex filename
   2228       4       4    2236     8bc LPC810_CodeBase.axf
Created checksum 0xeffff949 at offset 0x1c in file LPC810_CodeBase.bin

00:27:37 Build Finished (took 2s.752ms)
```

We now have our 'blinky' firmware (located in the same folder as our project files), and can get it onto our LPC810 using the UART bootloader built into the chips ...

Programming the LPC810 with Flash Magic

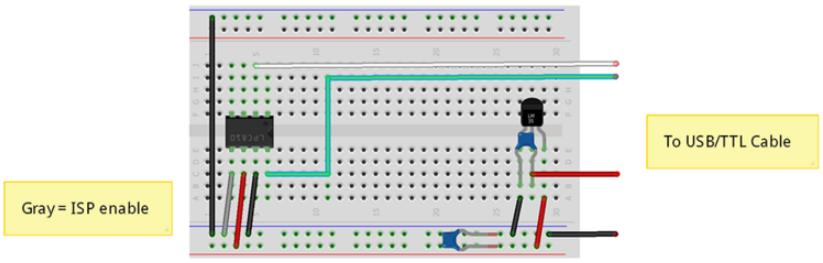
OK now is the time to pull out your Adafruit LPC810 Mini Starter Pack (http://adafru.it/1336)



Programming your LPC810 is relatively foolproof, and these chips are extremely hard to brick since they have a ROM-based bootloader and 'safety mode' that can always be executed before any bad firmware we might have programmed into the chip. We'll use this feature to program the MCUs with the firmware we built in the previous step.

But first ... a bit of wiring is in order! You'll need to hookup a UART to USB adapter, a 3V3 voltage regulator, a couple caps, and a few wires to get everything wired up safely with the LPC810. Just follow the wiring diagram below and you should be good.

The UART cable colors on the right (four wires) match the colors of Adafruit's USB to TTL Serial Cable (http://adafru.it/954).



Programming the LPC810

There are two main ways to take advantage of ISP mode to program your MCU:

- Use Flash Magic (https://adafru.it/cbd), a free GUI-based tool for Windows
- Use lpc2isp (https://adafru.it/cbf), an open source command-line tool that can be adapted to work on most operating systems, though you may need to build it yourself and add your MCU ID.

This tutorial will focus on Flash Magic as a starting point since it's the easiest tool to use, and since many Linux users should be able to figure out lpc2isp without too much effort.

Hooking Everything Up

This tutorial will assume that you are using the PL2303-based USB to TTL Serial Cable (http://adafru.it/954) included with the **LPC810 Starter Kit**.

1. Insert the LPC810 into your breadboard as shown, along with the 3.3V voltage regulator, and two decoupling capacitors.
2. Connect the white, green, red and black wires on the right-hand side of the diagram to the same colored cables on your USB to TTL serial adapter.
3. Connect the ISP pin to GND (the cable shown in gray above), which will cause the chip to enter ISP mode when it comes out of reset
4. Connect the other red and black cables on the breadboard which control the power supply to the MCU.

Note: Be sure to connect the ISP pin to GND before powering the board (with the red and black cables), since the LPC810 checks this pin as soon as it has power to device if it should enter ISP mode or not. When the ISP is at GND on boot, the chip enters ISP mode, when ISP is floating or logic high, th chip will boot normally, skipping ISP mode.

Using Flash Magic

If you haven't already done so, download and install the latest version of the free Flash Magic (https://adafru.it/cbd) tool.

The tool is relatively straight forward once you have used it once or twice, but this guide will show you everything you need to know to program the LPC810 using the Flash Magic GUI.

Get Your .HEX File Ready

Flash magic uses Intel Hex files to program the MCUs.

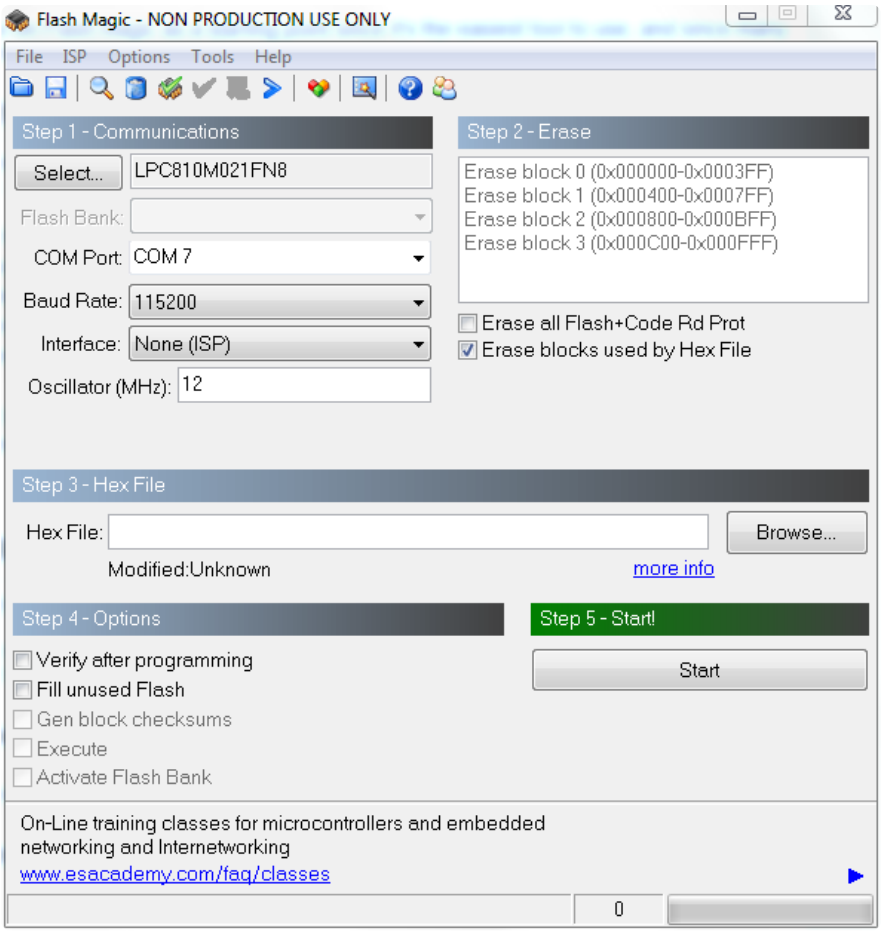
The LPC810 CodeBase (https://adafru.it/cbc) will produce a compatible .hex file from your own code, which we'll learn about later, but you can also find some sample .hex files at the bottom of this tutorial for convenience sake.

Configuring Flash Magic for the LPC810

Before you can program your LPC810, you need to supply some basic information to Flash Magic:

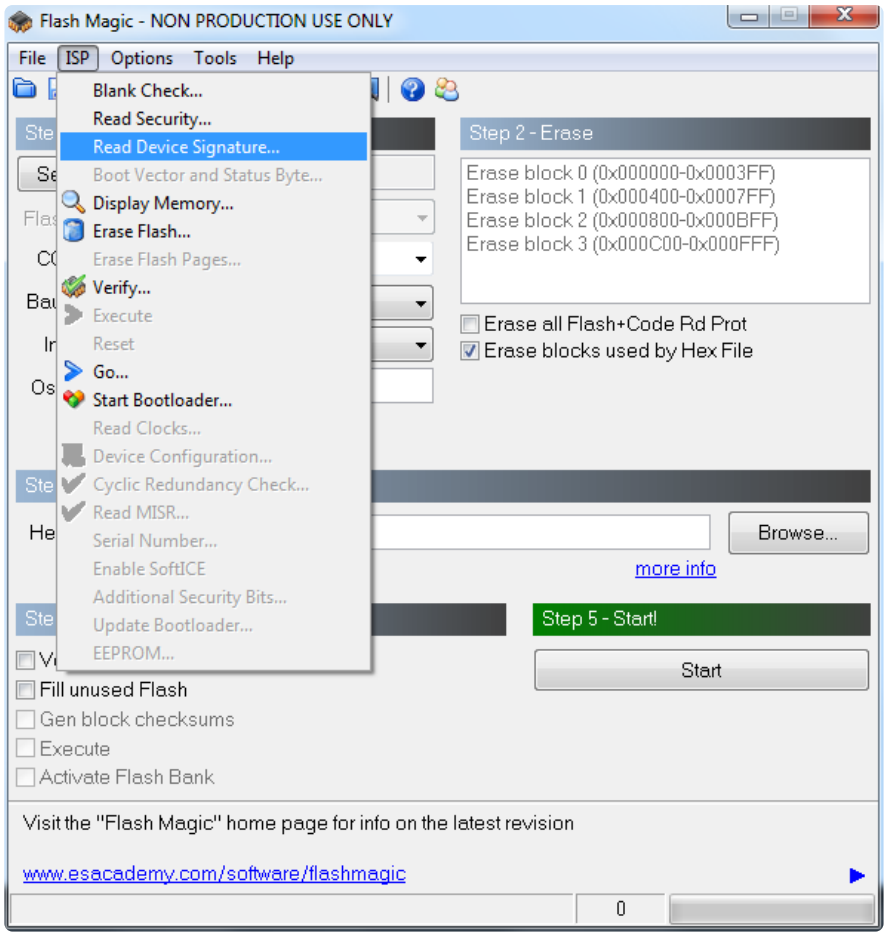
- Set the device to '**LPC810M021FN8**'
- Set the COM port to whatever COM port your USB to TTL cable is using (you can find this in the Device Manager on Windows, for example)
- Set the Baud Rate to '**115200**'
- Set the Oscillator to '**12**'
- Check the '**Erase blocks used by Hex File**' checkbox

Once these settings are entered (see the image below for a reference), you simply need to point to your .hex file by clicking on the '**Browse...**' button

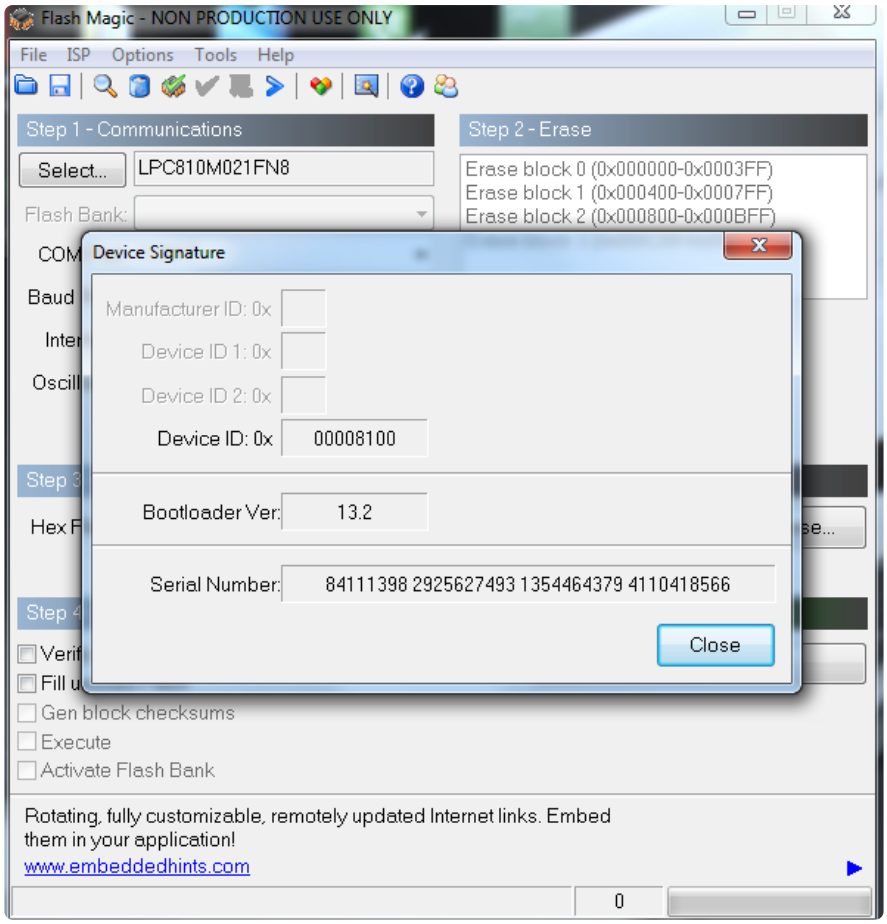


Checking your Connection

You can make sure that all of your settings are correct, and that everything is connected properly and that the chip is actually in ISP mode by selecting 'ISP > Read Device Signature ...' from the top menu bar:



If everything is OK, you should see something similar to this:

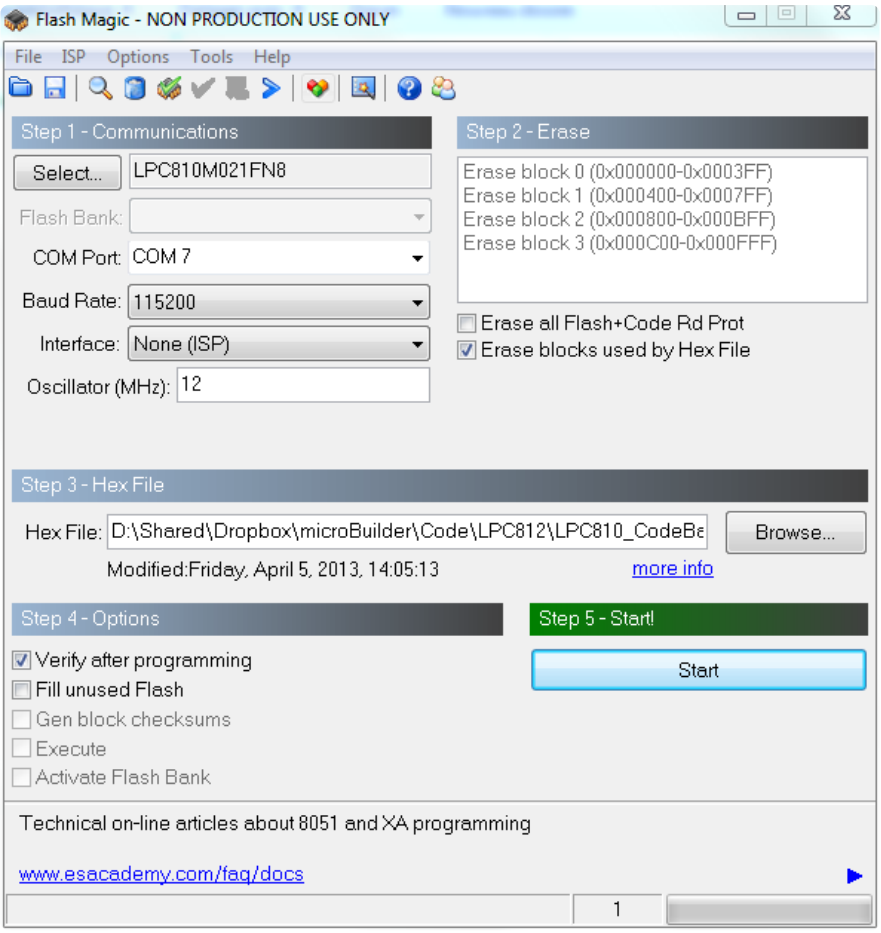


If you get an error message, check your COM port and the connections to your USB to TTL cable and ensure that the ISP pin is pulled low to GND before powering or resetting your board.

Programming the Device

After selecting your .hex file in the '**Hex File**' textbox, you simply need to click the **Start** button, and Flash Magic should start programming your device.

You may wish to check the '**Verify After Programming**' checkbox before doing this, but it isn't strictly necessary:



Testing Your Firmware

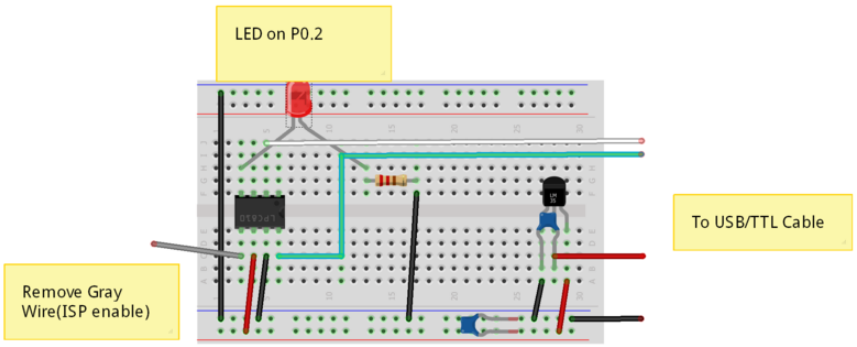
At this point, your device should be programmed. To test your firmware perform the following steps:

- Disconnect the ISP wire on your breadboard (the gray wire in the image above)
- Reset the device ... for example, disconnect then reconnect the red power cable on your USB to TTL adapter
- Once the board is powered up and if the ISP pin is not pulled low the chip will execute code normally

Sample Hex Files

For convenience sake, you can find some pre-compiled sample .hex files for the LPC810 in the '/tools' folder of the LPC810 code base (https://adafru.it/cbc), available on github:

- **LPC810_Blinky_P0_2.hex** - This hex file will cause an LED on P0.2 to blink at a regular rate. To use this firmware, connect the anode (the larger pin on your LED) to P0.2 on the LPC810, and the cathode (the shorter pin on the LED) to a 1K resistor and to GND, as seen in the following diagram:



OK ... But why the LPC810 at Adafruit?

I'll be honest ... I just thought this was an incredibly interesting chip. I've been using NXP's LPC series of MCUs since, oh, forever ... but this particular chip in DIP8 really jumped out at me since it's so different than what people usually think of when they hear 'ARM'.

The DIP8 LPC810 is still somewhat of a challenge to use precisely because it's so small (by ARM standards, anyway): 4KB flash and 1KB SRAM. That doesn't go far, and in reality you only have about 3KB flash to play with once you add in your startup code and get everything setup.

But that's actually part of what I found so fun about this chip! It's a genuine challenge but a fun one to do something creative and interesting with so much performance in such a small space!

'Small is Beautiful'

Warning: Shameless, unadulterated, highly-personal rant!

Really ... small is where it's at (https://adafru.it/cbh) if you care about writing clean, efficient code and really learning how to do something properly and understanding what you're doing!

I always start my new projects with the smallest chip I can, rather than the biggest one available. Why? Because it forces me to keep size in mind, to try to write better, more efficient code, and to get the most out of the limited resources I have. Why do I always start with ARM chips with 32KB flash and 8-12KB SRAM (https://adafru.it/cbj) when I could get something with 512KB flash and 100KB+ SRAM for not much more? Because if I started with those big chips I'd write even sloppier code that took more space than I needed, and I'd be stuck with bigger and more expensive chips and even worse code forever!

One of the biggest difficulties in deeply embedded systems is keeping things lean, without compromising on stability and reliability. That means error checking, data validation, etc., which takes space and time to implement, but you need to keep space in check, so it's an endless series of trade offs, decisions and optimisations.

Deeply embedded development has a lot more in common with mechanical watch-making than it does with traditional SW development: you're always trying to fit an impossible seeming number of gears and components into a ridiculously small package, and everything has to fit in in just the right manner to work at all. To throw some oil on the fire, it also needs to hold up to all manner of abuse and mistreatment that will get thrown at it, and keep smiling back at you day after day!

Something embedded SW engineers understand better than almost anyone else -- except maybe those crazy mechanical watch makers -- is the huge satisfaction in writing clean, concise, efficient code that solves real problems in concrete, elegant, reliable ways. Writing bloated code is easy ... writing elegant code and creating tiny solutions for big ambitions is harder, but also infinitely more satisfying!

Why the LPC810 with it's tiny package, and limited resources? Because it's a fun challenge and a really interesting way to learn ARM!

I really wanted to get this chip in people's hand just to see what people can do with 4KB flash and 1KB SRAM, and a reasonably limited set of peripherals. It's the kind of fun challenge that I enjoy working on, and I'm sure I'm not alone in that!