utatss-documentation Documentation

Release 1.0

Dylan Vogel, Bruno Almeida, Sidd Mahen, Ali Haydaroglu

Oct 23, 2021

Getting Started

All of the code and some of the hardware design for our project can be found on the UTAT Space Systems GitHub. If you're currently on the GitHub, our documentation can also be viewed on Read the Docs.

This documentation covers the software used on our satellite and some useful getting started information for new members. It has been created with the following goals in mind:

- To enable new UTAT SS members to quickly get up to speed and begin contributing to the codebase
- To document the functionality of the various libraries we develop
- To contain information regarding build instructions and debugging on our custom hardware

Our documentation is organized into the following sections:

- *Getting Started*
- *Software and Programming*
- *Electronics and Altium*

CHAPTER 1

Getting Started

1.1 Software Installation

These are the tools you will need to develop the satellite's software.

Please ask one of the leads if you encounter any issues or are not familiar with using the command line.

macOS: Open the Terminal application.

Windows: Open the Powershell or Command Prompt application.

Linux: Open the Terminal application.

1.1.1 Xcode Command Line Tools (macOS only)

This is a package of common development tools you will need. Note that this will just install a small package, not the full Xcode application.

Run the following:

```
$ xcode-select --install
```

If it says you already have this installed, continue to the next step.

If a dialog box pops up, click `Install`. When it is done, verify it is installed:

```
$ xcode-select -p
```

It should print something like `/Library/Developer/CommandLineTools`.

Reference information

1.1.2 Homebrew (macOS only)

This is a tool that allows us to install and update other command-line programs more easily. Run the command:

```
$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/
↪master/install)"
```

Homebrew is now accessible via the command-line utility `brew`. To make sure the installation was successful, run

```
$ brew help
```

If Homebrew is successfully installed, it will display help information.

1.1.3 AVR Toolchain

This is the software that allows us to compile and upload C programs to the microcontrollers.

macOS

To install AVR-GCC and AVR libc (compiler and C libraries), run

```
$ brew tap osx-cross/avr
```

Once it finishes, run

```
$ brew install avr-gcc
```

This will take a while (usually about 20-30 minutes). Be patient.

To install avrdude (program uploader), run

```
$ brew install avrdude --with-usb
```

To check that the installation completed successfully, run

```
$ which avr-gcc
```

This should print something like `/usr/local/bin/avr-gcc`. If so, you're all set.

Windows

1. Download and install the latest version of WinAVR. Remember where you installed it - we will call the path `<install>` here. By default, it is `C:\WinAVR-20100110`. Leave the default installation options, but uncheck "install programmer's notepad".
2. Open Control Panel and go to `System and Security > System > Advanced system settings` and click on Environment Variables. In the window that pops up, edit the Path variable in the **top** list. Add `<install>\bin` and move it to the top of the list. Add `<install>\utils\bin` and move it to second in the list. Click OK and exit the window.
3. Download this zip file, which contains necessary modifications we made to some files so it works. You may need to retry the download if it doesn't work. Unzip this file. Within the extracted folder, take the `WinAVR-20100110` folder and replace the `<install>` folder on your computer with it.

Reference instructions (**Do not actually follow these instructions**)

Linux

Run the following commands:

```
sudo apt-get update
sudo apt-get upgrade all
sudo apt-get install gcc-avr binutils-avr avr-libc
sudo apt-get install avrdude
```

1.1.4 Make

Compiling and uploading a program requires several complex commands. Instead of having to type these commands every time, we use a tool called Make to automate them using a `makefile` containing these commands.

macOS

This is already installed with the Xcode command line tools.

Windows

Download and run the setup program from here.

Linux

This should already be installed.

1.1.5 Git

Git is a version control system that allows teams to track changes and collaborate on the same codebase. It provides mechanisms for tracking versions of code, so you can see previous iterations and revert back if necessary.

macOS

This is already installed with the Xcode command line tools.

Windows

Download and install Git from here. In the installer, use all of the default settings.

Linux

This should already be installed.

1.1.6 GitHub

GitHub is a platform that builds on top of Git to provide extra features, including a web interface for viewing and managing projects and code.

Create an account here. If you are contributing code to the Heron Mk II codebase, ask one of the leads to add you to the Heron Mk II organization.

1.1.7 GitHub Desktop

This is a desktop application that makes it easier to view and modify code locally on your computer.

Download it from here.

macOS

Move the application to your Applications folder.

Windows

Run the installer.

Linux

This app is not available for Linux.

1.1.8 Atom

This is a text editor that we use to write our code. We recommend Atom, but you can use other text editors such as Sublime or Notepad++.

Download it from here.

macOS

Move the application to your Applications folder.

Windows

Run the installer.

Linux

Run the installer.

1.1.9 CoolTerm

This is a serial terminal program that allows us to view log messages and send commands from a laptop. We recommend using CoolTerm, but other options such as TeraTerm or Xterm are available.

Download it for your operating system from here.

macOS

Open the downloaded `CoolTermMac.dmg` file. In the window the opens, move the `CoolTerm` application to the Applications folder on your computer.

You may get the message `"CoolTerm" can't be opened becuase it is from an unidentified developer`. If this happens, open it by right-clicking the application and selecting `Open`.

Windows

Extract the zip file. CoolTerm does not have an installer; you can move it and run `CoolTerm.exe` from anywhere on your computer as long you move the entire `CoolTermWin` folder containing the necessary libraries. We recommend moving it to your `Documents` folder, but you can move it somewhere else if you want.

Linux

Run the application.

1.1.10 Linux Reference Instructions

The Linux instructions were based off of the following links (you can check them for reference):

- Reference 1
- Reference 2

If you want to use the Windows Subsystem for Linux, follow the instructions here. If you don't know which distro to use, pick Ubuntu.

1.2 Optional Software installation

This page contains miscellaneous software that is not essential, but that you may want to install sometime later when you need it.

1.2.1 Pololu USB AVR Programmer v2

This utility is for changing the settings of a hardware programmer device.

Download and install the Pololu USB AVR Programmer v2 for your operating system here. If you're using the older programmer, this might not work.

1.2.2 Python and Pyserial

These are needed to run the test harness, our automated software testing framework.

We are only supporting Python 3 with the test harness, not Python 2. Open your command line and type `$ python -V` to check the version.

If you don't have Python at all or need Python 3, download it from here. Get the latest version (v3.7.1 as of Nov. 28, 2018) and install it.

If you are on macOS, make sure to follow the instructions on the last page of the installer to install certificates.

If you are on Windows, make sure to check the box called "Add Python 3.7 to PATH".

Run `$ python -V` to check that you have Python 3. If you are on macOS, it probably distinguishes the versions using `python` for Python 2 and `python3` for Python 3. You can try running `$ python3 -V` to see what happens. If that works, use `python3` instead of `python` from now on so that you use Python 3.

Now you will need to install the Pyserial library, which the test harness requires to interface with serial ports. We will do this using the pip package manager.

```
$ pip install pyserial
```

If you are on macOS, you might need to use `pip3` instead of `pip`.

If this doesn't work, you might need to follow the instructions in the terminal to upgrade the version of pip on your computer.

Please ask for help if you encounter any problems.

1.2.3 SpeedCrunch

This app is useful for quickly converting numbers between decimal, hexadecimal, and binary.

Download and install here.

1.3 Patching avrdude for 64M1

To work with the 64M1 microcontroller, you need to modify the avrdude configuration manually because avrdude does not support it by default. This involves modifying the configuration file to add the settings it needs to use the 64M1.

Download the patched `avrdude.conf` file from here, which has the configuration settings for the 64M1.

Open the main `avrdude.conf` file on your computer. On macOS, this is located at `/usr/local/Cellar/avrdude/6.3_1/etc/avrdude.conf`. Open the patched `avrdude.conf` file, copy the ATmega64M1 section (at the end of the file, lines 1704-1887), and add it to the end of main `avrdude.conf` file on your computer, then save it. Now when you run avrdude with the 64M1, it should recognize it and be able to program it.

1.4 Real-Time Embedded Systems

An embedded system refers to a software system which is integrated into a larger project, often alongside other mechanical or electrical systems. The microcontroller which controls your coffee machine is an example of a embedded system.

A real-time system is a software system that has real-time constraints; the system must make certain guarantees regarding how quickly it responds to input. An electric braking system is an example of a real-time system.

The Heron Mk II is a real-time embedded system: the software onboard must integrate and interact with solar cells, external memory, fluid actuation mechanisms, radio antenna, all within certain (small) time frames.

This introduces unique challenges. If something goes wrong, there is no one to reset the system, and once the satellite is launched, there is no way to re-program its various components; therefore the system must be robust and handle errors gracefully. On board memory (RAM) and non-volatile memory (flash storage) are limited. Communicating with the satellite requires a radio link, which is only possible when the satellite is above the ground station.

1.5 Satellite Software

Some of the subsystems in the satellite have a software component. Each subsystem that uses software has its own **microcontroller**, which executes programs that we write in C. We write code that runs on the microcontrollers (each has its own code, but with some common components).

We use a microcontroller (MCU) called the **ATMega32M1** (32M1), which is part of the AVR family of microcontrollers.

The following subsystems each have an MCU:

1.5.1 OBC (On Board Computer)

- the main microcontroller on the satellite that coordinates the satellite's actions
- developed by the CDH (Command and Data Handling) subsystem

1.5.2 PAY (Payload)

- controls the experiment in the paylod
- developed by the Instrumentation subsystem

1.5.3 EPS (Electrical Power Systems)

- regulates the power/energy systems on the satellites
- developed by the EPS subsystem

1.6 Command Line

The command line is a useful way to interact with your computer. It allows you to navigate your computer and execute command line programs. As you will see in developing the satellite's software, there are particular programs and functionality that are only available through the command line.

It will probably seem scary and confusing at first, but will become natural as you use it more.

To access the command line, open the Terminal application (Mac) or Command Prompt application (Windows). The Mac command line is called a bash terminal, while the Windows command line is called a cmd terminal.

You always have a **current directory** (folder) where operations take place. It will prompt you with the current directory followed by `$` (Mac) or > (Windows) to enter a command.

Here are some common commands. Remember that a folder and a directory are the same thing. Any text inside <> is text you replace (without the brackets).

| Mac | Windows | Command | | :- | :- | :- | | `pwd <path>` | `chdir <path>` | Print working (current) directory | | `cd <path>` | `cd <path>` | Change directory | | `cd ..` | `cd ..` | Go up (back) one directory | | `ls` | `dir` | List files and folders in current directory | | `mkdir <name>` | `mkdir <name>` | Make new directory |

When you execute a command (program), sometimes you pass it **command line arguments** to give it additional information or instructions.

For example, `ls` displays just the file and folder names in a directory, while `ls -l` tells it to display more information about the files and folders.

1.7 The AVR Toolchain

The AVR toolchain is a collection of open-source tools which can be used together to compile and upload programs to AVR-based microcontrollers, such as the ATMega32M1 microcontroller used on the Heron Mk II subsystem microcontrollers.

Installing and understanding how to use the AVR toolchain is the first step to contributing to the Heron Mk II codebase.

1.7.1 Glossary

Before we dive in, we pause to introduce the following terminology. It will be used liberally throughout these docs.

- SSM: abbreviation for subsystem microcontroller; this refers to the microcontroller which controls a particular subsystem board.
- 32M1: abbreviation for the ATMega32M1.
- MCU: abbreviation for microcontroller.
- Library: a self-contained unit of code, intended to be reused.
- Header: a file describing the interface to a particular unit of code. Header files end in `.h`.
- SPI: serial peripheral interface; the mechanism by which the 32M1 communicates with peripherals, such as external sensors.
- CAN: controller area network; a robust bus on which all SSMs communicate with one another.

1.7.2 The AVR-GCC Compiler

The AVR-GCC compiler is a tool used to compile and link C code to create an executable that can run on the 32M1. Generally, compilation refers to the process of transforming structured data from one form into another. The AVR-GCC compiler transforms C code into machine instructions that the 32M1 can execute. It also links this code together into a executable; a program which can be run on the 32M1.

In fact, AVR-GCC accomplishes much, much more, including warning you of certain kinds of errors in your code (type errors) and performing optimizations to reduce code size.

Installing AVR-GCC also installs a host of other useful programs such as `avr-objcopy`, `avr-ar` and `avr-objdump`. You'll learn about these tools later.

The AVR-GCC is based on the well known GCC compiler. To read more about AVR-GCC and GCC, look here.

AVR-GCC can be invoked by running `avr-gcc` in the command line. You won't normally have to invoke the AVR-GCC compiler yourself; as you'll see the process of compiling your code can be automated to the point where it becomes a single command.

1.7.3 AVR libc

AVR libc is a library which provides all of the standard C headers and their implementations, as well as special code used to access features unique to AVR-based MCUs, such as accessing pins, handling hardware interrupts, and communicating with peripheral devices.

Whenever you use standard C libraries, for example, via `#include <stdint.h>` or `#include <avr/io.h>`, you are implicitly using AVR libc.

To learn more about AVR libc, read the online documentation here.

1.7.4 avrdude

Avrdude is a command-line application which, when used in combination with a programmer, can upload the contents of a specific file to the 32M1's flash memory. We use avrdude to upload executables onto the 32M1.

Avrdude writes to the 32M1's flash memory using a protocol called In-Circuit Serial Programming; ISP for short. A programmer is simply a device which allows your computer to interface with the pins of the 32M1 that are relevant to ISP. You will learn about which pins are relevant to ISP later, when you learn more about the 32M1.

The programmer available for general use in the UTAT lab is the Polulu USB AVR programmer v2. To learn more about the Polulu, read the online documentation here.

1.8 The ATMega32M1 Microcontroller

The ATmega32M1 microcontroller used in Heron Mk II SSMs is a part of a family of MCUs called the Atmel AVR family, produced by Atmel corporation (now owned by Microchip Inc.). All the MCUs in the AVR family have a similar architecture and instruction set.

The 32M1 has an 8-bit architecture. This means most registers are 8 bits wide, and the data bus is 8 bits wide. Because some registers are 16 bits wide, they must be accessed using special atomic instructions.

In practice, you won't often need to worry about the underlying details of the instruction set or architecture; you can simply write code in C and be confident `avr-gcc` will generate the correct instructions for you.

1.8.1 Pin Layout

The 32M1 has 32 pins, labelled from 1 to 32. Most pins have some specific functionality; for example, pin 4 is VCC, the supply voltage. Instead of referring to pins by their numeric label, you can also specify a pin using its mnemonic label; pin 8 is also labelled pin PB0, and read "port B zero".

There is no simple rule matching a pin's number to its mnemonic label; see the 32M1 datasheet for more details.

If a pin is associated to a port, it can be read and written to using software instructions. We will examine this more closely later.

Every pin associated to a port has two key properties: its data direction, and its value. A pin's data direction is either input (0) or output (1). A pin's value is either low (0) or high (1).

We describe two families of pins which will be referenced many times throughout these docs. Knowing where these pins are and how to measure the voltage across them can also be useful for debugging purposes.

In what follows we use mnemonic labels because this is how pins are specified in C code.

1.8.2 SPI Pins

See the Communication Protocols section for how the SPI protocol works.

- **PB7 - SCK/SCLK** - The 32M1 outputs the clock signal to this pin to synchronize all devices.
- **PB0 - MISO** - Peripherals (other devices) write to this pin to send data to the 32M1.
- **PB1 - MOSI** - The 32M1 writes to this pin to send data to a peripheral device.
- **PD3 - CS/SS** - This pin can be used by another peripheral to select the 32M1 as a slave.

In the current setup, the CS/SS pin is not used because each SSM is the master in the SPI master/slave model. It is included for completeness, and because it can be the source of many difficult-to-debug bugs. For example, accidentally setting this pin as an input pin and driving it high can cause the 32M1 to believe it is now a SPI slave, thus changing the MOSI and clock SPI pins to input pins.

1.8.3 CAN Pins

See the Communication Protocols section for how the CAN protocol works.

These pins connect the 32M1 to the CAN transceiver, which is the device that actually transmits and receives messages on the CAN bus.

- **PC2 - TX** - Transmit
- **PC3 - RX** - Receive

1.9 Hello World ATMega32M1

Now that you've installed the AVR toolchain, let's compile and upload our first program.

Later on, this compilation and uploading process will be made much less painful using tools such as Make; for most subsystems, this entire process can be accomplished using a single command.

1.9.1 Step 1

Create a new directory called `hello_32m1` as follows:

```
$ mkdir hello_32m1
```

This will be the root directory of the project. Now enter this directory by running `cd hello_32m1` and run the following command:

```
$ git init
```

This will initialize Git for this root directory and all sub-directories. You'll learn more about what this command is doing soon.

1.9.2 Step 2

Create a new file called `hello_world.c`, and copy the following contents into the new file:

```c
#include <uart/log.h>

int main() {
    init_uart();
    print("Hello ATMega32M1!\n");
    return 0;
}
```

The first line, `init_uart();`, initializes the UART circuitry on the 32M1. UART stands for Universal Asynchronous Receiver-Transmitter. This allows the 32M1 to communicate with external peripherals using a serial protocol.

The second line, `print("Hello ATMega32M1!\n");`, send the string "Hello ATMega32M1" to the UART system to be written to serial output.

The third line returns from the `main` function.

Run `$ git status` and read the output carefully. Notice that Git has noticed the new file you've created.

1.9.3 Step 3

Download the `lib-common` library by running the following command in the root directory:

```
$ git submodule add https://github.com/HeronMkII/lib-common
```

Run `ls` in the root directory. Notice a new directory has been added, called `lib-common`. Explore the contents of this directory by running `cd lib-common` (do *not* modify anything).

This is a library which contains many code components common to all subsystems, including SPI, UART and CAN functionality.

Before the library is used, it must be built. To do this, make sure you are in the `lib-common` directory and run

```
$ make
```

Inspect the `lib` folder in the `lib-common` directory. You should notice many files ending in `.a`. These are the library files we just generated.

1.9.4 Step 4

We're now ready to compile our program. Make sure you're in the root directory, and run

```
$ avr-gcc -std=gnu99 -Wall -mmcu=atmega32m1 -c hello_world.c -I ./lib-common/include
```

This will create a new `hello_world.o` object file. We must now link this object file to create an executable. To do this, run

```
$ avr-gcc -std=gnu99 -Wall -mmcu=atmega32m1 -o hello_world.elf hello_world.o -L ./lib-
↪common/lib -l uart
```

This will create a new file called `hello_world.elf`; this is a complete executable file. To upload it onto the 32M1, however, we must format it in the Intel Hex format. To do this, run

```
$ avr-objcopy -j .text -j .data -O ihex hello_world.elf hello_world.hex
```

This creates the final executable which we will upload to the 32M1, called `hello_world.hex`.

1.9.5 Step 5

We're now ready to upload our hello world executable onto a board.

To do this, you need to connect a 32M1 on a board to your computer using a programmer. Find a Polulu programmer in the lab, and connect it to an open USB port on your computer.

We will use `avrdude` to upload our `hello_world.hex` onto the 32M1. To do this, we need to tell `avrdude` which USB port the programmer is connected to. To find this out, run

```
$ ls /dev/tty.usb*
```

This lists all USB devices connected to your computer.

The USB device corresponding to the programmer is almost always the device with the lowest id. For example, if the command above returns `/dev/tty.usbmodem00100561 /dev/tty.usbmodem00100563`, the USB port corresponding to the programmer is `/dev/tty.usbmodem00100561`. Make sure to remember this port.

Connect power to the VCC pin on the board, and ground the GND pin. Set the voltage to 3.3 volts. Connect the programmer to the programmer port on the board. Check that the light on the programmer is green; this means the board is being powered correctly. If the board has insufficient power, the programmer will not upload to the board, because this has the potential to corrupt the MCU's memory.

Now run the following command, where `<port>` is replaced with the USB port you found above.

```
$ avrdude -p 32m1 -c stk500 -P <port> -U flash:w:./hello_world.hex
```

This should upload your program onto the 32M1! If this command runs successfully, congratulations! You've just compiled and uploaded your first program onto the 32M1!

1.9.6 Step 6

To verify that the program is running correctly, first connect the female RX pin on the programmer via a male to male wire to the MOSI pin on the board. This is wire along which the serial communication between the 32M1 and your computer will occur.

Now, open CoolTerm. Click the Options icon on the top of the new window. The default options do not need to be changed. Click the Port dropdown menu and select the port you identified above. Click OK on the bottom right. Click the Connect icon. Now, any UART output from the 32M1 should appear on your screen.

Reset the board by pressing the reset button. If all goes well, you should see the string "Hello world!" appear on your screen!

This process was very time consuming and error prone. We'll see how we can simplify this process in the future using tools such as Make.

1.10 Git and Github

As we mentioned before, Git is a tool used to manage software over time. Git keeps track of changes to files, and allows developers to attach messages to changes they've made to let others know what's changed. Git also supports a workflow that enables large teams to collaborate on a complex codebase without treading on each others toes.

For example, because Git preserves all of the changes in your codebase, if you discover a fatal bug, you can revert a specific file or the entire project back to an earlier bug-free state.

Git is a very powerful tool. We will only scratch the surface of what Git can do. If you're interested in software development, it is well worth the time to learn how to use Git productively.

Github is a separate website used to manage Git-based projects. It provides many convenient features, such as a visual interface to view code changes and browse the codebase.

1.10.1 A Brief Overview of Git

Git tracks changes you make to the codebase and bundles them into *commits*. Each commit includes data keeping track of the changes made to the codebase, as well as a message describing the changes that were made. Thus, a commit is a snapshot of the codebase at a particular point in time, along with a message describing what has changed since the last commit.

Each commit occurs on a particular *branch*. A branch is a lightweight copy of a project (from a particular commit) where you can experiment without having to worry about clobbering or interfering with other people's changes.

The idea is that each time you want to build a new feature, you can create a new branch, make commits on this new branch, and then *merge* this branch with the codebase.

This workflow allows multiple people do develop different features in parallel.

A Git-based project is called a *repository*, or repo for short.

1.10.2 Creating a Github Account

Visit Github and create an account if you don't already have one. Heron Mk II has a organization-wide Github account which hosts all subsystem code.

Let Siddharth, Ali or Dylan know what your account name is, and we'll make sure to add you to the organization. Unless you've been added to the organization, you won't be able to access any of the codebase via Git.

1.10.3 Setting Up Git

To make sure each commit you create is labelled correctly, Git needs to know your name and email address. You can set your name and email address using the following commands:

```
$ git config --global user.name "Your Name"
$ git config --global user.email "your@email.com"
```

1.10.4 Learning Git

To learn Git, work through the following tutorials: the official Git tutorial and the Git everyday tutorial.

From now on, we expect that you have some basic familiarity with Git commands and how they are used.

1.10.5 Reviewing Key Git Concepts: The Working Tree and The Index

We briefly summarize some key concepts from the tutorials above. This terminology is ubiquitous and important to understand.

A single Git repository can keep track of an arbitrary number of branches. The current branch that is being used, and all of the commits that were made on this branch, is called the *working tree*.

When you *check out* a new branch, you're replacing the contents of the current working tree with the contents of the last commit on the new branch. The last commit on a branch is often called the branch *HEAD*. The HEAD keyword can be used in many Git commands, see the documentation for more details.

The *index*, or *staging area*, consists of changes to files you wish to include in your next commit. When you run commands such as `git add file1 file 2`, you're modifying the index to include the changes made to `file1` and `file2` in the next commit. As you probably know by now, you can inspect which files in the working tree have changed since the last commit, and which are staged in the index, by running `git status`.

1.10.6 Git Best Practices

The following are a list of best practices which will help you and the team use Git effectively.

Use Descriptive Names

When creating new branches, tags, etc. try to use descriptive names. Long, descriptive names are preferable to short, uninformative ones.

Never Push Directly to `master`

In general, it is not a good idea to push directly onto the `master` remote branch. As a rule, `master` should always hold a version of the project which compiles without errors or warnings, and on which all tests pass. Pushing directly to `master`, especially without testing, can compromise this.

More generally, be wary when pushing to a remote branch. If you rebase a branch and then `git push -f` it onto a remote branch, you could be ruining the branch history for others working on the same branch. This is considered bad practice, and your teammates will not appreciate it.

Create Good Commits

A good commit consists of:

1. A set of related changes to a group of related files.
2. An informative, well-crafted commit message.

Creating good commits requires discipline.

Each commit should be a self-contained snippet of progress towards a certain goal. Trying to combine multiple unrelated changes in a single commit is bad practice. For example, if you're implementing two completely different features, they should each have their own commits. If these features require multiple commits, they should be developed on their own branches.

Every commit message should contain a short title summarizing *all* changes, followed by a more detailed summary of the changes in bullet point form, or a paragraph of descriptive text. Correct spelling, grammar and punctuation are expected. The commit message should allow other developers to quickly and easily identify whether the commit in question contains changes they care about.

If you create commits which glob unrelated changes together and have useless commit messages your commits will not be merged.

Read about `git amend` and `git rebase` to learn how to tidy up a branch's commit history.

Merge Changes Using Pull Requests

As mentioned above, you should never be pushing directly onto `master`.

Instead, create a pull-request on the project's Github page, and request a code review from one of the leads. Branches merged directly to `master` without review will be reverted.

Create Separate Branches for Bug Fixes

If you find a bug and fix it, it is important to create a seperate branch containing your changes, and open a separate pull request specifically for the bug fix. Do *not* just add the bug fix to the branch you're currently working on.

Suppose you're working on a new feature on the branch `new-feat` and you happen to find a bug in the file `src/a.c`. Once you fix the bug on your current branch, checkout the `master` branch and create a new branch called, say `a-bug-fix`. This branch will contain only the new bug fix. Now, apply your bug fix onto this new branch, in this case, by modifying `src/a.c`. Next, push this new branch to Github and create a pull request explaining what you found and how you fixed the problem.

Following this process allows other people to benefit from your find, and helps keep track of bug fixes across many branches.

1.10.7 Git Cheat Sheets

The following Git cheat sheets are quite useful: cheat sheet written by Github, cheat sheet written by Atlassian.

1.10.8 Additional Resources

To learn more about Git, or read about the many features we have not mentioned, run `git help`, `git help <command>` or read the documentation online.

To learn see a glossary of Git terminology, see the official glossary.

CHAPTER 2

Useful Websites

Mbedded Ninja - General electronics and software (**very helpful!**)

All About Circuits - General electronics and PCB design

Explore Embedded

AVR Beginners

PCB 3D - PCB design and footprints

SnapEDA - PCB schematic symbols, footprints, 3D CAD models

Ultra Librarian - PCB schematic symbols and footprints

Military standard wire parameter calculator

Signal Processing and Electronics

DC Motor Characteristics

DC Motor Control

ReStructuredText documentation format -

CHAPTER 3

Software Tools

3.1 Software Workflow

This page describes the general workflow to clone a GitHub repository to your computer, make changes to the code, upload them to a board, and push your changes back to GitHub.

Make sure you have already installed everything in Getting Started > Software Installation, created a GitHub account, and are added to the Heron Mk II GitHub organization.

This tutorial will use the `pay` (payload instrumentation) repository as an example, but this process applies to any repository.

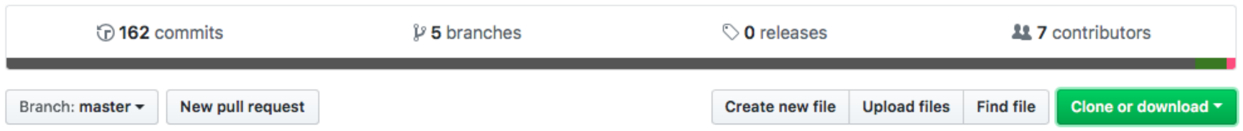
3.1.1 GitHub Repositories

If you go to the UTAT Space Systems organization on GitHub (https://github.com/HeronMkII), you will see a list of repositories. A **repository** is a project with a collection of code for a particular purpose. We have a repository for each of the circuit boards/microcontrollers.

3.1.2 Cloning a Repository

Cloning is the process of copying a repository on GitHub to your computer so you can make changes and push changes back to GitHub.

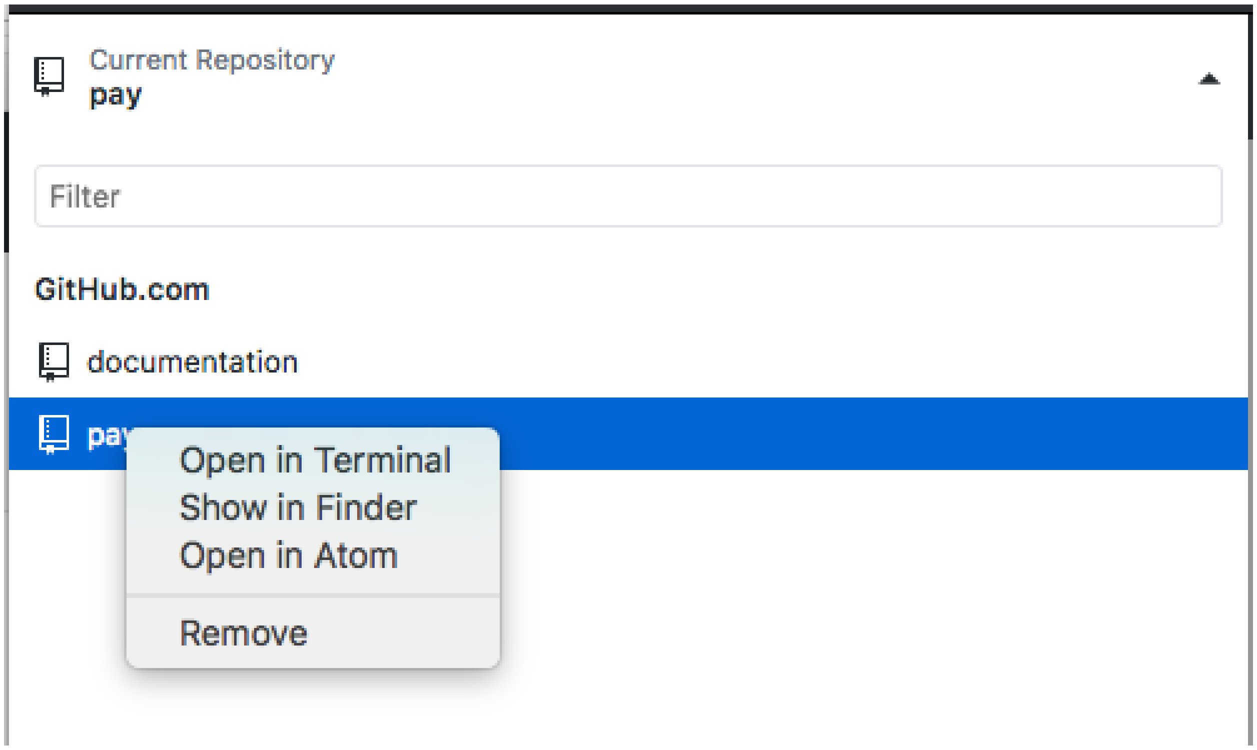
On the repository page (https://github.com/HeronMkII/pay), click the green button called `Clone or download`, then click `Open in Desktop`. Choose where to save the repository locally on your computer.



3.1.3 Editing Code

There are two ways to open the project in Atom:

1. Click `Current Repository` in the top left, right-click on `pay`, then click `Open in Atom`.



1. Open Atom, go to `File > Open`, navigate to where you saved the repository locally, and select the `pay` folder.

You can now pen and edit files in Atom. Atom is a text editor, so it can edit code but it can't run it. We need to use command line tools to compile and upload code to the board. Make sure to save your edited files in Atom before compiling and uploading the program.

3.1.4 Project Structure

When opening the repository folder in Atom, you will see many folders and files.

Here are the commonly used folders.

- **.git**: Not actually part of the project, so ignore it. It contains files that Git uses for version control.
- **build**: Compiled versions of the files.
- **examples**: Example code that demonstrates use of particular software components. This is not the actual code that will go on the satellite.
- **lib-common**: This is a Git **submodule**, which has code from another repository embedded in this repository so that we can use it. Our `lib-common` repository contains common code to be shared among all the repositories.
- **src**: The actual code that will go on the satellite (`.c` and `.h` files).
- **tests**: Automated testing files that will be used by the test harness.

`lib-common` has a slightly different structure:

- **bin**: For the automated test harness (`harness.py`).
- **include**: All the `.h` files.
- **lib**: Compiled versions of the common libraries.
- **src**: All the `.c` files (the `.h` files are moved to `include`).

Here are the commonly used files.

- **.gitignore**: Specifies which files Git should ignore (not saved to the GitHub repository).
- **.gitmodules**: Contains information about the Git submodule used in this repository.
- **makefile**: Contains instructions that Make will use to compile the project.
- **README.md**: Documentation about this repository.

3.1.5 Compiling and Uploading a Program

We need to compile the project from the command line. Open your command line (Terminal on Mac or Command Prompt on Windows) and navigate to the `pay` folder.

```
$ cd <folder>/<other folder>/.../pay
```

Make sure `lib-common` is up to date:

```
$ git submodule update --remote
```

We need to compile `lib-common` separately from the `pay` project.

```
$ cd lib-common
$ make
```

When you run `make`, the Make tool looks inside the folder for a `makefile` and executes the instructions in it to compile `lib-common`.

Go back (up) one folder:

```
$ cd ..
```

Now compile the `pay` project.

```
$ make
```

If you are connected to the board, upload the program to the 32M1.

```
$ make upload
```

If you get an error about the device not being found, see the `Finding the Correct USB Port` section below.

Note for future reference that you can use `make upload` to both compile and upload the program.

If there are any compilation errors, fix them in Atom, save the files, and run `make` again.

Note: If you get errors about C99, go to the makefile and add `-std=c99` to the CFLAGs. Ask a lead if you are not sure.

3.1.6 Finding the Correct USB Port

If you get an error about the device not being found, you need to change the port used to communicate with the device. In Atom, open the `makefile` in the `pay` folder. Find the line starting with `PORT =`. Find the port on your computer as shown below, modify this line to change the port, save the file, and run `make upload` again.

When you do this process, you should find two ports for the programmer. Usually, the lower number is for uploading programs while the higher number is for viewing UART (serial) output.

Mac: To see all connected USB devices, run

```
$ ls /dev/tty.usb*
```

Windows: Open the `Device Manager` application on your computer, then select `Ports`. Find the appropriate port. You want the programmer port, not the serial port. In the makefile, modify the line to use the programming port, such as `PORT = COM7`.

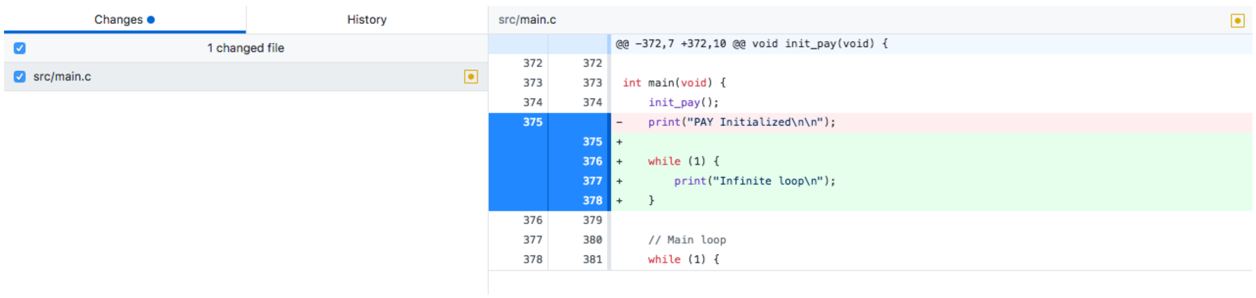
Linux (Windows Subsystem): Follow the above instructions for Windows to get the port number from the Device Manager. In the Linux subsystem, these ports are mapped with the same numbers but slightly different names. As described here, `/dev/ttyS<N>` on Linux is tied to `COM<N>` on Windows. For example, use `dev/ttyS7` instead of `COM7`.

Linux (Native): Generally, the programming port is `ttyACM0` and the serial port is `ttyACM1`. You may need to run the command for giving permission to use the serial port: `sudo chmod 666 /dev/ttyACM0` (or the same with `ttyACM1`).

3.1.7 Committing and Pushing to GitHub

After making changes to the code and running it, you will probably need to modify, compile, upload, and test it a few times. Remember to save the files in Atom and run `make upload` to compile and upload it each time.

When you are ready to save your changes to GitHub, go to the GitHub Desktop application. It will show which lines in which files you have changed. Red lines have been deleted and green lines have been added. Review these changes to be sure.



```
@@ -372,7 +372,10 @@ void init_pay(void) {

 int main(void) {
     init_pay();
-    print("PAY Initialized\n\n");
+
+    while (1) {
+        print("Infinite loop\n");
+    }

     // Main loop
     while (1) {
```

Now, we will create a commit. A **commit** is a version of the project where you have made a particular set of changes. In the `Summary` box, type a summary of the changes you made. In some cases, you may want to add a more detailed description. Click the `Commit` button to make your commit. Push it to GitHub by clicking the `Push Origin` button in the top right. Now your commit is saved on GitHub.

3.1.8 Branches and Pull Requests

A **branch** is like a timeline of development within a repository. We have a **master** branch that always contains a version of the code that has been tested and that we know works.

Never commit directly to the master branch (except for rare cases when one of the leads tells you otherwise). We create separate branches to develop and test code, then integrate those changes into master when we know they work.

To create a new branch on the GitHub repository page or in GitHub Desktop, select the *base* branch from which to create the new branch. Go back to the list of branches, type in a name for the new branch, and create it. When editing and committing code on this new branch, make sure you have that branch selected in GitHub Desktop.

When you want to integrate your commits on this branch into the master branch, create a **pull request**. Go to the GitHub repository page, click `New pull request`, choose the base (master) and the compare (your branch). Add a description and people to review it (usually your subsystem lead). If they approve your pull request, they will **merge** it into the master branch, integrating those commits into master.

3.2 Test Harness

The **test harness** is a program we use to automate testing parts of our code. It takes a folder of test programs, uploads each program to the microcontroller, runs it, observes its output, determines whether the tests passed or failed, and prints the results.

Each **test suite** contains a set of tests, which contain assertion statements. An **assertion** is a condition we expect to be true at a certain time if the program is working correctly. If all assertions succeed, the program is working correctly. If any assertion fails, we know the program is not working correctly and can isolate the problem.

The test harness is written in Python and requires the `pyserial` library to be installed.

3.2.1 Running the Test Harness

The test harness Python script is located in `lib-common/bin/harness.py`.

The command to run the test harness is:

```
$ python harness.py -p <port> -d tests
```

IMPORTANT NOTES:

You must replace `<port>` with the port the programmer is connected to (what you normally use to upload programs), for example `$ python harness.py -p /dev/tty.usbmodem00208212 -d tests` (Mac) or `$ python harness.py -p COM3 -d tests` (Windows).

You may have to replace `harness.py` with the path to the Python script, such as `bin/harness.py` if you are running from the `lib-common` folder.

You may have to replace `tests` with the path to the folder of your test files.

If you are running tests that use 2 boards (e.g. CAN), you must specify a second port with a space after the first port, e.g. `$ python harness.py -p /dev/tty.usbmodem00208212 /dev/tty.usbmodem00187462 -d tests`.

3.2.2 Useful Links

Pull request for test harness development for more details.

Available assertion commnads

3.3 Our Toolchain

Developing code for a real-time embedded system like HERON Mk II as part of a large, interdisciplinary team of programmers is no easy task, and we have collected a set of tools to make the process as effective as possible. To make the software we develop reliable and sustainable, it is crucial to properly set up, use, and understand the tools described in this article.

3.3.1 Background

The ATmega32M1 microcontroller we use is a part of the Atmel AVR family. Atmel is the company that produces them (and is now owned by Microchip), and AVR refers to the architecture and the instruction set used in these microcontrollers (MCU for short). As discussed earlier, in order to program these devices, we need to translate the C code we write into machine code through a *Compiler*. The compiled machine code is then loaded to the AVR device through a protocol called *In-Circuit Serial Programming*, ISP for short, which is just a fancy name for "program your device with just 6 wires so you don't have to rip it out of your circuit board every time you want to change a line of code". This is a standard feature nowadays, but at some point in the past "burning" a program onto your MCU was a much harder, non-reversible process.

Often, the process of writing code, compiling it for the specific MCU that you are using, and loading that program onto the MCU is done through what is called an *Integrated Development Environment*, or an IDE. Atmel has a nice IDE for its own microcontrollers, named *Atmel Studio*, and it integrates and simplifies the process a fair bit. However, for HERON Mk II, we've decided to move to a more bare-bones approach that lets you see a little deeper into the inner workings of the process. This approach also allows us to keep better track of the *libraries* that we use in our code. Libraries, which are pre-written, self-contained pieces of useful code that can be reused, make up most of the code in any embedded system, and we have a few that we have written that you will definitely end up using if you write any code for HERON.

We call the set of tools we use to develop and program our MCUs our *toolchain*, and I'll give a brief overview of what every part of it is about.

3.3.2 Our AVR Toolchain

From the point of view of the programmer, there are three steps to get some code on your MCU. For the sake of simplicity, let's call them the following:

- **Code**: Use any text editor you want on your computer to create the `.c` and `.h` files and make use of libraries.
- **Make**: Use a compiler to compile your code into machine code
- **Upload**: Use a *programmer*, a small circuit board with wires on either end, to connect your computer to your MCU and to upload the program

Once you've completed the setup of the environment, this whole process will be as simple as saving your code, and typing the following into your command line:

```
make upload
```

The rest of this section describes these steps in more detail, and explains how they apply to our specific project.

Coding with Libraries

Libraries are collections of well-tested, useful, and nicely documented code that can be *included* in many projects. They can have many different purposes, including:

- provide easy access to basic functions, such as "wait 50 ms", "turn pin PB5 on" or "allocate 8 bits of memory for this integer"
- allow easy use of complex features of your microcontroller, such as the SPI or CAN data transfer protocols
- let you use an external device (a _peripheral_), such as writing data to an SD card or running a motor.

There are three sources of libraries that we use:

1. **Standard C Libraries**: a set of libraries that can be used in any given C program running on any device. Functions from these libraries are used so often that you don't think about them much. Math functions, variable types and sizes, floating point numbers, strings are all dealt with here.
2. **avr-libc**: a library created specifically for writing C programs for AVR microcontrollers. It lets you do things that are specific to the microcontroler you are using. For example, it can let you read/write to the I/O pins on your microcontroller; it can allow you to count a certain number of milliseconds to have accurate delays; or it can provide access to certain power-saving functions.
3. **lib-common**: our constantly-evolving, home-made library just for HERON Mk II. Here we have functions that are used in all of the different subsystems on the satellite, protocols that allow the subsystems to communicate, and drivers for peripherals that are used on the satellite.

Since libraries typically make up most of a project, it is not convenient to store them as `.c` and `.h` files with the rest of your code. If you were to do that, they would need to be recompiled every time you make a small change in your main code, which would take ages. Instead, they are pre-compiled and stored in `.a` files, which you can see in `lib-common/lib/` in the link above. This is why later on, once you have completed setting up the toolchain, you will not be able to compile one of the subsystem projects until after you have compiled the contents of lib-common.

Any library can be used in your code as long as it is accessible to the compiler, however it needs to be included properly. The snippet below shows an example of how to include various libraries and other project files in your main code (taken from obc.h in the On-Board Computer code). Note how the angle brackets are used for the libraries, and double quotes are used for the project files - can you guess why?

```
// standard C libraries
#include      <stdbool.h>
#include      <stdint.h>

// avr-libc includes
#include     <avr/io.h>
#include     <util/delay.h>

// lib-common includes
#include     <spi/spi.h>
#include     <uart/uart.h>
#include     <uart/log.h>
#include     <can/can.h>

// project file includes
#include     "rtc.h"
#include     "mem.h"
```

3.4 lib-common

`lib-common` is a repository of common software used in multiple subsystems of the satellite. It has its own GitHub repository for development (`lib-common`), and is embedded in other subsystem repositories as a Git submodule so it can be used in those repositories.

A submodule is code from one repository that is embedded in another repository to be used. We use this to embed code from the `lib-common` repository in other repositories such as `pay`.

3.4.1 Update lib-common

To update the `lib-common` submodule to the latest code from the `lib-common` repository, run this command from your other repository's root:

```
$ git submodule update --remote
```

You must then recompile `lib-common`:

```
$ cd lib-common
$ make
$ cd ..
```

3.5 Switch lib-common to a Different Branch

Normally, when you run `$ git submodule update --remote`, it fetches the latest version of `lib-common` from `master`. You can get Git to track a different branch of `lib-common` by modifying the `.gitmodules` file. If you open it, you will see something like this:

```
[submodule "lib-common"]
        path = lib-common
        url = https://github.com/HeronMkII/lib-common.git
```

Add the following line to track a different branch:

```
branch = <branch name>
```

For example, if you want to use the branch `flash-dev`, your `.gitmodules` file will look like this:

```
[submodule "lib-common"]
        path = lib-common
        url = https://github.com/HeronMkII/lib-common.git
        branch = flash-dev
```

3.6 UART Terminal

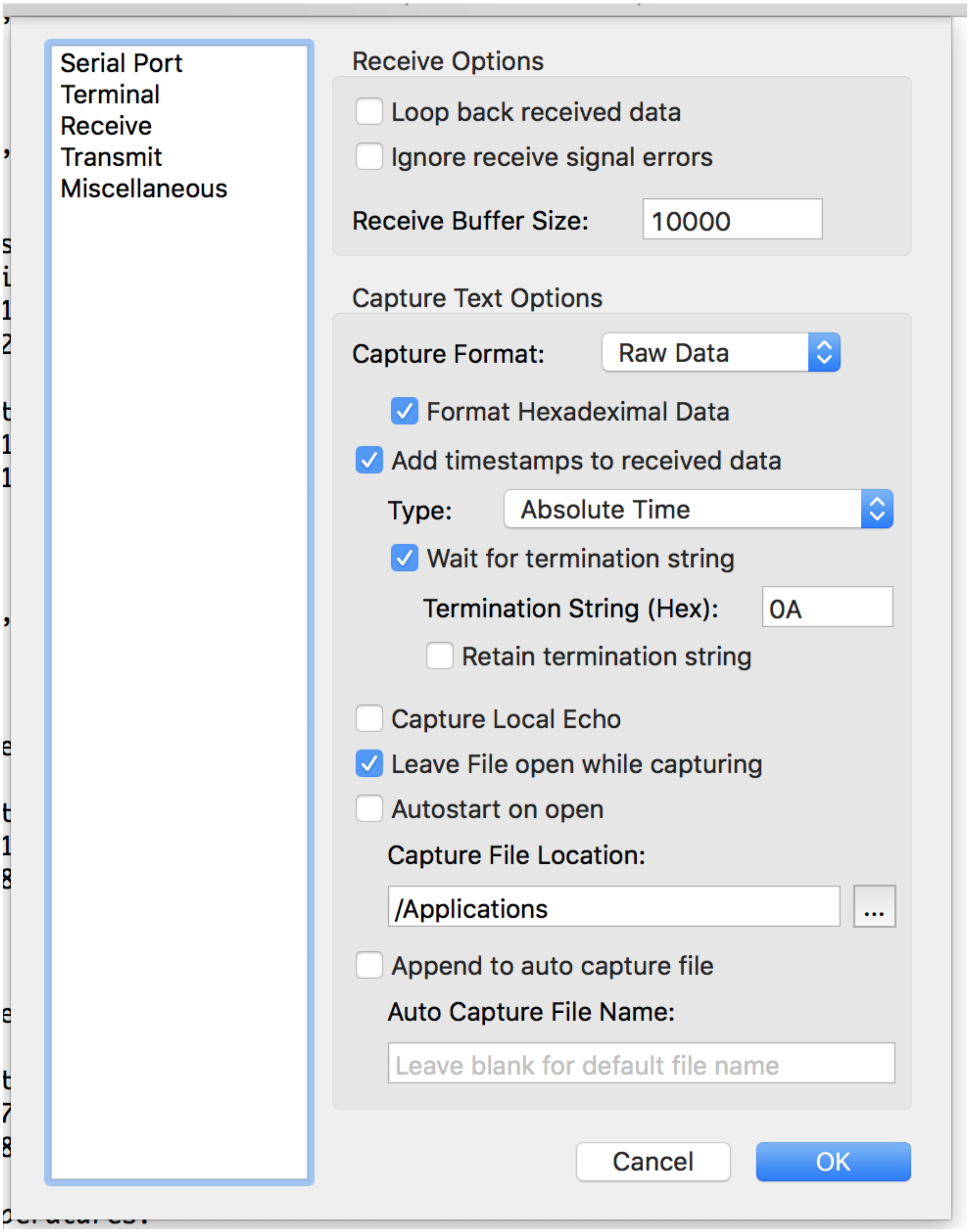
The following settings will probably work:

- Baud Rate: 9600
- Data: 8 bit
- Parity: None
- Stop: 1 bit
- Flow control: none

3.7 CoolTerm Capture

You can capture the microcontroller's UART output using CoolTerm and save it to a file.

Open CoolTerm, click Options at the top, then click Receive on the left side. Change the settings as follows (making any modifications as necessary), then click OK.



Receive Options

- [ ] Loop back received data
- [ ] Ignore receive signal errors

Receive Buffer Size: 10000

Capture Text Options

Capture Format: Raw Data

- [✓] Format Hexadeximal Data
- [✓] Add timestamps to received data

Type: Absolute Time

- [✓] Wait for termination string

Termination String (Hex): 0A

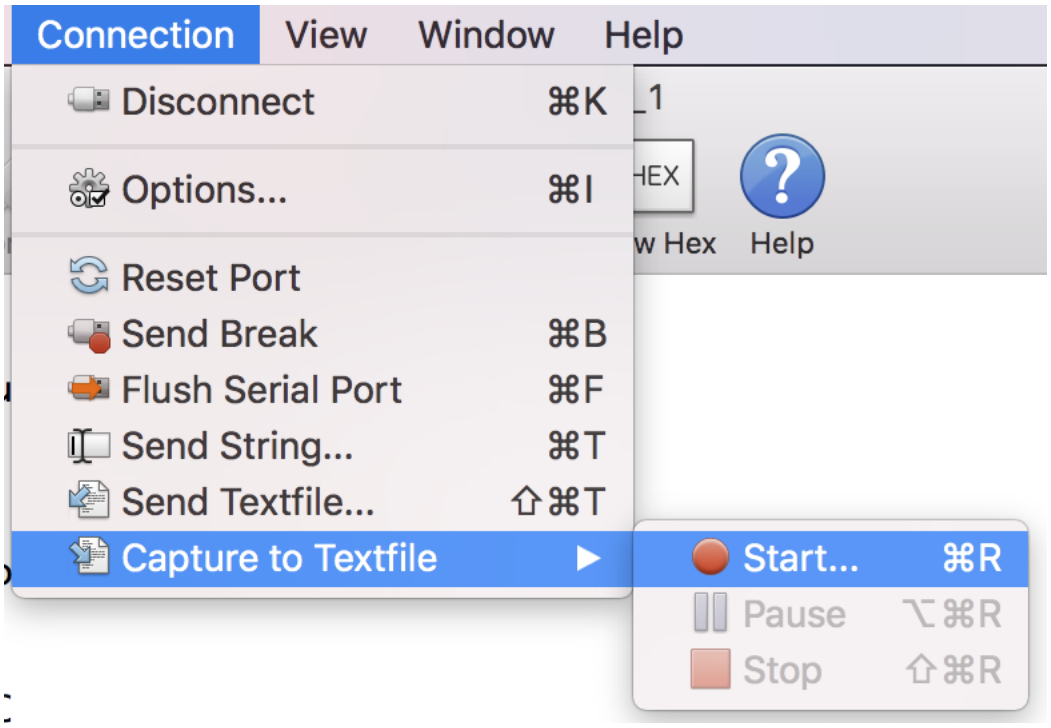
- [ ] Retain termination string
- [ ] Capture Local Echo
- [✓] Leave File open while capturing
- [ ] Autostart on open

Capture File Location: /Applications

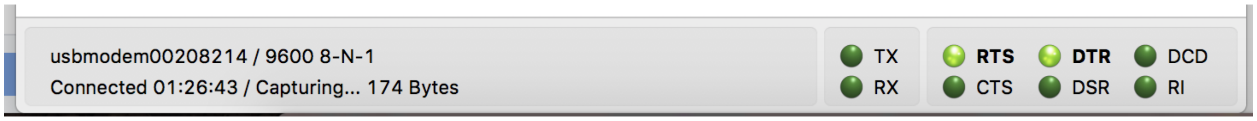
- [ ] Append to auto capture file

Auto Capture File Name: Leave blank for default file name

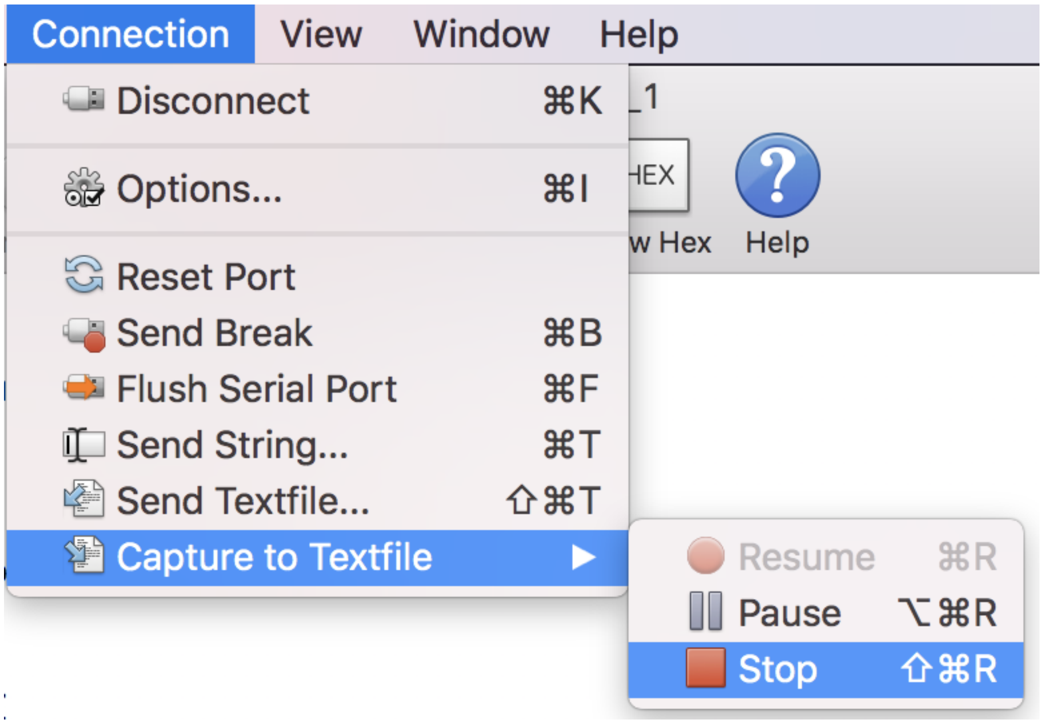
Click on Connection > Capture to Textfile > Start, and choose the file name and destination to save the data.



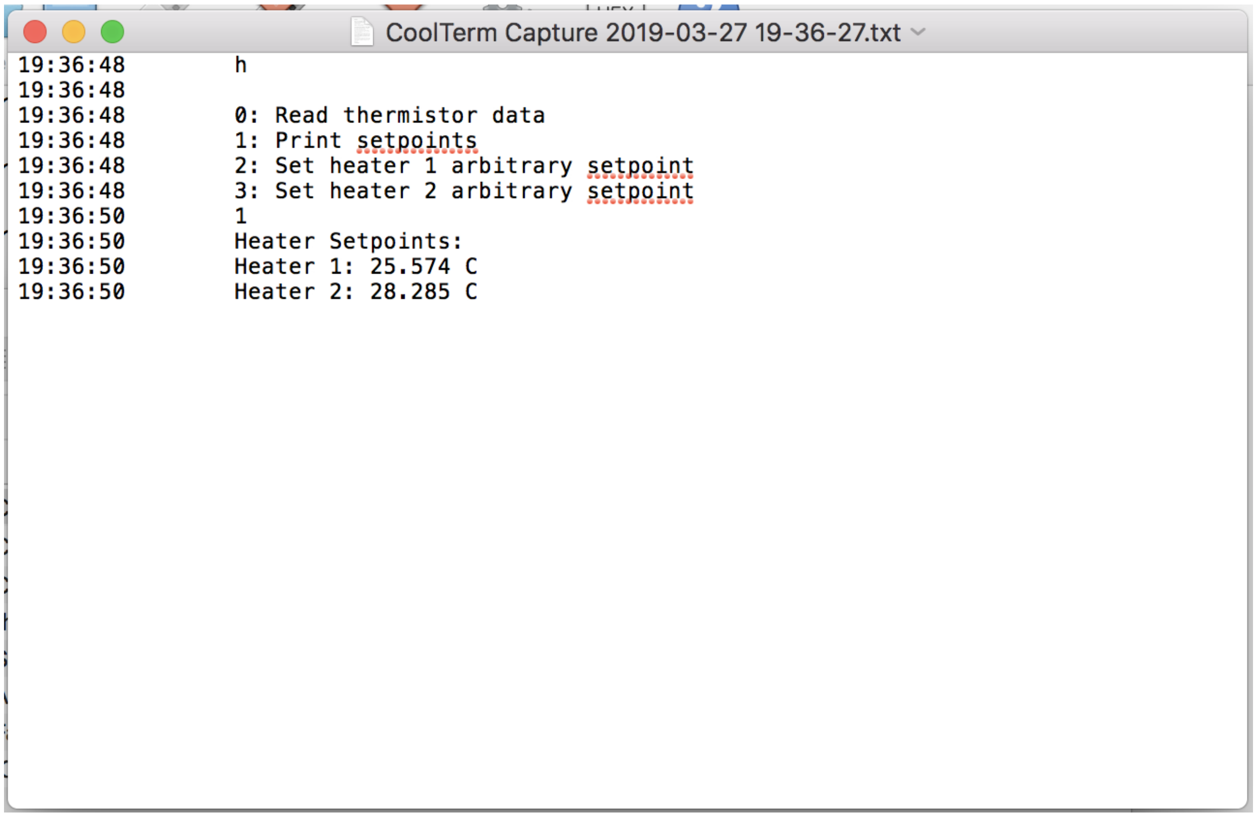
Recording is now started. Run the program and collect your data. The bottom of the window should say "Capturing…".


usbmodem00208214 / 9600 8-N-1
Connected 01:26:43 / Capturing... 174 Bytes

When finished running your program and collecting data, click on Connection > Capture to Textfile > Stop.



Open your file of output data, which should look like this.



```
19:36:48        h
19:36:48
19:36:48        0: Read thermistor data
19:36:48        1: Print setpoints
19:36:48        2: Set heater 1 arbitrary setpoint
19:36:48        3: Set heater 2 arbitrary setpoint
19:36:50        1
19:36:50        Heater Setpoints:
19:36:50        Heater 1: 25.574 C
19:36:50        Heater 2: 28.285 C
```

3.8 Debugging Programs Atmel-Studio

Using the Atmel ICE debugging board, we can step through and debug programs. Note that setting breakpoints and inspecting variables only works if you use the Atmel ICE board instead of the normal programmers.

You will need to install 'Atmel Studio' (Windows only):

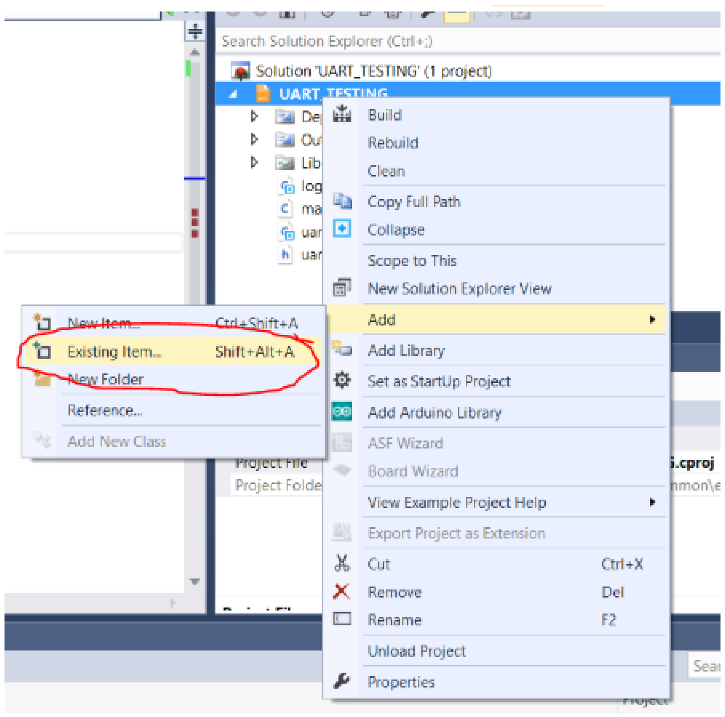
When programming, you must first setup and check everything is detected by going to 'Tools > Device Programming'. You must change the interface to use 'debugWire' instead of ISP.

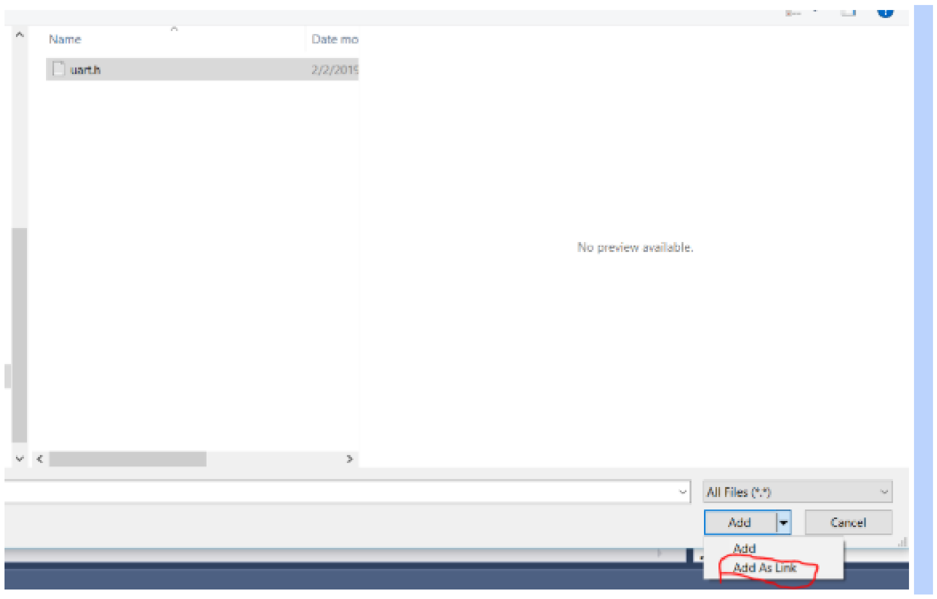
When you are done with your debugging, always remember to disable debugWire mode by changing the interface to ISP (and clicking Apply). You should disable the debugWire and close by going to the 'debug' heading. If you do not disable debugWire, the board will not work with normal programmers.

When prompted, toggle the power (remove and reinsert vcc) to set the fuses.

Some notes: You may need to run Atmel Studio as administrator to get programs to compile and build. If the program isn't running as expected, it may be due to compiler optmizations.

Create a new project for the program you want to debug. Currently, to include other header and source files, you will have to manually link each and every one. To do this, add each file to the project as a link (this includes all files in lib-common that your program uses):





Note that when using Atmel Studio, you need to create an Atmel Studio project that uses an auto-generated makefile instead of our own makefiles. This even applies to lib-common files - we tried compiling lib-common with our own makefiles, but this did not work since we could not set breakpoints or step into code in lib-common, likely due to differences in compiler settings from the Atmel Studio build system.

To run your program in debugWire mode with breakpoints, click the "Debug Program" button instead of the "Run Program" option. Note that you cannot change breakpoints while your program is running, so you need to set them before running or reupload your program.

3.8.1 Breakpoints

Note that when setting breakpoints, it might not function exactly as you expect due to optimizations in the compilation of your program. For example, if you set a breakpoint on a line that modifies a variable whose value is never actually used for anything, the compiler might optimize it out, causing it to never hit that breakpoint or produce strange behaviour.

To be continued…

3.9 Debugging Programs MPLAB X IDE

This is one of the software debugger tools available for Atmega and Atmel ICE on Mac OS.

3.9.1 Installation

To install the IDE, visit this website. Make sure that your installed version is v5.05 or later.

Next, install compiler XC8 for Mac on this website. When prompted in the installation process, set compiler settings to ‘all users of this machine’.

3.9.2 Project Creation

To make a new project, which is required to run code on MPLAB X IDE, follow the steps outlined below

1. Open MPLAB X IDE
2. Click ‘New Project’ in the menu bar
3. You then be allowed to set the settings of the project. The following settings should be selected for the project (ordered by when they are prompted):
    - Microchip embedded
    - Standalone project
    - Family: AVR MCUs
    - Device: ATmega64M1 or Atmega32M1 (depending on which microcontroller you are using)
    - Hardware Tools: Atmel-ICE
    - Compiler: XC8
4. Name your project (you can choose the name)

Your newly created project will be initialized with a couple of folders. The header folder is where the header/.h files are found. The source folder has the src/.c files. The important files contain the makefile for the XC8 compiler (you shouldn’t have to change this folder/file).

On the menu, the hammer icon builds your project (equivalent of make). The downward arrow onto a microcontroller icon build and uploads your project (equivalent of make upload). And the red square on a white sheet with a green arrow is the debug icon.

Side-note: If you have multiple projects, only the main project (bolded) will be uploaded. You can switch main projects by right clicking the project you want to switch to, and clicking ‘Set as Main Project’

3.9.3 Adding Files

The UTAT software files on git need to be manually added. To add a file:

1. Right click on the folder you want to add a file to
2. Click 'add existing file' on the drop-down menu
3. Set store path as: 'Absolute'
4. Find file within your computer
5. Set file format (.c/.h etc..)
6. Click 'Select'

Note: - Add '#include <xc.h>' and '#include <stdint.h>' somewhere all files can access (such as a header file)

If you receive an error about the IDE not finding header files, perform the following steps:

1. Click 'File' on the IDE top menu
2. Click 'Project Properties'
3. Under Conf: [default]
4. Select your compiler (XC8 Compiler)
5. In 'Option categories' dropdown menu choose 'Preprocessing and messages'
6. Click '. . . ' beside 'Include directories'
7. Click 'Browse. . . '
8. Find and select the location of the include. For example, if the include is <uart/uart.h>, whose directory path is lib-common->includes->uart->uart.h. You would include the lib-common/includes path

3.9.4 Debugging

To debug, you must enable the DeBug Wire:

1. Under 'Project Properties', Conf: [default], Atmel-ICE
2. Set 'Option categories' to 'communication'
3. Change interface to 'debugWIRE'
4. Run your program. It will give you a message to configure to debugWIRE, click agree

Remember to disable the DeBug Wire to allow for normal programming when you are finished. To disable the DeBug Wire, do the opposite of the above steps (change debugWIRE to ISP and run)

Print statements

Note that the XC8 compiler has trouble compiling our print statement function. Therefore, you can comment out all print statements (go to atom, Command F all print statements and comment them out with //). If you require print statements, declare another character array variable and set it to what you want to print out (char printVar[] = "print this out"). Then in the print function, pass in that variable (printVar).

3.10 Git Troubleshooting

Sometimes Git can be annoying.

3.10.1 Cannot Unlink File

If you try to pull and get an error saying “cannot unlink file”, this means that Git failed to update a file because it is locked by another program and Git cannot modify it. The solution is to close any programs that might have that file open.

For example with Altium Designer, the program might not appear to be running but it might have a process still running in the background. Open the Task Manager and end the process called “X2.EXE”, which belongs to Altium Designer. Then try to Git pull again, which should work now.

CHAPTER 4

Troubleshooting

This document is to be used as a guide when programming to ensure all critical points for error are considered.

Below is a quick checklist for points to consider when programming.

- [ ] Lib-common, if included, is compiled.
- [ ] Added files compile.
- [ ] 'PORT' in makefile is set to the correct port. See below for details.
- [ ] All necessary lib-common modules are initialized.
- [ ] Build folder is included.
- [ ] CoolTerm / Xterm baud rate is set to 9600. Appropriate port is set. See below for details.
- [ ] 6-pin programming header is connected to the PCB.
- [ ] If using CAN, the CANH and CANL pins are connected between PCBs.
- [ ] In your program, all the necessary components/libraries are initialized.

If the quick checklist has not cornered your error, let us first troubleshoot the software.

4.1 Hardware

4.1.1 Is the LED on the board on?

- [ ] Check power supply, use different channel
- [ ] Change wires

4.1.2 Is UART working? Can you see your print statements on CoolTerm / Xterm?

- [ ] Check that you have clicked on the Connect button

- [ ] Click on the Options button and ensure your port is correct. This port should not be the same as the port in your makefile.
- [ ] Check the baud rate (use 9800)
- [ ] Ensure the UART wire is connected to MOSI pin on the board and to the RX pin on the Pololu programmer

4.2 Software

Read compiler warnings!!

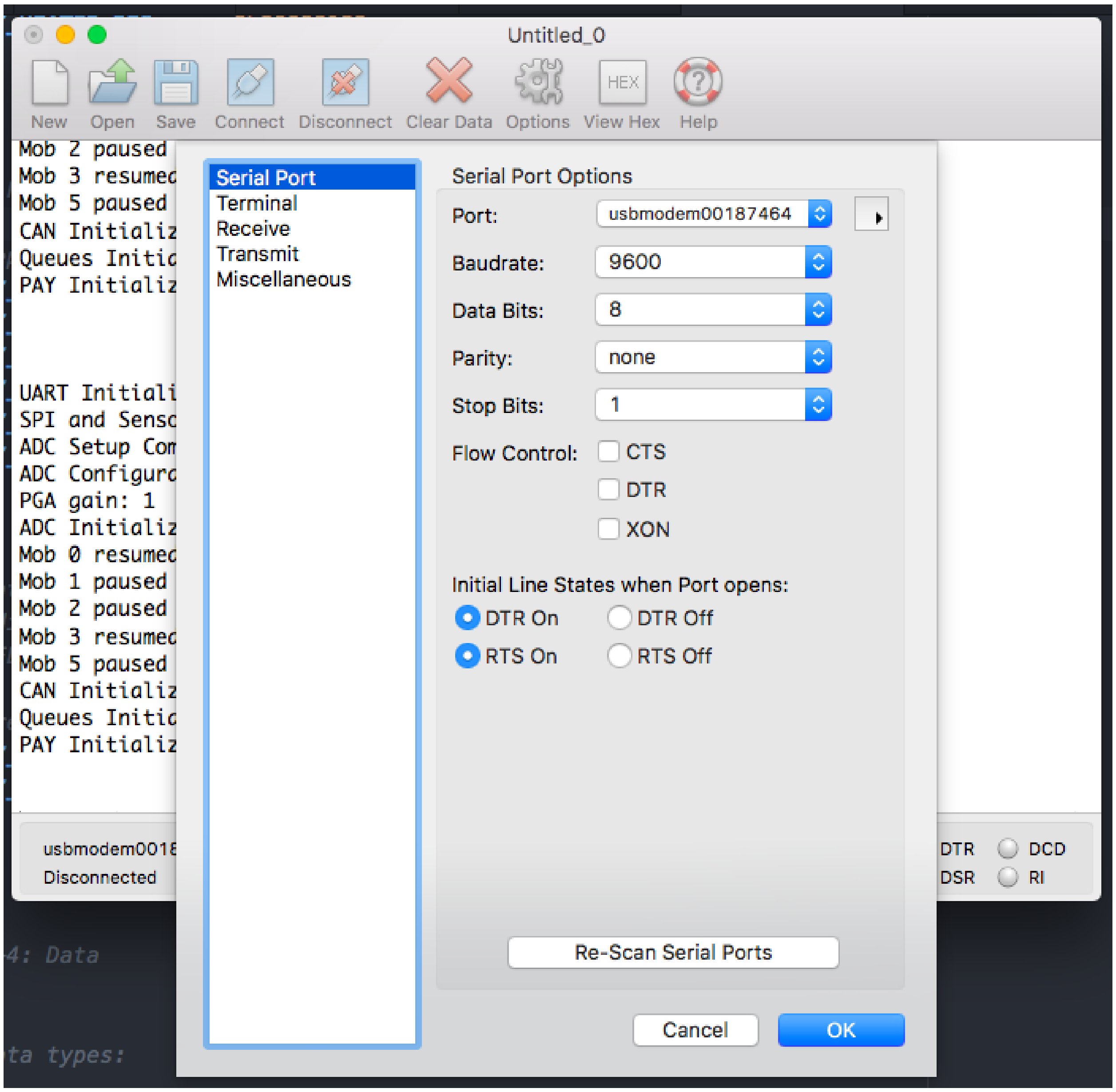
4.2.1 Does AVR say completed and uploaded to board?

- [ ] Check your port on the makefile. (For macOS use ls /dev/tty. *), for Windows use device manager)
- [ ] Change programmer micro-usb
- [ ] Change programmer
- [ ] Change PCB

4.2.2 Debugging through print statements (like the old times)

- [ ] Add a print statement after init_uart() to see if your program executes on the board
- [ ] Use various print statements throughout the program to see how your program executes (print useful variables)

4.2.3 Appendix - Figures



```
Mob 2 paused
Mob 3 resumed
Mob 5 paused
CAN Initializ
Queues Initia
PAY Initializ


UART Initiali
SPI and Senso
ADC Setup Com
ADC Configura
PGA gain: 1
ADC Initializ
Mob 0 resumed
Mob 1 paused
Mob 2 paused
Mob 3 resumed
Mob 5 paused
CAN Initializ
Queues Initia
PAY Initializ
```

usbmodem0018
Disconnected

4: Data

ta types:

Serial Port Options

Port: usbmodem00187464

Baudrate: 9600

Data Bits: 8

Parity: none

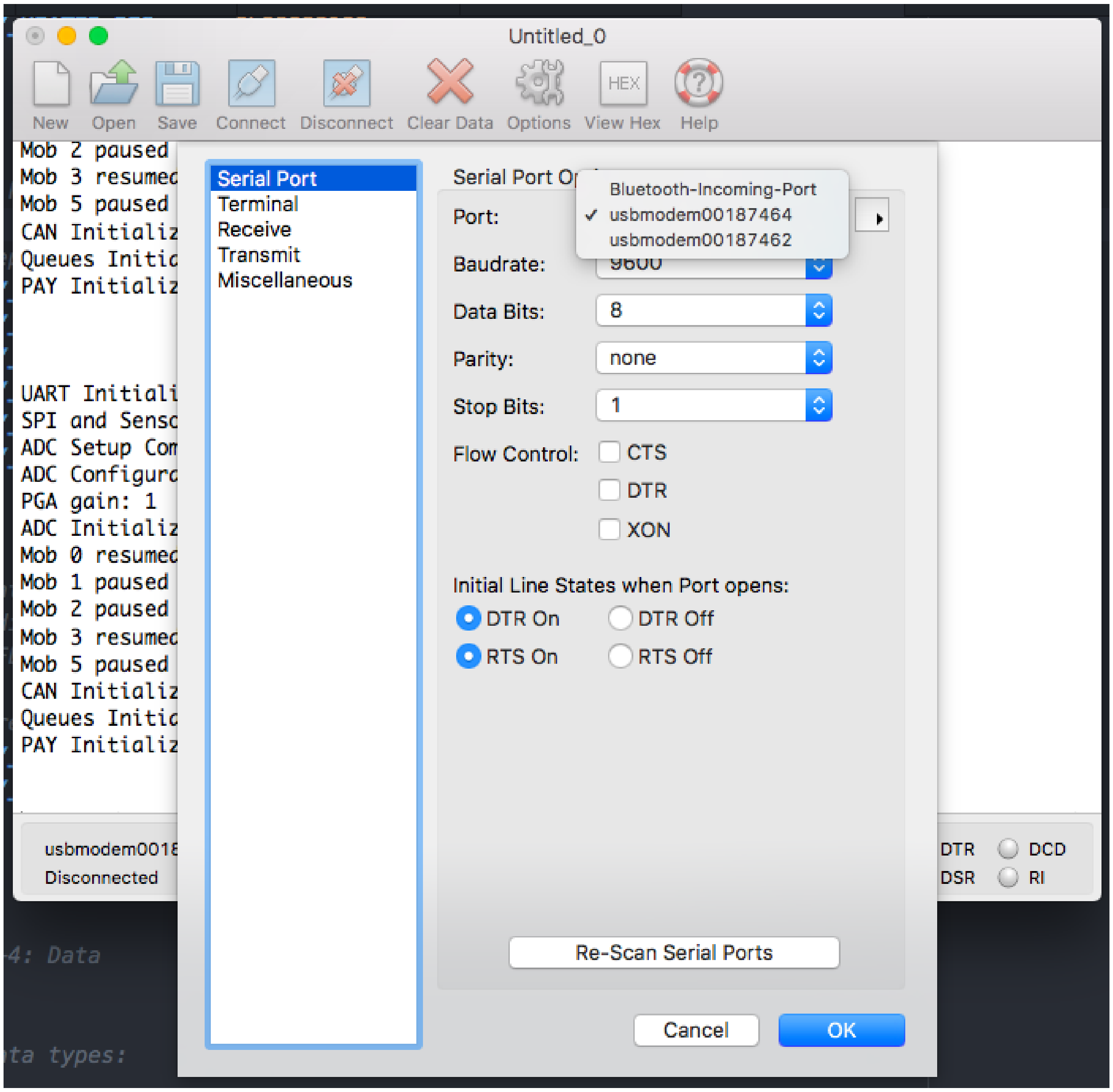
Stop Bits: 1

Flow Control: ☐ CTS ☐ DTR ☐ XON

Initial Line States when Port opens:

[✓ DTR On] DTR Off

[✓ RTS On] RTS Off



```
Mob 2 paused
Mob 3 resumec
Mob 5 paused
CAN Initializ
Queues Initic
PAY Initializ


UART Initiali
SPI and Sensc
ADC Setup Com
ADC Configura
PGA gain: 1
ADC Initializ
Mob 0 resumec
Mob 1 paused
Mob 2 paused
Mob 3 resumec
Mob 5 paused
CAN Initializ
Queues Initic
PAY Initializ
```

usbmodem0018
Disconnected

DTR DCD
DSR RI

Serial Port O

Port:
- Bluetooth-Incoming-Port
- ✓ usbmodem00187464
- usbmodem00187462

Baudrate: 9600

Data Bits: 8

Parity: none

Stop Bits: 1

Flow Control: CTS [ ] DTR [ ] XON [ ]

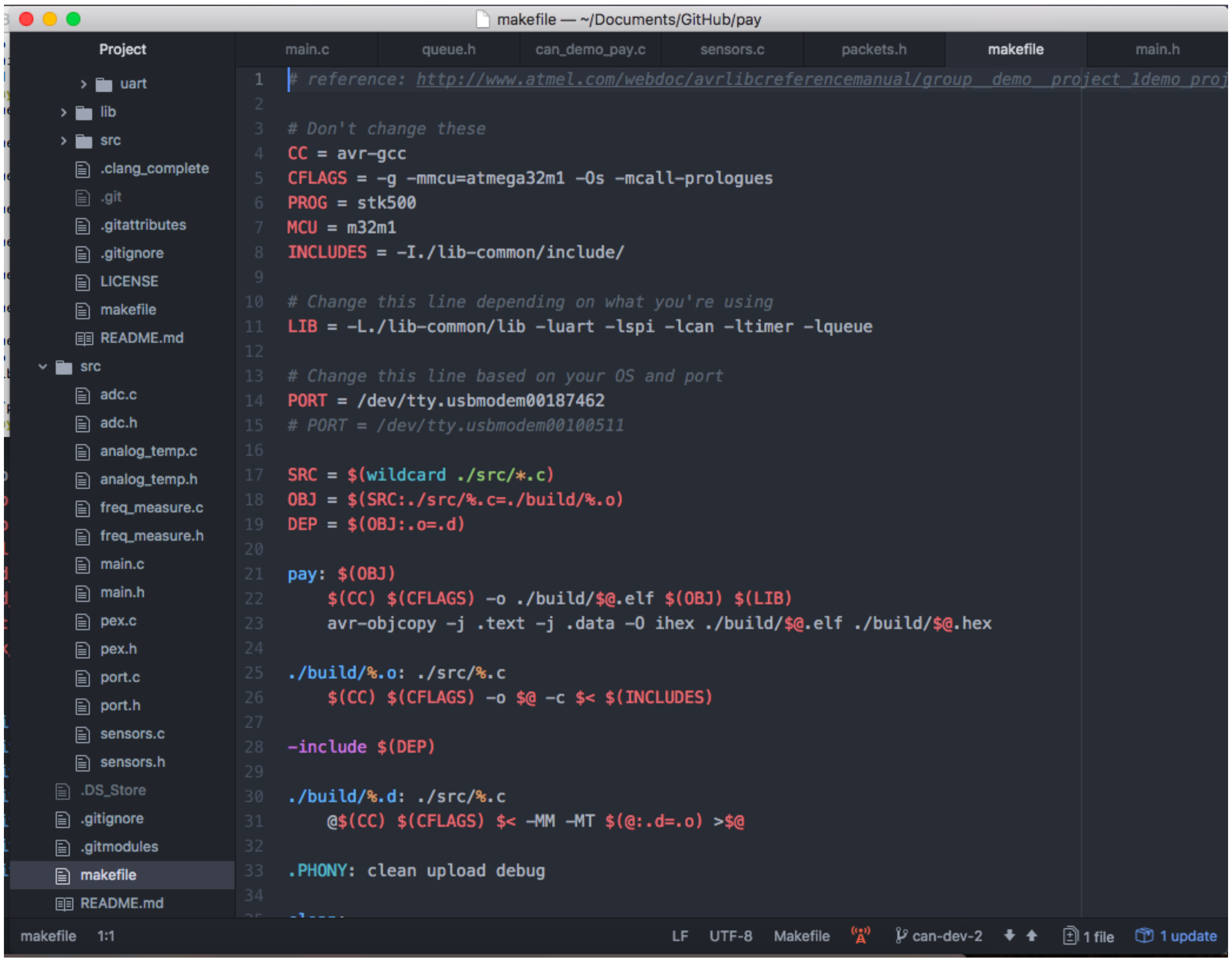
Initial Line States when Port opens:

[✓ DTR On] DTR Off

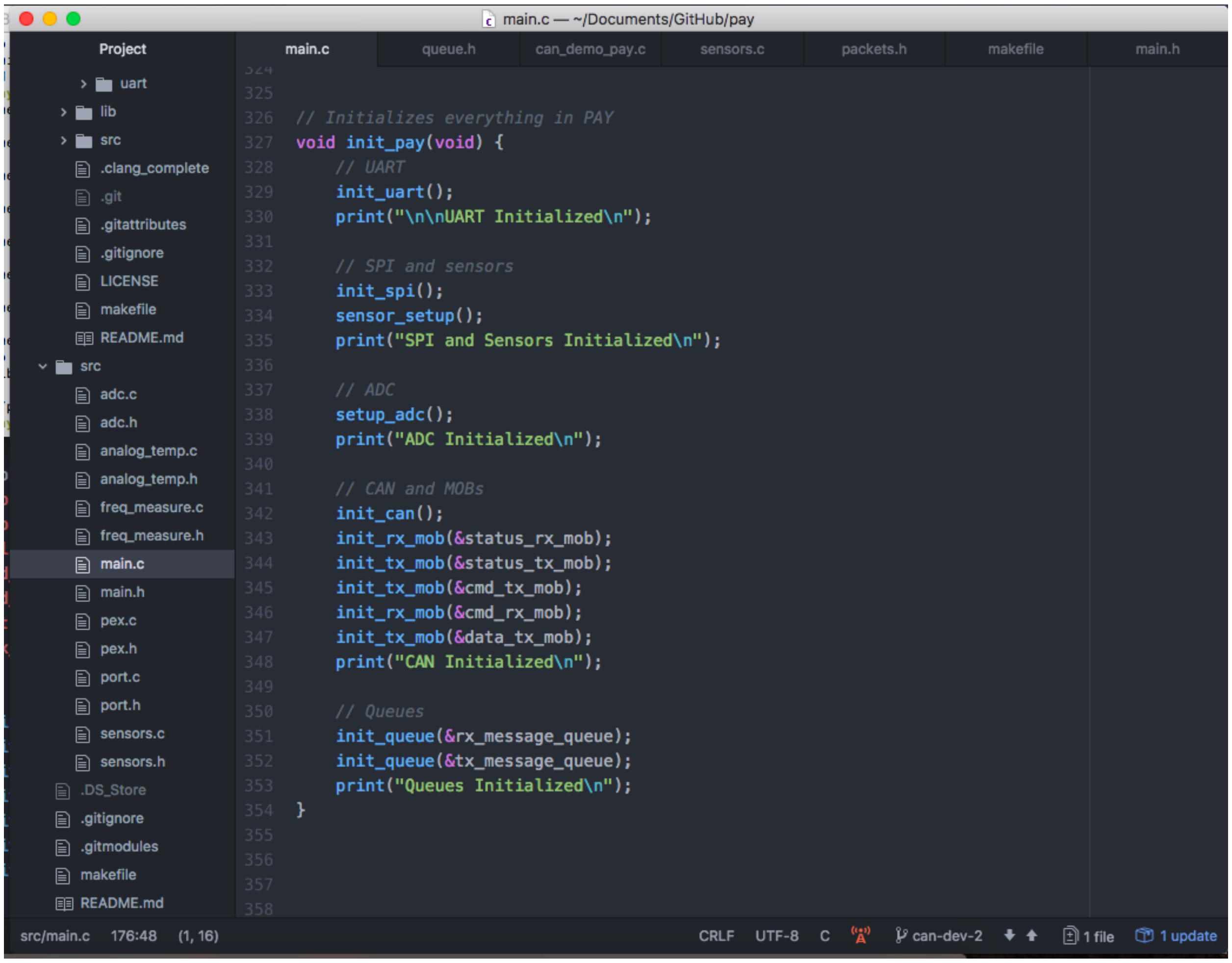
[✓ RTS On] RTS Off

-4: Data

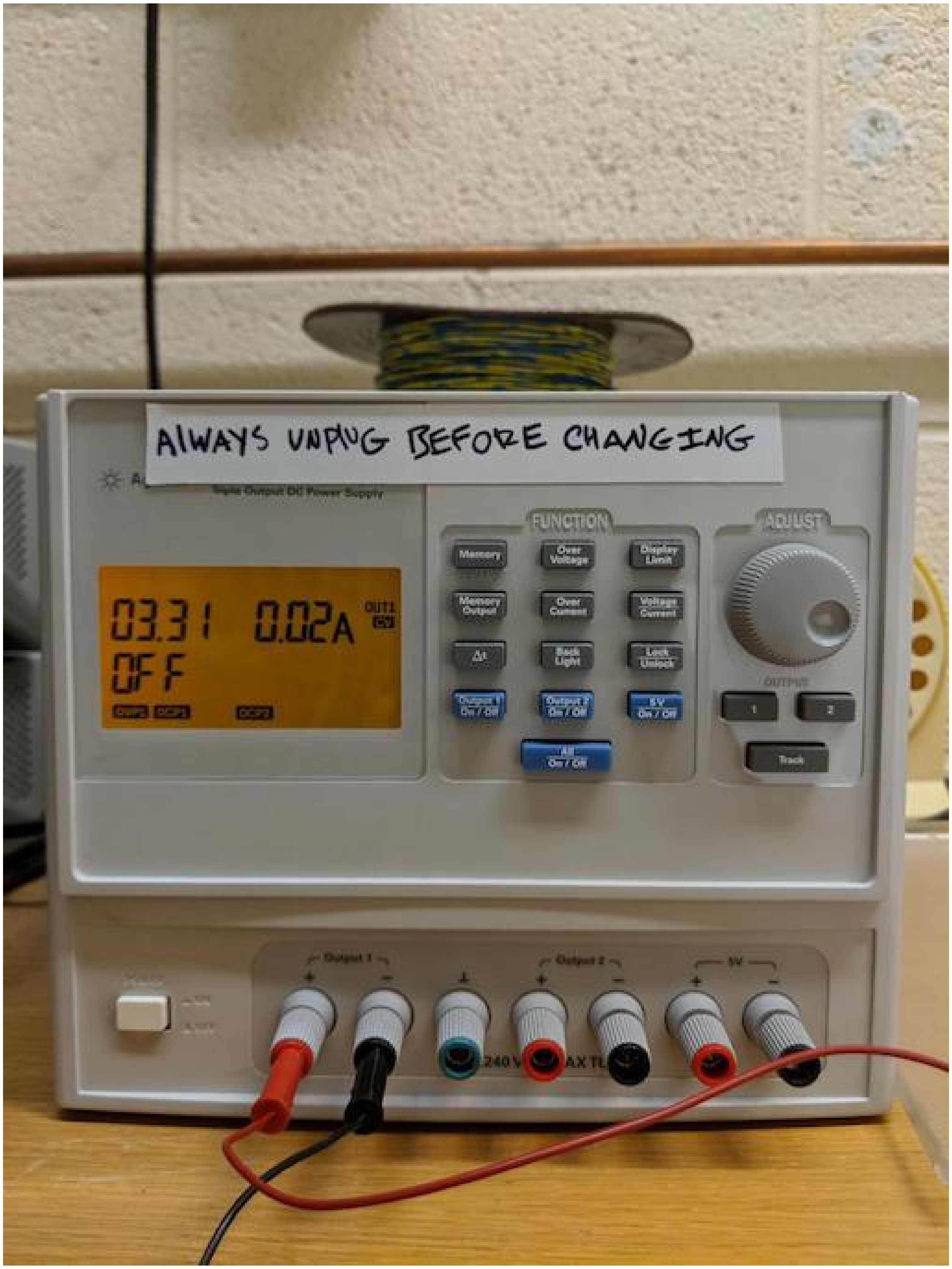
ta types:

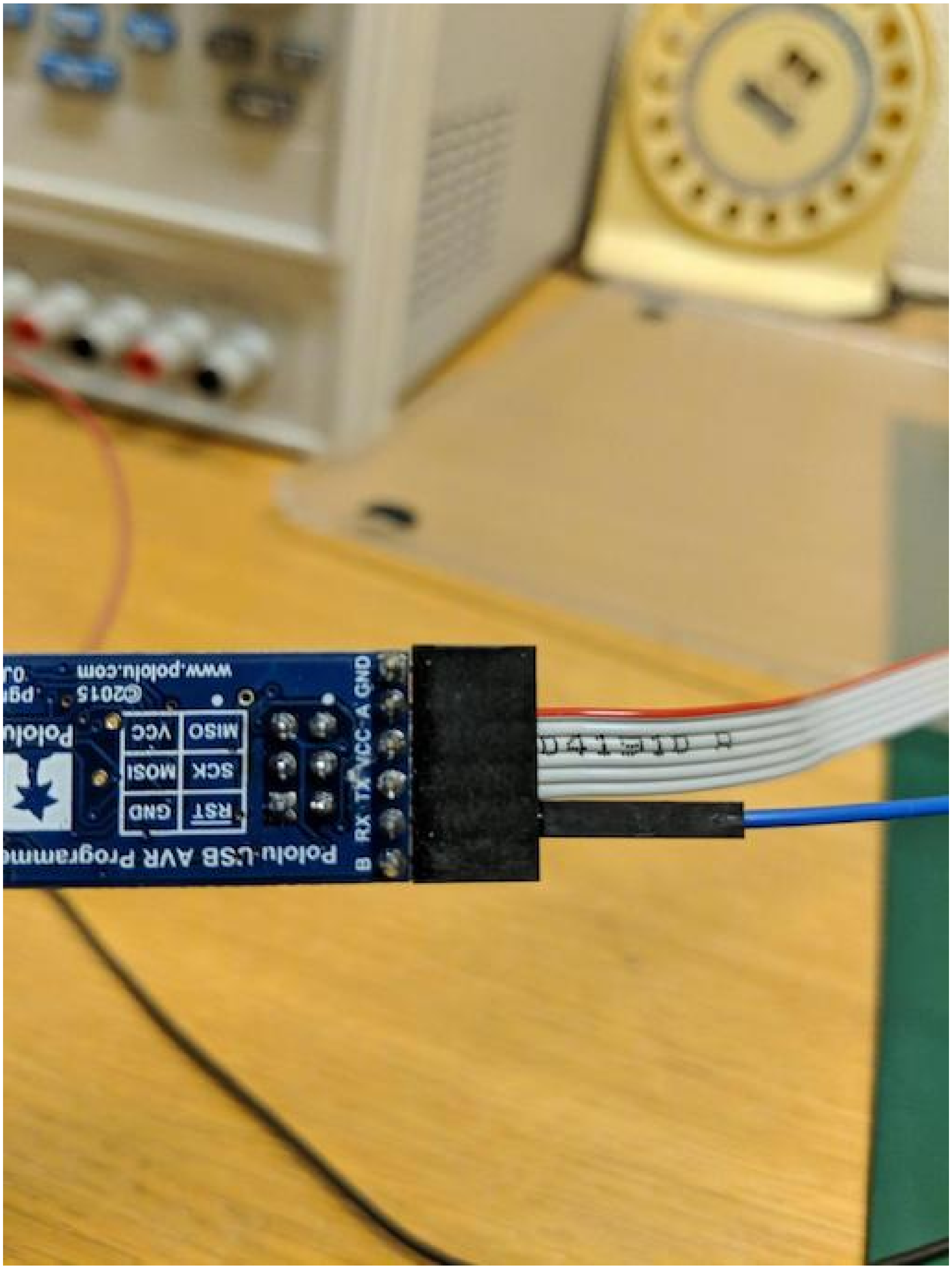


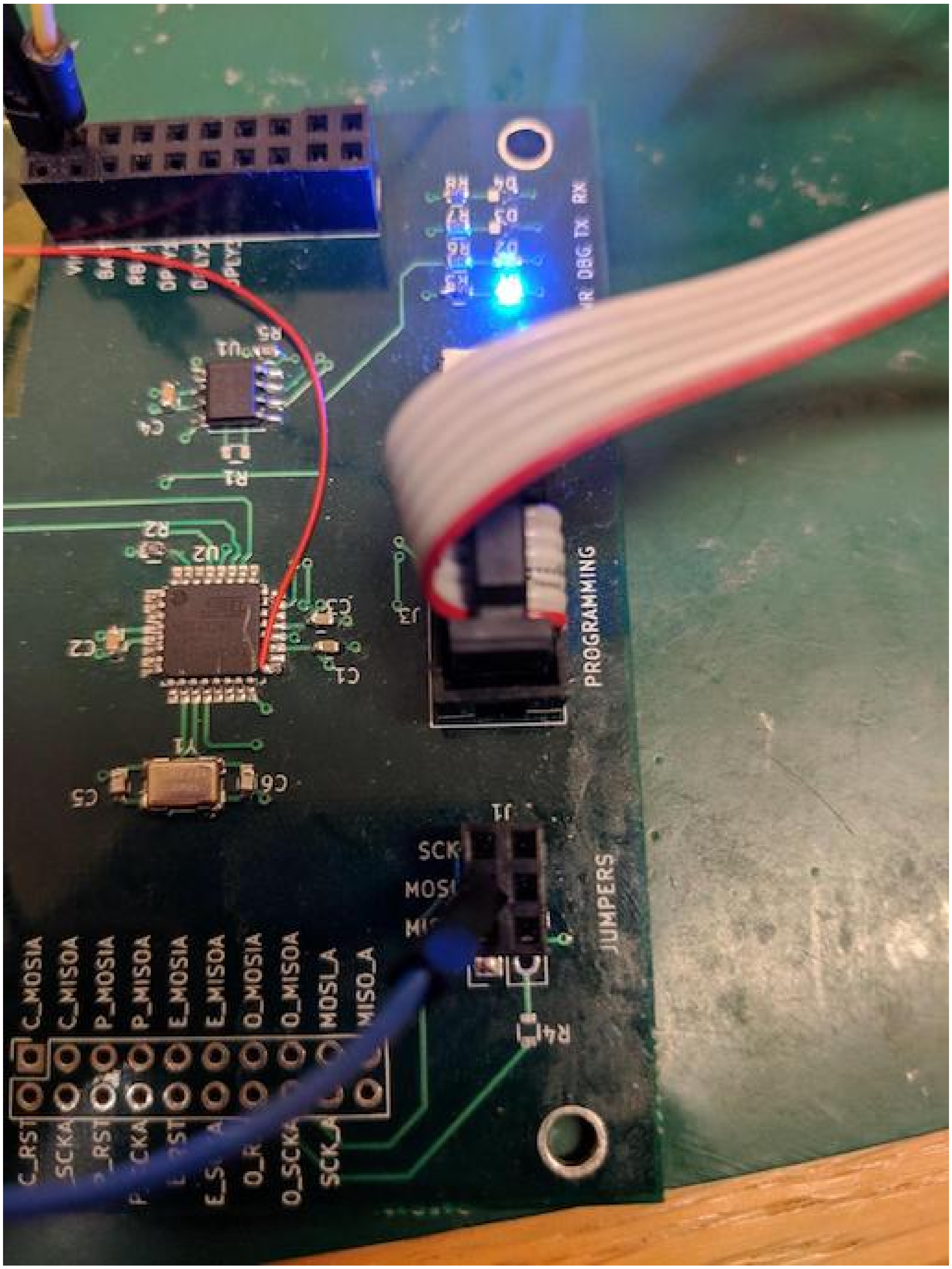
```
# reference: http://www.atmel.com/webdoc/avrlibcreferencemanual/group__demo__project_1demo_proj

# Don't change these
CC = avr-gcc
CFLAGS = -g -mmcu=atmega32m1 -Os -mcall-prologues
PROG = stk500
MCU = m32m1
INCLUDES = -I./lib-common/include/

# Change this line depending on what you're using
LIB = -L./lib-common/lib -luart -lspi -lcan -ltimer -lqueue

# Change this line based on your OS and port
PORT = /dev/tty.usbmodem00187462
# PORT = /dev/tty.usbmodem00100511

SRC = $(wildcard ./src/*.c)
OBJ = $(SRC:./src/%.c=./build/%.o)
DEP = $(OBJ:.o=.d)

pay: $(OBJ)
	$(CC) $(CFLAGS) -o ./build/$@.elf $(OBJ) $(LIB)
	avr-objcopy -j .text -j .data -O ihex ./build/$@.elf ./build/$@.hex

./build/%.o: ./src/%.c
	$(CC) $(CFLAGS) -o $@ -c $< $(INCLUDES)

-include $(DEP)

./build/%.d: ./src/%.c
	@$(CC) $(CFLAGS) $< -MM -MT $(@:.d=.o) >$@

.PHONY: clean upload debug
```



```c

// Initializes everything in PAY
void init_pay(void) {
    // UART
    init_uart();
    print("\n\nUART Initialized\n");

    // SPI and sensors
    init_spi();
    sensor_setup();
    print("SPI and Sensors Initialized\n");

    // ADC
    setup_adc();
    print("ADC Initialized\n");

    // CAN and MOBs
    init_can();
    init_rx_mob(&status_rx_mob);
    init_tx_mob(&status_tx_mob);
    init_tx_mob(&cmd_tx_mob);
    init_rx_mob(&cmd_rx_mob);
    init_tx_mob(&data_tx_mob);
    print("CAN Initialized\n");

    // Queues
    init_queue(&rx_message_queue);
    init_queue(&tx_message_queue);
    print("Queues Initialized\n");
}
```


ALWAYS UNPLUG BEFORE CHANGING
Triple Output DC Power Supply
03.31
0.02A
OFF
OUT1
CV
FUNCTION
ADJUST
Memory
Over Voltage
Display Limit
Memory Output
Over Current
Voltage Current
Δt
Back Light
Lock Unlock
Output 1 On / Off
Output 2 On / Off
5 V On / Off
All On / Off
OUTPUT
1
2
Track
Output 1
Output 2
5V
+
−
+
−
+
−


Pololu USB AVR Programmer
www.pololu.com
©2015
B
RX
TX
VCC-A
GND
MISO
SCK
RST
VCC
MOSI
GND


PROGRAMMING
JUMPERS
SCK
MOSI
C_MOSIA
C_MISOA
P_MOSIA
P_MISOA
E_MOSIA
E_MISOA
O_MOSIA
O_MISOA
MOSI_A
MISO_A
C_RST
SCKA
O_SCKA
SCK_A
PWR DBG TX RX

CHAPTER 5

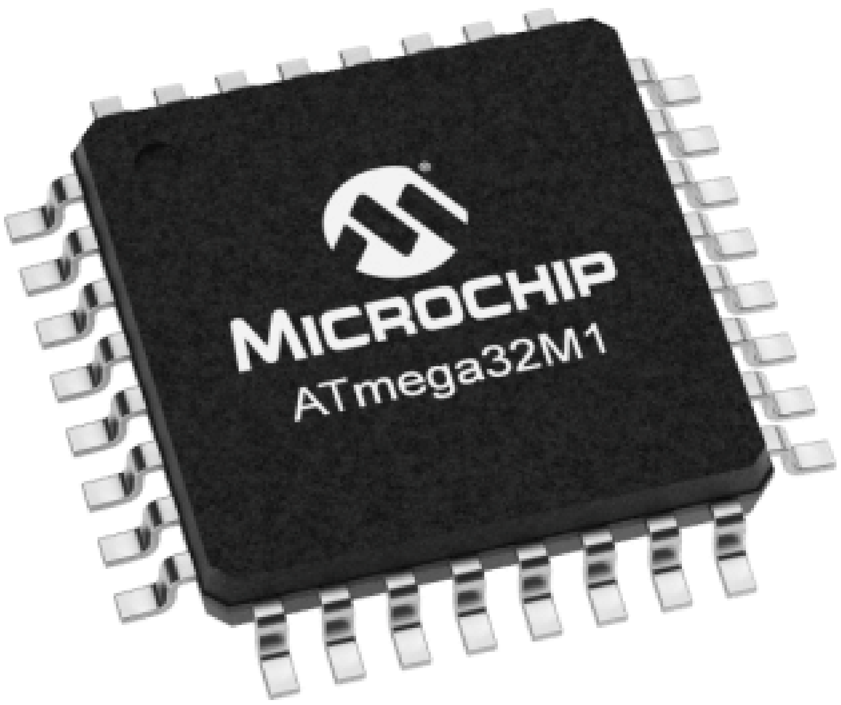
Embedded Systems

5.1 Microcontrollers

A microcontroller is a processing unit, like the brain of a circuit. It is slightly different from the microprocessor in your computer because it also has other components like RAM and ROM integrated on the same chip. A microprocessor just has the processing unit.

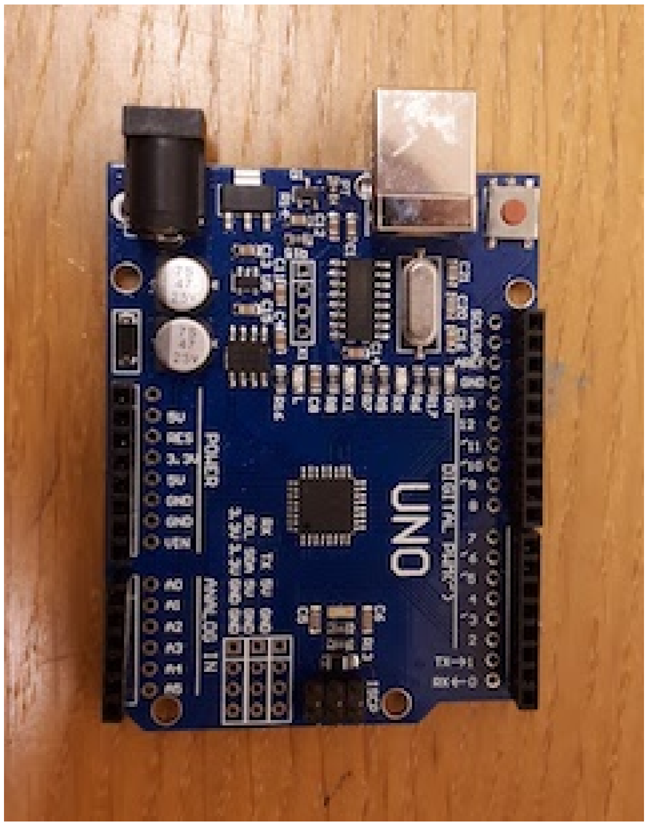
C is the most common language for programming microcontrollers. This is because of its wide availability and compatibility between platforms, low performance and memory overhead, and high level of control for the programmer. This high level of control for the programmer also creates more potential for mistakes, so be careful.

5.1.1 ATmega32M1 (AVR)



Our subsystem's components will be controlled by the ATmega32M1 microcontroller on the satellite, part of the AVR family of microcontrollers. It is programmed in the C language, and you will need to install the AVR software to compile and upload code to it (which we will help you to get set up).

5.1.2 Arduino



Arduino is an open-source platform of microcontrollers, which we occasionally use to test components and prototype software quickly. Since Arduino has more built-in code libraries, it is faster to write and upload code to test individual components than using the AVR microcontroller. Arduino is programmed using a slightly modified version of the C++ language.

5.2 Registers

A **register** is a small piece of memory in a digital device. Each register is generally used to control settings and behaviours of the device. Most registers are 8 bits, but some can be 16 bits, 24 bits, or more.

For the 32M1, you write to its registers directly using built-in AVR constants. For SPI devices, you can read and write register values by sending specific SPI messages as described in the component's datasheet. Descriptions of registers are generally one of the most important things to look for in a component's datasheet.

5.3 Interrupts

An interrupt is an event where specialized hardware in the microcontroller (MCU) is triggered to immediately notify it of an important event. For example, a sensor could trigger an interrupt when new data is ready, or a CAN communication transceiver could trigger an interrupt when it receives a message.

When an interrupt occurs, the MCU immediately stops executing the main program and starts executing a piece of code called an ISR (interrupt service routine). When the ISR finishes running, the MCU returns to executing the main program. It automatically preserves the values of local variables inside of your functions, so you don't need to worry about those values changing.

Generally you do not need to deal with interrupts directly, but you need to be aware that interrupts could pause your program at any point. Local variable data will always be preserved, but you need to be careful if using global variables. If an ISR modifies the value of a global variable, it could occur at any point without your code knowing.

5.3.1 Handling Interrupts

Here is an example of an interrupt handler in our codebase:

```
ISR(CAN_INT_vect){
    print("Interrupt received\n");
    for (uint8_t i = 0; i < 6; i++) {
        mob_t* mob = mob_array[i];

    ...
}
```

An ISR (interupt service routine) is automatically triggered by the 32M1 when a particular kind of interrupt occurs in the hardware. In this case, this code runs when the `CAN_INT` interrupt occurs. See the 32M1 datasheet for other kinds of interrupts.

5.4 Brownout Detection

Microchip - AVR Brownout Detection

https://www.embedded.com/electronics-blogs/break-points/4430317/Brown-Out-Reset

https://www.reddit.com/r/AskElectronics/comments/3qbu2q/voltage_drop_rebooting_my_microcontroller_when_i/

5.5 AVR Fuses

The microcontroller contains fuses (fuse bits) that control settings such as the clock speed.

We use the same fuse bits for all microcontrollers and you should never change them (except when setting up the MCU on a new PCB for the first time).

You can get help with calculating fuse bit configurations and what they mean at http://www.engbedded.com/fusecalc/.

5.5.1 Our Fuse Bit settings

The default fuse bits on the 64M1 from the factory are `62 6F FF` (in hex).

We always use the fuse bits `F9 D7 FF` (in hex).

Key Settings:

- BODLEVEL2 = 0, BODLEVEL1 = 0, BODLEVEL0 = 0 - brownout detection at 2.6 V - see datasheet p. 37, 62, 67-69, 366, 377-378, 399, 411 - want this for flash/EEPROM protection, although it consumes some power for the BOD circuit

5.5.2 Setting Fuse Bits

If you need to set the fuse bits, `cd` to the `lib-common` directory and run:

```
$ avrdude -U efuse:w:0xf9:m -U hfuse:w:0xd7:m -U lfuse:w:0xff:m -c stk500 -C
avrdude.conf -p m64m1 -P <port>
```

You must replace `<port>` with the port the programmer is connected to (what you normally use to upload programs), such as `/dev/tty.usbmodem00208212` (on Mac, remember the programming port number normally ends in 2) or `COM3` (on Windows).

NOTE: You must download the Pololu USB AVR Programmer v2 Configuration Utility application on your laptop. You must check the frequency of your programmer before running the fuse configuration command. We normally have ISP Frequency set to 1714kHz, the highest possible value that our programmer supports, so that it uploads programs faster. However, this will not work for a new microcontroller because it uses a much slower clock by default. Make sure to change the ISP Frequency to 114kHz (and click Apply Settings) before running the fuse command. After the fuses are configured, change the ISP Frequency back to 1714kHz (and click Apply Settings).

The reason we use `-C avrdude.conf` is because the default `avrdude.conf` file in the avrdude installation does not support the 64M1, so we created a modified `avrdude.conf` file, put it in the `lib-common` repository, and tell avrdude to use that configuration file instead.

If you are using the 32M1, replace `m64m1` with `m32m1`.

5.6 Uploading Code (Mac)

These instructions may be a duplicate of previous instructions. These instructions have not been thoroughly checked yet.

Here is how to upload your code to the AVR ATmega32M1 microcontroller.

1. Follow the instructions at https://github.com/HeronMkII/coms-board to download and install the AVR software.
2. Download the CoolTerm application from http://freeware.the-meiers.org. This will be used to view transmissions from UART, a communication protocol to transmit log messages (from the print() function) from the board to the computer.
3. Get the programmer hardware device. Connect the 6-pin connector to the "Programming" header on the PCB, which is used to upload code. Connect the "RX" pin on the programmer to the "MOSI_A" pin on the PCB. RX refers to the "receive" pin, while MOSI_A refers to the "alternative" MOSI line, used for UART which is separate from SPI.
4. Turn on the power supply, set the output to 3.3V, and connect the power and ground lines to the "3V3" and "GND" header pins on the PCB.
5. See the instructions at https://github.com/HeronMkII/coms-board for finding the correct USB port and modifying the makefile (it might already be the correct one).
6. Open CoolTerm and modify the options to set the correct port (see https://github.com/HeronMkII/coms-board; the port for UART is the **opposite** of the port for uploading code). Click Connect.
7. Navigate to the folder for the local copy of the Git repository on your computer. Run `make upload` to compile the program and upload it to the board. Fix any compile-time errors if they occur.

CHAPTER 6

Communication Protocols

There are various communication protocols used in the satellite to communicate data between devices. Different protocols have different hardware (communication lines) and transfer data in different formats. Different protocols prioritize different things, such as error checking, minimal hardware wires, or message priorities.

The satellite uses three communication protocols: - CAN - Used to communicate between the 32M1 microcontrollers in different subsystems. - SPI - Used by each 32M1 microcontroller to communicate with sensors and other devices within its subsystem. - UART - Used to communicate between a 32M1 microcontroller and a laptop. The 32M1 can send log messages to the laptop (useful for debugging) or receive keyboard input from the laptop (not used much, but sometimes for controlled testing).

6.1 CAN (Controller Area Network)

CAN is a communication transfer protocol that we use to communicate between 32M1 microcontrollers in different subsystems. CAN transmits **messages**, which can each contain up to 8 bytes of data.

6.1.1 Acronyms

RX - Receiving

TX - Transmitting

MOB/MOb - A message object (like a mailbox) that transmits or receives particular types of messages.

CAN Transceiver - The hardware device connected to each 32M1 that transmits and recives CAN messages on the bus.

CAN Bus - The set of wires connecting all CAN transceivers.

CANH - CAN high (bus wire)

CANL - CAN low (bus wire)

6.1.2 Hardware

Each 32M1 is connected to its own CAN transceiver, which handles transmitting and receiving CAN messages. The CAN bus connects all the CAN transceivers using two wires called CANH and CANL. The signal on the bus is defined as the difference between the CANH and CANL voltages.

6.1.3 MOBs

A MOB (message object) behaves like a mailbox that transmits or receives particular types of messages. It must be designated as either RX (receiving) or TX (transmitting). Each RX MOB has a mask (filter) that determines which messages it receives.

Each MOB is defined as a struct in C.

6.1.4 RX MOBs

Here is an example of an RX mob on PAY that receives commands for requesting sensor data. See later documentation for how these values are assigned.

```
mob_t cmd_rx_mob = {
        .mob_num = 3,                   // MOB number
        .mob_type = RX_MOB,             // MOB type (RX)
        .dlc = 3,                       // Expected length of CAN message (number of␣
↪bytes)
    .id_tag = PAY_CMD_RX_MOB_ID,    // Tag is used to determine which messages to␣
↪receive
        .id_mask = CAN_RX_MASK_ID,      // Mask is a filter that is combined with the␣
↪tag to determine which messages to receive
    .ctrl = default_rx_ctrl,        // Control bits (leave as default)

    .rx_cb = cmd_rx_callback        // The function to be called when this MOB␣
↪receives a message
};
```

If you want the RX MOB to receive all messages, such as for testing, set its mask to 0.

```
.id_mask = { 0x0000 }
```

When the RX MOB receives a CAN message, it will trigger an interrupt and call the function specified in `rx_cb`. The CAN system passes the received data bytes and the length (number of bytes) to your function, which you can process however you want. For example:

```
void cmd_rx_callback(const uint8_t* data, uint8_t len) {
    print("RX Callback\n");

    // Print the bytes in hex (%02x means 2 digits at a time)
    print("Received Message:\n");
    for (uint8_t i = 0; i < len; i++) {
        print("0x%02x ", data[i]);
    }
    print("\n");

    // ...
    // Process the data
}
```

6.1.5 TX MOBs

Here is an example of an TX mob on PAY that transmits commands with sensor data. See later documentation for how these values are assigned.

```
mob_t data_tx_mob = {
    .mob_num = 5,                   // MOB number
        .mob_type = TX_MOB,             // MOB type (TX)
    .id_tag = PAY_DATA_TX_MOB_ID,   // The receiving MOB will match this tag with its␣
↪own tag and mask
    .ctrl = default_tx_ctrl,        // Control bits (leave as default)

    .tx_data_cb = data_tx_callback  // The function to be called before this MOB␣
↪transmits a message
};
```

Unlike an RX MOB, the TX MOB does not activate on its own. When you want to transmit a CAN message, you must activate it by "resuming" it. In this example, when you call `resume_mob(&data_tx_mob)`, it will resume the TX MOB, triggering an interrupt and calling the function specified in `tx_data_cb`. The CAN system allocates space for the data bytes and the length (number of bytes) to transmit, passing pointers to them to your function. You must get the necessary data and place it in those memory locations (using the pointers). When the function ends, the CAN system takes that data and transmits the CAN message. The MOB then "pauses" itself until you call `resume_mob(&data_tx_mob)` again. For example:

```
void data_tx_callback(uint8_t* data, uint8_t* len) {
    print("TX Callback\n");

    // Normally we would retrieve some sort of sensor data
    // For this example, just transmit the bytes 0x00 0x01 0x02 0x03

    // Want to transmit 4 bytes
    * len = 4;

    // Set the data
    data[0] = 0x00;
    data[1] = 0x01;
    data[2] = 0x02;
    data[3] = 0x03;

    // After this function ends, the CAN system will transmit this message
}
```

6.1.6 Initializing CAN

At the beginning of your program, you must initialize the CAN system and each individual MOB. For example:

```
// Enable CAN interrupts and the CAN transceiver
init_can();

// Initialize MOBs (note the different functions for RX and TX)
init_rx_mob(&cmd_rx_mob);
init_tx_mob(&data_tx_mob);
```

6.1.7 Message Objects (MObs)

Before sending messages over CAN, the message objects (MObs) need to be initialized. A universal MOb structure is defined in `lib-common/include/can/can.h`:

```
typedef struct {
    // common
    uint8_t mob_num;
    uint8_t dlc;
    mob_id_tag_t id_tag;
    mob_ctrl_t ctrl;
    mob_type_t mob_type;

    // rx specific
    mob_id_mask_t id_mask;
    can_rx_callback_t rx_cb;

    // tx specific
    can_tx_callback_t tx_data_cb;
    uint8_t data[8];
} mob_t;
```

mob_num

Each 32M1 has space for **six** MObs which can be active at any given time, denoted by `mob_num`. In the 32M1 datasheet, you might see these referred to as 'pages', although `mob_num` and the page number are functionally the same. Before editing MOb variables, the mob needs to be selected, which is handled automatically by the CAN library. *Further reads or writes to MOb-related registers will only affect the selected MOb.*

Priority is given is given to the MOb with the smallest `mob_num` when choosing which MOb to send/receive. For example, if two TX MObs are initialized and resumed on Board A, then MOb 0 will send first. Likewise, if a TX from Board A can be handled by two RX MObs on Board B, then the RX MOb with the lower `mob_num` will trigger the RX interrupt.

id_tag and id_mask

Each MOb on the *entire CAN bus* should be given a *unique* `id_tag`. ID tags are used by RX MObs when masking incoming transmissions via `id_mask`. Both `id_tag` and `id_mask` are *eleven* bits long and should be defined as single-element structs during initialization:

You may see references to AUTO MObs in documentation and code. We are not using them, so ignore them.

```
// MOb A
.mob_type = TX_MOB,
.id_tag =   { 0x0001 },

// MOb B
.mob_type = TX_MOB,
.id_tag =   { 0x0002 },

// MOb C
.mob_type = RX_MOB,
.id_tag =   { 0x0003 },
.id_mask =  { 0xFF01 },
```

From the above example, notice that each MOb has been given a unique ID and that the RX MOb has been given the mask `0xFF01`. Assume that the TX MObs are initialized on Board A and the RX MOb on Board B.

Masks work such that if `id_mask` has a 1 in a certain bit position, then the incoming `id_tag` and the `id_tag` of the RX MOb must be *equal* in that bit position. If `id_mask` has a 0 in that position, then the mask treats that position as a "don't care".

In this example, the RX MOb will ignore transmissions from MOb B. The RX mask cares about equality in bits 10-8 and bit 1. All the ID tags have 0's in bits 10-8, but since the RX MOb has a 1 in bit 0 and MOb B has a 0, the RX MOb will mask transmissions from MOb B. Meanwhile, both the RX MOb and MOb A share a 1 in bit 0, so the RX MOb will accept incoming transmissions from MOb A.

ctrl and mob_type

The `mob_type` variable defines the type of MOb and should be one of `TX_MOB` or `RX_MOB`, depending on the type of MOb you would like to define. The `ctrl` variable is a six-element struct defined in `lib-common/include/can/can.h` which holds CAN control parameters. Default TX and RX control configurations are also defined in `can.h` (and work pretty well for the majority of cases).

```
// struct to hold RTR, IDE, IDE Mask, RTR Mask and RBnTag bits;
// all boolean
typedef struct {
    uint8_t rtr; // 1 for remote frames, 0 for data frames
    uint8_t ide; // specifies CAN rev; should always be 0, for rev A
    uint8_t ide_mask; // masking bits for RX
    uint8_t rtr_mask; // masking bits for RX
    uint8_t rbn_tag; // masking bit for RX
    uint8_t rplv; // RPLV bit
} mob_ctrl_t;

// TODO: change these; ide_mask SHOULD matter
#define default_rx_ctrl { 0, 0, 0, 0, 0, 0 }
#define default_tx_ctrl { 0, 0, 0, 0, 0, 0 }
```

dlc

The `dlc` variable stores the number of bytes to be sent/received, and can be up to 8 bytes long. DLC needs to be defined upon initialization for RX MObs. If the incoming message does not have the expected DLC an error will be thrown.

When the MOb is initialized for TX, `dlc` needs to be set to the length of the data (in bytes) to be transmitted. This is assigned in the TX callback function pointed to by `tx_data_cb`, which is described in the following section.

rx_cb, tx_data_cb and data[8]

The `rx_cb` and `tx_data_cb` variables store function pointers to the RX and TX callback functions for a specific MOb. Different callback functions can be defined for each individual MOb. These functions are called from their respective interrupt-handling functions in `lib-common/src/can/can.c`. They are passed a pointer to the array of data to be sent/received, and either the length of the array (to the RX callback) or a pointer to the length of the array (to be set in the TX callback).

`tx_data_cb` is called upon initialization and after a transmission has been sent. After a transmission is sent, it generates a `TXOK` interrupt and calls `tx_data_cb` from `load_data` to get fresh data. `rx_cb` is called from `handle_rx_interrupt()` after once a transmission has been sucessfully recieved, generating a `RXOK` interrupt.

The functions should be defined in the file which includes CAN, and the two variables can be set simply by passing them the name of the function.

```
void rx_callback(uint8_t* data, uint8_t len) {
    print("TX received!\n");
    print("%s\n", (char *) data);
}

void tx_callback(uint8_t* data, uint8_t* len) {
    * len = 7;
    char str[] = "Hello!";

    for(uint8_t i = 0; i < *len; i++) {
        data[i] = str[i];
    }
}
```

6.2 SPI (Serial Peripheral Interface)

SPI is a data communication protocol that we use for controlling devices within a subsystem, primarily sensors.

In this section, we will describe the SPI protocol and how to use our library with the ATmega32m1.

6.2.1 What is SPI?

SPI is a communication protocol used to communicate between microcontrollers and peripheral devices, such as sensors. Put simply, it's a system that allows us to send a byte to some device and receive a byte in return.

SPI uses what's called a Master-Slave architecture. In this system there is one master device that communicates to multiple slave devices. In our system the master device is our microcontroller and the slave devices are mostly sensors. The master can only communicate with one slave device at a time and the slaves cannot communicate with each other.

6.2.2 SPI Bus

SPI is a synchronous serial communication protocol used to communicate between devices. SPI uses a master-slave architecture, with a single master device initiating the communication frame, and operates in full-duplex mode (data is sent and received at the same time).

In digital systems, data is transferred in the form of 1's and 0's. In hardware, 1's and 0's are represented as high and low voltages. Our satellite uses 3.3V, so a 1 corresponds to ~3.3V on a wire, and a 0 corresponds to ~0V.

SPI is a synchronous communication protocol, meaning data is timed with clock pulses. is sent and received at the same time. This means that two of the four lines (MOSI, MISO) are for data, and one of the lines (SCK) is for timing.

SPI uses four wires to communicate, which are referred to as the *SPI bus*. Three lines (SCK, MOSI, MISO) are shared between *all devices* on a SPI bus. The fourth line (CS/SS) is *unique to every slave device*, and all CS/SS lines are connected to the master device.

The master device always transmits a square wave on SCK (source clock) to synchronize all devices. To initiate communication, the master device sets CS/SS (chip select/slave select) low for a particular device. The master device sends data to the slave device on the MOSI (master out, slave in) line. At the same time, the slave device sends data to the master device on the MISO (master in, slave out) line. To stop communication, the master devices sets CS/SS high again.

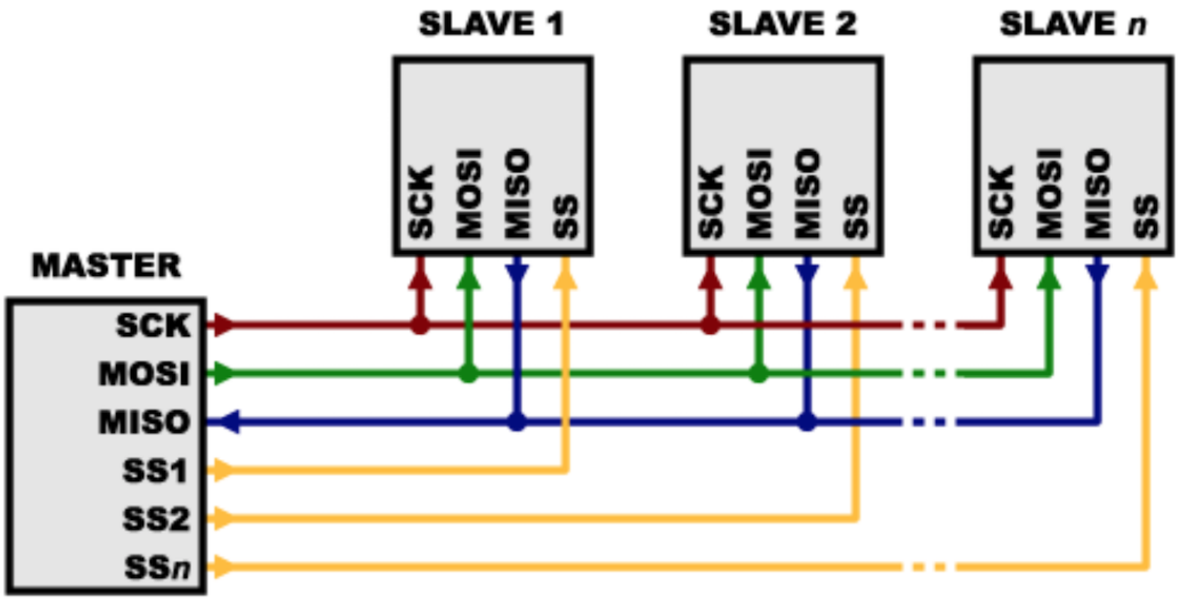


Diagram of a general SPI bus

SCK/SCLK (Source Clock)

The clock keeps the data lines and devices in sync. The clock is an oscillating signal produced by the master device that tells the receiving device when to read the data. Depending on the device properties, data is either sent/received on the rising/falling edge of SCK. We will discuss this in more detail shortly. This line is shared by all slave devices.

MOSI (Master Out Slave In)

This is the line where data is sent from the master device to the slave device. This line is shared by all slave devices.

MISO (Master In Slave Out)

On this line the data is being sent out of the slave device received by master. This line is shared by all slave devices.

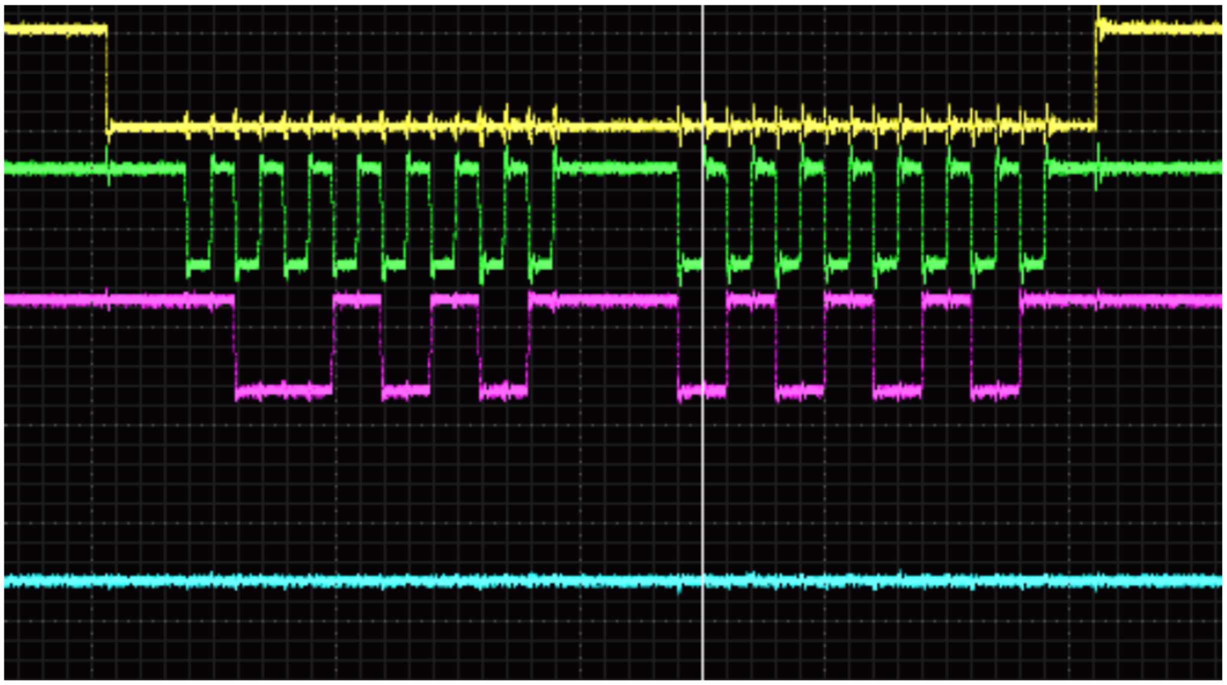
CS/SS (Chip Select / Slave Select)

This line is referred to as CS or SS interchangeably. It is active low, which means the slave device is active when CS is set low. Only one CS can be low at a time or there will be conflicts on the SPI bus resulting in garbage data. We use a pull-up resistor on the CS pin to set a default value.

A pull-up resistor is a large resistor (typically 10K) which bridges between VCC (3V3 in our case) and another pin. When no load is applied to the pin, no current flows through the resistor. This allows us to hold CS at a known (3V3) state when the CS pin isn't being driven by other circuitry. Then, we can drive another pin on the CS line low (GND) to select the device. Current will flow through the resistor and drop 3V3 across it.

Hardware Signal

If you look at the SPI lines on an oscilloscope, this is what a SPI transfer should look like:


SPI signals on an oscilloscope

This image is taken from online but SPI transfers look very similar on our equipment.

I'll pose the following question:

Which lines are which?

You can't really distinguish between MISO and MOSI in this picture but just pick one to be MOSI and the other to be MISO. What is being sent and received? The answer is given below.

Answer:

1. The **yellow line is CS**. This is because it is being lowered before and raised after the SPI transfer is complete.
2. The **green line is SCK**. It oscillates 8 times for each byte sent.
3. The **pink and blue are MOSI and MISO**. The blue line has `0x00` and `0x00` and the pink line has `0b10010101` and `0b01010101`.

6.2.3 Using our SPI Library

Here are the basic software steps to send SPI messages.

Setup:

1. Initialize SPI
2. Initialize CS as an output pin
3. Set CS high

Transmission:

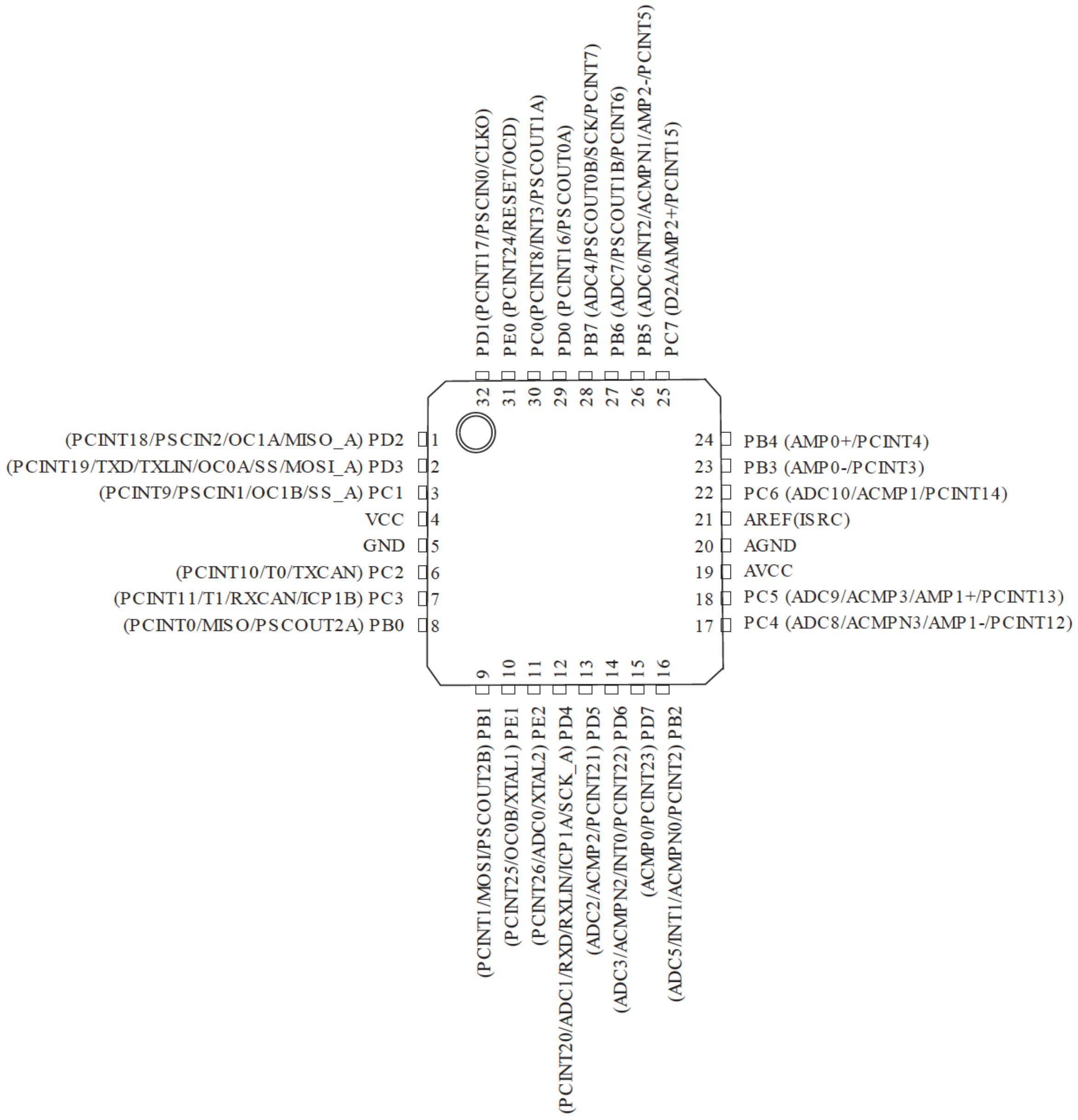
1. Set CS low
2. Send SPI message

3. Set CS high

CS Pin

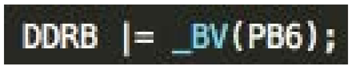
On our microcontroller each IO pin has three registers that control it. We will only need to use two of them. There is the data direction register that controls if the pin is input our output and there is the port register that lets you write high or low on the pin.

Once you figure out what pin you are using for CS you can check the microcontroller's datasheet to get the name of the pin. Figure 1 shows the pin configuration for the ATmega32m1 microcontroller.


(PCINT18/PSCIN2/OC1A/MISO_A) PD2 1
(PCINT19/TXD/TXLIN/OC0A/SS/MOSI_A) PD3 2
(PCINT9/PSCIN1/OC1B/SS_A) PC1 3
VCC 4
GND 5
(PCINT10/T0/TXCAN) PC2 6
(PCINT11/T1/RXCAN/ICP1B) PC3 7
(PCINT0/MISO/PSCOUT2A) PB0 8
9 (PCINT1/MOSI/PSCOUT2B) PB1
10 (PCINT25/OC0B/XTAL1) PE1
11 (PCINT26/ADC0/XTAL2) PE2
12 (PCINT20/ADC1/RXD/RXLIN/ICP1A/SCK_A) PD4
13 (ADC2/ACMP2/PCINT21) PD5
14 (ADC3/ACMPN2/INT0/PCINT22) PD6
15 (ACMP0/PCINT23) PD7
16 (ADC5/INT1/ACMPN0/PCINT2) PB2
17 PC4 (ADC8/ACMPN3/AMP1-/PCINT12)
18 PC5 (ADC9/ACMP3/AMP1+/PCINT13)
19 AVCC
20 AGND
21 AREF(ISRC)
22 PC6 (ADC10/ACMP1/PCINT14)
23 PB3 (AMP0-/PCINT3)
24 PB4 (AMP0+/PCINT4)
25 PC7 (D2A/AMP2+/PCINT15)
26 PB5 (ADC6/INT2/ACMPN1/AMP2-/PCINT5)
27 PB6 (ADC7/PSCOUT1B/PCINT6)
28 PB7 (ADC4/PSCOUT0B/SCK/PCINT7)
29 PD0 (PCINT16/PSCOUT0A)
30 PC0(PCINT8/INT3/PSCOUT1A)
31 PE0 (PCINT24/RESET/OCD)
32 PD1(PCINT17/PSCIN0/CLKO)

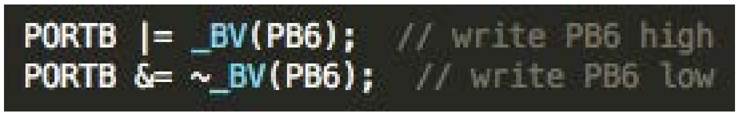
ATmega32m1 pinout

There are four banks of ports (B, C, D and E) with eight pins on each. There are 8-bit data direction and port registers for each of the four ports. Each bit in the register is for a separate pin. The data direction register is called DDRx and the port register is just called PORTx where x is the port. So if you wanted to initialized PB6 as output the code would be the following.



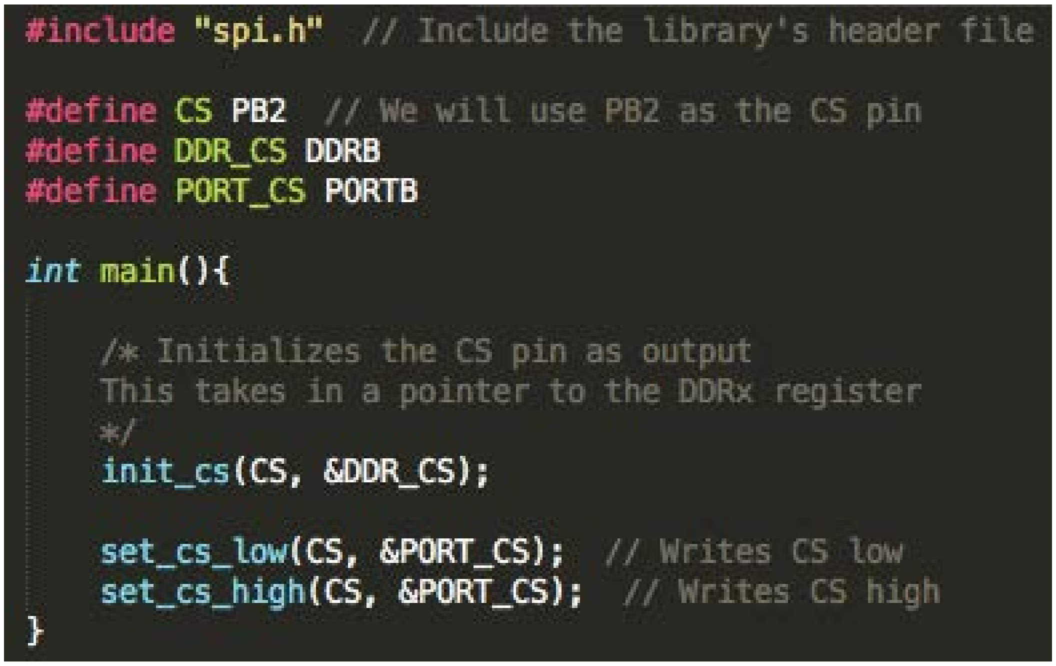
```
DDRB |= _BV(PB6);
```

The macro `_BV(PB6)` expands to `1 << PB6` and PB6 is a macro that expands to 6. Here is the code to write high or low on PB6.



```
PORTB |= _BV(PB6);  // write PB6 high
PORTB &= ~_BV(PB6);  // write PB6 low
```

In our SPI library we have functions that will do this for you. This is how you use them.



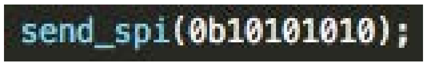
```
#include "spi.h"  // Include the library's header file

#define CS PB2  // We will use PB2 as the CS pin
#define DDR_CS DDRB
#define PORT_CS PORTB

int main(){

    /* Initializes the CS pin as output
    This takes in a pointer to the DDRx register
    */
    init_cs(CS, &DDR_CS);

    set_cs_low(CS, &PORT_CS);  // Writes CS low
    set_cs_high(CS, &PORT_CS);  // Writes CS high
}
```

Initialize SPI

We have a function, `init_spi()` that does this. It initializes SCK and MOSI as output and sets the SCK frequency to 8 MHz / 64. The 32M1's internal clock frequency is 8 MHz.

Sending a SPI message

SPI sends 8 bit messages. If you want to send more than a byte you can send consecutive SPI messages. This is how you do it.



```
send_spi(0b10101010);
```

Example SPI Code

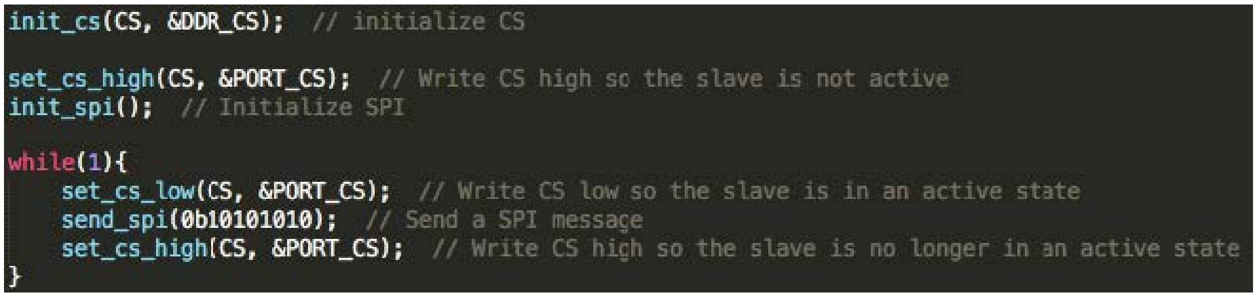
```
/*
Example from PAY:
Say we want to use the pin labelled PB5 on the 32M1 as the CS pin for the SPI device.
```

(continues on next page)

(continued from previous page)

```
It uses pin 5 on DDR B (data direction register) and Port B (output).
*/

// This would be in a header file
#define CS PB5
#define CS_PORT PORTB
#define CS_DDR DDRB




// Setup: Just do this once

// Initialize SPI
init_spi();

// Initialize CS pin as an output pin
init_cs(CS, &CS_DDR);    // pin, DDR

// Set CS pin high (disable SPI device by default)
set_cs_high(CS, &CS_PORT);




// Transmission: do this every time you want to transmit

// Start transmission: set CS pin low (enable SPI device)
set_cs_low(CS, &CS_PORT);  // pin, port

// Send and/or receive data: call send_spi() for each byte
uint8_t received1 = send_spi(0xA4); // if you want to both send and receive
uint8_t received2 = send_spi(0x00); // if you just want to receive (send 0)
send_spi(0xA4);                     // if you just want to send (ignore received)

// End transmission: set CS pin high (disable SPI device)
set_cs_high(CS, &CS_PORT);
```

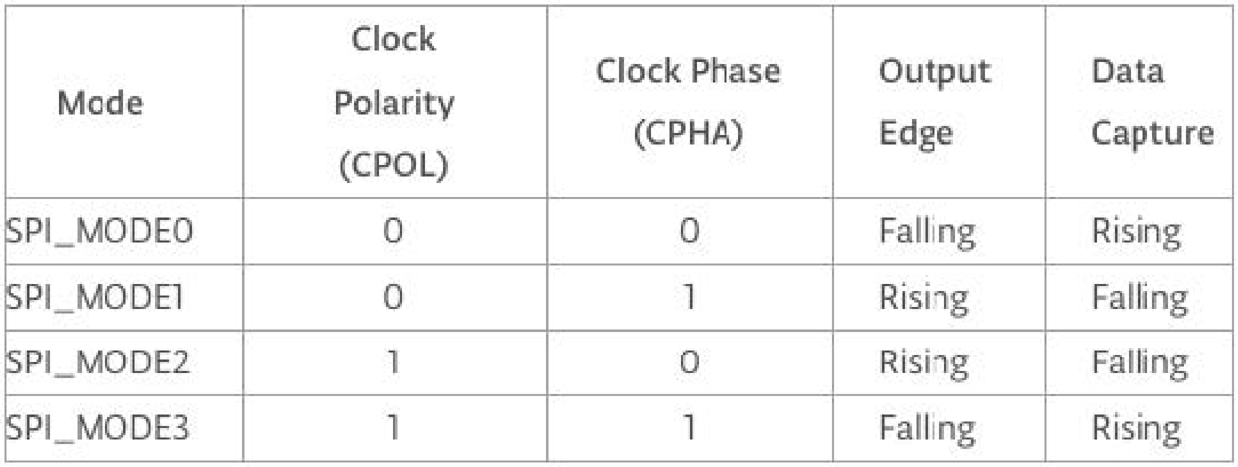
This is another full SPI program to make sure SPI is running correctly on the microcontroller.



```
init_cs(CS, &DDR_CS);  // initialize CS

set_cs_high(CS, &PORT_CS);  // Write CS high so the slave is not active
init_spi();  // Initialize SPI

while(1){
    set_cs_low(CS, &PORT_CS);  // Write CS low so the slave is in an active state
    send_spi(0b10101010);  // Send a SPI message
    set_cs_high(CS, &PORT_CS);  // Write CS high so the slave is no longer in an active state
}
```

This repeatedly sends `10101010`.

6.2.4 Clock Settings (advanced)



Mode	Clock Polarity (CPOL)	Clock Phase (CPHA)	Output Edge	Data Capture
SPI_MODE0	0	0	Falling	Rising
SPI_MODE1	0	1	Rising	Falling
SPI_MODE2	1	0	Rising	Falling
SPI_MODE3	1	1	Falling	Rising

SPI has settings for the clock polarity (CPOL) and phase (CPHA), as described here. Usually the default settings work and you don't need to worry about this, but occasionally there is a device where these settings must be modified. You can see what a device's CPOL and CPHA settings should be based on the SPI timing diagram(s) in its datasheet.

The two clock settings introduced here are Clock Polarity and Clock Phase. Clock Phase determines whether data is shifted in and out on the rising or falling edge of the data clock cycle. Clock Polarity determines determines whether the clock is idle when high or low.

6.3 I2C (Inter-Integrated Circuit) Protocol

I2C is a master-slave protocol (similar to SPI) and only uses 2 wires (similar to UART). We are using it to send and receive antenna deployment commands.

SparkFun

I2C Info

6.4 UART (Universal Asynchronous Receiver Transmitter)

UART is a protocol that allows us to use the 32M1 to print messages to the terminal in CoolTerm/XTerm.

```
init_uart();
print("Hello World!\n");
```

The `print()` function behaves like C's `printf()` function (but prints to UART instead of `stdout`), so you can use format specifiers (see documentation).

6.4.1 Implementation of UART

Stanford CS140

Wikipedia

Circuit Basics

AVR Beginners - UART

CHAPTER 7

C Programming

7.1 C Programming

We will provide an overview of the fundamentals of the C programming language. For certain topics universal to all programming languages, some programming experience is assumed and there is only an example of the C syntax with a brief explanation. In addition to this document, we recommend reading parts of **"The C Programming Language"** by Brian Kernighan and Dennis Ritchie. **You can find a pdf of this book on the Google Drive in /Instrumentation/Literature.** This book describes all the features of C in great detail and is suitable for all levels of programming experience.

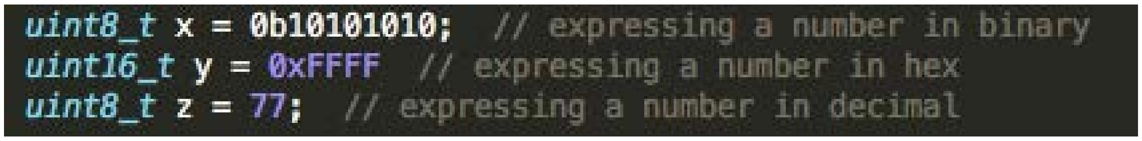
7.1.1 So, what makes C different than other programming languages?

Well, one thing is that it's not an object oriented programming language like Java or C++ or a whole lot of other programming languages. In addition to that, C is considered a relatively low level programming language since most C operations can be moderately easily translated into assembly. Because of this, C is a great choice for developing software that interfaces directly with hardware. Another unique part of C is pointers and dynamic memory allocation. C allows its programmers to have a generous amount of control over the computer's memory.

7.2 Variables

C includes the following most common types: `char`, `int`, `long`, `double` and `float`. A `char` is always a byte but the size of the other variables is machine dependent. Due to the confusion caused by variable sizes on different machines, in our software we use the types defined in `stdint.h`. These include the following unsigned types: `uint8_t`, `uint16_t`, `uint32_t` and `uint64_t` and the following signed types: `int8_t`, `int16_t`, `int32_t` and `int64_t`, where the numbers are the number of bits the variable holds. These types allow us to have better control over the sizes of our variables. So don't use a 64 bit variable in a for loop that loops 10 times.

Here you can see how to initialize and set variables using binary, hex and decimal. The `//` before text is how you make a comment in C.



```
uint8_t x = 0b10101010;  // expressing a number in binary
uint16_t y = 0xFFFF  // expressing a number in hex
uint8_t z = 77;  // expressing a number in decimal
```

You may have noticed no mention of strings or booleans, which are common types in several other languages. While a char represents a byte, it also can represent a character, hence the name char. So a string of characters is just an array of chars with a terminating character (`\0`) as the last element. `char[] s = “Hello World”` is a away to initialize a char array.

As for booleans we just use a `char` or `uint8_t` to represent them. 0 represents false and any other number is true. By convention we use 1 to represent true. So the statement `5 == 5` evaluates to 1 whereas `5 != 5` evaluates to 0.

7.3 Operators

7.3.1 Mathematical

`*` (multiplication)`/` (division)`+` (addition)`-` (subtraction)`%` (modulus)

7.3.2 Comparative

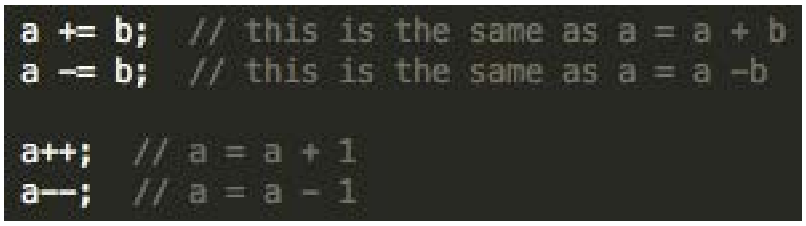
`>` (greater than)`>=` (greater than or equal to)`<` (less than)`<=` (less than or equal to)`==` (checks equality) `!=` (checks inequality)

7.3.3 Logical

`&&` (and) - `a && b` is true if and only if a and b are true`||` (or) - `a || b` is true if and only if at least one of a and b are true

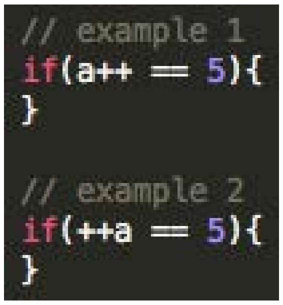
7.3.4 Increment and Decrement

There are the following short forms for incrementing or decrementing numbers.



```
a += b;  // this is the same as a = a + b
a -= b;  // this is the same as a = a -b

a++;  // a = a + 1
a--;  // a = a - 1
```

In addition to `a++` and `a--` there is `++a` and `--a` which are indistinguishable without a context.



```
// example 1
if(a++ == 5){
}

// example 2
if(++a == 5){
}
```

In *example 1*, a is used first in the `a == 5` check and then is incremented, whereas in *example 2*, a is first incremented and then used in the `a == 5` check.

7.4 Bitwise Operators

These operators directly manipulate the bits in integer numbers (generally represented as `uint` types). A lot of our software involves manipulating 8 bit registers on our microcontroller, so bitwise operators are very common in our software.

The first two are very similar to `&&` and `||` mentioned above. They are the "and" and "or" bitwise operators, represented by `&` and `|`. They both combine two numbers into one.

7.4.1 Single Bit Operations

First, we'll discuss how bitwise operators are applied between two bits. In these cases, we take two 1-bit values and apply the operation to get a 1-bit result.

Bitwise operation behaviour is commonly represented using a **truth table** as shown below. It shows all possible combinations of inputs and the resulting outputs.

AND

The AND operation produces a 1 if both of the inputs are 1. Otherwise, it produces a 0.

A | B | A AND B – | – | – 0 | 0 | 0 0 | 1 | 0 1 | 0 | 0 1 | 1 | 1

OR

The OR operation produces a 1 if either or both of the inputs are 1. Otherwise, it produces a 0.

A | B | A OR B – | – | – 0 | 0 | 0 0 | 1 | 1 1 | 0 | 1 1 | 1 | 1

XOR (Exclusive OR)

The XOR operation produces a 1 if either but not both of the inputs are 1 (the meaning of exclusive). Otherwise, it produces a 0. This is similar to OR, but if both inputs are 1 the result is 0.

A | B | A XOR B – | – | – 0 | 0 | 0 0 | 1 | 1 1 | 0 | 1 1 | 1 | 0

NOT (Compliment/Inverse)

The NOT operation is called an **unary** operator because it only has one input. The NOT operation "flips" a bit.

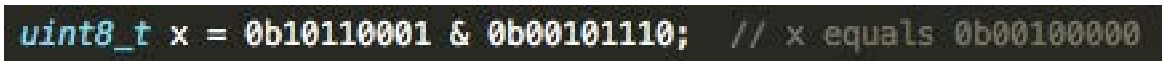
A | NOT A – | – 0 | 1 1 | 0

7.4.2 Multi-Bit Operations

We generally use the AND, OR, and XOR operations on numbers with multiple bits. These operations simply apply the operator between each pair of bits.

7.4.3 AND (&)

We represent AND using the & operator in C.



```
uint8_t x = 0b10110001 & 0b00101110;  // x equals 0b00100000
```

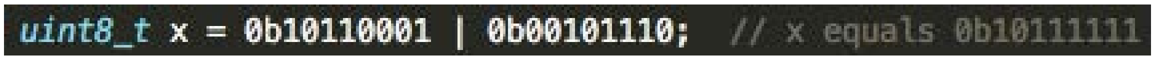
```
  10110001
& 00101110
----------
  00100000
```

There is only a 1 in the 6th bit position because that is the only position where there is a 1 in both the numbers.

7.4.4 OR (|)

We represent OR using the | operator in C.

If at least one of the bits is a 1, the result will have a 1 in that position.



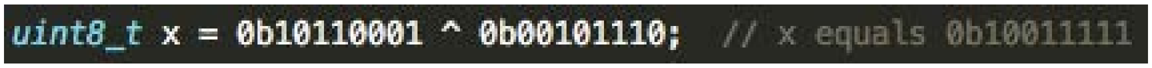
```
uint8_t x = 0b10110001 | 0b00101110;  // x equals 0b10111111
```

```
  10110001
| 00101110
----------
  10111111
```

7.4.5 XOR (^)

We represent XOR using the ^ operator in C.

If one and only one of the bits is a 1, then the result will have a 1 in that position.



```
uint8_t x = 0b10110001 ^ 0b00101110;  // x equals 0b10011111
```

```
  10110001
^ 00101110
----------
  10011111
```

7.4.6 NOT (~)

This still only applies to one number, and it flips all the bits so 1's become 0's and vice-versa.

Example 1:

```
uint8_t x = 0b10100011;
x = ~x;
```

x becomes `0b01011100`.

Example 2:

```
uint8_t x = 0b11110000;
x = ~x;
```

x becomes `0b00001111`.

7.4.7 Bitwise vs. Logical Operators

Note that `&` and `|` are different from `&&` and `||`. `&` and `|` are **bitwise operators**, which take two integers and produce another integer. `&&` and `||` are **logical operators**, which take two booleans and produce another boolean (commonly used in conditional expressions).

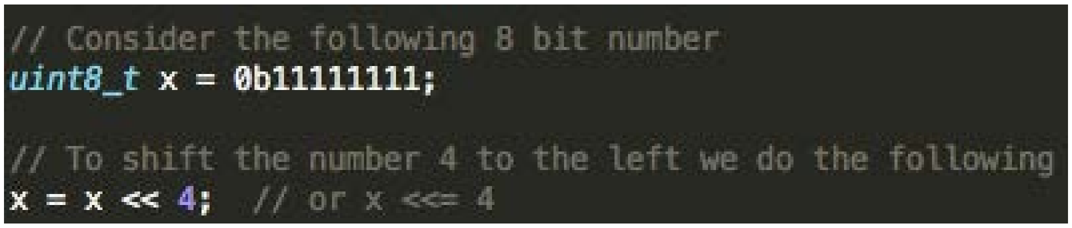
Also note that `~` (bitwise NOT) is different from `!` (logical NOT).

7.4.8 Modified Assignment

You can use the following operators to modify and assign a variable in one step: `&=`, `|=`, and `^=`.

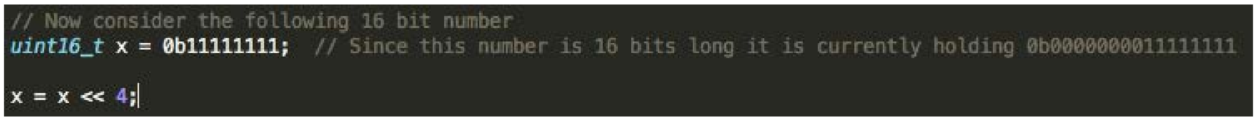
7.4.9 Bitwise Shifts (<<, >>)

C has bitwise shift operators for left (<<) and right (>>) bit shifts. They do what the name suggests, shift the bits in a number by a certain number of places.



```
// Consider the following 8 bit number
uint8_t x = 0b11111111;

// To shift the number 4 to the left we do the following
x = x << 4;  // or x <<= 4
```

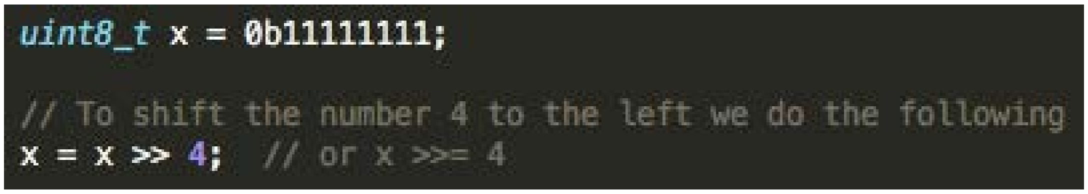
The bits are shifted to the left and 0's are shifted into the empty spots. You might think x is now `0b111111110000`, but in this case that is incorrect because that is no longer an 8 bit number. It is actually `0b11110000`. With bit shifts you have to consider the size of the variable because if you shift a number to the left past its limit those bits will be cut off.



```
// Now consider the following 16 bit number
uint16_t x = 0b11111111;  // Since this number is 16 bits long it is currently holding 0b0000000011111111

x = x << 4;
```

When we shift this number by four, we end up with `0b0000111111110000` (or `0x0FF0`) because this 16 bit variable can hold those bits.

The right bit shift does the exact same thing except in the opposite direction. No matter the size of the variable the bits will be cut off.



```c
uint8_t x = 0b11111111;

// To shift the number 4 to the left we do the following
x = x >> 4;  // or x >>= 4
```

In this case the bits are shifted out of the variable to the right and 0's are shifted into the variable on the left. Here we will end up with x as `0b00001111`.

7.4.10 Applications

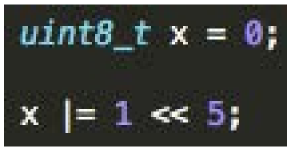
Here are the most common uses of bitwise operators in our software.

Very frequently we have to manipulate specific bits in numbers (usually 8 bit, 16 bit, or 32 bit). For example, this is necessary to change settings in the microcontroller or other devices. We use bitwise operators to change certain bits while not affecting others.

OR

If you OR any bit with 0, you always get the same bit. If you OR any bit with 1, you always get 1. We can use the OR operator to force specific bits to be 1 while leaving other bits as they are.

For example, this is how to set bit 5 to 1.



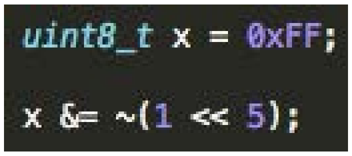
```c
uint8_t x = 0;

x |= 1 << 5;
```

Now the value of x is `0b00100000`.

AND

If you AND any bit with 0, you always get 0. If you AND any bit with 1, you always get the same bit. We can use the AND operator to force specific bits to be 0 while leaving other bits as they are.

For example, you can set bit 5 to 0 with the following:



```c
uint8_t x = 0xFF;

x &= ~(1 << 5);
```

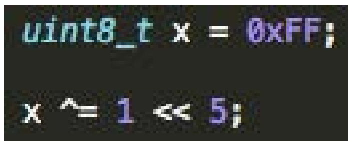
The expression `1 << 5` is `00100000`, so `~(1 << 5)` is `11011111`.

Now the value of x is `0b11011111`.

XOR

If you XOR any bit with 0, you always get the same bit. If you XOR any bit with 1, you always get the opposite bit. We can use the XOR operator to flip specific bits while leaving other bits as they are.

For example, you can switch the value of a bit with the following:



```
uint8_t x = 0xFF;

x ^= 1 << 5;
```

Now the value of x is `0b11011111`.

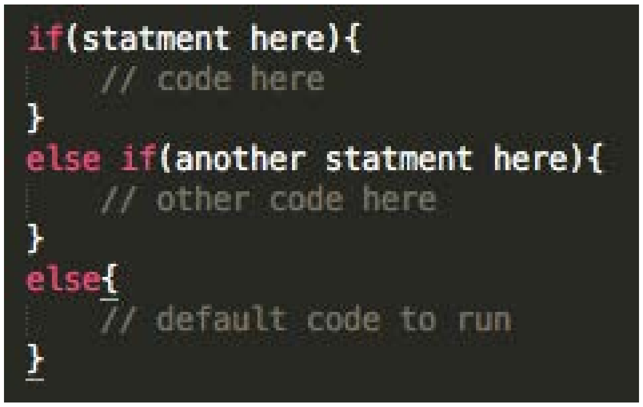
NOT

If you NOT any bit, you always get the opposite bit. We can use the NOT operator to flip all bits.

7.5 Control Structures

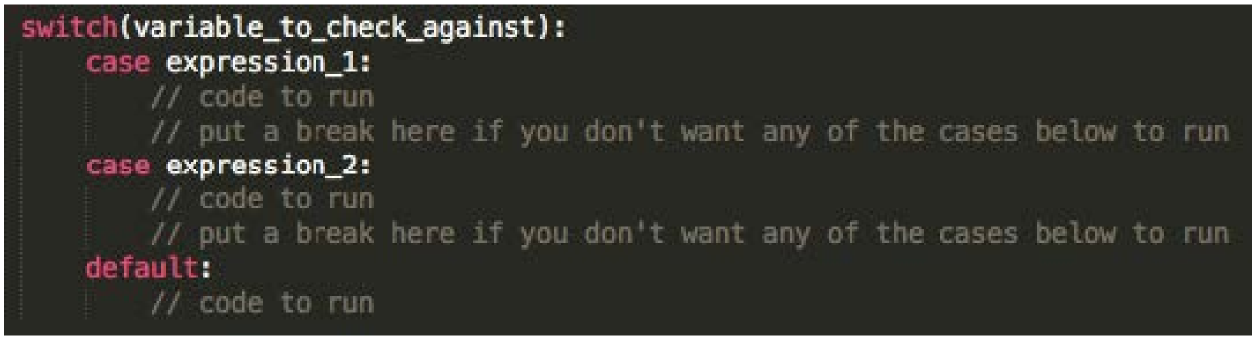
Control structures alter program flow.

7.5.1 If Statements



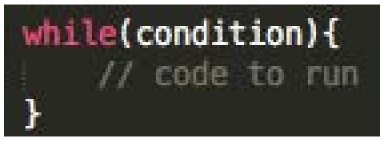
```
if(statment here){
    // code here
}
else if(another statment here){
    // other code here
}
else{
    // default code to run
}
```

7.5.2 Switch Statements



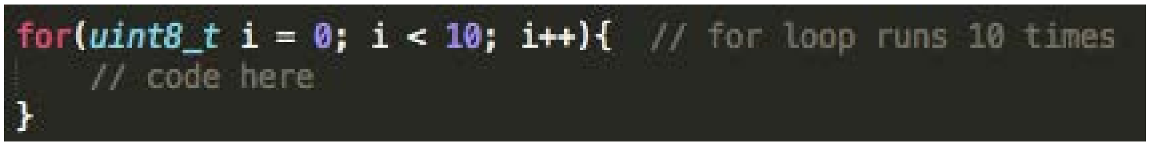
```
switch(variable_to_check_against):
    case expression_1:
        // code to run
        // put a break here if you don't want any of the cases below to run
    case expression_2:
        // code to run
        // put a break here if you don't want any of the cases below to run
    default:
        // code to run
```

7.5.3 While Loops



```
while(condition){
    // code to run
}
```

7.5.4 For Loops



```
for(uint8_t i = 0; i < 10; i++){  // for loop runs 10 times
    // code here
}
```

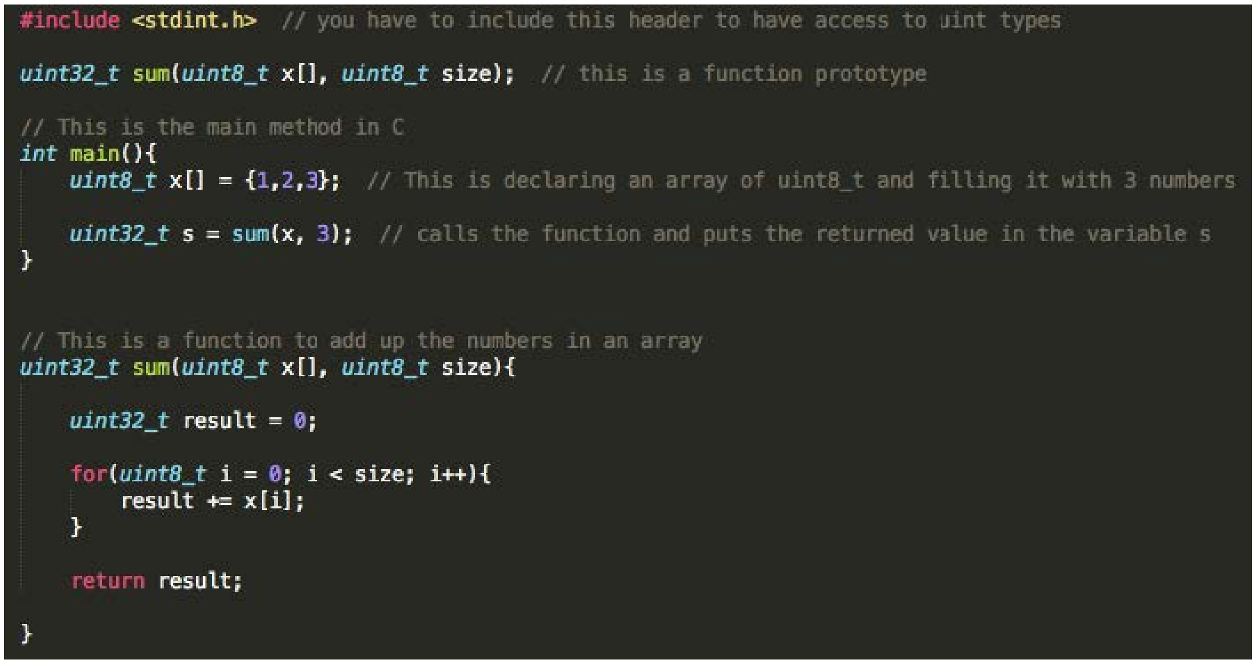
7.5.5 Break and Continue

These are useful commands for conditional statements and loops.

> **break** - exits the current statement or loop**continue** - skips the current iteration of the loop and continues with the next

7.6 Functions

We'll look at the following code to learn how to do functions in C and see what a C file should look like.



```
#include <stdint.h>  // you have to include this header to have access to uint types

uint32_t sum(uint8_t x[], uint8_t size);  // this is a function prototype

// This is the main method in C
int main(){
    uint8_t x[] = {1,2,3};  // This is declaring an array of uint8_t and filling it with 3 numbers

    uint32_t s = sum(x, 3);  // calls the function and puts the returned value in the variable s
}


// This is a function to add up the numbers in an array
uint32_t sum(uint8_t x[], uint8_t size){

    uint32_t result = 0;

    for(uint8_t i = 0; i < size; i++){
        result += x[i];
    }

    return result;

}
```

First, include `stdint.h`. See the section about header files.

In C the `main()` function returns an int. By default it returns `0`. All C projects must contain one and only one `main()` function.

The function `sum` takes in a `uint8_t` array and its size and returns a `uint32_t`. In C you have to always pass the size of an array with the array.

The only difference with functions in C is that for the function to be recognized throughout the file it needs to have a function prototype as show above.

7.7 Header Files

The purpose of header files is to have definitions and declarations that can be shared amongst multiple files by including the header. By putting function prototypes in a header file and including it, you can call those functions in other files. Standard library headers (such as for C or AVR) are included with `<file_name.h>`, such as `stdint.h`. Non-library header files (such as the code we write) are included using `#include "file_name.h"`. Header files can also include other header files.

The `#define` and `typedef` statements are commonly used in header files, but can be used in any file.

7.8 #define

This tells the preprocessor to replace a particular piece of text with another piece of text before compiling the C program. It is commonly used to associate a readable name with a commonly used constant value.

```
#define REGISTER_DEFAULT 0x4B
```

For example, we can use `#define` to clarify the use of a register's default value. Now whenever you put `REGISTER_DEFAULT` in your code, the compiler will substitute that with the value `0x4B` before compiling. This ensures constants are defined in one place and have a clear name.

7.9 typedef

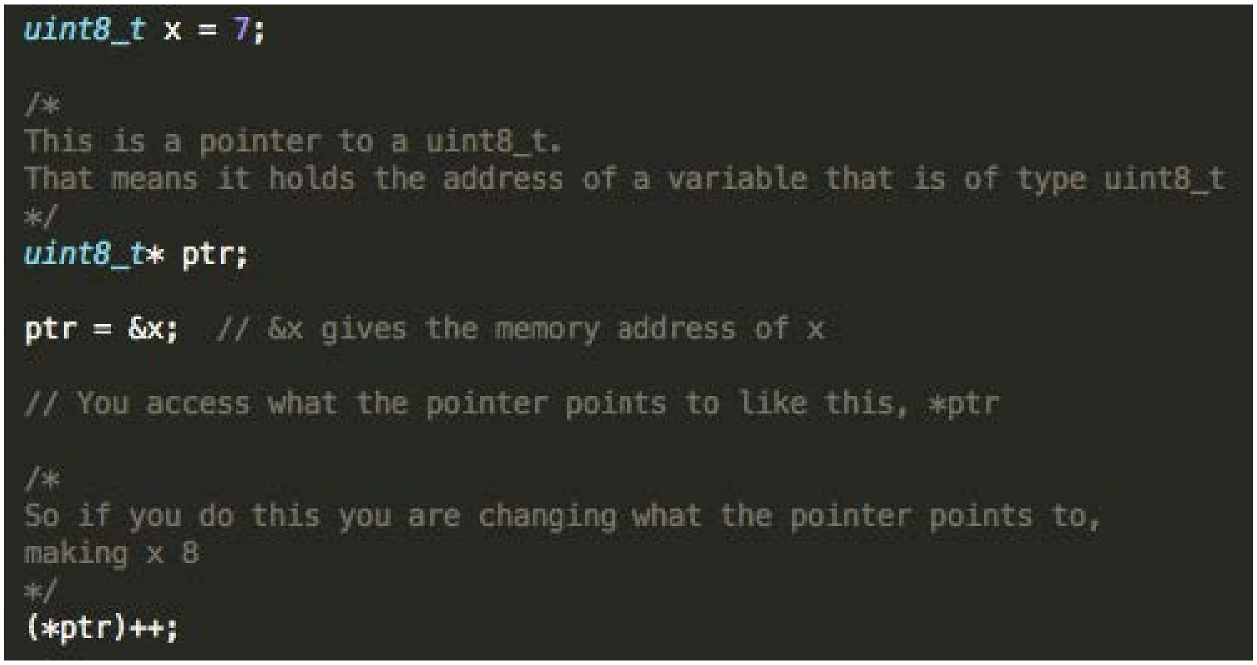
This creates a new type and associates it with some other existing type. This is commonly used to give a simpler and clearer name for a type for a particular use instead of writing a complex type many times.

```
typedef volatile uint8_t* port_t;    // Don't need to worry about what `volatile` means
```

This example creates a new type `port_t` and associates it to mean the same type as `volatile uint8_t*`. This is a standard type to represent a port on the 32M1, so using the name `port_t` is more clear.

7.10 Pointers

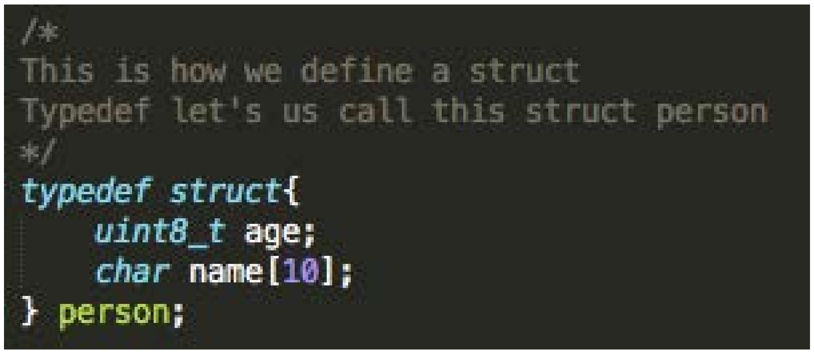
A pointer is a type of variable whose value is the memory address of another variable. This is how you declare and use a pointer.



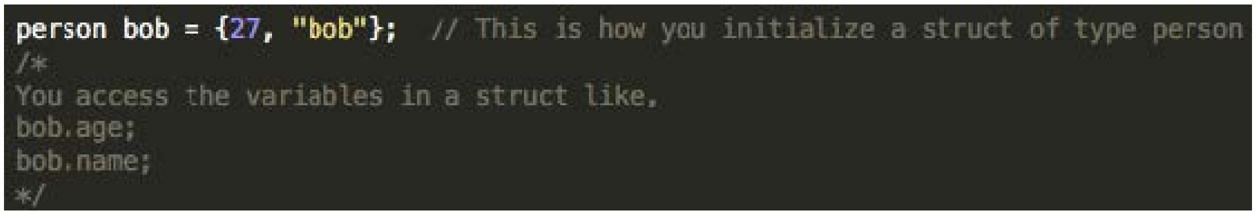
```c
uint8_t x = 7;

/*
This is a pointer to a uint8_t.
That means it holds the address of a variable that is of type uint8_t
*/
uint8_t* ptr;

ptr = &x;  // &x gives the memory address of x

// You access what the pointer points to like this, *ptr

/*
So if you do this you are changing what the pointer points to,
making x 8
*/
(*ptr)++;
```

7.11 Structs

Structs are a useful way of bunching multiple variables together. This is probably the closet C gets to objects. We tend to declare our structs in a header file using `typedef` for easier use.



```c
/*
This is how we define a struct
Typedef let's us call this struct person
*/
typedef struct{
    uint8_t age;
    char name[10];
} person;
```



```c
person bob = {27, "bob"};  // This is how you initialize a struct of type person
/*
You access the variables in a struct like,
bob.age;
bob.name;
*/
```

7.12 `.c` and `.h` Files

In our software repositories, you will often see pairs of files with the same name but different extensions (`.c` and `.h`). The `.c` file is often called the C file or implementation, while the `.h` file is often called the header file or interface.

`.c` files contain the implementation of the code, while `.h` files exist to provide interfaces that allow a file to access functions, global variables, and macros from other files.

When you `#include "file.h"` or `#include <folder/file.h>` in a file, it includes the header file in the current file so that you can call functions from `file.h`.

For functions, the `.c` file contains the entire function. You copy the corresponding **function prototype** to the `.h` file, which is just the first line containing the function name, return type, and parameters.

7.12.1 Example

For example, say we have a function called `foo()` in a file called `test.c`.

```
double foo(double a, double b) {
    return a + b;
}
```

In the corresponding header file, `test.h`, we put just the function prototype so other files can call this function.

```
double foo(double a, double b);
```

Say we want to call this function from `main.c`.

```
#include "test.h"

int main(void) {
    foo(1.1, 2.3);
    ...
}
```

7.13 Binary (0b) and Hexadecimal (0x) Literals

Binary is a base-2 numbering system (digits are either 0 or 1), while hexadecimal is a base-16 numbering system (digits 0-F). Look online for descriptions of how these systems work.

In C, you can express integers using their binary or hexadecimal representations by prefixing them with `0b` or `0x`.

```
// These represent different numbers
uint8_t binary = 0b01101011;
uint8_t hex = 0xF7;
```

7.14 Integer Types

https://www.nongnu.org/avr-libc/user-manual/group__avr__stdint.html https://en.cppreference.com/w/cpp/language/integer_literal

You will often see variable types such as `uint8_t` and `int16_t` in our code. Instead of using the standard `int` type, we use these types to specify how many bits each integer has. This is because the number of bits in `int` varies depends on the compiler used, so we want to be explicit about the number of bits used. This is also to prevent accidental integer overflow if the number of bits is too small.

Standard integers are signed (can be positive, zero, or negative). The `u` prefix means unsigned (can only be positive or zero). The number specifies the number of **bits** (not bytes). The `_t` suffix is a C convention for naming types.

When writing **integer literals** (constant integer values written out in code), the AVR compiler defaults to 16-bit integers. If we want to express a literal that uses more than 16 bits, we need to use an **integer suffix** for a larger bit size.

For example, we can't use the literal `1000000` because the maximum 16-bit unsigned literal is `65535` (2^16 - 1) and the maximum 16-bit signed literal is `32767` (2^15 - 1). To make this work, we need to use the literal `1000000L` or `1000000UL`. The `L` suffix means `long` (signed 32-bit) while the `UL` suffix means `unsigned long` (unsigned 32-bit).

7.14.1 Available Types

Type | Integer | Literal Suffix | sizeof :— | :— | :— | :— `int8_t` | signed 8-bit | None | 1 `uint8_t` | unsigned 8-bit | None | 1 `int16_t` | signed 16-bit | None | 2 `uint16_t` | unsigned 16-bit | None | 2 `int32_t` | signed 32-bit | `L` (long) | 4 `uint32_t` | unsigned 32-bit | `UL` (unsigned long) | 4 `int64_t` | signed 64-bit | `LL` (long long) | 8 `uint64_t` | unsigned 64-bit | `ULL` (unsigned long long) | 8

7.15 Print Formatting

C printf() formatting

The UART `print()` function behaves almost identically to the standard C `printf()` function. This gives a powerful way to format values as strings for output.

Using '%f' or '%lf' requires a special compiler flag to be enabled. See 'lib-common/src/uart/log.c' about this, which is enabled when compiling testing programs but not for main programs.

AVR's `printf()` doesn't support printing 64-bit integers.

Here are the format specifiers for common data types:

Data Type(s)	Format Specifier(s)
`uint8_t`, `uint16_t`	`%u` (Unsigned)
`uint32_t`	`%lu` (Long Unsigned)
`int8_t`, `int16_t`	`%d` (Decimal - Signed)
`int32_t`	`%ld` (Long Decimal - Signed)
`uint8_t`, `uint16_t`, `int8_t`, `int16_t`	`%x` (Hexadecimal - Lowercase), `%X` (Hexadecimal - Uppercase)
`uint32_t`, `int32_t`	`%lx` (Long Hexadecimal - Lowercase), `%lX` (Long Hexadecimal - Uppercase)
`float`, `double`	`%f` (Float)

7.15.1 Integer Formatting

An example of controlling integer formatting output as hex:

```
print("0x%02x", value);
// lowercase hex, always with 2 hex digits (zero padded if necessary)
// add "0x" at the beginning to indicate to the reader that the output is in hex
```

7.16 Volatile Variables

The `volatile` keyword instructs the C compiler to not make assumptions about the value of a variable. Normally, the compiler optimizes the program based on things guaranteed to be true about variable values. But these optimizations are under the assumption that only the current program can change a variable's value.

The 32M1 has **MMIO** (Memory-Mapped Input/Output) at specific memory locations. This means that values in specific locations in RAM are directly mapped to hardware devices such as port and peripherals. Things other than the C program can change the values of these variables.

In these special cases, you need to use the `volatile` keyword on a variable so the compiler doesn't assume that only the C program can change its value. If you don't put `volatile` and the compiler makes that assumption, the program will probably behave strangely and produce incorrect values.

For example, in `spi.h`:

```
typedef volatile uint8_t* port_t;
```

The `volatile` keyword is used because the port location in memory is directly mapped to pins in the hardware which could change outside of the program. This ensures correct behaviour.

CHAPTER 8

Style Guide

8.1 Software Style Guide

This is a collection of the software programming style guide for the Heron Mk II codebase. Feel free to suggest revisions or new guidelines.

The "Technical" section describes programming patterns and practices that reduce the chance of error, while the "Formatting" section describes stylistic changes for consistency and making code easier to read.

NASA's 10 Coding Rules

8.2 Technical

8.2.1 Memory Allocation and `malloc()`

You are not allowed to use `malloc()`. This is because our satellite is an embedded system and `malloc()` is a non-deterministic operation.

- There is a limited amount of memory, so it is likely to fail.
- It may be very slow, blocking the system.

All arrays must be statically declared with a fixed number of elements, preferably using a named constant that can be easily changed.

8.2.2 Integer Types

Do not use the `int` type. All integer types should be declared with explicit sizes using types in the `<stdint.h>` library, such as `uint8_t`, `uint32_t`, and `int16_t`. This is to be clear about the maximum value the integer can hold and should support.

Be careful about integer sizes. We need to avoid integer overflows at all costs, so make sure your integer types are large enough for the values they will contain.

8.2.3 Infinite Loops and Timeouts

Do not allow the possibility of an infinite loop. Every loop should have a guaranteed maximum number of times it can execute. Do not assume hardware will always behave as expected. Always add a timer as a fallback for loops waiting for a hardware event so the MCU cannot get stuck in an infinite loop.

The most common timeout structure starts at 65,535 (2^16 - 1, declared as the built-in `UINT16_MAX` constant), with a `uint16_t` type counting down to 0.

For example, say we are waiting for the pin PB0 to go low before proceeding.

```
// bad, could infinite loop
while (bit_is_set(PINB, PB0)) {
    continue;
}

// good, protect against rare case of infinite loop
uint16_t timeout = UINT16_MAX;
while (bit_is_set(PINB, PB0) && timeout > 0) {
    timeout--
}
// might want to check if timeout == 0 here if you want to detect a timeout
```

8.2.4 Atomics

Remember that when running this embedded system, the program may be interrupted at any time. See this link on interrupts. Generally, your program is prepared to be interrupted at any line of code. But sometimes, there are certain operations (blocks of code) where you want to guarantee that the program will not be interrupted. This is called an **atomic operation**.

When you want to guarantee that a section of code will not be interrupted, you enclose it in an atomic block (see here and here). You put `ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {` before and `}` after the atomic code. This will temporarily disable interrupts in the microcontroller, then restore the previous interrupt state.

A common use case is when reading or writing a 16-bit register. The microcontroller can only read/write 8 bits per instruction, so you should enclose a 16-bit read/write operation in an atomic block.

For example, in the UART library, we want to write the 16-bit LINBRR register with two separate write operations (high and low):

```
ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
    ... (other atomic code)

    // Set LINBRR 16-bit register to LDIV (high and low registers separate)
    LINBRRH = (uint8_t) (ldiv >> 8);
    LINBRRL = (uint8_t) ldiv;
}
```

According to https://www.avrfreaks.net/forum/i-am-confused-atomicforceon-and-atomicrestorestate and https://hackaday.com/2015/10/02/embed-with-elliot-interrupts-the-ugly/, this macro automatically takes care of all scope exit paths. This means it is fine to have a return statement inside an atomic block because it will restore the interrupt state before returning. In our testing, the compiler does not recognize return statements inside atomic blocks (gives an error of a non-void function missing a return statement). Just add `return 0` (or whatever is appropriate) outside the atomic block to silence the warning, even though the code will never actually reach there.

8.2.5 Volatile Variables

For any variables, structs, data structures, etc. that you modify in an interrupt handler, make sure to declare the variable as `volatile`. See this link for an explanation of what `volatile` does.

8.2.6 Functions

If you declare a function without any parameters (and nothing in the parentheses), C considers the function to accept any arguments. Functions that do not take any parameters should have `void` in the parentheses, to explicitly declare it as taking no parameters.

See a technical explanation here

```
// bad
void do_something();
...
void do_something() {
    ...
}

// good
void do_something(void);
...
void do_something(void) {
    ...
}
```

8.2.7 Header File Guards

All header files (`.h`) should have a guard statement to prevent duplicate declarations when included multiple times.

Say we have a header file called `file.h`:

```
#ifndef FILE_H
#define FILE_H

<code>

#endif
```

8.2.8 Compiler Warnings

You should not have any compiler warnings. These often detect non-obvious bugs such as bitshift overflows.

8.2.9 Delays

The following comes from the `<util/delay.h>` header (located in `avr-gcc/8.2.0/avr/include/util/delay.h`) with our MCU running at 8 MHz.

The maximal possible delay for `_delay_ms()` is 262.14 ms / 8 = **32.7675 ms** at the exact resolution. Any delays higher than this will have a 1/10 ms resolution up to 6.5535 seconds. Be aware of this in cases where you may need the precision.

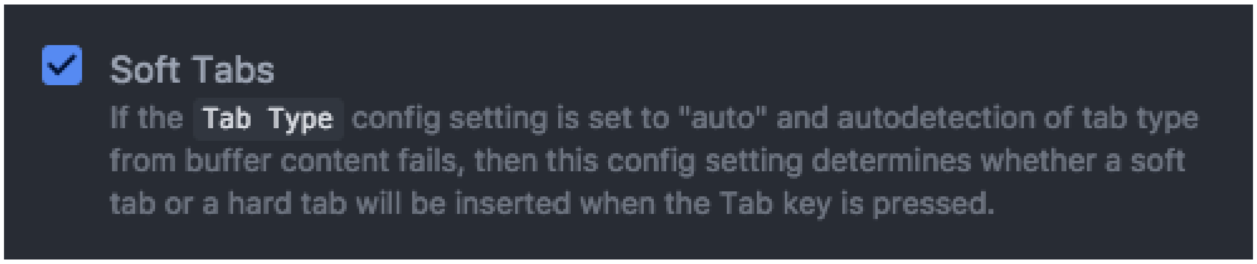
The same applies for `_delay_us()`, but the maximal possible delay at the exact resolution is 768 us / 8 = **96 us**.

8.3 Formatting

8.3.1 Spaces

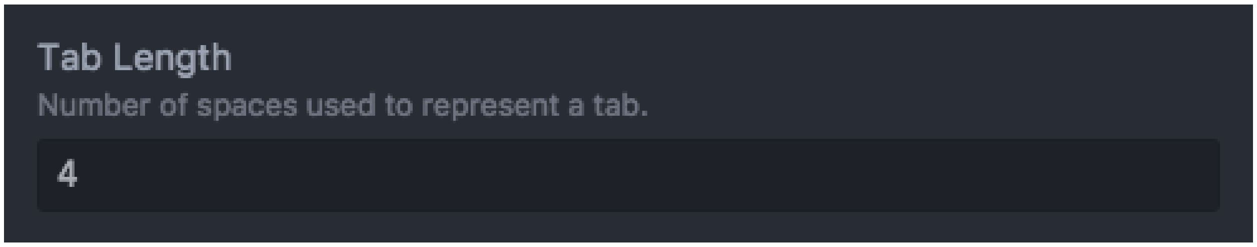
All code is formatted with **spaces, not tabs**, using **4 spaces** to represent an indent.

If you are using Atom, this is done as follows: in the menu bar, select `Atom > Preferences`, select the `Editor` tab, and change the settings as follows:



[✓ Soft Tabs]

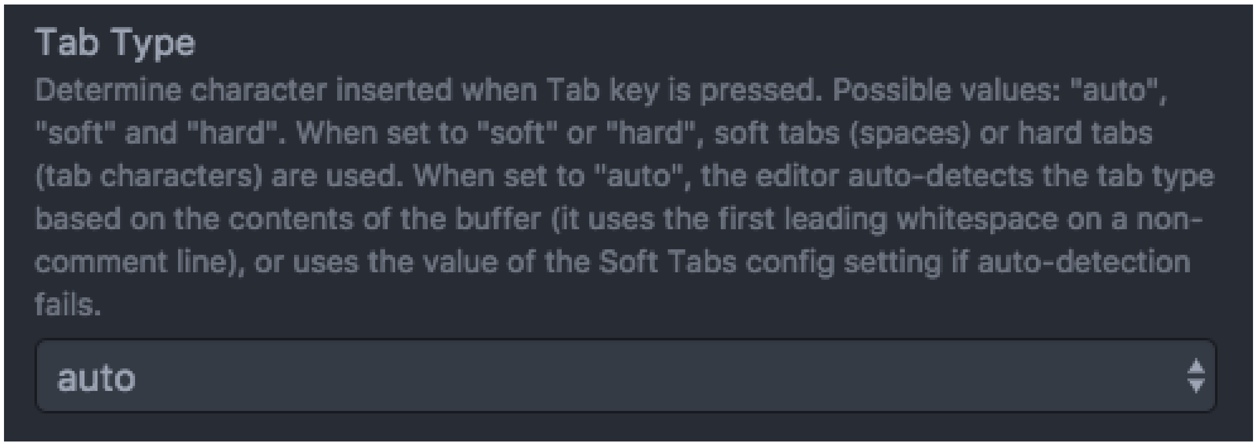
If the `Tab Type` config setting is set to "auto" and autodetection of tab type from buffer content fails, then this config setting determines whether a soft tab or a hard tab will be inserted when the Tab key is pressed.



Tab Length

Number of spaces used to represent a tab.

4



Tab Type

Determine character inserted when Tab key is pressed. Possible values: "auto", "soft" and "hard". When set to "soft" or "hard", soft tabs (spaces) or hard tabs (tab characters) are used. When set to "auto", the editor auto-detects the tab type based on the contents of the buffer (it uses the first leading whitespace on a non-comment line), or uses the value of the Soft Tabs config setting if auto-detection fails.

auto

8.3.2 Naming

Functions should be named similar to a command or action, as close as reasonably possible to how you would describe an action in English.

```c
// bad, not as natural
void uart_init(void) {
}

// good, closer to English action
void init_uart(void) {
}
```

A function's name should give some indication of which library is comes from, generally by having the library's name (or a shortened form/acronym of it) in the name.

```c
// bad, don't know what kind of callback we are registering
// (could be UART, timer, CAN, etc.)
void register_callback(void) {
}

// good, we know  this is a UART callback
void register_uart_callback(void) {

}
```

Variables, functions, and types are named using the "lower snake case" convention, where all letters are lowercase and all words are separated with underscores.

```c
// bad
void dosomethingnow() {
}

// bad
void DoSomethingNow() {
}

// bad
void Do_Something_Now() {
}

// good
void do_something_now() {
}
```

Preprocessor macros (including constants) should be named with the "upper snake case" convention, where all letters are uppercase and all words are separated with underscores.

```c
// bad
#define adc_cs PB3

// bad
#define ADCCS PB3

// good
#define ADC_CS PB3
```

Struct and enum names should end with `_t` to represent a type.

```c
// bad
typedef struct {
    ...
} mob;
typedef enum {
    ...
} clk;

// good
typedef struct {
    ...
} mob_t;
```

(continues on next page)

(continued from previous page)

```
typedef enum {
    ...
} clk_t;
```

8.3.3 Operator Spacing

There should be a space on each side of a binary operator (operates on two values) or a ternary operator (operates on three values).

```
// bad
a=b+c;
x=a?b:c;
if (thing1&&thing2) {
}
for (int i=0; i<5; i++) {
}

// good
a = b + c;
x = a ? b : c;
if (thing1 && thing2) {
}
for (int i = 0; i < 5; i++) {
}
```

For unary operators (operates on one value), there should be a space opposite the value but no space beside the value.

```
// bad
if (! enable) {
    x = ~ x;
    i ++;
}

// good
if (!enable) {
    x = ~x;
    i++;
}
```

8.3.4 Punctuation Spacing

There should be a space after commas and semicolons (except semicolons that end a line).

```
// bad
my_func(a,b,c);
for (int i = 0;i < 5;i++) {
}

// good
my_func(a, b, c);
for (int i = 0; i < 5; i++) {
}
```

8.3.5 Functions

A function should generally not be more than 40 lines long (including comments and whitespace). An easy check is that all of its code should be able to fit on your screen at once.

8.3.6 Pointers

Pointers should be declared with `*` beside the type name, not beside the variable name.

```
// bad
void my_func(uint8_t *ptr) {
}

// good
void my_func(uint8_t* ptr) {
}
```

8.3.7 Parentheses

If statements, while loops, functions, structs, etc. should have the first curly brace on the same line as the declaration, not the following line. There should be a space between the closing parenthesis and the opening curly brace.

```
// bad
int main(void)
{
    ...
}

//  bad
int main(void){
    ...
}

// good
int main(void) {
    ...
}
```

```
// bad
if (condition){
    ...
}

// good
if (condition) {
    ...
}
```

8.3.8 Switch Blocks

Do not nest switch blocks within each other; only one switch block at a time. This is because having nested switch blocks is hard to read and make it easy to forget a `break` statement.

8.3.9 Comments

Every file should have a comment at the top with a high-level description of the code in the file. It should describe what the code does, and list the author(s) of the file. If the file contains code to control a particular component, it should give the part number, a link to its datasheet, and a list of important page numbers in the datasheet. It should also describe any assumptions or operational modes used.

Example:

```
/*
MCP23S17 port expander (PEX)
Datasheet: http://ww1.microchip.com/downloads/en/DeviceDoc/20001952C.pdf

A port expander is a device with many GPIO (general purpose input/output) pins.
Each GPIO pin can function as either an input or an output, depending on what
you want to use it for. Using a port expander gives us more GPIO pins to work
with since we have a limited number on the 32M1 itself.
...
AUTHORS: Dylan Vogel, Shimi Smith, Bruno Almeida, Siddharth Mahendraker
*/
```

Every function should have a comment before it with a description of what it does. On separate lines, it should describe each of its parameters (including scientific units if applicable) and its return value (if applicable).

```
// bad

uint8_t send_spi(uint8_t cmd) {
    ...
}


// good

/*
Transmits 8 bits of data while simultaneously receiving 8 bits of data over the SPI␣
↪bus.
cmd - byte of data to transmit
Returns - byte of data received
*/
uint8_t send_spi(uint8_t cmd) {
    ...
}
```

Variables should generally having a comment describing what they represent. This is especially important for global variables or struct fields, but also useful for local variables within functions.

```
// bad

queue_t rx_message_queue;


// good

// CAN messages received but not processed yet
queue_t rx_message_queue;
```

Comments should not explain things that are obvious from C syntax.

```
// bad

// An integer representing the number of errors occurred
uint8_t num_errors = 5;


// good

// The number of errors occurred
uint8_t num_errors = 5;
```

8.3.10 Scope

Declare variables at the smallest possible level of scope for their use. Do not repeat a variable name from a larger scope within a smaller scope. Even though this is valid C, it can lead to subtle bugs and confusion.

```
// bad
uint8_t count = 0;
...
for (uint8_t i = 0; i < 5; i++) {
    uint8_t count = 0;
    ...
}


// good
uint8_t count = 0;
...
for (uint8_t i = 0; i < 5; i++) {
    uint8_t count_inner = 0;
    ...
}
```

8.3.11 Whitespace

Only one line of code per line.

Code should generally be organized into logical blocks or sections with spaces between them.

8.3.12 File Organization

For a pair of `.c` and `.h` files, `#include` statements should generally be in the `.h` file, except for the `.c` including its corresponding `.h` file.

Includes, constants, macros, and global variables should generally come first in a file.

Here is the general structure of a `.c` file, say it is called `file.c`:

```
introductory file/library comment block
#include "file.h" (corresponding header file)
global variables
functions (with comments)
```

Here is the general structure of a `.h` file, say it is called `file.h`:

```
#ifndef FILE_H
#define FILE_H

standard C library includes (e.g. <stdint.h>)
AVR library includes (e.g. <avr/interrupt.h>)
lib-common includes (e.g. <uart/uart.h>)
local file includes (same folder, e.g. "other_file.h")
#define constants
typedefs, struct definitions, enum definitions
function prototypes

#endif  // for FILE_H
```

8.3.13 Magic Numbers

Try to avoid hard-coding constants such as conversion ratios and array sizes. These should be declared as easy to read constants using `#define`.

```
// bad
uint8_t can_message[8];
for (uint8_t i = 0; i < 8; i++) {
    can_message[i] = ...
}

// good
#define CAN_MESSAGE_LENGTH 8
...
uint8_t can_message[CAN_MESSAGE_LENGTH];
for (uint8_t i = 0; i < CAN_MESSAGE_LENGTH; i++) {
    can_message[i] = ...
}
```

8.3.14 Line Length

Lines should be no more than 80 characters wide. This is a common standard for text width in editor windows.

8.3.15 Testing

When developing libraries of code, do not put temporary testing code in `main.c`. Any code that is for testing specific functions in a library should be written in a separate program in the `examples`, `harness_tests`, or `manual_tests` folders, separate from the main program.

8.3.16 Manual tests

When developing manual tests, try to print messages during the test to indicate what the user should check to see if the test is successful. This makes it more obvious for the person running the test so they don't have to memorize what things to check for. For example, print "CHECK:" when the user should check some result.

For example,

```
// ... code to set DAC output
print("CHECK: DAC pin 1 (VOUT_A) is 2.1V\n");

// ... other code to change DAC output
print("CHECK: DAC pin 1 (VOUT_A) is 0.8V\n");
```

CHAPTER 9

Testing

9.1 Unit Testing

Unit testing is a method of testing components of software for correct behaviour.

Say you want to test if a particular function works as expected. You come up with a particular set of inputs and know what the expected output is. Then you call the function, giving it those inputs and saving the actual output. Then you compare the actual output to the expected output. If you do this for a few different test cases and they all work, you can be reasonably confident the function works. This also applies if you want to check some condition or state of the program before and after calling a function, even if it is not a return value of the function.

Output values are compared to the expected values using assertions, which describe the expected behaviour. You can assert that some value is true or false, that two values are equal or not equal, etc.

Say in the future, you go back and modify a function, perhaps to make it more efficient or to restructure your program. If you wrote your unit tests properly, as long as your newly implemented function passes all the unit tests, you should be confident the function works.

9.1.1 Writing Good Unit Tests

- Your tests should not only cover cases where an operation succeeds, it should also check cases where an operation should fail (e.g. invalid message, array/queue is full, passed a NULL pointer)
- Try doing multiple operations in different orders, not always in the same sequence

9.1.2 Links

Our unit testing framework

Excellent unit test example

CHAPTER 10

Altium

This section covers various topics related to electronic schematic and PCB design within the Altium Designer CAD software. Altium is the primary CAD tool used for electronic design within UTAT Space Systems.

10.1 Getting Started

This tutorial covers getting started with Altium Designer.

10.1.1 Cross Probe

The cross probe, when clicking on a net, will expand the internal net names that Altium has given to that net. This is *very* useful for multi-channel design.

10.2 Handling Errors at Compile Time

Sometimes, for some reason or another, you need to do something in your schematic that raises an error at compilation time. The most common case you'll run into is when you need to connect an input or output pin to a bidirectional pin (say, a pin coming from a microcontroller). This will return an error in Altium, because you've potentially created a situation either where you have two outputs driving the same line (bad) or two inputs with no driving source which are left floating (less bad, but bad).

From a practical point of view, neither of these are situations you actually want in your real circuit. But, perhaps you know that you'll only be operating your microcontroller GPIO pin in input mode, so connecting it to an output makes sense. This is still dangerous (make sure your software person knows it can only be an input), and in some cases it might make sense to throw a current limiting resistor in series as a failsafe, like in the image below. This will remove the error in Altium and would be the correct way to deal with this problem. For the two inputs case, consider connecting a pull-up or pull-down resistor to the line to define a default state.

In either case, the fact that Altium is giving you an error is *telling you something important* about your circuit, and you would do well to understand how you would fix that error under normal circumstances. That said, there are still cases where you might understand that there's an error, but are OK with it and just want to move ahead

10.2.1 Showing Suppressed Messages

Under **Project > Project Options > Error Reporting** go ahead and enable the "Report Suppressed Violations in Messages Panel" checkbox at the bottom of the window. This will allow you to see the messages at compile time that you've suppressed, and give anyone checking over your work the chance to catch any potential errors before sending out the design.

10.2.2 The Generic No ERC

The Generic No ERC (third from the right in the schematic toolbar, shown as a red "X") is your go-to for handling compilation errors you want to suppress on a case-by-case basis. This will prevent you from bulk suppressing any errors unknowingly. Either select the Generic No ERC tool from the toolbar, or right click on a message in the message panel and select "Place specific No ERC for this violation…". Then, place the red "X" on the net or line that's giving you the error.

This will suppress any and all errors on this net by default, which might have unintended consequences. I would recommend going into the "Properties" panel for that ERC and selecting "Specific Violations" from the "Suppressed Violations" tab. This will give you the option to only suppress the violations you want to waive, and still warn you if other, real errors occur on that net later on.

10.3 Creating your CAM Files for Manufacturing

To be populated

10.4 Creating your Bill of Materials

To be populated.

Check that your footprints actuall exist

10.5 Creating your Output Job File

To be populated

CHAPTER 11

Electrical Systems

This section details different aspects of electronics design.

11.1 Components

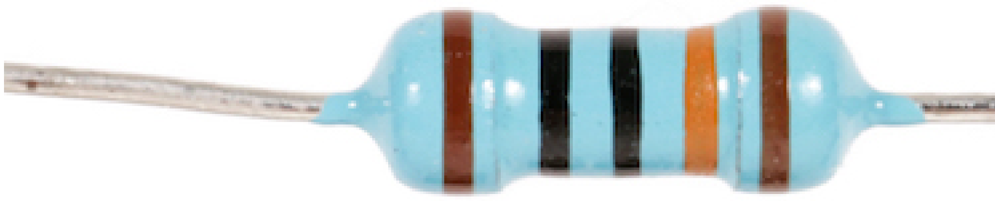
Here are some of the common electrical components used.

11.1.1 Header

Headers are used on PCBs to provide an easy interface to connect wires between boards or with a laptop interface.

11.1.2 Resistor

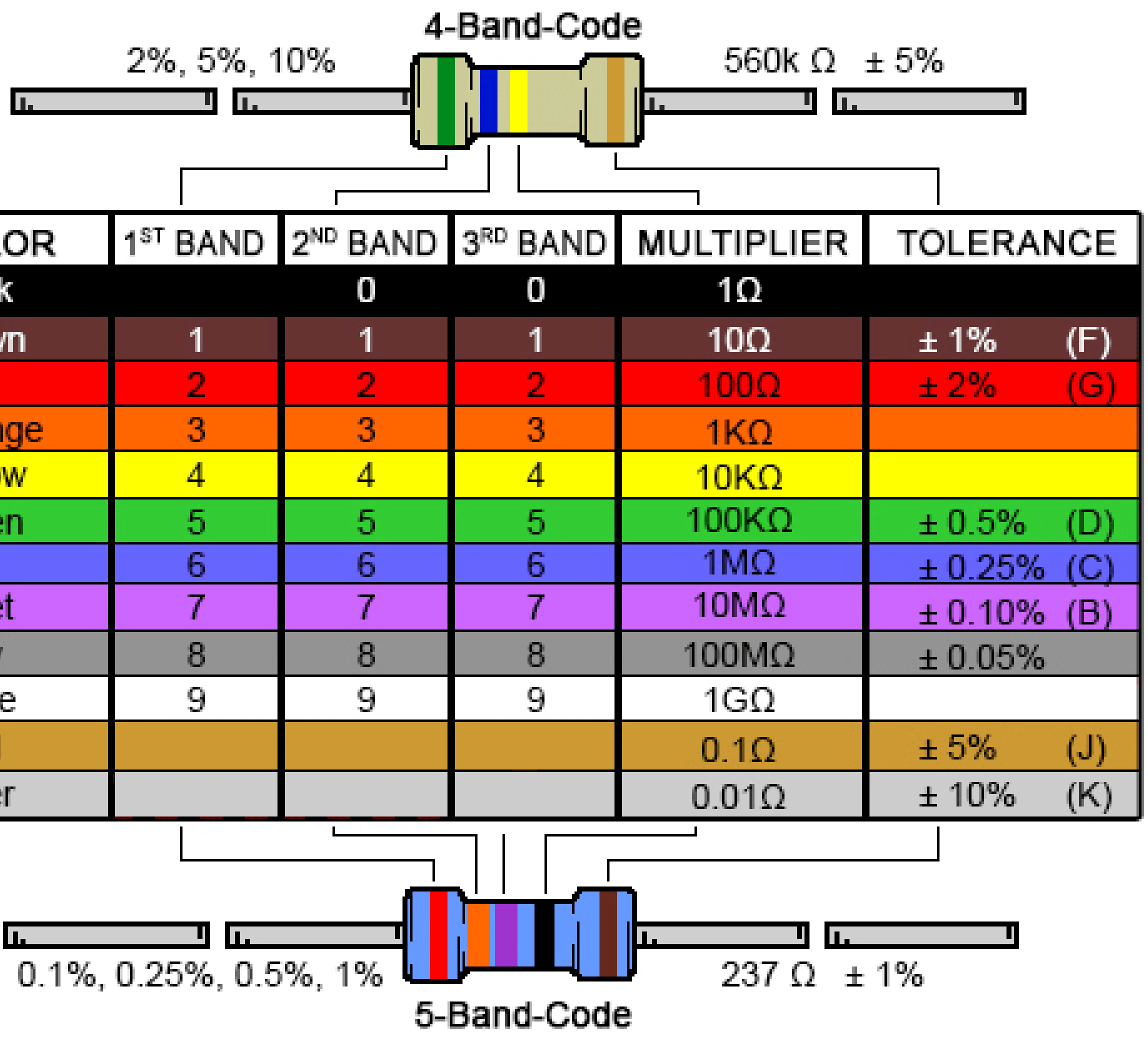
Resists the flow of electrical current.



| Through Hole | Surface Mount | | :-: | :-: | |

| |

Through hole resistors use colour-coded bands to indicate their value.


4-Band-Code
2%, 5%, 10%
560k Ω ± 5%
0.1%, 0.25%, 0.5%, 1%
237 Ω ± 1%
5-Band-Code

COLOR	1ST BAND	2ND BAND	3RD BAND	MULTIPLIER	TOLERANCE
Black		0	0	1Ω	
Brown	1	1	1	10Ω	± 1% (F)
Red	2	2	2	100Ω	± 2% (G)
Orange	3	3	3	1KΩ	
Yellow	4	4	4	10KΩ	
Green	5	5	5	100KΩ	± 0.5% (D)
Blue	6	6	6	1MΩ	± 0.25% (C)
Violet	7	7	7	10MΩ	± 0.10% (B)
Grey	8	8	8	100MΩ	± 0.05%
White	9	9	9	1GΩ	
Gold				0.1Ω	± 5% (J)
Silver				0.01Ω	± 10% (K)

11.1.3 Capacitor

Stores energy in the form of electrical energy. It can store charge and release it when necessary, so it is often used as a *decoupling capacitor* to smooth out fluctuations in a voltage.

Decoupling Capacitors

11.1.4 Inductor

Choke Coil

11.1.5 Diode

Only allows current to flow in one direction. If the current flows into the diode in the proper direction, this is called forward biased and ideally behaves as a short circuit. If the current flows into the diode in the opposite direction, this is called reversed biased and ideally behaves as an open circuit.

An LED (light emitting diode) also produces light when forward biased.

11.1.6 ADC (Analog to Digital Converter)

Converts an analog input signal within a particular voltage range to a digital signal with a specific number of bits. This is used because the microcontroller can only take digital signals as input, so analog inputs must go through an ADC before the microcontroller.

We measure fluorescence and optical density from the samples using sensors whose output goes through an amplifier chain to produce a voltage. This voltage is an analog signal (can be one of an infinite number of values in a range), but we need to convert it to a digital (discrete) format that can be used by the 32M1.

11.1.7 DAC (Digital to Analog Converter)

Converts a digital input signal with a specific number of bits to an analog output signal within a particular voltage range. This is used because the microcontroller can only output digital signals, so producing an analog output requires it to go through a DAC first.

We use DACs to output reference voltages that control the heater temperature, as well as to output reference voltages for precise optical measurements.

11.1.8 DC/DC Converters

DC/DC converters, also known as regulators, are a class of switching devices used to change one DC voltage to another, with minimal power loss. They are used to supply loads that require a constant input voltage, and which can be sensitive to changes in the input voltage. The goal of a DC/DC converter is to output a constant voltage, regardless of changes in the input voltage or output current (current drawn by the load). Depending on whether the input voltage is lower or higher than the output (desired) voltage, different types of converters can be used.

The three main types of converters are discussed below. For all converters, there are three important variables: input voltage Vg, output voltage V, and duty ratio D.

What doesn't work?

Consider the following goal: A sensitive load requires 5 V but you only have a voltage source at 10 V. How would you reduce the voltage to be appropriate for the load?

A possible response would be to use a resistor divider network. Using the equation V = Vin*(R1/(R1+R2)), all you need to do is connect two identical resistors (say, 10 Ω each) in series, and the voltage between them is half of the input. Equivalently, using the load itself as R2 and a resistor equivalent to the load resistance as R1 yields the same result. However, this is hardly a desirable solution when **efficiency** is considered.

In this resistor divider circuit, the source is supplying 10 V at, by Ohm's law, 0.5 A. According to the power equation, P = VI, the source is supplying 5 W of power. Now, the load is being supplied 5 V and draws the same 0.5 A, giving a power draw of 2.5 W. The efficiency is 50%; half of the power from the supply is wasted!

Voltage regulators, on the other hand, are able to perform this voltage conversion with minimal power loss, typically achieving over 90% efficiency.

Buck converter

Buck converters are designed to "buck" down the voltage while maintaining close to 100% efficiency. The schematic of a buck converter is shown below:

Key equations

The equation governing the operation of the buck converter is:

$$M(D) = \frac{V}{V_g} = D$$

In terms of conversion ratio, the buck converter is the most straightforward. The duty ratio simply sets the ratio of the output voltage to the input voltage! Remember that since D is always less than 1, the output voltage will always be lower than the input voltage.

Boost converter

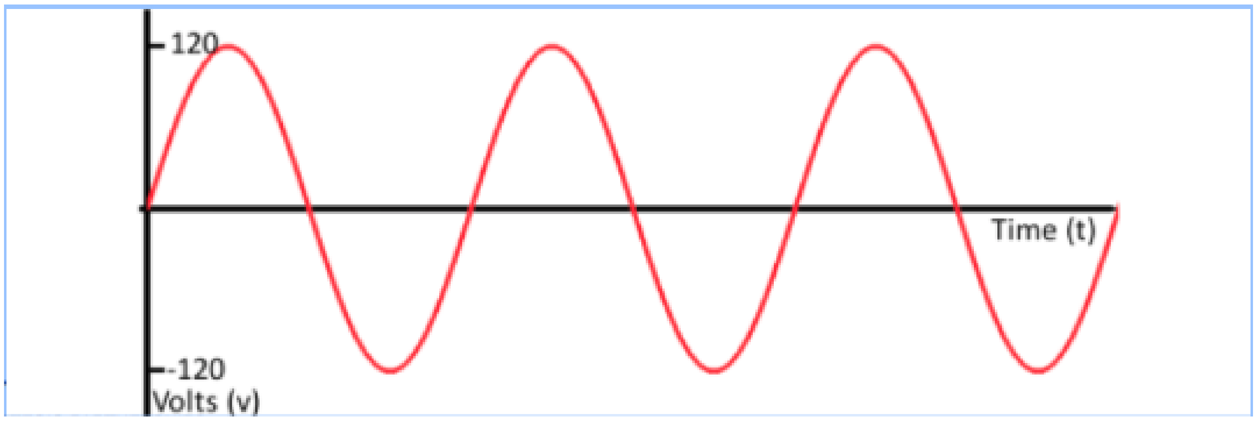
A boost converter takes a lower input voltage and outputs a higher output voltage. The schematic of a boost converter is shown below:

11.2 Analog and Digital Systems

In our satellite we have both analog and digital types of data.

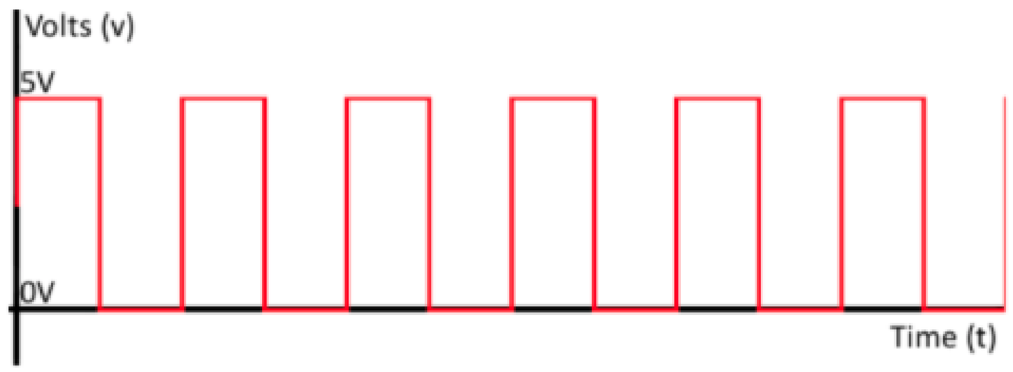
11.2.1 Analog Data

Analog data can take on any value within some range, usually represented as a voltage (infinite number of possible states).



11.2.2 Digital Data

Digital data can only take on a “high” or “low” value, usually represented as a “0” or “1” bit (two possible states).



11.2.3 Conversion

Most data in the real world is analog, but our microcontrollers can only work with digital data. That is why we generally use Analog to Digital Converters (ADCs) to convert analog inputs to digital forms, and Digital to Analog Converters (DACs) to convert digital outputs to analog forms.

11.3 Equipment

Here is some electrical/hardware equipment you will see and use in the lab (MP 099). We can demonstrate how to use the equipment in-person. These are some of the tools you will use to build, test, and debug circuits.

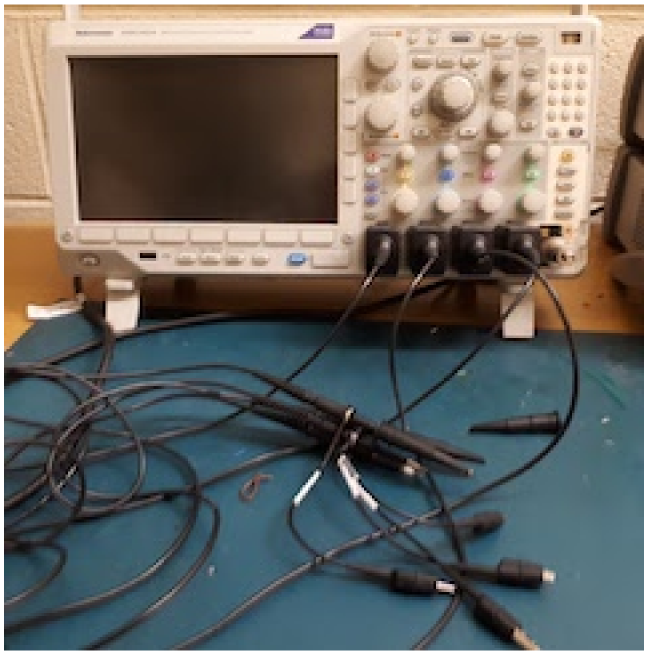
11.3.1 Multimeter

Used to measure voltage, current, resistance, and connected points in parts of a circuit.

- **Voltage** - **measured across a component** because voltage is the relative energy between two points
- **Current** - **measured through a wire**; you have to break (disconnect) the circuit at the point you want to measure, then insert the multimeter as a component in series
- **Resistance** - **measured across a component** because resistance is measured between two points
- **Short circuit mode** - used to determine if there is a short circuit **between two points** in the circuit (a direct connection through wires with no components in between)
  - After assembling a circuit, can check that you have made the intended connections
  - Can check that you have not made accidental connections that change your circuit. An unintentional short circuit can change the circuit's behaviour and/or produce a high current that can damage components.

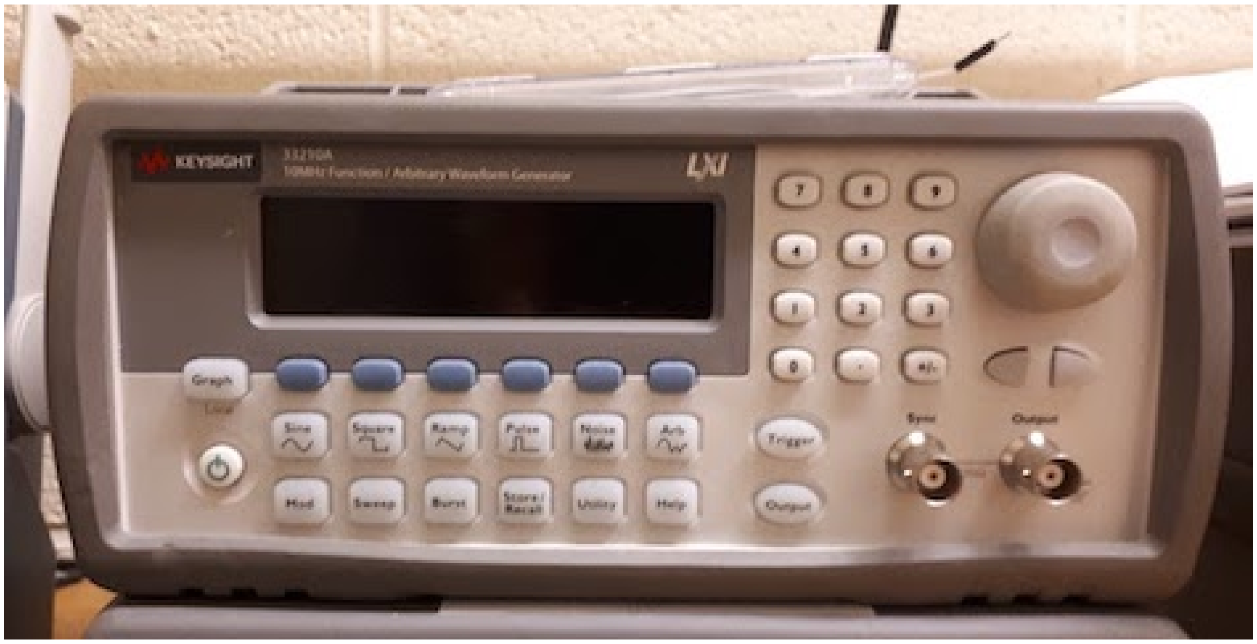
11.3.2 Oscilloscope

- Used to measure waveforms (signals) over time in a circuit
- This is useful for viewing the raw signal data in a wire, such as a sensor's output or communication lines



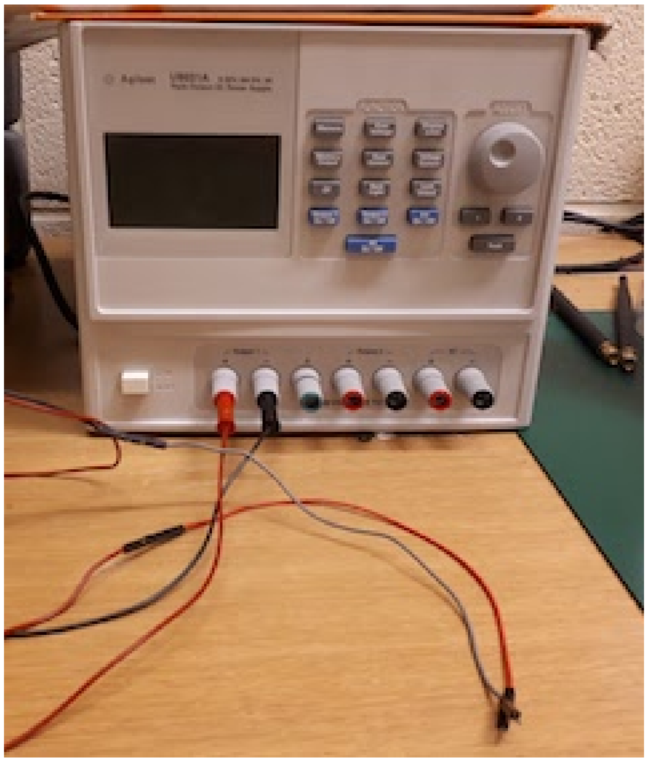
11.3.3 Function Generator

- Used to generate an AC (alternating current) signal with a specific voltage and waveform



11.3.4 Power Supply

- Used to generate DC (direct current) power with a specific voltage or current

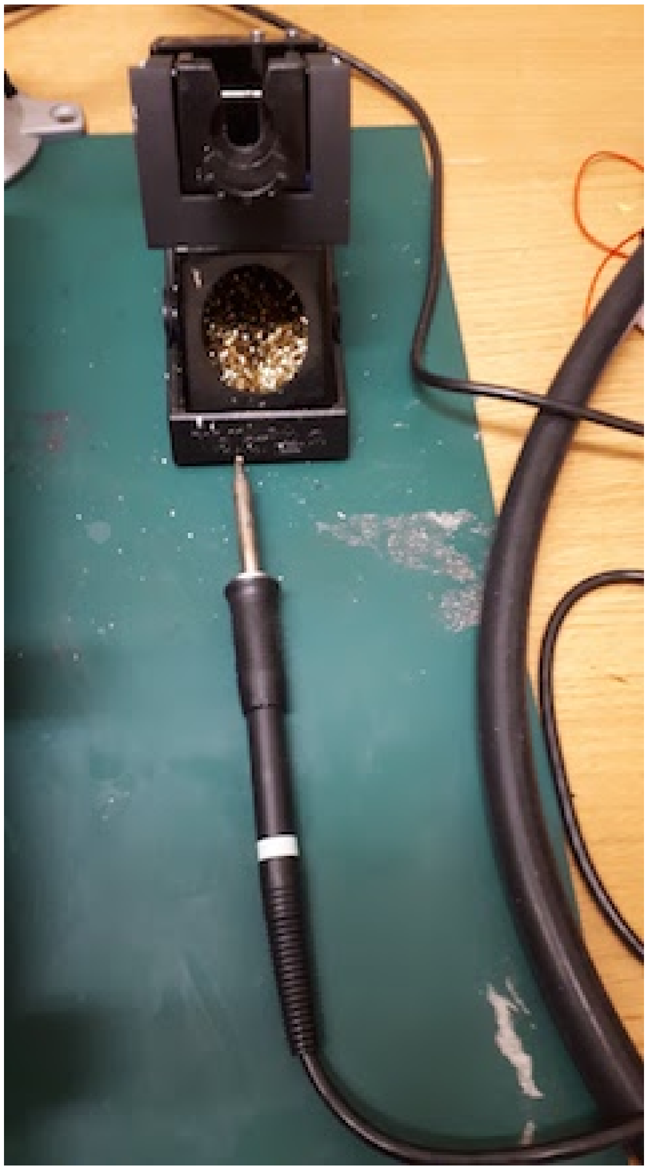


11.3.5 Wire Cutters/Wire Strippers

- **Wire cutter** - to cut specific lengths of wire to use on breadboards
- **Wire stripper** - to remove some of the insulation on the end of a wire so it can be connected to a breadboard

11.3.6 Soldering Iron

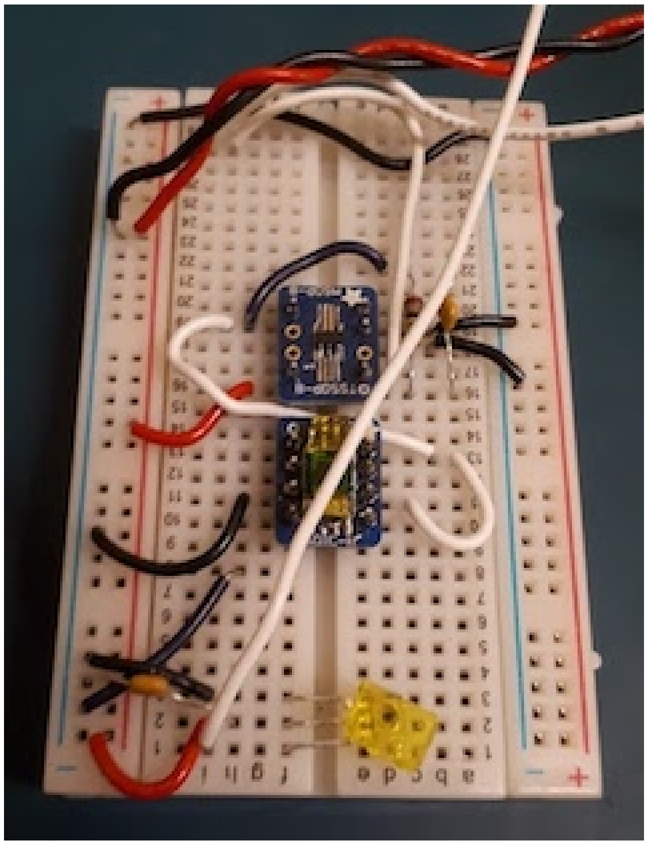
- Used to form strong electrical connections between components on PCBs or protoboards



11.4 Circuit Platforms

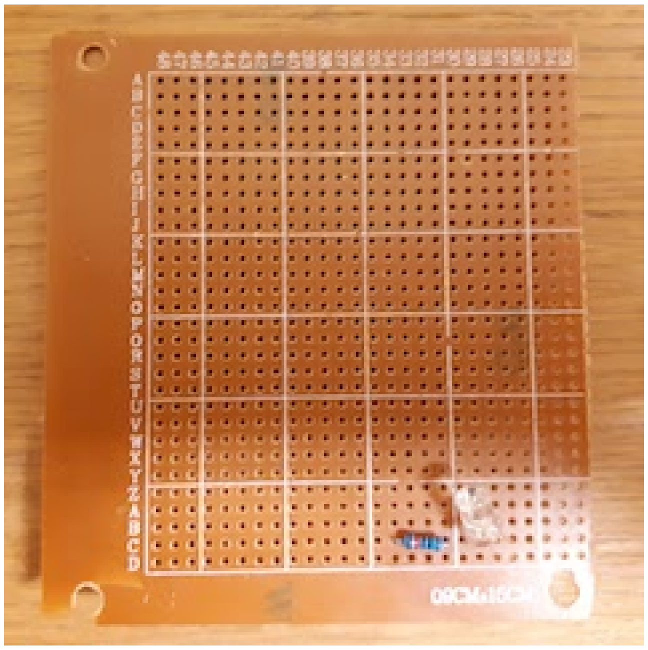
Here are the different base platforms for building circuits.

11.4.1 Breadboard



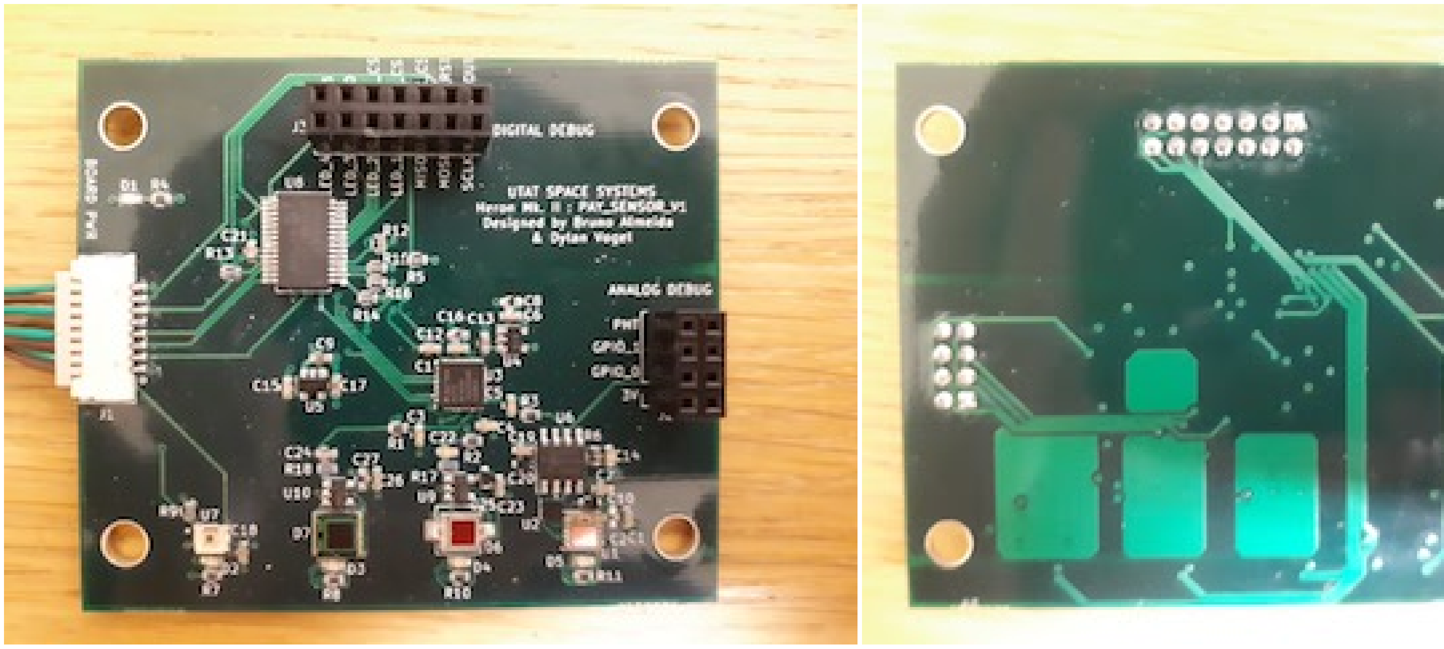
- Used to quickly build and change circuits for fast prototyping
- Has holes to insert through-hole components and wires
- Components and wires can be moved easily, but can also be knocked loose easily
- Uses through hole (TH) components
- Particular sets of holes are connected under the board:
  - Each long rail of holes is connected together (there are two rails on each side). It is general convention to use the red rails for power and the blue rails for ground.
  - Each row of 5 holes in the main part of the board is connected together. These are used to connect components together.
  - If you forget which holes are connected, follow the lines and notice the breaks in the lines (add diagram/photo with lines indicating the connected sets of holes)

11.4.2 Protoboard



- Has connected tracks like a breadboard, but through hole components are soldered and are more likely to stay on the board

11.4.3 Printed Circuit Board (PCB)



- Used to create final circuits or major prototype versions of a circuit
- A PCB's connections can't be changed after it is ordered and printed (with some exceptions, but it is very difficult and unreliable)

- Generally uses surface mount (SMD/SMT) components, which are much smaller than through hole

11.5 PCB Design

A PCB (Printed Circuit Board) is a circuit board that is designed and manufactured using CAD (Computer Aided Design) software, also called EDA (Electronic Design Automation) software.

Getting a functional PCB requires several steps:

- Design the board's connections and layout using CAD software
- Send the design to a manufacturer, who manufactures the board with just the traces (electrical wires)
- Order all of the components
- Receive the board from the manufacturer
- Purchase the necessary components
- Solder the components onto the board.

This process usually takes at least a month, which is why PCBs are not used for fast prototyping.

11.5.1 Software

We use two different pieces of CAD software.

KiCad (website)

- Free and open source
- Easier to learn
- Available for Windows, Mac, and Linux

Altium Designer (website)

- Paid, but UTAT is sponsored by Altium and gets free licenses
- More difficult to learn
- Only available for Windows

11.5.2 Design Process

Designing a PCB is generally separated into two main steps.

Schematic

The schematic specifies all the components and how they are logically connected to each other to create circuits (which pins are connected). This does not have anything to do with the physical arrangement of components.

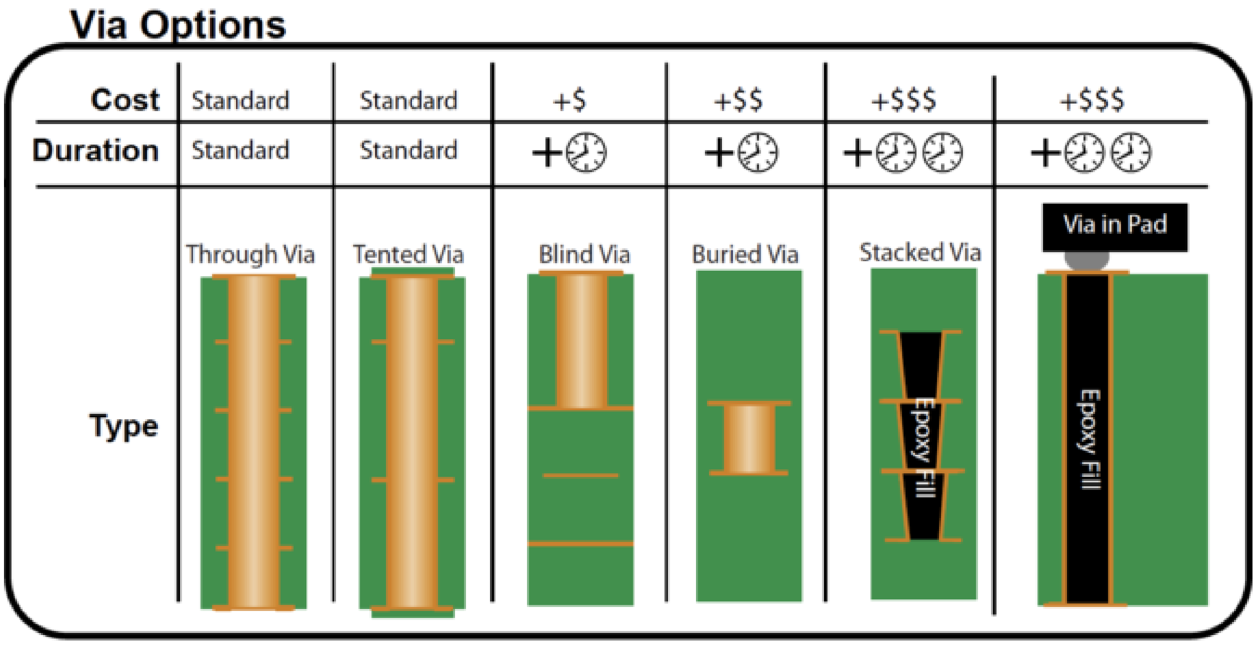
PCB Layout

The PCB Layout involves taking the schematic and physically laying out the components and connections as they will be located on the PCB. After placing the components, you route the wires (also called traces or tracks) so they do not cross one another. Altium Designer has an autorouter that can automate this process.

11.5.3 Layout

The simplest PCB uses 1 layer and places all component on one side, but multiple layers can be used. Some PCBs have 2 layers, where you can put components and traces on the other side as well. Some have 4 layers, such as the payload sensor PCB which has dedicated layers for power and ground connections to minimize electrical noise. In any case, you can only put components on a maximum of 2 layers (top and bottom). Other layers are useful for dedicated power/ground planes or to route traces that are impossible to put on one layer without crossing.

Vias



A via is a hole that connects a trace on one layer to another layer. They are useful for routing traces that cannot be routed on a single layer without crossing other traces. For example, you can start a trace on the top layer, place a via to the bottom layer, route it along the bottom layer, then place another via bringing it back to the top layer.

11.5.4 Components

Schematic Components

A schematic component describes a component's type and pins.

Footprints

A footprint describes a component's physical size and shape.

11.6 PCB Checklist

A good checklist before finalizing and ordering PCBs, courtesy of Jaden.

1. PCB design rule file, pad and via libraries have been imported to prototype PCB project
2. Schematic components are connected according to datasheet specifications (eg. pull-up-or-down resistors, etc.) and schematic compiles without error
3. Schematic components have legal paths to the correct footprints for the design
4. Footprints reflect the pad layout of the component itself (as per datasheet). Effort has been made to add 3D models to components
5. Components are laid out on the board with appropriately sized traces/pours in high current applications, and follow any configuration recommendations found in the datasheet (eg. bypass capacitors being in proximity to their associated pin and with their own connection to GND)
6. Silk screen (top overlay, the yellow one) labels are 1mm high and are placed near their respective components. Pin 1 symbols for components are indicated
7. PCB is as small as can be made without compromising board functionality. Boards are usually 4 layers, sometimes 2 layers
8. Important components in need of being spec'd have part numbers from DigiKey associated in Altium for BOM
9. You have exported a schematic PDF of your PCB for viewing and have sat back to be proud of your design

11.7 Footprints

Every electrical component has a **footprint** which describes its physical size and shape.

Surface mount device naming conventions: http://www.pcb-3d.com/tutorials/ipc-7351b-naming-convention-for-surface-mount-device-3d-models-and-footprints/

Through hole device naming conventions: http://www.pcb-3d.com/tutorials/ipc-7251-naming-convention-for-through-hole-3d-models-and-footprints/

11.7.1 Surface Mount and Through Hole

Surface mount (SMT/SMD) components lay on the surface of a PCB face (only on one side). Most of our components are surface mount because they are smaller and take up less space.

Through hole (TH) components go through a hole in the PCB from one side to the other. These are less common because they are bigger and take up less space, but are generally used for header pins and other connectors.

11.7.2 Resistors and Capacitors

Resistors and capacitors generally use either the 0603 or 0402 footprints. As described here, this is an imperial naming convention for width and height in thousands of an inch (sorry, but it's the general standard).

Generally we try to use 0603 resistors and capacitors since they're easier to solder. Sometimes, we have to use 0402s because of space constraints, which are more difficult to solder. Occasionally we have used 1210 resistors if they need to dissipate a lot of power.

http://blog.mbedded.ninja/electronics/components/resistors http://blog.mbedded.ninja/electronics/components/capacitors

11.7.3 Selecting Footprints

Generally, selecting which footprints to use is a tradeoff between size and ease of soldering. Bigger components are easier and more reliable to solder, but smaller components are often required due to space constraints.

11.8 Differential Signals

Differential Signalling Article

Generally we use **signal-ended signalling**, which means we have a single point/wire for the ground (voltage reference) and one wire for each signal we send. This means adding a signal only means adding one wire. The voltage of a signal is measured as the voltage of the wire relative to the **common ground**. For most applications, this is the best method.

Some applications use **differential signalling**, which means we have two wires (high and low) for each signal we send. Instead of measuring the voltage of the single wire relative to the common ground, we measure the signal as the voltage difference between the two wires. This is useful for protocols such as CAN where the data bus is differential (CANH and CANL signals). This is sometimes used because it produces less interference and is less affected by surrounding interference, so signal integrity is improved. This is because the two wires are routed right beside each other, so any interference received is received by both wires and cancels out.

CHAPTER 12

Reading Datasheets

Presumably, this section will contain lots of information on reading datasheets.

12.1 Reading Datasheets

A datasheet is a technical document that provides detailed information about the operation of a component, written by the company that designed it.

Datasheets are generally very dense and technical, so they are often confusing and difficult to read. This is an important skill and resource, so the key is learning how to find the information you want for a particular purpose.

For your subsystem, the datasheets you work with will usually be located in a Literature folder within your subsystem folder on the Google Drive. You can also search for datasheets on the Google Drive or using a general Google search.

Here is some of the most commonly used information in datasheets for different purposes.

12.1.1 Electrical (prototyping or PCB design)

- pin configurations and descriptions
- package/footprint/dimensions
- power supply voltage or other voltage reference
- grounding and layout

12.1.2 Software

- pin configurations and descriptions
- register descriptions and bit allocation
- digital interface

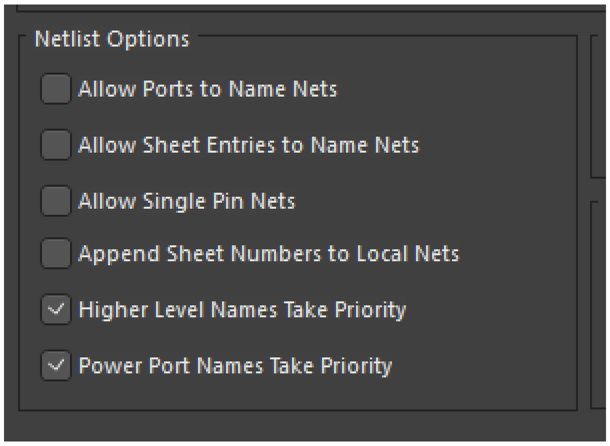
- timing diagrams (usually SPI)
- operational modes

CHAPTER 13

Electrical Style Guide

13.1 Electrical Style Guide

Use the following netlist settings under Project Options -> Options:



Netlist Options

- [ ] Allow Ports to Name Nets
- [ ] Allow Sheet Entries to Name Nets
- [ ] Allow Single Pin Nets
- [ ] Append Sheet Numbers to Local Nets
- [✓] Higher Level Names Take Priority
- [✓] Power Port Names Take Priority

In PCB layouts, reduce clutter in the silkscreen overlay - clean up repeated component names:

- Click an empty space in the PCB layout
- Open the Properties panel and scroll to the bottom
- Change “Designator Display” from Physical to Logical

CHAPTER 14

Remote Access

This section details different aspects of remote access, including remote programming.

14.1 VPN Setup

The following set of instructions will tell you how to get onto the VPN from your home computer.

Required files (Ask IT members for them):

- cacert.pem (certificate file)
- client.pem (client cert file)
- client.key (client key file)
- er.ovpn (configuration file)

Note that the names may be vaguely different and that the .pem extension may be replaced with .crt

You will be prompted to enter a username and password to login to the VPN. Ask an IT member to create an account for you and for the account's details.

14.1.1 Steps

Mac

1. Create a directory to place your vpn configuration in (such as Desktop/openvpn)
2. Place the ca.cert, client.crt, er.ovpn and client.key files in that directory
3. Download Tunnelblick (it's free!) or some other OpenVPN client management software
4. If you download tunnelblick, make sure to upload the er.ovpn file by either clicking it or adding it to the Tunnelblick Configurations
5. Click `connect` to connect to the VPN

6. You will be prompted to enter a username and password. Enter the account given to you by the IT member
7. The time taken to connect shouldn't be more than a minute. You will be informed if the connection is successful
8. Congrats you should be on!

Windows

1. Create a directory to place your vpn configuration in (such as Desktop/openvpn)
2. Place the ca.cert, client.crt, er.ovpn and client.key files in that directory (need to unzip the folder)
3. Download OpenVPN for Windows
4. Import the er.ovpn file from the OpenVPN GUI menu or drag the file into the GUI window
5. You will be prompted to enter a username and password when the file is successfully imported. Enter the account given to you by an IT member
6. You should be connected within a minute or so.

Linux

1. Create a directory to place your vpn configuration in, then place the four required files mentioned above in that directory.
2. If you are on Ubuntu/Debian Linux (or its variations), open your terminal and run `sudo apt-get install openvpn` to install openVPN for Linux.
   - OpenVPN also supports Fedora/CentOS/RedHat (install with `yum` instead of `apt`)
3. Once the installation is successful, navigate to the directory where your er.ovpn file is placed in terminal and run `sudo openvpn --config er.ovpn`.
4. Texts will start flying through the terminal and you will be prompted to enter your username and password at some point.
5. Once the connection is established, you should see something like "Initialization Sequence Complete" and you should be good to go. The connection will stay established as long as the terminal windows is open.

Resource for Linux OpenVPN setup instructions

Guideline for OpenVPN setup instructions

CHAPTER 15

Remote Programming

15.1 Introduction

This page describes setup and general best-practices for using a remote programming setup. For connection details and login credentials, please talk to someone from the software team.

Before getting started it's useful to have an uderstanding of basic command line usage and tmux commands. Also, you'll have to ask someone on the Slack what hardware is currently connected or set it up yourself. In the future we may have some standard way of determining current hardware setups.

15.2 Setting up the Hardware

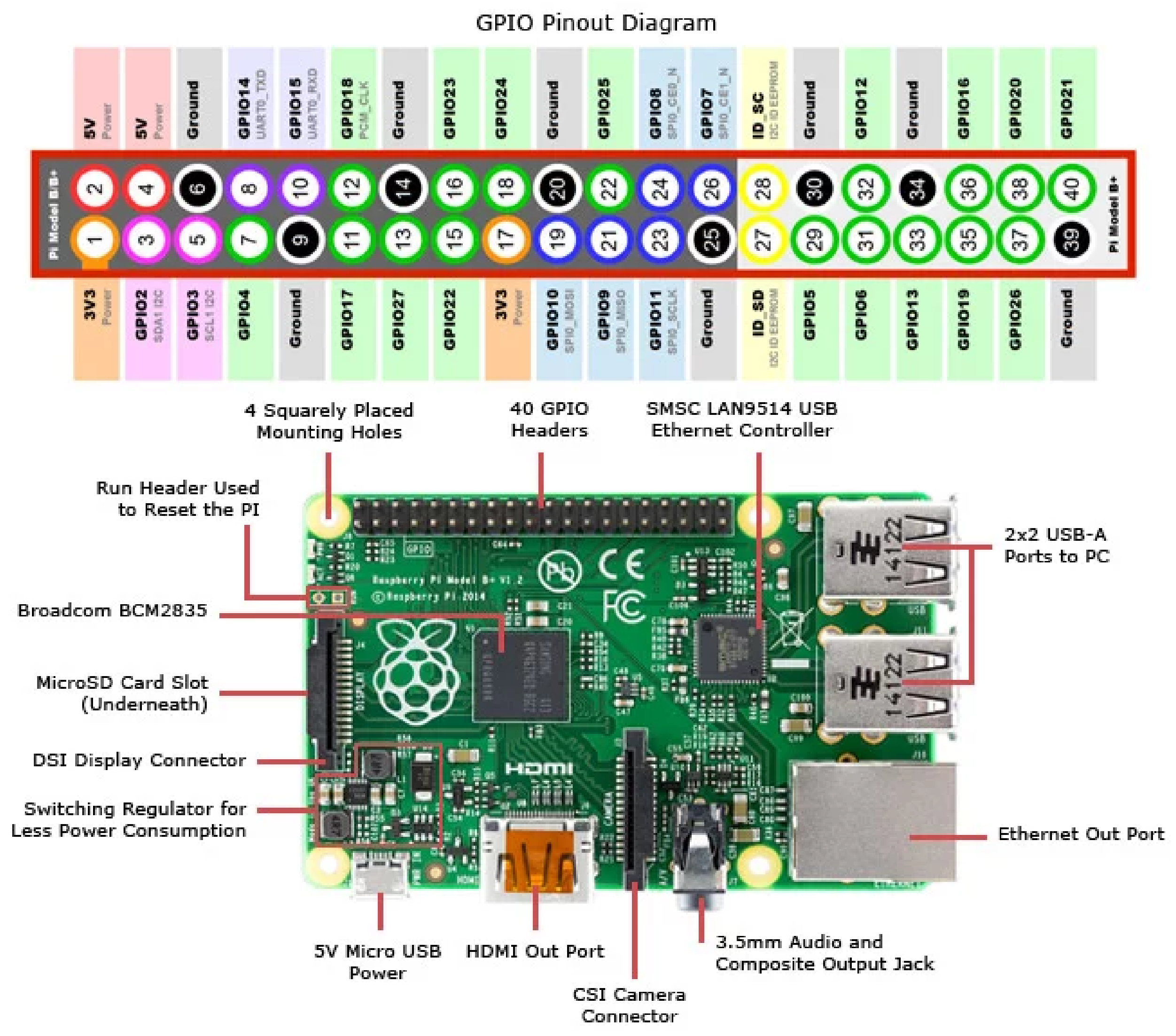
If you require the use of UART TX (i.e. sending serial from your computer to the PCB), then the only hard requirement is that the RXD pin on the microcontroller must be directly accessible through a debugging header. **The standard 4-pin UART header will not work** because the PGM<->RUN mode switch disconnects RXD on the microcontroller during programming. However, since we have direct control over the Raspberry Pi GPIO, we can simply toggle the UART on the Pi before and after programming to get around this issue while developing remotely. Look for "TX" and "RX" labels on your debugging header to see if your board supports it. **Remember to leave the PGM<->RUN mode switch in program mode** to ensure that you can program remotely.

For example, in the case of the OBC PCB, you would use the TX and RX pins on the horizontal header across the south side of the board rather than the 4-pin header in the north-east corner.

15.2.1 Wired Connections

The Raspberry Pi requires a 5V power supply with at least ~1A of current capability. The official Raspberry Pi foundation recommends 2.5A to ensure that the board always has a steady power source. The Raspberry Pi also needs a wired ethernet connection, but otherwise can be run entirely headless.

We'll be making use of the Pi's GPIO header. An image of the standard 40-pin header used on all Raspberry Pi's is included below for reference.



The Pi will be used to supply 3V3 and UART to the PCB. Connect jumper wires between 3V3 and GND on the Pi and 3V3 and GND on your PCB. In some cases, the inrush current will cause the Pi to restart. We recommend that you first shutdown the Pi safely using `sudo shutdown now` and disconnect the power before changing any of the hardware connections.

While you have the jumpers out, connect UART_TXD (GPIO14) to RX on your PCB and UART_RXD (GPIO15) to TX. This finishes the hardware setup.

Plug back in the Pi and check that it starts up normally. If the red LED on the Pi blinks during startup and seems to power cycle, this is probably *because it's power cycling*. Something on your PCB (or the Pi itself) is drawing too much current during startup and the power adapter you're using doesn't have enough juice to keep the rail up. Disconnect your PCB and see if the issue goes away, or try using a different power adapter.

15.3 Getting Your Bearings

Log into the remote programming machine over SSH. By default, the user directory should contain a `github` directory. This folder is where you should checkout any GitHub repositories you're developing on.

List out the tmux sessions using `tmux ls` to see if anyone else is currently using your machine. If you get something like `error connecting to ...` then tmux isn't running and you're safe to use the setup. If you do get a list of sessions, get in touch with whoever is using it and see if they're still active.

Start a new session from the command line by calling `tmux new -s <your descriptive session name>` and make sure to include your own name in the title (e.g. `dylan-obc-dev`). This way if someone else logs into the machine (and follows these instructions) they'll see your session and get in touch to see if you're still using it.

While you should read up on available tmux commands, a few basic ones to get you started are:

Ctrl+b %	Split your tmux pane vertically
Ctrl+b "	Split your tmux pane horizontally
Ctrl+b o	Cycle through your active panes
Ctrl+b d	Detach your current tmux session
Ctrl+b $	Rename your current session
Ctrl+b [	Scroll through your terminal history ('q' to exit)
tmux a -t <session name>	Attach to a detached session

15.4 Developing on a Remote Machine

Since anyone could have been logged onto the machine before you, make sure to call `git status` and `git pull` to check the current branch and pull any updates. You should ensure you're working on your own branch when on a shared machine. Also, make sure to **commit any work-in-progress** and push it before logging out, lest someone delete your work.

Code development on the remote programming setup is about as you would expect. Write some code, compile it using a call to `make` and upload it to your PCB using `make upload`. Feel free to code in Vim for that authentic experience, otherwise VS Code offers an excellent Remote - SSH extension which allows you to develop on a remote machine from inside VS Code. Just remember to launch up the terminal inside VS Code and perform the same tmux setup as described in *Getting your Bearings*.

15.4.1 Checking Connected Programmers

It's possible to poll connected programmers from the command line using the `pavr2cmd` command. Call `pavr2cmd --help` to get a list of the valid commands. A few useful ones are summarized below:

pavr2cmd -s	Get the status of the connected programmer
pavr2cmd –prog-port	Get the programming port of the connected programmer

15.5 UART TX and RX on a Remote Machine

If you followed *Setting up the Hardware*, your target board should be connected to the `/dev/serial0` port. This is the UART port mapped to pins 14 and 15 (BCM numbering) of the the Raspberry Pi. There is some setup involved in getting these pins to work as UART on a Raspberry Pi, but this should have already been done on your machine.

If you already have a good understanding of how to use a serial port in Linux, feel free to do it your way. Otherwise, this is the workflow that we recommend.

15.5.1 Getting Started

This method involves direct reads and writes from the `/dev/serial0` file. Serial configuration is done using the `stty` command line utility. To make life easier, we've added a few macros to `~/.bashrc` to make serial interaction easier.

To begin, call `start_serial` from the command line to set up the `stty` configuration. We also recommend creating a new tmux window or pane and splitting it to view TX and RX at the same time.

15.5.2 UART RX

Just call `cat /dev/serial0` from the command line. This command should block if there's nothing in the serial buffer, and display new lines as they come in. Hit `Ctrl+c` to quit RX.

In case you forget the port, we've also defined the `$serial_port` bash variable. So a call to `cat $serial_port` should have the same effect.

15.5.3 UART TX

We've defined the bash macro `ws <some text here>` to write to the serial port. For example, to send "Hello, World!" over the serial port simply call `ws "Hello, World!"` from the command line.

For context, `ws` is meant to be short for "write serial". The macro is simply calling `echo <some text here> > /dev/serial0` in the background to write to the serial port file.

If you would like to contribute to our documentation, create a branch and start a pull request. If you would like to report an error, please submit an issue.