# Dash Hacking: Bare-Metal STM32 Programming

Created by Tony DiCola


https://learn.adafruit.com/dash-hacking-bare-metal-stm32-programming

Last updated on 2023-08-29 02:59:06 PM EDT

# Table of Contents

# Overview

The Amazon Dash button () is a tiny device that orders products from Amazon.com () a t the press of a button.  It's designed to be put wherever you store consumeables like paper towels, trash bags, etc. so that you can easily order more when they run out.  The Dash is great at what it's designed to do, but did you know inside the Dash is a powerful ARM Cortex-M3 processor and WiFi module that are very similar to wireless development boards like the Particle Photon ()?  You'll even find there are easily accessible test pads on the Dash which allow you to reprogram its CPU and turn it into your own $5 internet button!  This guide will explore how to take apart the Dash and reprogram its CPU to run your own code.

This is a good introduction to 'bare-metal' embedded development where you write code to run on a chip without any operating system.  Just like an Arduino you have total control over what the CPU does, but unlike Arduino you need to get closer to the hardware to tell it exactly what to do.  Be warned that you'll want to have some experience soldering, programming C, and using development tools from the command line to follow this guide--this is not a good intro to electronics project!

This guide builds on some great work by others to understand the hardware available on the Dash.  In particular this Exploring Amazon Dash Button project () and Amazon Dash Teardown blog post () are good sources of info that describe the Dash hardware:

- The CPU is a STM32F205RG6 () processor which is an ARM Cortex-M3 that can run up to 120mhz and has 128 kilobytes of RAM and 1 megabyte of flash memory for program storage.
- The WiFi module is a BCM943362 module () which in combination with the CPU make it a platform for Broadcom's WICED SDK ().
- There's a 16 megabit SPI flash ROM which is typically used in conjunction with the WICED SDK for storing application data.
- An ADMP441 microphone is connected to the CPU and used by the Dash iOS application to configure the device using the speaker on a phone/tablet.
- There's a single RGB LED and a button.

It's still early days in the undestanding of the Dash hardware so this guide will only show how to program the Dash CPU and use its LED.  Unfortunately the WiFi module isn't useable yet until a little more investigation is done to understand how it's connected to the Dash CPU and exposed to the WICED SDK.  For now you can control the LEDs and even output data on a serial UART with the example code in this

guide.  In the future as more Dash functionality is understood later guides can explore using more Dash features like its WiFi radio.

WARNING: Follow the steps in this guide at your own risk!  The Dash isn't designed to be taken apart and reprogrammed.  Your Dash hardware might be completely different from the hardware in this guide and could be damaged or destroyed--you have been warned!

Continue on to learn about how to disassemble the Dash and access test pads for reprogramming the CPU.

---

# Connections

# Parts & Tools

To disassemble and program the Dash you'll need at least the following parts and tools:

- T5 Torx (star-shaped) driver. (http://adafru.it/4523904)
- Small and large flat-head screwdrivers or a jimmy (http://adafru.it/2414) for prying.
- Soldering iron/station (http://adafru.it/1204) with a fine tip (http://adafru.it/1249).
- Thin solder (http://adafru.it/1886) (~0.02" or less thick) and thin wires (http://adafru.it/1446) (26-30 AWG).
- STLink V2 programmer (http://adafru.it/2548) and female jumper wires () for accessing its pins.
- Vice (http://adafru.it/151), helping hands (http://adafru.it/2474), or other board holding tools.
- Magnification and light for inspecting small connections.
- Solder wick (http://adafru.it/149) and/or sucker (http://adafru.it/148) in case you make a mistake.

It will also help, but isn't required, to have a 3.3 volt serial to USB cable () so you can output debug information from the Dash CPU and read it on your computer.

# Disassembly

You'll need to start by disassembling the Dash so you can access programming test pads on the circuit board.

WARNING: Disassemble the Dash at your own risk!  Remember the Dash is not designed to be opened by users and will require some force to open.  Take the proper safety precautions to protect your eyes and body from harm when using tools to open the Dash.  The instructions below are a suggestion and they might not work with current or future Dashes.

Start by prying the sticker off the top of the dash using a thin screwdriver.

I found it was easiest to start from a long edge like the top or bottom and wedge the screwdriver inbetween the case and sticker, then pry up the sticker to get started.  Don't let the screwdriver slip and hurt you!

Then unscrew the 3 screws that hold the top of the case to the Dash.  You will need a T5 size Torx (star-shaped head) driver to remove these screws.

After removing the screws you will need to pry the top cover off the bottom of the Dash as it is glued firmly in place.


I found sticking a large flat screwdriver into the groove between the two case halves and twisting it slowly while pushing in was the quickest way to separate the halves of the case.  Note that unless you are very careful you will probably mar the plastic of the case (it's just a beauty scar to show the world you've hacked your Dash!).


Pull the Dash circuit board out of the case. You will notice on the back of the Dash board a AAA lithium battery is connected. Unfortunately the battery is welded to tabs on the circuit board so it is not a simple process to remove or replace the battery.

To remove the battery I found it was easiest to use flush cutters and snip the metal tabs that hold it to the circuit board. Remove the ground / negative side of the battery first, then remove the positive side and pull the battery from the board (it is glued in place lightly). Don't try to remove the positive side first as you will likely short the battery against the grounded metal shield of the microphone near it.

Do NOT try to remove the battery from its tabs with a soldering iron! The heat of the iron will potentially damage the battery or even cause it to explode.

Under normal use the Dash will spend most of its time in a low power sleep mode where it consumes extremely small amounts of power. When the button is pressed the Dash's processor wakes up, connects to the WiFi network, and sends a signal to Amazon's servers. This means the tiny battery of the Dash can last for a very long time, even years, because it is rarely ever awake.

However for your own projects unless you are careful to sleep and use the processor as little as possible you will likely exhaust the Dash's battery in a short amount of time (hours). Since the battery can't easily be replaced the best option is to completely remove the battery from the Dash and power it from an external power source.

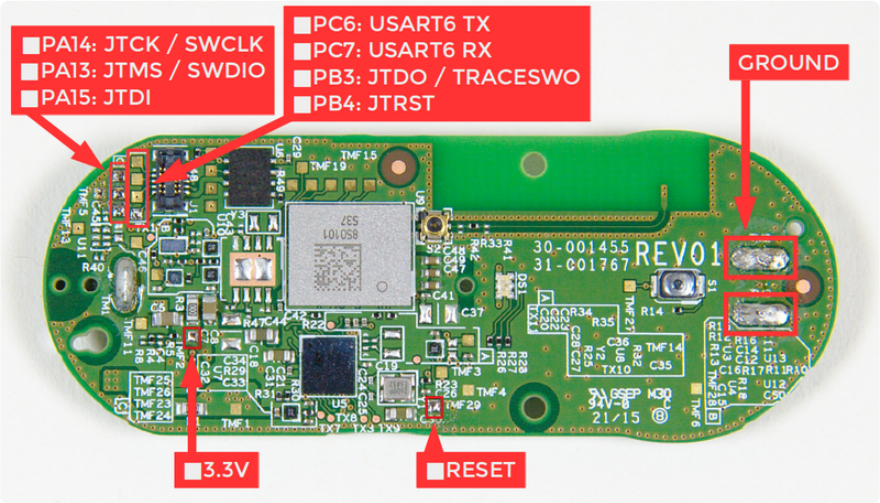
Do NOT leave the battery connected and apply an external power source to the Dash! The Dash is NOT designed to recharge the battery.

After removing the battery your Dash is completely disassembled. Now follow the steps below to solder to the programming test pads of the Dash's circuit board.

# Soldering

Once you have access to the circuit board first check the revision number printed on the board.  Be aware this guide was written with a Dash revision 01 (notice the REV01 printed on the board).  Different revisions might not have the same test pads exposed so be careful to check before you solder any connections.

Below is a diagram of the Dash circuit board with the important test pads labeled.  Much of this Dash reverse engineering comes from Matthew Petroff's excellent Dash teardown blog post () and dekuNukem's Exploring Amazon Dash button project () (in particular check out his more complete test pad diagram ()).




Note that some of the highlighted test pads above are silver colored because they have solder left on them from previous connections that were made.  An untouched board will have shiny copper pads everywhere.

To connect to the Dash's CPU you can use its single-wire debug interface by soldering wires to at least the following connections:

- PA14 SWCLK in the upper left corner.
- PA13 SWDIO immediately below SWCLK.
- 3.3V power in the bottom left of the board.
- RESET in the bottom middle of the board.
- GROUND on either of the large battery holder solder joints on the far right of the board.

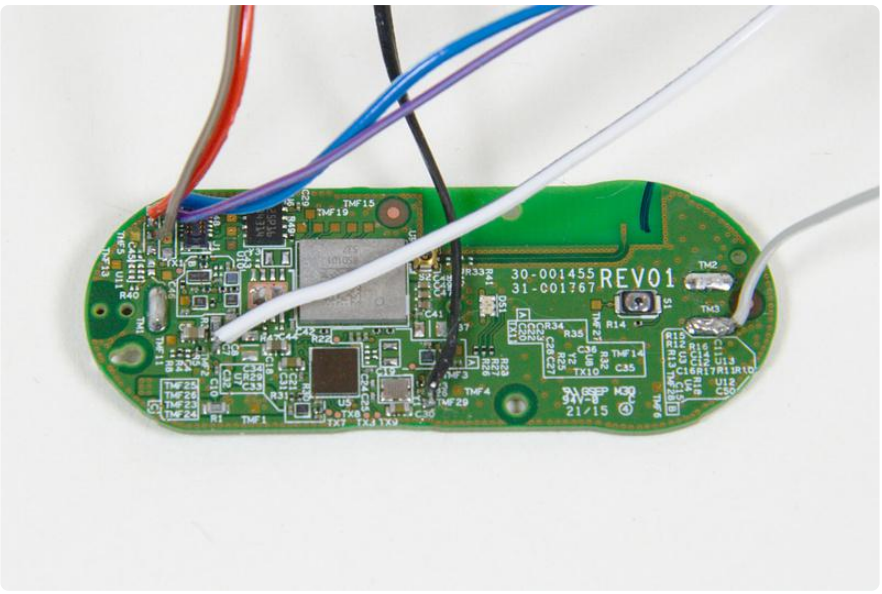
Although the test pads are somewhat small it's actually pretty easy to solder to them if you follow this advice:

- Use thin wires like 26-30 AWG thickness.
- Use a fine point soldering iron tip. There aren't a lot of obstructions near the test pads so a larger iron tip might work, but you'll have a much easier time with a good quality soldering station and fine tip.
- Use thin solder like 0.02" or smaller diameter.
- Tin each test pad and wire separately by heating them with the iron and melting a bit of solder on them. Then hold the tinned wire to the tinned pad and touch them with the iron to flow the solder and make a connection.
- Use light and magnification to inspect each joint to make sure there aren't shorts with other solder pads or parts.
- After a joint cools gently tug on the connection to ensure the joint holds firm and doesn't break. If a connection isn't solid touch it with the iron to flow the solder again, or even remove the wire completely and start over.
- Before you solder anything think through where each wire will go and how you'll hold the iron. You don't want to solder a wire in the way of where you need to move the iron to reach another test pad.
- If you make mistakes don't worry, just flow the solder with the iron and pull off the wire. Use a solder wick (http://adafru.it/149) or sucker (http://adafru.it/148) to pull off excess solder and start over. Just be careful not to use too hot a temperature or hold the iron too long on the board to prevent lifting the pad or burning the board.

In addition to the required test pads above for programming you might also solder wires to these pads:

- PC6 USART6 TX in the upper left. You can use this pin as a serial output to print debug messages from the CPU, or just as a GPIO pin.
- PC7 USART6 RX immediately below USART6 TX. This can be used as a serial input (not currently used in the example code), or as a GPIO pin.

Below you'll see a picture of a board with wires soldered to all the programming test pads, and the PC6 & PC7 connections:




After you've followed this guide and have written code to the Dash you might consider putting it back in its case to keep it protected.  If you drill a small hole in the cover of the Dash on the opposite end of the button you can pull the wires through like below:

Unfortunately the case itself needs to be glued or taped shut since it's not designed to be securely closed again by itself.

# Programmer Connections

To connect the Dash to the STLink V2 programmer make the following connections between the wires and programmer:

- Dash PA14 SWCLK to STLink V2 SWCLK.

- Dash PA13 SWDIO to STLink V2 SWDIO.
- Dash RESET to STLink V2 RST.
- Dash GROUND to STLink V2 GND.
- Dash 3.3V power to STLink V2 3.3V power.  Don't try to send 3.3V power into the positive battery terminal!  The Dash uses a small boost converter to take the ~1.7V battery voltage up to 3.3V and will be damaged if you send in a higher voltage.  Instead send 3.3V power into the test pad noted in this guide.  If you can't use the test pad you can instead power the Dash with a single AA or AAA battery (i.e. a ~1.5-1.7V source) connected to the positive and negative battery connections.

Once you've connected the Dash to the programmer continue on to learn how to setup a toolchain that can compile and upload code to the board.

---

# Toolchain Setup

To program the CPU on the Dash you'll need to setup a toolchain that can compile code for the Dash's ARM Cortex M3 processor.  The GNU compiler collection (GCC) () i s a great opensource toolchain with excellent ARM CPU support.  This page will walk through setting up a Linux-based virtual machine (VM) that uses GCC to compile code for the Dash.

Why use a VM for the toolchain?  The reason is that setting up a compiler for a different CPU architecture, AKA a cross compiler, can be quite challenging.  Each platform like Windows, OSX, etc. has a different process for installing and setting up the software so there's no single set of instructions that will work for everyone.  Using a VM for the toolchain makes it easy to follow one set of steps regardless of your operating system, and isolates the toolchain from conflicting with any other tools on your system.

This guide uses Vagrant () and VirtualBox () to manage and run the VM.  Both are excellent pieces of open source software that you can use for free to easily create VMs from the command line.  With just a few commands you'll have an entire toolchain provisioned and setup.

# Dependencies

Before you get started you'll want to make sure you have the following software installed:

- VirtualBox () - This guide was written using the 5.0 release.
- Vagrant () - This guide was written using the 1.7.4 release.
- VirtualBox Extension Pack () - Download and install the version that matches your version of VirtualBox.  To install on Windows just double click the downloaded .vbox-extpack file, or on Linux & Mac OSX run in a command terminal: VBoxManage extpack install /path/to/downloaded/vbox-extpack
- Git source control system () - On Windows you should install the command line version of Git as you can use its shell to access the VM without having to install and use a separate SSH client.  On Linux or Mac OSX install git using your appropriate package manager, Homebrew (), etc.

In addition you'll need around 5-10 gigabytes of free space for the VM's virtual hard disk.

On Windows ensure you have the STLink V2 USB driver () installed.

On Linux you need to add your user to the vboxusers group so that VirtualBox can access USB devices on your computer.  In a terminal run the following command:

```
sudo usermod -a -G vboxusers $USER
```

# Toolchain Setup

Once you have the software installed open a command line terminal (in Windows open a 'Git Bash' terminal), navigate to where you'd like the toolchain to be created, and run the following command to clone its repository:

```
git clone https://github.com/adafruit/ARM-toolchain-vagrant.git
```

After a few moments the repository will be cloned, then run the following command to navigate inside the cloned directory and start up the VM:

```
cd ARM-toolchain-vagrant
vagrant up
```

Note that the very first time you start the VM it will take around 10-30 minutes to download an OS image and provision itself.  Future startups will just take a few seconds.

Whenever you want to control the VM, like starting it with the up command, you'll need to make sure you're inside the ARM-toolchain-vagrant directory that contains the Vagrantfile VM configuration.

After the VM starts run the following command to connect to it with SSH:

```
vagrant ssh
```

You should see welcome text and your terminal change to indicate you're logged in with user vagrant on the vagrant-ubuntu-trusty-64 VM:

```
Welcome to Ubuntu 14.04.2 LTS (GNU/Linux 3.13.0-55-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

  System information as of Wed Aug  5 21:15:55 UTC 2015

  System load:  0.0               Processes:           77
  Usage of /:   5.0% of 39.34GB   Users logged in:     1
  Memory usage: 30%               IP address for eth0: 10.0.2.15
  Swap usage:   0%

  Graph this data and manage this system at:
    https://landscape.canonical.com/

  Get cloud support with Ubuntu Advantage Cloud Guest:
    http://www.ubuntu.com/business/services/cloud

65 packages can be updated.
32 updates are security updates.


Last login: Wed Aug  5 21:15:55 2015 from 10.0.2.2
vagrant@vagrant-ubuntu-trusty-64:~$
```

Congrats, you're connected to the VM and ready to use the toolchain!

However if you see an error or Vagrant fails to start the VM, make sure you have all the required software installed.  In particular Vagrant is very sensitive about the version of VirtualBox that's installed so you might need to update both VirtualBox and Vagrant to ensure they're at the latest versions and then try again.  Don't forget you need the VirtualBox Extension Pack installed too.

For reference, to exit the VM and turn it off run the following command to leave the SSH session:

```
exit
```

Then run the following command to shut off the VM:

```
vagrant halt
```

Remember the virtual machine will always be running even after you log out of its SSH session.  You need to run the halt command to explicitly stop the VM.

Finally if you ever wish to completely remove the VM or even start back from scratch provisioning again, run the following command to destroy the VM image:

```
vagrant destroy
```

# Installed Tools

With the VM running and a connection to it open with SSH you can examine the tools that are available.

The GCC ARM compiler toolchain is installed and available in the system path under the arm-none-eabi-* names.  For example to see the version of the C compiler you can run:

```
arm-none-eabi-gcc --version
```

Which will show output similar to:

```
arm-none-eabi-gcc (GNU Tools for ARM Embedded Processors) 4.8.3 20140228 (release)
[ARM/embedded-4_8-branch revision 208322]
Copyright (C) 2013 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

In addition to GCC a few tools are installed for interacting with a STM32 processor using the STLink V2 programmer.  The first tool is OpenOCD () and it's a very flexible debugger with support for the STLink and other programmers.  You can see its version by running:

```
openocd --version
```

Which will respond with something like:

```
Open On-Chip Debugger 0.9.0 (2015-08-04-23:24)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.org/doc/doxygen/bugs.html
```

The second tool is the Linux stlink command line tool () which is a little simpler than OpenOCD but only works with the STLink programmer. This tool is available in the system path under the st-util and st-flash programs.

Finally the VM is automatically configured to pass through the STLink V2 programmer from your host computer to the VM. This means you can program and flash chips directly from the VM instead of having to copy out firmware files and flash from your main operating system. To check that the VM can see your STLink V2 programmer connect it to the computer hosting the VM and then after a few seconds run the following command in the VM:

```
lsusb
```

You should see a ST-LINK/V2 device listed like below:

```
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 017: ID 0483:3748 STMicroelectronics ST-LINK/V2
Bus 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
```

# Syncing Files

The general workflow is to use the VM to compile and flash code to a chip and your host operating system to edit the code with your preferred code editor. To facilitate this workflow you can use Vagrant's synced folders () to automatically transfer files between the VM and your operating system.

By default the VM is setup to sync anything in its /vagrant folder with the directory where the VM was created on the host computer (the ARM-toolchain-vagrant folder). Try creating a text file in the ARM-toolchain-vagrant folder, then run the following commands in the VM to move to the synced folder and list the files:

```
cd /vagrant
ls
```

You should see listed all the files that are in the ARM-toolchain-vagrant folder on your host computer, including the file you created. Any files creat or modify here will be kept in sync between the VM and host computer.

Once you've setup the toolchain VM continue on to learn how to program the Dash CPU with example code.

---

# Programming

Once a toolchain is setup to compile code for ARM processors you're ready to program the Dash's CPU with new code. In this section I'll describe how to download, compile, and flash example code to control some functions of the Dash like its LED and a serial UART.

If your hardware experience is mostly with platforms like Arduino this is a great opportunity to learn how those platforms work at a lower level. With Arduino you don't need to worry about setting up a toolchain, writing code to directly access the CPU, etc. as that's all taken care of for you by Arduino's libraries and IDE. However the trade-off is that you can only write code that works on chips that Arduino supports, and you can only use features that Arduino has exposed for you. By writing your own code to control a chip 'bare-metal' you have much more flexibility, but you have to do more work yourself.

For the Dash's STM32F205RG6 CPU the definitive source for how everything on the chip works is the STM32F205xx reference manual (). This document describes all of the functions of the CPU, but be warned it's almost 1500 pages long! Instead of trying to read the entire reference manual start by skimming the STM32F205xx datasheet () which is a high-level summary of the chip's features. Then lookup specific parts of the chip in the reference manual when you want to understand how they work.

Although the reference manual is all you technically need to start programming the CPU, you'll be much more productive using a library that implements some of the tedious and boilerplate code to access the hardware. One option is STMicro's STM32 CubeF2 HAL () (hardware abstraction layer), and another is the libopencm3 library (). For these examples I've chosen to use the libopencm3 library because it's mature, easy to setup, and open source. There's no right or wrong option when choosing a library--experiment with using different hardware libraries to find one that supports the features you need and makes you the most productive.

WARNING: Once you reprogram the Dash with your own code you will no longer be able to use it to order products from Amazon! Your firmware will completely overwrite Amazon's firmware and replace its functionality. Don't reprogram a Dash you aren't willing to lose completely.

# Examples

To download the example code () first make sure you are running the toolchain VM and have connected to it with SSH.  It will be easiest to download the code to the /vagrant folder in the VM so that it can be edited from your host operating system with any text editor (I recommend Atom () as a good cross-platform programmer's text editor).

Run the following commands to clone the example code to the /vagrant folder:

```
cd /vagrant
git clone --recursive https://github.com/adafruit/dash-examples
cd dash-examples
```

Make sure to specify the --recursive option so that the libopencm3 library (which is a git submodule) is downloaded too.

After the code is downloaded you'll first need to build the libopencm3 library which the examples use.  Run the following command from inside the dash-examples root folder:

```
make
```

You should see text pass by as libopencm3 and the examples are compiled.  Once the compilation finishes you can navigate to each example's subdirectory and run its Makefile to build and program the example code.

Make sure to compile libopencm3 as described above before running and compiling the examples!  If you don't compile libopencm3 then the examples will fail to compile with an error like 'no target for xxxx.elf'!

The examples are available inside the examples subdirectory of the root and include:

- examples/blink - A basic example to rotate through blinking the red, green, and blue LED for a second each.
- examples/pwm - A more advanced example that uses timers on the CPU to generate a PWM signal that can light the LED to any RGB color.  The example will cycle through all possible color hues.
- examples/uart - Example of sending data out USART6 TX (pin PC7) using printf calls.  This is useful for printing debug messages and other data from the Dash.  Note that receiving data on the RX pin isn't yet in the example.

To run an example first make sure the Dash and STLink V2 programmer are connected to the computer running the VM.  Then navigate inside the example's folder and run the following command (to run the blink example):

```
cd examples/blink
make stlink-flash
```

The stlink-flash target of the makefile will use the st-flash utility to program the Dash.  You should see an output that looks something like:

```
FLASH  blink.bin
2015-08-09T03:42:38 INFO src/stlink-common.c: Loading device parameters....
2015-08-09T03:42:38 INFO src/stlink-common.c: Device connected is: F2 device, id
0x20036411
2015-08-09T03:42:38 INFO src/stlink-common.c: SRAM size: 0x20000 bytes (128 KiB),
Flash: 0x100000 bytes (1024 KiB) in pages of 131072 bytes
2015-08-09T03:42:38 INFO src/stlink-common.c: Attempting to write 916 (0x394) bytes
to stm32 address: 134217728 (0x8000000)
EraseFlash - Sector:0x0 Size:0x4000
Flash page at addr: 0x08000000 erased
2015-08-09T03:42:39 INFO src/stlink-common.c: Finished erasing 1 pages of 16384
(0x4000) bytes
2015-08-09T03:42:39 INFO src/stlink-common.c: Starting Flash write for F2/F4
2015-08-09T03:42:39 INFO src/stlink-common.c: Successfully loaded flash loader in
sram
size: 916
2015-08-09T03:42:39 INFO src/stlink-common.c: Starting verification of write
complete
2015-08-09T03:42:39 INFO src/stlink-common.c: Flash written and verified! jolly
good!
```

Immediately after finishing you should see the Dash's LED start blinking through red, green, blue colors.  Woo hoo, you've reprogrammed the Dash!

If you receive an error make sure you've compiled libopencm3 in the example root directory, and that the STLink V2 is plugged in to the computer.  Also check the STLink V2 is visible to the VM using the lsusb command.  If the STLink V2 isn't visible to the VM make sure VirtualBox's extension pack is installed and try running VirtualBox's GUI program to view the ARM-toolchain-vagrant VM's settings (look in the USB settings to see if the STLink V2 can manually be added to the VM).

For the uart example you'll want to connect the Dash to a serial to USB cable so you can see the UART output on your computer.  You'll need a wire soldered to the Dash's PC6 USART6 TX pad and then connected to the serial to USB cable's RX pin.  In addition connect the serial to USB cable's ground pin to the Dash's ground.  Finally open the serial port with 115200 baud, 8 data bits, no parity, and 1 stop bit in a tool like PuTTY () (Windows) or screen () (Linux & Mac OSX).  Every second you should see a count printed to the serial port.

If you connect the Dash PC7 USART6 RX pad to the cable's TX pin be sure the cable is a 3.3 volt cable and NOT a 5 volt cable! The Dash CPU's GPIO pins are NOT 5 volt safe and will be damaged.

# Blink Example

To help understand how the Dash CPU is programmed I'll walk through the blink example in a little more detail.

First to understand how the project is built look at the Makefile () in the root of the examples/blink folder:

```
# Dash Blink Example Makefile
# Copyright (c) 2015 Tony DiCola
# Released under a MIT license: http://opensource.org/licenses/MIT

BINARY = blink

# Note if you have multiple source files, list them all in an OBJS variable.
# The Makefile rules will pick them up and compile their source appropriately.
# For example if you have a foo.c and bar.c to include set the OBJS variable:
# OBJS = foo.o bar.o

include ../Makefile.include
```

This Makefile is trivially simple because all of the rules and logic are in a separate Makefile.include () and Makefile.rules () file. Those files are based on the Makefiles from the libopencm3-examples project () and provide all the typical rules for making and programming an ARM project, like compiling code to a .bin or .hex file and uploading with a programmer.

For an example project it only needs to specify the name of the output binary with the BINARY = name value. The Makefile will then try to build a binary.o file from a binary.c or binary.cpp file in the directory.

Notice the commented section of the Makefile which tells how to reference other sourcefiles too. Just set the OBJS variable to a list of object files that need to be built and the Makefile will take care of the rest.

You can set other variables in the Makefile to customize how an example is built too. I recommend skimming the Makefile.include and Makefile.rules file and examining other examples from the libopencm3-examples () project to learn more.

Next open the blink.c file () to see the code that powers the blink example.  After including a few libopencm3 headers the code defines a few functions that deal with the system timer, or systick:

```
// Global state:
volatile uint32_t systick_millis = 0;  // Millisecond counter.


// Delay for the specified number of milliseconds.
// This is implemented by configuring the systick timer to increment a count
// every millisecond and then busy waiting in a loop.
static void delay(uint32_t milliseconds) {
  uint32_t target = systick_millis + milliseconds;
  while (target &gt; systick_millis);
}

// Setup the systick timer to increment a count every millisecond.  This is
// useful for implementing a delay function based on wall clock time.
static void systick_setup(void) {
  // By default the Dash CPU will use an internal 16mhz oscillator for the CPU
  // clock speed.  To make the systick timer reset every millisecond (or 1000
  // times a second) set its reload value to:
  //   CPU_CLOCK_HZ / 1000
  systick_set_reload(16000);
  // Set the systick clock source to the main CPU clock and enable it and its
  // reload interrupt.
  systick_set_clocksource(STK_CSR_CLKSOURCE_AHB);
  systick_counter_enable();
  systick_interrupt_enable();
}

// Systick timer reload interrupt handler.  Called every time the systick timer
// reaches its reload value.
void sys_tick_handler(void) {
  // Increment the global millisecond count.
  systick_millis++;
}
```

The system timer () is a common feature of ARM processors and provides a timer to keep track of program execution time.  This is very useful to understand how long something takes to execute, and in this example the systick timer is used to implement a millisecond delay function.

The millisecond delay works by configuring the systick timer to increment a count every millisecond and waiting for that count to reach a specific value.  To configure systick to update each millisecond its reload value is set to 16,000.  This value was computed based on the fact that the main CPU clock & systick timer are running at 16mhz (since no external clock source was configured the CPU defaults to its internal oscillator) and solving:

16,000,000 mhz / 1,000 time a second target = 16,000 systick reload value

The systick timer will count from 0 to 16,000 in one millisecond of wall-clock time, fire an interrupt which increases the millisecond count, and wrap back to zero to start the process again.

The next part of the code deals with how to configure the GPIO outputs that drive the Dash's LEDs:

```
// Setup and configure the GPIOs to control the LEDs on the Dash.
static void gpio_setup(void) {
  // Enable the GPIO clocks for the two GPIO ports that will be used (A &amp;amp; B).
  rcc_periph_clock_enable(RCC_GPIOA);
  rcc_periph_clock_enable(RCC_GPIOB);

  // Set each LED GPIO as an output.
  gpio_mode_setup(GPIOA, GPIO_MODE_OUTPUT, GPIO_PUPD_NONE, GPIO8); // PA8, blue LED
  gpio_mode_setup(GPIOB, GPIO_MODE_OUTPUT, GPIO_PUPD_NONE, GPIO6); // PB6, red LED
  gpio_mode_setup(GPIOB, GPIO_MODE_OUTPUT, GPIO_PUPD_NONE, GPIO7); // PB7, green LED
}
```

The gpio_setup function does a couple important thing:

- The GPIO clock is enabled for both GPIO ports that will be used (A and B). In general most peripherals on the CPU like GPIOs, UARTs, timers, etc. need to have their clock enabled before they can be used.
- The GPIOA and GPIOB ports are configured to drive PA8, PB6, and PB7 as outputs. This is similar to the pinMode function you might use in an Arduino sketch to setup a pin as an input or output.

Finally the main entry point and loop of the program looks like:

```
int main(void) {
  // Setup systick timer and GPIOs.
  systick_setup();
  gpio_setup();

  // Main loop.
  while (true) {
    // Light the red LED.  Note that LEDs light up when the GPIO is pulled low
    // with the gpio_clear function, and turn off when pulled high with the
    // gpio_set function.
    gpio_clear(GPIOB, GPIO6);  // Red LED on
    gpio_set(GPIOB, GPIO7);    // Green LED off
    gpio_set(GPIOA, GPIO8);    // Blue LED off
    delay(1000);  // Wait 1 second (1000 milliseconds).
    // Now light just the green LED.
    gpio_set(GPIOB, GPIO6);    // Red LED off
    gpio_clear(GPIOB, GPIO7);  // Green LED on
    gpio_set(GPIOA, GPIO8);    // Blue LED off
    delay(1000);
    // Finally light just the blue LED.
    gpio_set(GPIOB, GPIO6);    // Red LED off
    gpio_set(GPIOB, GPIO7);    // Green LED off
    gpio_clear(GPIOA, GPIO8);  // Blue LED on
    delay(1000);
  }

  return 0;
}
```

The main function is where execution of the example starts.  Unlike an Arduino sketch there is no explicit setup or loop function, instead the main function performs both of those tasks itself.  At the start of the main function it calls the previous systick and GPIO setup functions to initialize those parts of the example.  Then the main function jumps into an infinite loop that will blink the LEDs forever.

Inside the main loop each LED is enabled and disabled using the gpio_clear and gpio_set functions to drive the GPIO pins low and high respectively.  You can learn more about the GPIO functions from the libopencm3 GPIO function reference ().

Between lighting each LED the delay function built from the systick timer is used to pause for a second, and the whole process repeats forever.  That's all there is to the blink example code!

# Reviving A Bricked Dash

As you explore the Dash and reprogram it to run your own code you might find yourself accidentally 'bricking' the device and being unable to reprogram it.  This can happen if you accidentally disable the single-wire debug test headers, misconfigure the clock or other peripherals, etc.  Luckily there's a 'safe mode' you can put the chip in to help get it into a good state to reprogram again.

If you try to program the Dash's CPU and receive errors that the STM32 processor cannot be detected (and you know for sure it's not a physical connection with the programmer, or software issue passing the programmer through to the VM) you can follow the steps mentioned in the workaround section of this page () to get the Dash back into a good state.

First connect the RESET pin of the Dash to ground. This will force the chip to stop operating any faulty program and reset itself to receive new instructions.  Make sure to keep the RESET pin connected to ground as you run the next steps. Then inside the VM run OpenOCD's telnet server (which is already installed) with the following command:

```
openocd -f interface/stlink-v2.cfg -f target/stm32f2x.cfg
```

This command will not return, instead OpenOCD will sit waiting for a connection.

Now open a new terminal and connect to the VM again (using the vagrant ssh command) and run the following to connect to OpenOCD's telnet server:

```
telnet localhost 4444
```

Once connected to OpenOCD's telnet server run the following command:

```
reset halt
```

This command should fail with a timed out error, like:

```
timed out while waiting for target halted
TARGET: stm32f2x.cpu - Not halted
in procedure 'reset'
in procedure 'ocd_bouncer'
```

Now remove the connection between RESET and ground. You should see OpenOCD report that the CPU is halted:

```
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x080002cc msp: 0x20020000
```

Now run the following command to erase the flash memory and any faulty program:

```
stm32f2x mass_erase 0
```

After a few seconds the erase should complete:

```
device id = 0x20036411
flash size = 1024kbytes
stm32x mass erase complete
```

Finally stop OpenOCD and close the telnet session by running the shutdown command:

```
shutdown
```

You should now be able to flash an example program back to the chip (like the blink example).

Be careful not to flash the same program that caused the problem or else you might need to un-brick again!

# Going Further

Now that you know how to get code running on the Dash's CPU there's almost no limit to what you can do.  Here are some good resources to help go further:

- libopencm3 Library () - Check out the documentation () for the libopencm3 library, and in particular look at the examples () it has for the STM32 F2 and other chips. You don't have to limit yourself to just using libopencm3 too, be sure to check out STMicro's STM32F2xx HAL () or other similar hardware libraries.
- STM32F2xx datasheet () and technical reference manual () - These are great to skim and reference to understand the features of the Dash's STM32F205RG6 CPU.
- ARM Cortex-M Series Documentation () - The Dash's CPU is an ARM Cortex-M3 processor and this reference has all the details on ARM's instruction set, features, etc.  Like the CPU's technical reference manual this is a very large document that's good to skim and reference later when needed.  An easier to digest introduction to ARM Cortex-M3 processors is The Definitive Guide to the ARM Cortex-M3 and M4 Processors () by Joseph Yiu.
- Broadcom WICED SDK () - Although this guide didn't touch on the Dash's WiFi module yet, it's good to know the WICED SDK from Broadcom is what the Dash was designed to use for control of the CPU and WiFi module.  The WICED platform provides a basic hardware abstraction layer (like libopencm3 provided in this guide) and WiFi, TCP/IP protocol, SSL/encryption support, and much more.
- Exploring Amazon Dash Button () and Amazon Dash Button Teardown () - These are a couple great resources with more information about the Dash hardware.  For example the Dash has a ADMP441 microphone () connected to the CPU which might allow for recording sound in your own code.
- Migrating Away from the Arduino IDE () at Contextual Electronics - This is a new series from Contextual Electronics that will be exploring in more detail bare-metal CPU programming, similar to what was shown in this guide with the Dash.

Good luck hacking the Dash and be sure to share interesting things you create or discover with the hardware!