Learning to Code with Flybrix

Robb Walters, PhD
August 7, 2016
Version 0.1

Introduction

Did you know that the flight board included in your Flybrix kit is also a powerful computer that you can learn to program? You'll need to install a few extra tools from the internet to get started, but it shouldn't take you more than an hour to create and run your first program - even if you've never programmed before. If you do run into problems, just send us an email at support@flybrix.com and we'll help you sort it out. For those of you who are experts, we would welcome your help in improving the quality of the lessons we're sharing in this document - get in touch!

Let's be ambitious. Our goal for this document will be to guide a smart and motivated young person (that's you!) with no prior experience in either electronics or programming to an advanced level of understanding of how Flybrix works on both a hardware and software level. All you will need is your Flybrix kit, a personal computer, and an internet connection. If we're successful, you'll be able to imagine a new feature for Flybrix and add it to the source code yourself.

We're really excited for you! The rabbit hole is as deep as you want it to be.

Table of Contents

Chapter 1: From Zero to One Blinking LED

Introduction

This chapter guides you through the installation of the tools you'll need to reprogram your flight board and shows you how to write programs that interact with the indicator LEDs on your flightboard.

Setting up your development environment

The most important chip on your Flybrix flightboard is the microcontroller that runs the code that lets you fly. The microcontroller (MCU) is attached to the bottom of the flight board printed circuit board (PCB). It's the chip in the middle of the board that is about 10 mm on each side. If you read the markings on the top, you'll see the model number for this MCU (MK200X256) which you can search on the internet to discover as a 32-bit ARM-based chip made by NXP. These are all breadcrumbs for you to follow later when you're interested. For now, the important thing to know is that the brain of your flight board speaks the language of ARM processors. We're going to need to gather some tools together that will let you speak "ARM" too, so you can tell your MCU what you want it to do.

Installing the Arduino Development Environment

The Arduino project has brought together a fantastic community of people interested in using embedded systems (small cheap computers) to make cool stuff. It was an easy decision for us to design our products to be compatible with the Arduino ecosystem. The most important benefit of Flybrix being Arduino-compatible is that you'll be able to find a lot of information online if and when you get stuck. Arduino maintains an entire website worth of tutorials. You'll be able to run through these tutorials with your Flybrix board after you complete all the setup steps in this chapter. Let's get started:

1. Read about the Arduino Integrated Development Environment (IDE) here
2. Download the IDE Installer version 1.6.9 for your computer from here. This is not the most recent release! It takes awhile for us to verify that all of our code is compatible with the newest features from Arduino so we'll generally be a version or two behind them. (We encourage those of you who are financially able to support the Arduino project to do so when downloading their code, but note that there is also a free download option.)
3. Follow the installation instructions for your computer.

Installing the Teensyduino Libraries

For cost and simplicity reasons, the Arduino project is focused on lower performance microcontrollers than the brain on your Flybrix flightboard. We will need to install some additional software to use the Arduino IDE with the ARM processor core inside your MCU. One of the great things about the open source software movement is that people can build on the work of other talented programmers who have come before them. We hope you will aspire to join this global effort by sharing code of your own someday!

We are grateful for the significant effort that Paul Stoffregen and Robin Coon have put into creating the Teensyduino library that lets the Arduino IDE work with our MCU. Paul and Robin make and sell a tiny computer called the Teensy. When we first started developing Flybrix, we used the Teensy computer with external sensors and motor drivers. Because the Teensy is open source hardware, we were able to incorporate the parts of the Teensy electronic circuit that we wanted to keep into the circuit of the Flybrix flightboard. If you look on the top of your board, you'll find a 3x3mm chip labelled on the circuit board as "Teensy 3.2". This chip is a little harder to identify by its markings than the main MCU (see if you can read them with a magnifying glass!), but it is in fact a second microcontroller (also based on the ARM core). These chips are installed on your flight board with some proprietary code written by Paul pre-installed that allows the Flybrix flightboard to appear to your computer as a Teensy 3.2 board over the USB connection. (In principle you could reprogram this chip too, but it would be extremely hard to reach all the electrical connections you would have to contact for this purpose. We also wouldn't be able to help you restore Paul's software since we don't have a copy of it!) If you're curious, you can learn more about this chip on the "pjrc.com" website here. You can safely ignore all of this detail, but we thought you might want to know why your computer thinks a Teensy 3.2 is attached when you plug in the Flybrix flightboard.

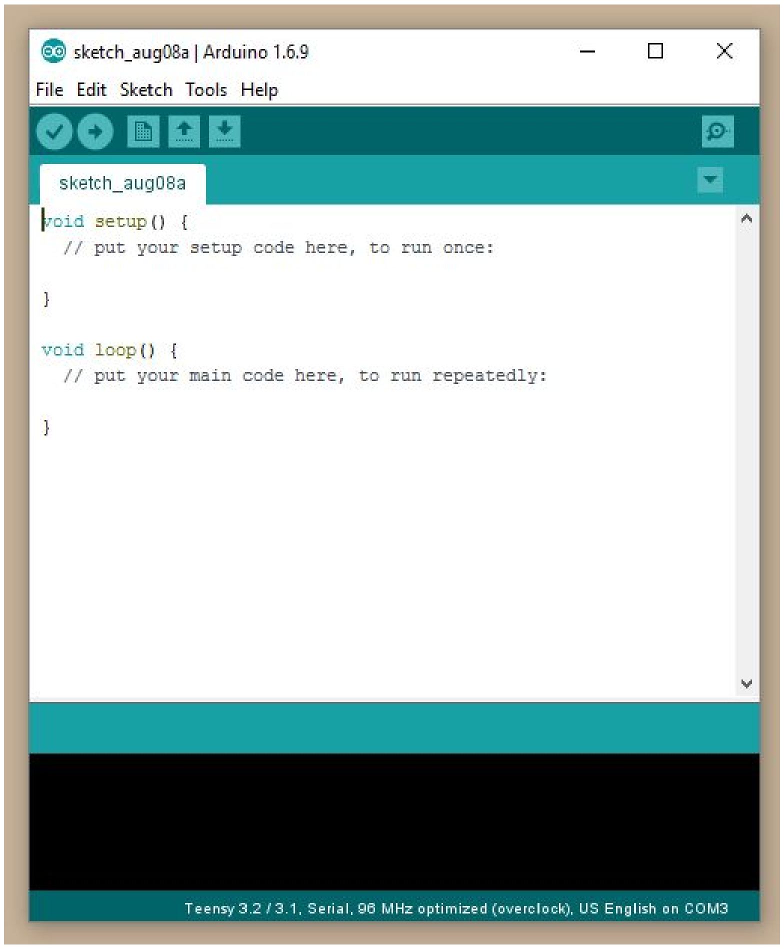
The Teensyduino libraries provide the software you need to install to use the Arduino IDE with the flightboard.

1. Read and follow the Teensyduino installation instructions here. You'll want to use "Teensyduino Version 1.29", which is matched to Arduino v1.6.9. Notice that the first step is to install the Arduino IDE, so you're already ahead of schedule!

Your First Program

We're going to write a very simple program using the Arduino IDE and upload it to your Flybrix flightboard. Be aware that when you upload new firmware to your flight board, it will overwrite the firmware that lets you fly your drones! But don't worry, as soon as you're done with a few experiments, we'll help you reinstall the Flybrix firmware.

1. Open up the Arduino IDE and create a New Project from the file menu. You should see a window that looks about like this:



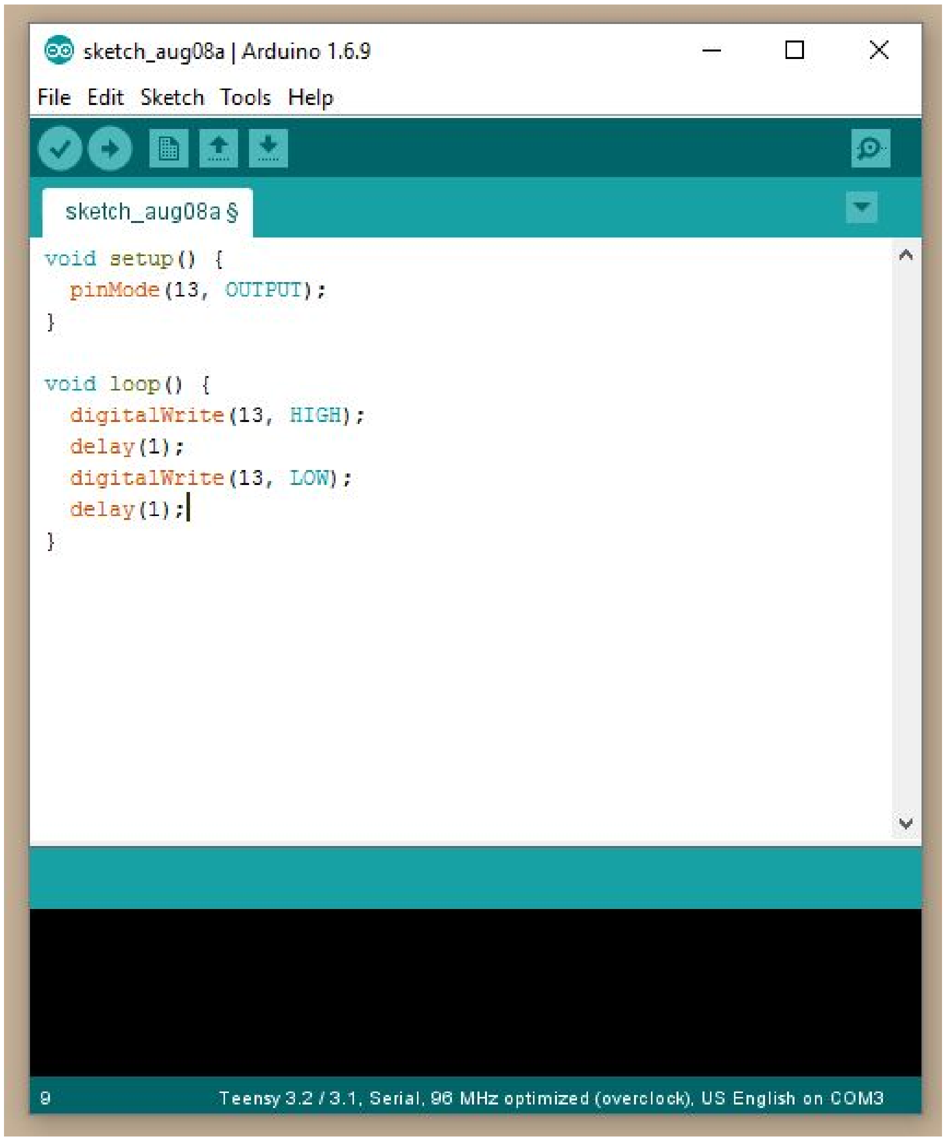
```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

```
Teensy 3.2 / 3.1, Serial, 96 MHz optimized (overclock), US English on COM3
```

If this is the first time you've seen code before, you might want to review some of the basic syntax. There are a number of good tutorials online (like this one) that will explain the basics of the "tokens" (symbols that have special meaning in the code). The file you see has two functions "setup" and "loop" and a couple of helpful comments that explain what each will do when we compile and run the code.

We're going to add some statements to make a green LED blink on the flightboard.

2. Type in the following code:

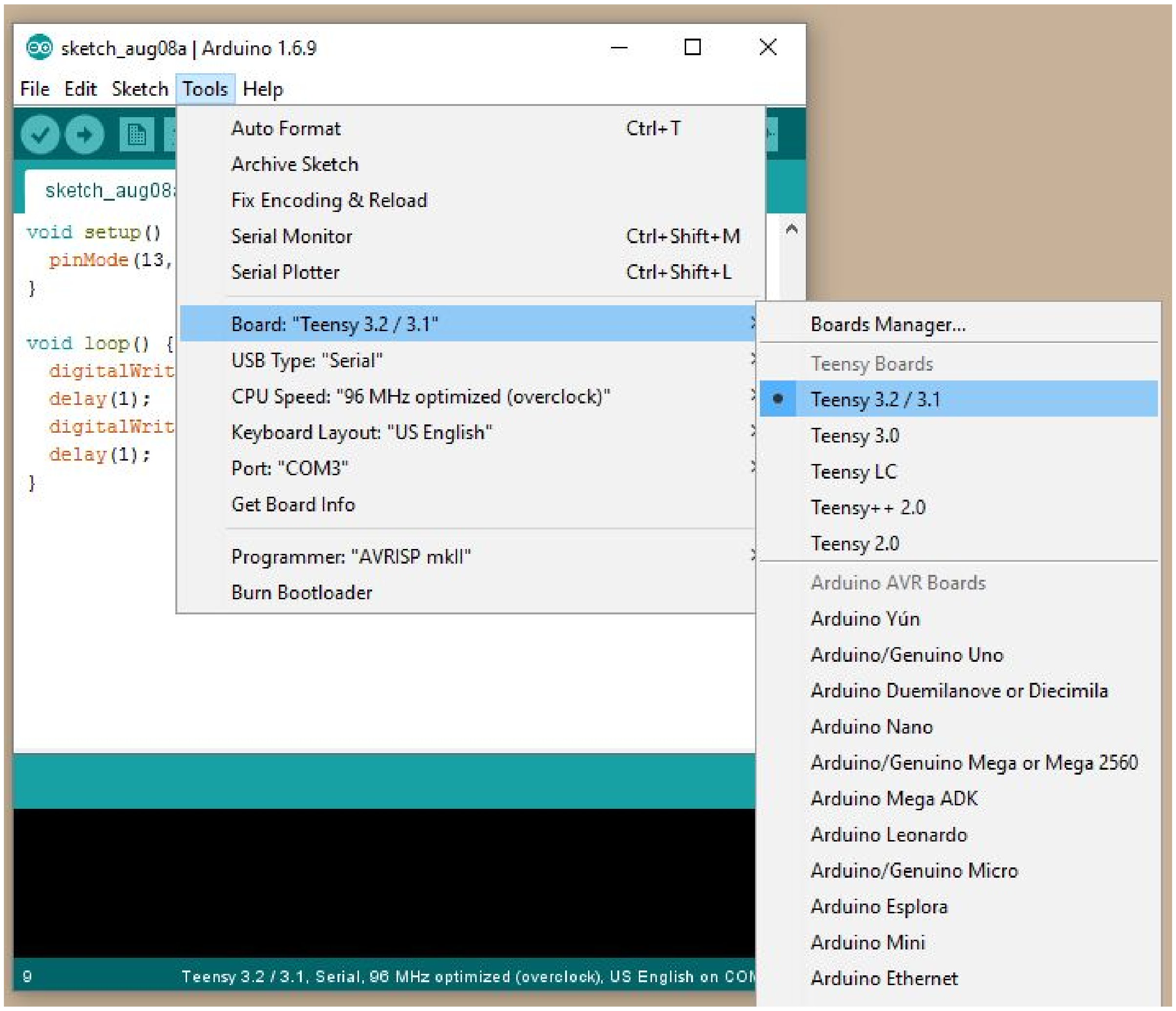


```
void setup() {
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, HIGH);
  delay(1);
  digitalWrite(13, LOW);
  delay(1);
}
```

```
9                Teensy 3.2 / 3.1, Serial, 96 MHz optimized (overclock), US English on COM3
```

We're now using three new functions: "pinMode", "delay", and "digitalWrite". We're also using the words "OUTPUT", "HIGH", and "LOW", and the number "13". We'll go over where these all come from in a moment, but let's go ahead and run this code now just to see what happens.

3. Tell the Arduino IDE that you're using a Teensy board:


sketch_aug08a | Arduino 1.6.9

File Edit Sketch Tools Help

```
void setup()
  pinMode(13,
}

void loop() {
  digitalWrit
  delay(1);
  digitalWrit
  delay(1);
}
```

Auto Format	Ctrl+T
Archive Sketch	
Fix Encoding & Reload	
Serial Monitor	Ctrl+Shift+M
Serial Plotter	Ctrl+Shift+L
Board: "Teensy 3.2 / 3.1"	
USB Type: "Serial"	
CPU Speed: "96 MHz optimized (overclock)"	
Keyboard Layout: "US English"	
Port: "COM3"	
Get Board Info	
Programmer: "AVRISP mkII"	
Burn Bootloader	

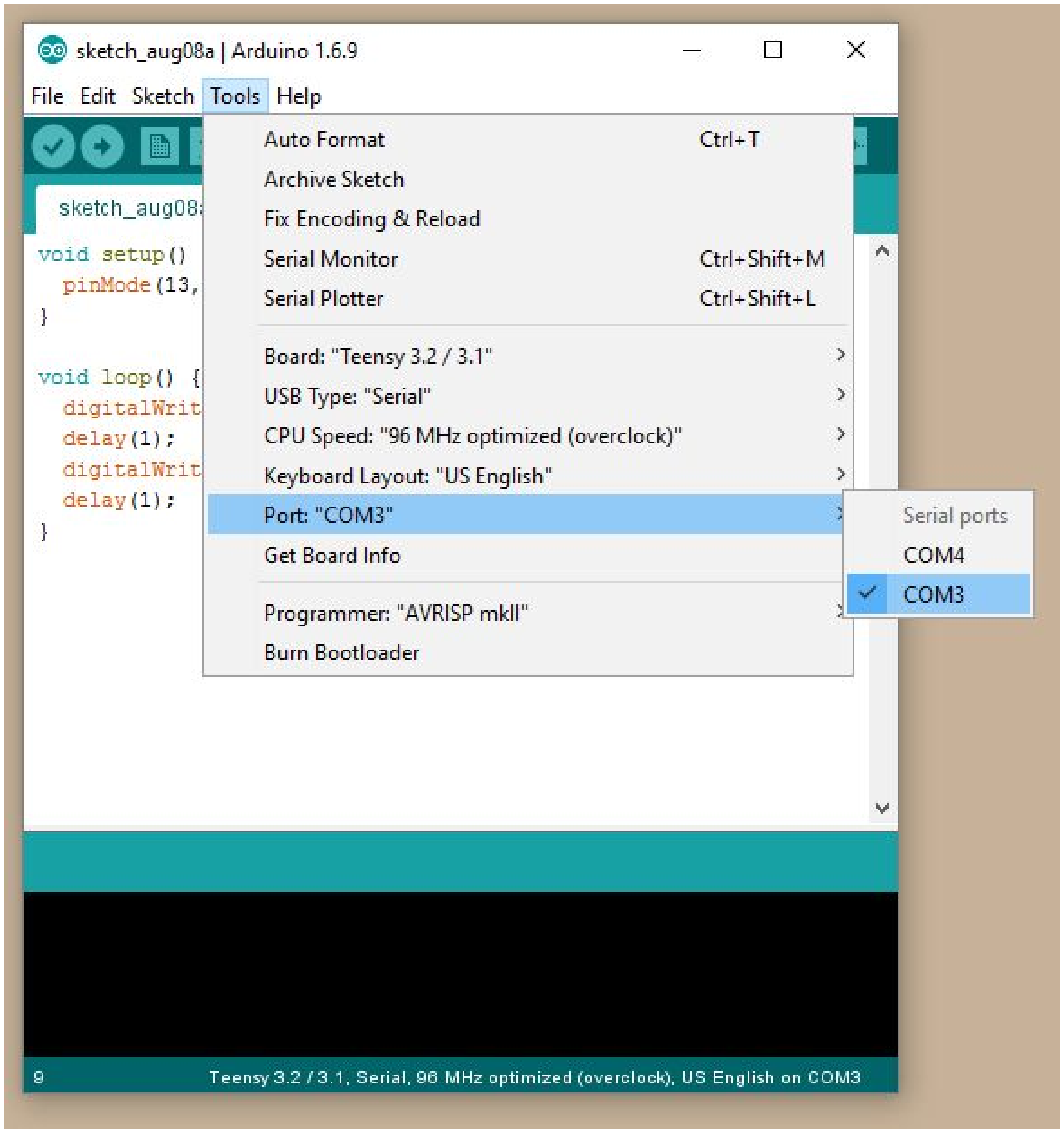
Boards Manager...

Teensy Boards
- ● Teensy 3.2 / 3.1
- Teensy 3.0
- Teensy LC
- Teensy++ 2.0
- Teensy 2.0

Arduino AVR Boards
- Arduino Yún
- Arduino/Genuino Uno
- Arduino Duemilanove or Diecimila
- Arduino Nano
- Arduino/Genuino Mega or Mega 2560
- Arduino Mega ADK
- Arduino Leonardo
- Arduino/Genuino Micro
- Arduino Esplora
- Arduino Mini
- Arduino Ethernet

```
9        Teensy 3.2 / 3.1, Serial, 96 MHz optimized (overclock), US English on COM
```

4. Plug a battery into your flight board and connect it to your computer with the USB cable.Tell the Arduino IDE where the flightboard is connected:



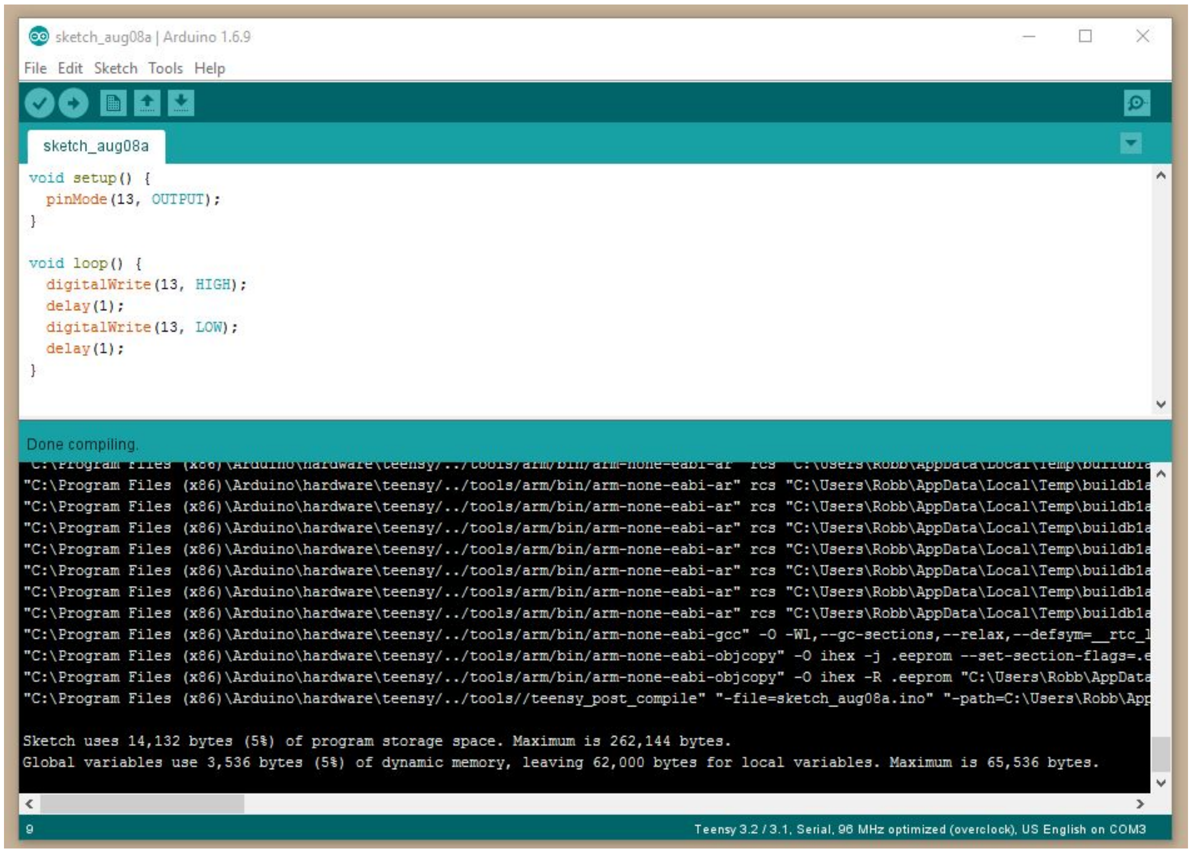
```
void setup()
  pinMode(13,
}

void loop() {
  digitalWrit
  delay(1);
  digitalWrit
  delay(1);
}
```

Board: "Teensy 3.2 / 3.1"
USB Type: "Serial"
CPU Speed: "96 MHz optimized (overclock)"
Keyboard Layout: "US English"
Port: "COM3"
Programmer: "AVRISP mkII"

```
9        Teensy 3.2 / 3.1, Serial, 96 MHz optimized (overclock), US English on COM3
```

5. Click the circle with the check mark to compile the code (“Verify”). You will probably be prompted to save the file somewhere on your computer. You might also see another program open automatically in the background called “Teensy Loader”, which you can leave open and ignore for now. Note that Arduino files are called “sketches” and end with the file extension “.ino”. The Arduino IDE also insists that each sketch is stored inside of a folder with the same name.

You will see a bunch of text appear in the black box at the bottom of the window. This is the output that is generated by the “Compiler” as it runs in the background. The compiler is a program that translates the human readable code we’re writing into “machine code” that can be uploaded to the microprocessor. Make the window a little wider so you can see the output more clearly:



```
void setup() {
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, HIGH);
  delay(1);
  digitalWrite(13, LOW);
  delay(1);
}
```

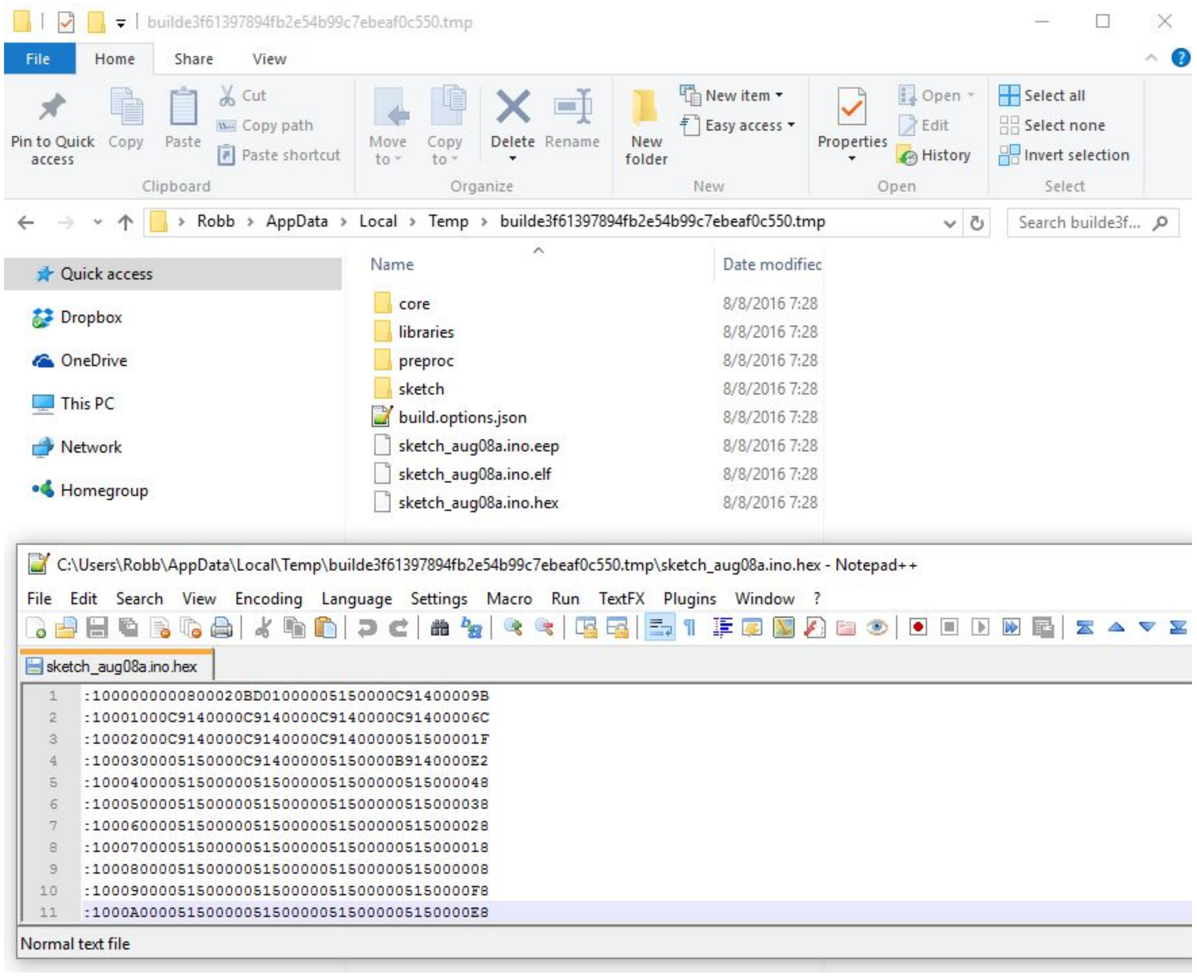
Done compiling.

```
"C:\Program Files (x86)\Arduino\hardware\teensy/../tools/arm/bin/arm-none-eabi-ar" rcs "C:\Users\Robb\AppData\Local\Temp\buildb1a
"C:\Program Files (x86)\Arduino\hardware\teensy/../tools/arm/bin/arm-none-eabi-ar" rcs "C:\Users\Robb\AppData\Local\Temp\buildb1a
"C:\Program Files (x86)\Arduino\hardware\teensy/../tools/arm/bin/arm-none-eabi-ar" rcs "C:\Users\Robb\AppData\Local\Temp\buildb1a
"C:\Program Files (x86)\Arduino\hardware\teensy/../tools/arm/bin/arm-none-eabi-ar" rcs "C:\Users\Robb\AppData\Local\Temp\buildb1a
"C:\Program Files (x86)\Arduino\hardware\teensy/../tools/arm/bin/arm-none-eabi-ar" rcs "C:\Users\Robb\AppData\Local\Temp\buildb1a
"C:\Program Files (x86)\Arduino\hardware\teensy/../tools/arm/bin/arm-none-eabi-ar" rcs "C:\Users\Robb\AppData\Local\Temp\buildb1a
"C:\Program Files (x86)\Arduino\hardware\teensy/../tools/arm/bin/arm-none-eabi-ar" rcs "C:\Users\Robb\AppData\Local\Temp\buildb1a
"C:\Program Files (x86)\Arduino\hardware\teensy/../tools/arm/bin/arm-none-eabi-ar" rcs "C:\Users\Robb\AppData\Local\Temp\buildb1a
"C:\Program Files (x86)\Arduino\hardware\teensy/../tools/arm/bin/arm-none-eabi-gcc" -O -Wl,--gc-sections,--relax,--defsym=__rtc_l
"C:\Program Files (x86)\Arduino\hardware\teensy/../tools/arm/bin/arm-none-eabi-objcopy" -O ihex -j .eeprom --set-section-flags=.e
"C:\Program Files (x86)\Arduino\hardware\teensy/../tools/arm/bin/arm-none-eabi-objcopy" -O ihex -R .eeprom "C:\Users\Robb\AppData
"C:\Program Files (x86)\Arduino\hardware\teensy/../tools//teensy_post_compile" "-file=sketch_aug08a.ino" "-path=C:\Users\Robb\App

Sketch uses 14,132 bytes (5%) of program storage space. Maximum is 262,144 bytes.
Global variables use 3,536 bytes (5%) of dynamic memory, leaving 62,000 bytes for local variables. Maximum is 65,536 bytes.
```

9 Teensy 3.2 / 3.1, Serial, 96 MHz optimized (overclock), US English on COM3

There's a lot of information in the output of the compiler and we're going to pass over most of it for now. The last few lines tell us where to find the machine code (the "-path=C:\Users…" part in the third to last line), how big our compiled machine code turned out to be (14132 bytes = 113056 zeros and ones) and how that compares to the largest program we can run on the flight board, and how much of the "dynamic memory" we're using up. It's interesting to note that our simple program was only about 130 characters long -- meaning that it would take up about 130 bytes to store on your computer - so it got about 870 times bigger when we compiled it into machine code! The compiler does a lot of work for you in the background.

It's kind of fun to dig up the compiled source code file to see what it looks like. In the third to last line, you can find the path that the output file was generated at. In this example, the path was "C:\Users\Robb\AppData\Local\Temp\builde3f61397894fb2e54b99c7ebeaf0c550.tmp". Let's take a look inside the compiled ".hex" file that we see there:



builde3f61397894fb2e54b99c7ebeaf0c550.tmp

Robb › AppData › Local › Temp › builde3f61397894fb2e54b99c7ebeaf0c550.tmp

Name	Date modified
core	8/8/2016 7:28
libraries	8/8/2016 7:28
preproc	8/8/2016 7:28
sketch	8/8/2016 7:28
build.options.json	8/8/2016 7:28
sketch_aug08a.ino.eep	8/8/2016 7:28
sketch_aug08a.ino.elf	8/8/2016 7:28
sketch_aug08a.ino.hex	8/8/2016 7:28

C:\Users\Robb\AppData\Local\Temp\builde3f61397894fb2e54b99c7ebeaf0c550.tmp\sketch_aug08a.ino.hex - Notepad++

sketch_aug08a.ino.hex

```
:100000000C800020BD0100000515000C91400009B
:10001000C9140000C9140000C9140000C91400006C
:10002000C9140000C9140000C91400000051500001F
:10003000005150000C9140000051500000B9140000E2
:100040000051500000515000005150000051500048
:100050000051500000515000005150000051500038
:100060000051500000515000005150000051500028
:100070000051500000515000005150000051500018
:100080000051500000515000005150000051500008
:1000900005150000051500000515000005150000F8
:1000A000051500000515000005150000051500000E8
```

Neat, right? This is the file that we will copy over to the memory inside the microcontroller on your flight board to run the program we compiled.

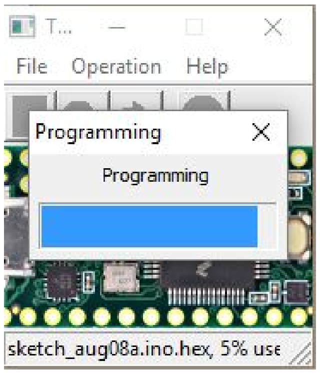
6. Next to the “Verify” button, you should see a circle with an arrow in it (“Upload”). Press it to recompile the code again and upload the result directly to the flightboard.

You might get a message in the compiler output area like this:

> Teensy did not respond to a USB-based request to automatically reboot.
> Please press the PROGRAM MODE BUTTON on your Teensy to upload your sketch.

If you do, go ahead and press the small program mode button. You’ll find it on the top of the flight board next to the USB connector, marked “program” on the circuit board silkscreen layer.

You should see the “Teensy Loader” application pload the code to the flightboard:

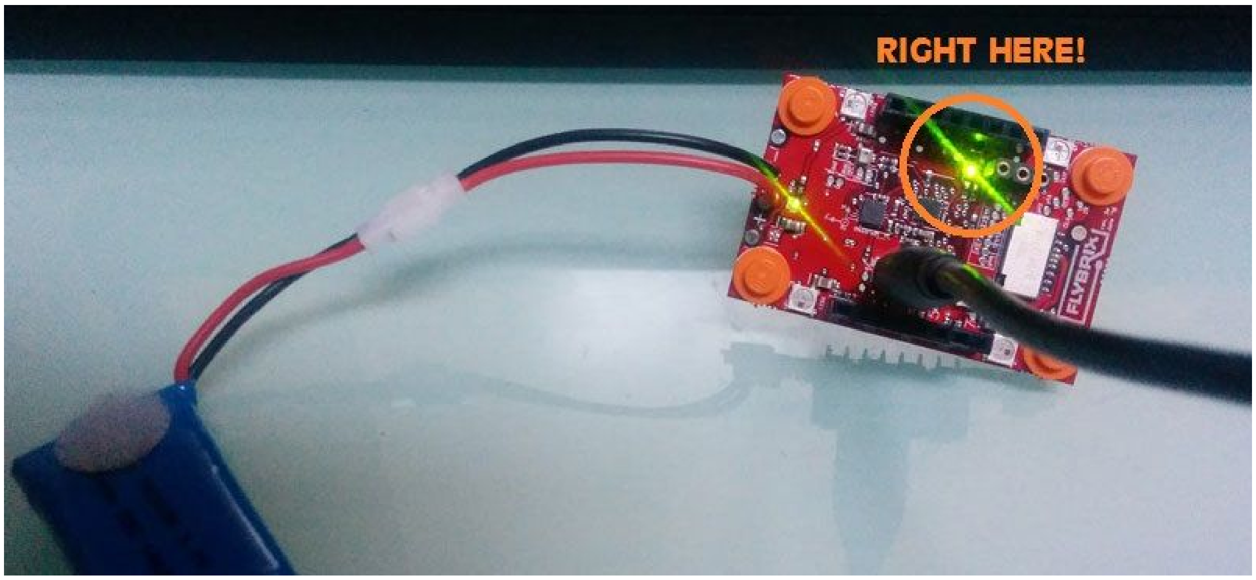


Programming

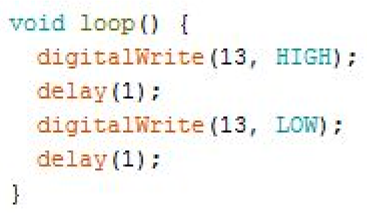
Programming

sketch_aug08a.ino.hex, 5% use

Take a look at your flightboard. Do you see the green indicator LED?



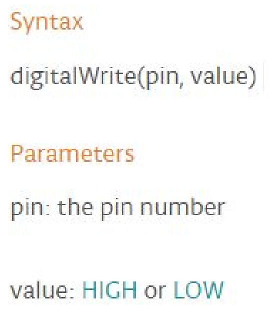
Let’s look again at the code we wrote in the “loop” part of our sketch:



```
void loop() {
  digitalWrite(13, HIGH);
  delay(1);
  digitalWrite(13, LOW);
  delay(1);
}
```

What exactly this this code supposed to do anyway? Well, let’s go one function at a time and search for more information online.

Let’s start by [searching for information on digitalWrite](). The first result is [a tutorial page]() from the official Arduino site. It explains that “digitalWrite” is a function that takes two arguments:


Syntax

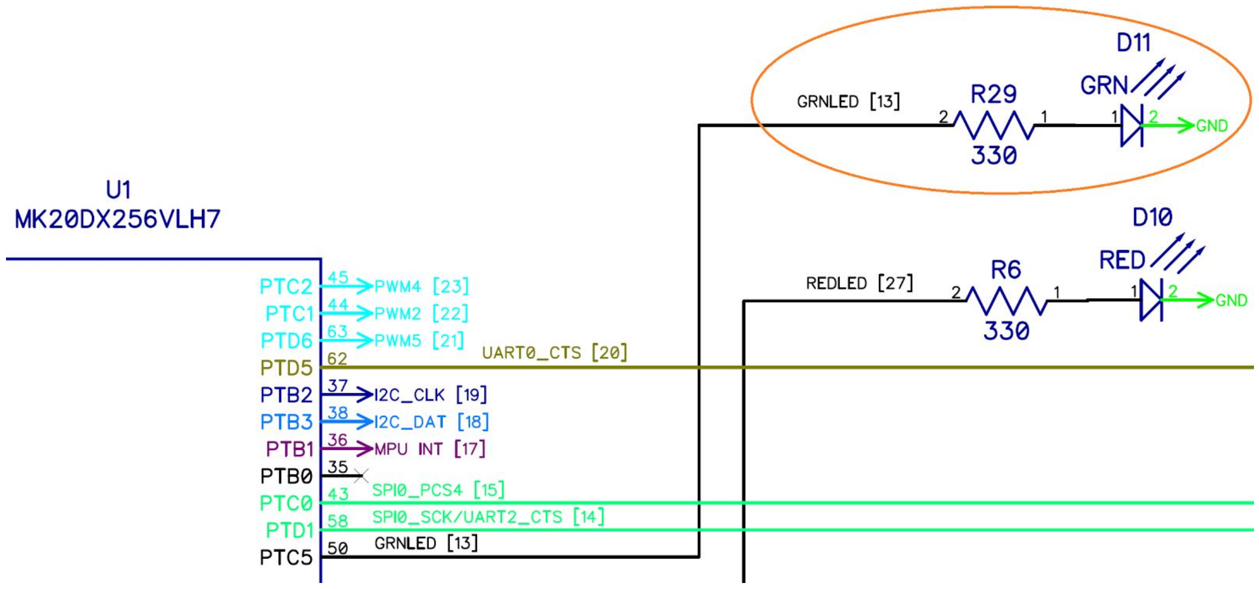
digitalWrite(pin, value)

Parameters

pin: the pin number

value: HIGH or LOW

If you read the entire page, you'll also learn that HIGH and LOW mean two different voltage levels. For our board "HIGH" will be 3.3V and "LOW" will be ground (0V). We also learn from the tutorial page that the "13" in our code specifies the particular pin that is being set at these voltage levels. Let's look at part of the electrical circuit schematic for the Flybrix flightboard:



When we change the voltage on pin 13 using the digitalWrite function, the microcontroller changes the voltage on the wire leading to the resistor R29, which sits in front of the diode D11. We asked the MCU to set the voltage to 3.3V. This means that the microcontroller has to push a current through the resistor and the diode. Let's dig a little deeper on this and read about the diode D11 online. If you look in the Flybrix "Bill of Materials" file, you can find out that the diode D11 is manufactured by Kingbright company and has the part number "APHHS1005CGCK". Search for "Kingbright APHHS1005CGCK" and find the datasheet for the part. A datasheet is full of all kinds of interesting information. For example, you can find out that the dominant wavelength of this particular LED is 570nm. You can also find out that the "forward voltage drop" is typically about 2.1 volts. This means that when the microcontroller sets the voltage on pin 13 to 3.3V ("HIGH"), the voltage at the other side of the resistor will climb up to 2.1V -- meaning that the voltage drop across the resistor is going to be (3.3V - 2.1V = ...) 1.2 volts. You may recall that Ohm’s Law can be used to calculate the current that must be flowing through the resistor to create a drop of 1.2V. In this case the current (I = V/R = 1.2V/330Ohms) is ~3.6mA. This current generates light (and a little bit of heat) in the LED and is supplied by the microprocessor when it interprets the code that you just wrote! This is about as close to Harry Potter’s “Lumos” as you can get.

So now we have “Lumos”.. but what about “Nox”? Let’s return to the code inside the “loop” function. We also wrote “digitalWrite(13, LOW)” with the intention of turning off the LED - but if you look at your flightboard, it probably looks like the light is on all the time, right?

We need to learn more about the “delay” function. Go back to your browser and search for “arduino delay”. The first result should again be a tutorial page from the official arduino website. Aha! The numerical parameter we’re passing to the delay function is interpreted with units of milliseconds. So we’re blinking the LED on and off 500 times every second! This is fast enough to appear as a solid color to your eyes and brain -- that is unless you use a trick related to the persistence of vision effect. Try waving the flight board around in the air -- do you see a row of green dots? Compare to the solid trail created by the other green LED that is powered constantly to show that the battery is attached.

Changing an LED between “on” and “off” five hundred times every second is easy for your flightboard. In fact your MCU could switch the lights “on” and “off” a million times a second without any problems. You should get used to thinking about very small slices of time when you program on a microcontroller!

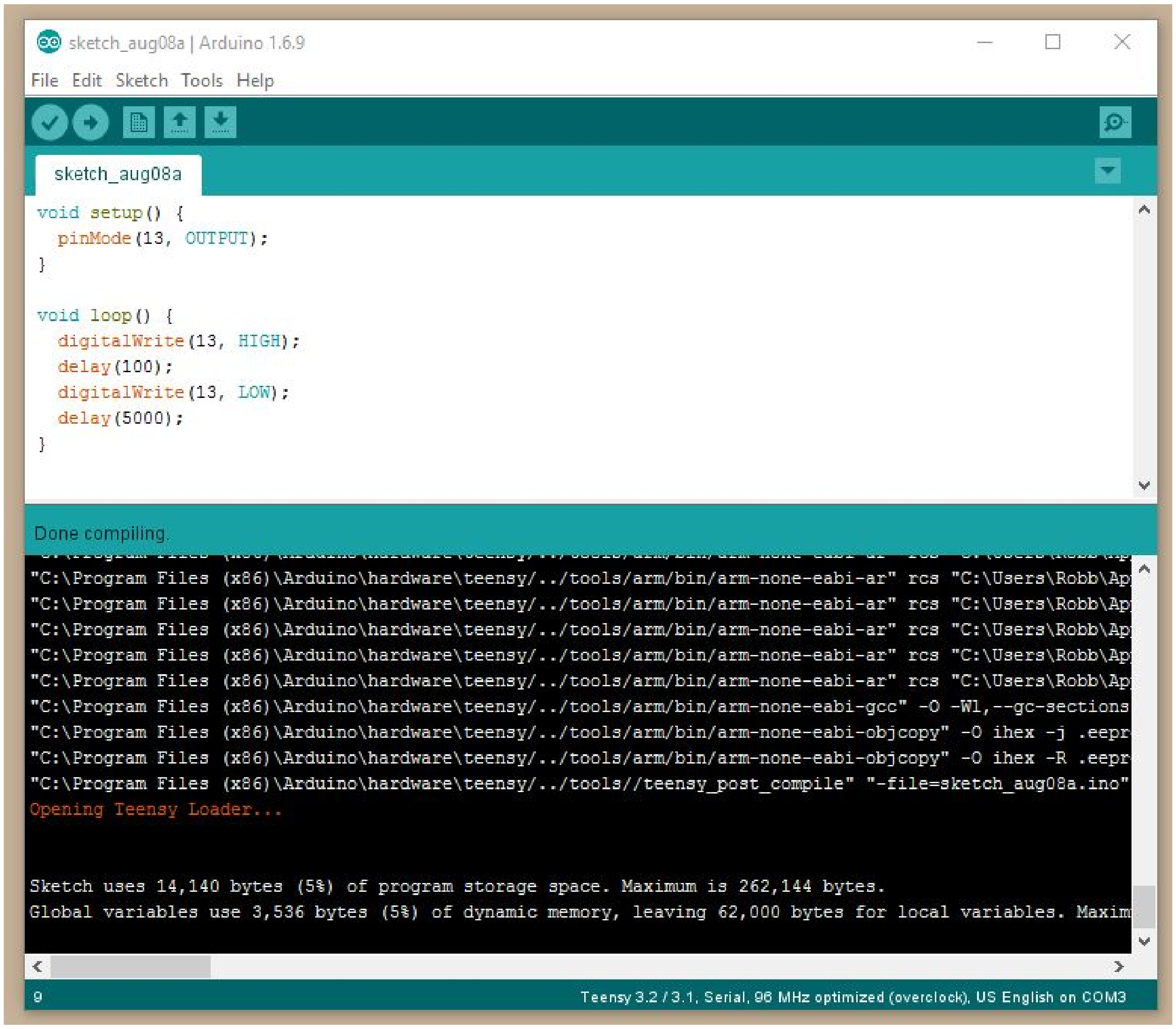
As another aside, let’s think for a moment about how your flight board knows anything about time at all in the first place. Look again at the bottom of your flight board, next to the microcontroller chip. Do you see a small silver box near one corner? It has some marks on top of it that you could read and search for online, or maybe you could ask in a user forum for help discovering that this component is the “oscillator” listed in the bill of materials as “Y1”. This part is manufactured by TXC Corporation and has the part number “8Y-16.000MAAV-T” -- see if you can find the datasheet online. This part is a “quartz crystal” - that’s the only breadcrumb you need to learn all about this type of electronic component if you’re interested. Here’s the wikipedia page. If you found the datasheet, you’ll learn that the part number tells us that this is a 16MHz crystal. So our microcontroller knows about time because it can watch a voltage (generated with the crystal) rise and fall 16 million times a second. If you’re a really good detective, perhaps you’ve noticed that the Arduino IDE has an option to let you change the CPU speed and that the default is 96Mhz and not 16 Mhz. If you wanted to understand this, you might explore circuits that perform frequency division or go out and hunt for the datasheet for the microcontroller itself.

Back to our LEDs. How would you slow things down so that you can see the LED blink without shaking the board around?

7. Change the program so that you can see the LED blink.

See if you can guess how to do this on your own before you continue - there is no right answer here! When you finish making your changes, you can save the file or just press the “Upload” button again and the file will be saved automatically. Are you blinking now?

The next page shows how we changed the code...



```
void setup() {
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, HIGH);
  delay(100);
  digitalWrite(13, LOW);
  delay(5000);
}
```

```
"C:\Program Files (x86)\Arduino\hardware\teensy/../tools/arm/bin/arm-none-eabi-ar" rcs "C:\Users\Robb\Ap
"C:\Program Files (x86)\Arduino\hardware\teensy/../tools/arm/bin/arm-none-eabi-ar" rcs "C:\Users\Robb\Ap
"C:\Program Files (x86)\Arduino\hardware\teensy/../tools/arm/bin/arm-none-eabi-ar" rcs "C:\Users\Robb\Ap
"C:\Program Files (x86)\Arduino\hardware\teensy/../tools/arm/bin/arm-none-eabi-ar" rcs "C:\Users\Robb\Ap
"C:\Program Files (x86)\Arduino\hardware\teensy/../tools/arm/bin/arm-none-eabi-ar" rcs "C:\Users\Robb\Ap
"C:\Program Files (x86)\Arduino\hardware\teensy/../tools/arm/bin/arm-none-eabi-gcc" -O -Wl,--gc-sections
"C:\Program Files (x86)\Arduino\hardware\teensy/../tools/arm/bin/arm-none-eabi-objcopy" -O ihex -j .eepr
"C:\Program Files (x86)\Arduino\hardware\teensy/../tools/arm/bin/arm-none-eabi-objcopy" -O ihex -R .eepr
"C:\Program Files (x86)\Arduino\hardware\teensy/../tools//teensy_post_compile" "-file=sketch_aug08a.ino"
Opening Teensy Loader...


Sketch uses 14,140 bytes (5%) of program storage space. Maximum is 262,144 bytes.
Global variables use 3,536 bytes (5%) of dynamic memory, leaving 62,000 bytes for local variables. Maxim
```

9 Teensy 3.2 / 3.1, Serial, 96 MHz optimized (overclock), US English on COM3

This code makes the green indicator LED blink once every ~5.1 seconds for ~100 msec. (How much time do you think is needed to change the voltage on pin 13?)

If you want to keep playing, you might try also changing the red indicator LED using pin 27 (can you see this in the circuit schematic above?). You'll need to add a line to the setup routine to tell the microcontroller that pin 27 is also being used as an output in your program.

Can you write a program that alternates between red and green indicator lights? How about a program that shows a line of alternating red and green dots when you shake the board around?

Reinstalling the Flybrix Firmware

When you're finished experimenting with the indicator LEDs you'll probably want to reinstall the Flybrix firmware that lets you fly drones! We're going to show you two different ways to do this -

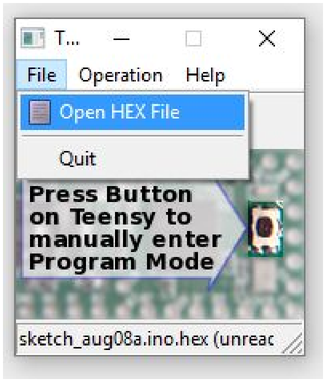
both are equivalent, so you can follow whichever you would like. Method 1 is faster if you're in a rush.

Method 1: Uploading the compiled firmware

Have you visited the Flybrix Github page before? We use Github to keep track of the code that we're writing for the Flybrix flightboard. If you're ever curious about what we're working on, you can check out the differences between the "master" branch (which stores the code for the last official firmware release) and the "development" branch (which stores all of our work in progress).

You can also download precompiled ".hex" files from each official release of the firmware. We're going to use the latest of these compiled files with the "Teensy Loader" program to restore the Flybrix flight code to your flightboard.

1. Download the file "firmware.hex" file from the latest release here.
2. Run the "Teensy Loader" application.
3. Load the file "firmware.hex" from the file menu:


T...
File
Operation
Help
Open HEX File
Quit
Press Button on Teensy to manually enter Program Mode
sketch_aug08a.ino.hex (unreac

4. With your flightboard powered up and plugged into the USB port (is it still blinking?) hit the program button.
5. You should see a progress bar while the Flybrix firmware is uploaded. You should see the boot-up light sequence -- you're ready to go!

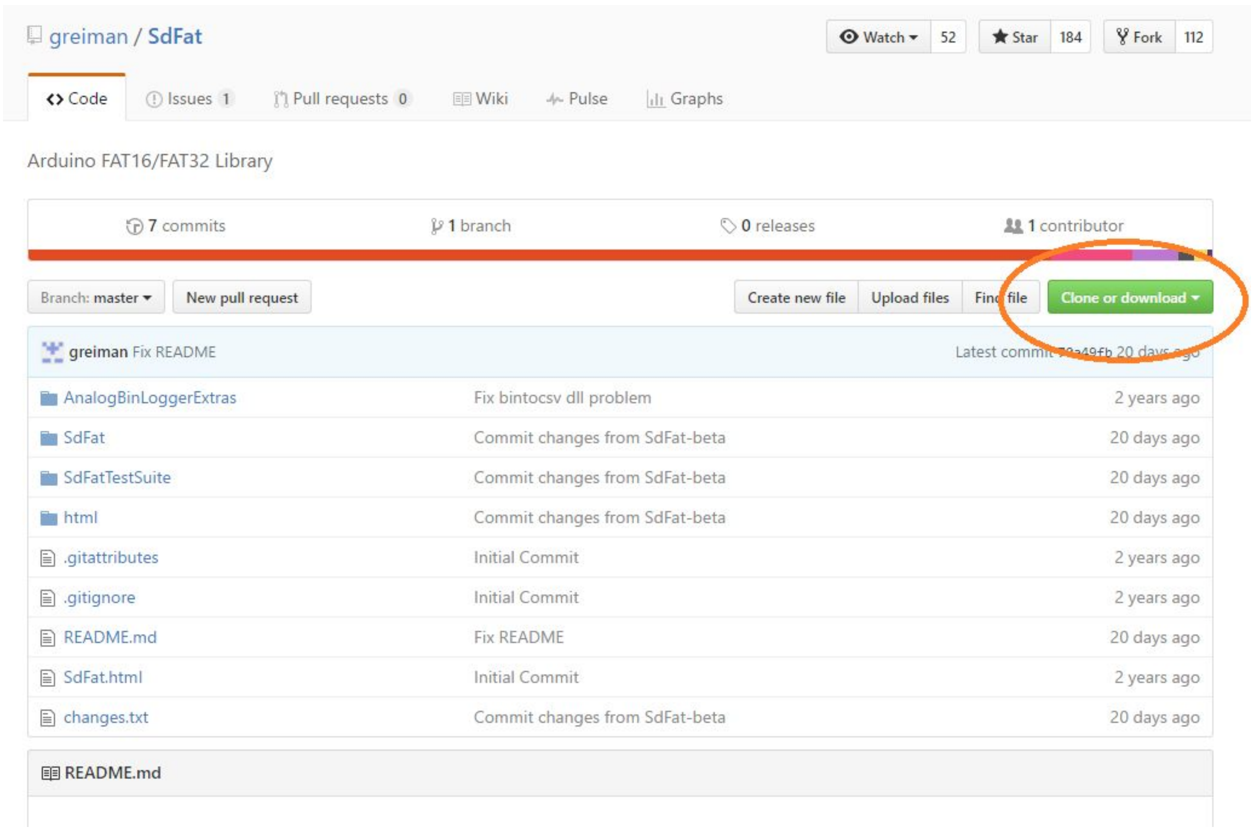
Method 2: Compiling from the source code

You don't have to understand all of the source code for the Flybrix firmware before you can compile it on your own computer. The process is very similar to how you uploaded your blinking indicator LED sketches earlier. There are a few extra steps, but we'll guide you through the process one step at a time.

1. Download the compressed archive of the source code from "Source Code (zip)" from the Flybrix release page on Github here.

2. Extract the zip file somewhere convenient. Remember that the main sketch file "flybrix-firmware.ino" must be inside a folder named "flybrix-firmware". Alternatively, you might decide to rename the sketch to "flybrix-firmware-x.y.z.ino" so that it matches the folder name.
3. Open the main sketch file in the Arduino IDE.
4. Click the round button with the check mark ("Verify").

At this point the compilation will fail because you're missing a library that we're going to need to install separately. The code is called "SdFat" and is written by Bill Greiman. We're going to download Greiman's code from his github page. See the green button labelled "Clone or Download"?



Arduino FAT16/FAT32 Library

greiman Fix README — Latest commit 70a49fb 20 days ago

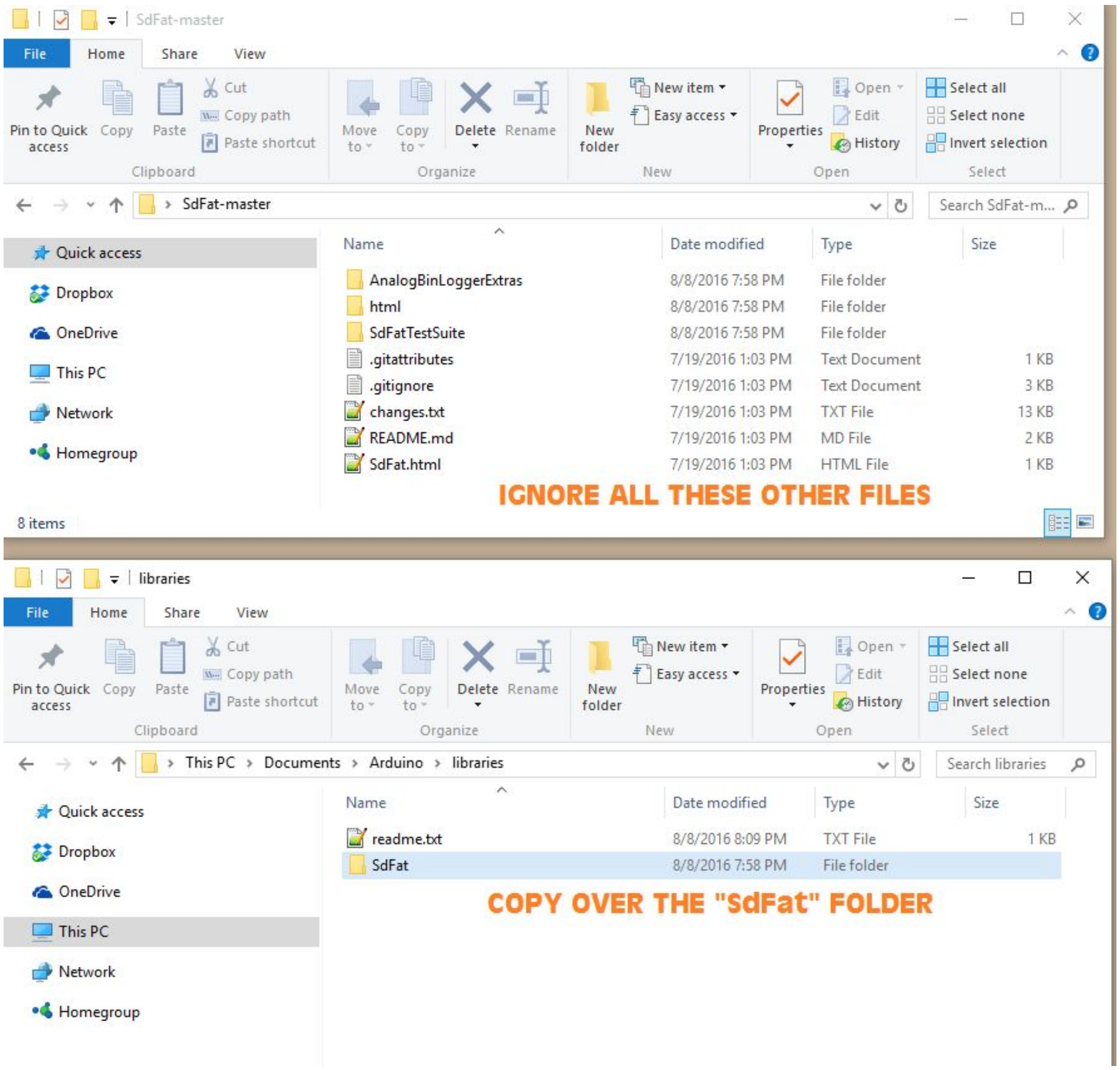
AnalogBinLoggerExtras	Fix bintocsv dll problem	2 years ago
SdFat	Commit changes from SdFat-beta	20 days ago
SdFatTestSuite	Commit changes from SdFat-beta	20 days ago
html	Commit changes from SdFat-beta	20 days ago
.gitattributes	Initial Commit	2 years ago
.gitignore	Initial Commit	2 years ago
README.md	Fix README	20 days ago
SdFat.html	Initial Commit	2 years ago
changes.txt	Commit changes from SdFat-beta	20 days ago

Download the zip file and extract the contents somewhere convenient.

5. Install the SdFat library for Arduino using the manual method described in the Arduino documentation into your "libraries" folder. The only tricky part is finding the right place to put the library, but here's where to look:

   Under Windows, it will likely be called "My Documents\Arduino\libraries". For Mac users, it will likely be called "Documents/Arduino/libraries". On Linux, it will be the "libraries" folder in your sketchbook.

You'll want to copy the "SdFat" folder from inside the unzipped "SdFat-master" folder to the "libraries" folder. You can ignore all the other stuff:



Name	Date modified	Type	Size
AnalogBinLoggerExtras	8/8/2016 7:58 PM	File folder	
html	8/8/2016 7:58 PM	File folder	
SdFatTestSuite	8/8/2016 7:58 PM	File folder	
.gitattributes	7/19/2016 1:03 PM	Text Document	1 KB
.gitignore	7/19/2016 1:03 PM	Text Document	3 KB
changes.txt	7/19/2016 1:03 PM	TXT File	13 KB
README.md	7/19/2016 1:03 PM	MD File	2 KB
SdFat.html	7/19/2016 1:03 PM	HTML File	1 KB

libraries
This PC > Documents > Arduino > libraries
Search libraries
COPY OVER THE "SdFat" FOLDER

Name	Date modified	Type	Size
readme.txt	8/8/2016 8:09 PM	TXT File	1 KB
SdFat	8/8/2016 7:58 PM	File folder	

6. Edit the SdFat Library.

This is the most complicated thing we're going to ask you to do today. Here's the deal - the MCU on your flightboard is much more than just an ARM core processor. It also has a bunch of hardware peripherals that are implemented using electronics rather than software. You've already been using one of these peripheral subsystems all day to talk to the microprocessor through the USB port. The MCU actually has more peripheral subsystems than you can use simultaneously! If you count the pins on the chip, you'll see that there are 64 of them. Many of these pins are connected to more than one peripheral. Remember when we used the function "pinMode" in the "setup" function? We didn't go over the function in detail, but it was needed to tell the chip that we wanted to use pin 13 as a general purpose input output device (GPIO) and more specifically, as an OUTPUT version of this device.

The Flybrix flight software lets you read and write from an SD card using a different peripheral subsystem of the MCU. We need to tell the SdFat library which pins on the MCU are connected to the SD card on the flightboard. Ready?

Let’s go back into the “libraries” folder where you copied the SdFat folder. We’re going to navigate down to a file called “SdSpiTeensy3.cpp” at “...Arduino\libraries\SdFat\src\SdSpiCard”. You will need to open this file in a text editor. It’s probably best not to use a word processor. On a Windows machine you can use Notepad or you can install “Notepad++”. On a Mac, you might use “Text Edit” or download TextWrangler or Sublime.

We’re going to look in the file for the following three lines:

```
    CORE_PIN11_CONFIG = PORT_PCR_DSE | PORT_PCR_MUX(2);
    CORE_PIN12_CONFIG = PORT_PCR_MUX(2);
    CORE_PIN13_CONFIG = PORT_PCR_DSE | PORT_PCR_MUX(2);
```

and replace them with these three lines instead:

```
    CORE_PIN7_CONFIG = PORT_PCR_DSE | PORT_PCR_MUX(2);
    CORE_PIN8_CONFIG = PORT_PCR_MUX(2);
    CORE_PIN14_CONFIG = PORT_PCR_DSE | PORT_PCR_MUX(2);
```

The general idea is that the SdFat library is expecting our SD card to be attached to the MCU at pins 11, 12, and 13, but in fact our SD card is attached to pins 7, 8, and 14.

Phew.

7. Close and reopen the flybrix-firmware sketch (the Arduino IDE isn’t very good at reloading libraries in our experience) and then compile and upload to your flightboard using the “Upload” button.

Congratulations! You just built our flight control software from scratch!

Summary

In this chapter, you installed a development environment on your computer and compiled your first program. You learned some strategies for how to find help online, and you also should feel confident that you can restore your Flybrix flightboard to its factory configuration whenever you want.

We hope you’ll continue to learn how the hardware on your flightboard works while you build your programming skills. The next chapter shows you how to read one of the sensors on your flightboard using your computer as the display.

Chapter 2: Terminal Velocity

Introduction

Receiving Data in the Arduino IDE Terminal

I2C Communication

The MPU9250 Sensor

Estimating Velocity

Building a Simple Pendulum

Summary

Future chapters….

- Connecting to python
- Connecting to the browser
- COBS Encoding for optimal throughput
- Some pendulum experiments
- State estimation
- Basics of control theory
- Driving the motors
- Controlling a real pendulum

- Flightboard Electronics Overview
- The BMP280 sensor
- FastLED
- Radio Control
- Bluetooth
- SD Cards
- SPI Peripherals