Cross-Platform Cortex-M Development

```
{
    “Host Program”:”Visual Studio Code”,
    “Targets”:”Arm Cortex-M”,
    “Language”:”C”,
    “Platforms”: [
        “Mac”,
        “Windows”,
        “Linux”
     ],
     “debug.extension”:”Cortex-Debug”,
     “debug.debugger”:”GDB”,
     “debug.protocols“: [
            “JTAG”,
            “SWD”
      ],

     “debug.gdb_servers”:[
        “J-Link”,
        “OpenOCD”,
        “ST-Util”,
        “pyOCD”,
        “bmp”
    ],
    “Cost”:”little to nothing”
}
```

With Visual Studio Code

— by David Clifton

Cross-Platform Cortex-M Development

With Visual Studio Code

David Clifton

This book is for sale at http://leanpub.com/cross-platform-cortex-m-development

This version was published on 2019-05-14

Contents

Introduction

The majority of Arm Cortex development is done with integrated development environments (IDEs). IDEs can be obtained from chip vendors, such as TI or ST, compiler makers like IAR or Keil, debugger probe makers such as SEGGER, or independent providers like EmBitz, CodeBlocks, Eclipse, or CodeLite. What you get with your IDE is a text editor, one or more compiler suites, library code, project manager, build tool, terminal, debugger-GUI and debugger. IDE's cost anywhere from zero to thousands of dollars.

Once you have an IDE, you have to familiarize yourself with the use of its perspectives, windows, menus, and dialogs; and configure it for your compiler, debug probe, circuit board, processor, RTOS if any, and third party code libraries.

Some IDEs, such as those by IAR and Greenhills, require less configuration than others; but they use proprietary compilers, startup code and libraries. That can lock you into their product, and require payment of yearly maintenance fees. And it still does not exempt you from learning everything about the details of their tool.

An alternative to obtaining and learning an IDE is to select an editor, build tool, compiler suite, debugger, debugger GUI, and terminal. Install them on your development computer, and setup environment variables and scripts that glue the parts together.

No matter which path you choose, there is a lot to learn before you can implement that great design you created.

Is there an optimal path through this development tool jungle?

The answer is a hopeful maybe. One solution candidate comes from an unlikely source: Microsoft. It supplies a free, relatively easy way to build your own Arm Cortex development environment. Nobody knows how long Visual Studio Code will be free, or whether it will be feature-enriched beyond usability. At the moment, it is a good framework upon which to hang the individual components of your Arm Cortex development system.

About Visual Studio Code

Visual Studio Code, a.k.a. VSCode, grew out of the need for a flexible tool for website production. It includes a variety of build tools: a good code editor, multiple script and compiled languages, and flexible, JSON-based configuration files. A Task configuration file permits ad-hoc creation of automated build tasks, and it has a Debug GUI which can be integrated with most of the debuggers you might need. TypeScript or JavaScript extensions can be coded to take care of complex situations.

VSCode is supported on macOS, Windows, and Linux.

Instead of a galaxy of menus and dialogs, which must be mastered before development can begin, VSCode requires just a few JSON configuration files to be defined. Those files often have helpful defaults.

This apparent simplicity can be deceptive. Many of the VSCode features are supplied by TypeScript programs, which are updated frequently by the VSCode developer team. Features are many, varied, and subject to change. A best practice with VSCode is to use only those features and extensions that are absolutely necessary for your project.

Sparse, simple Arm Cortex-M projects can be cobbled together pretty quickly with the VSCode framework.

More complex projects such as Nordic bluetooth programs can also be handled simply within the framework.

Embedded Project Example

This eBook illustrates the use of VSCode with a non-trivial, Arm Cortex example project.

The example project, a digital Theremin, was written for an STM32F7 Discovery Board. That circuit board is widely available from electronics distributors.

Complete digital source code and VSCode example-project build directories for macOS, Linux, and Windows are supplied as extras with the purchase of this eBook.

The requirements, architecture, design, and tests for this project, are available in another eBook by the same author at leanpub.com.

The example project uses the embedded version of the Gnu Arm compiler tools, the OpenOCD debug server, some library modules supplied by the chip/board supplier, the application code, and a simple make file.

The macOS and Linux projects use GNU make. The Windows project uses Cygwin make, the same makefile, and variations of the unix-ish json scripts.

OpenOCD is the debug probe interface used on all three platform examples.

JLink can also be used as the debug probe.

The VSCode Debugger GUI uses an extension called Cortex-Debug, which provides a JSON-configurable interface to GDB, and helps to configure OpenOCD or JLink.

The next sections describe, in some detail, how to prepare Mac, Windows, and Linux computers to use VSCode to build, load, and debug the example application.

Caution

The OSX build works just fine on the Yosemite OS, but not on the Sierra OS. Apple is apparantly on a quest to keep third party software and technicians from accessing the platform. This has led them

to quarantine third party software downloaded from the internet.

As often as users find work arounds to undo the effects of the quarantine, Apple finds new ways to discourage third parties. Unless you want to engage in a running battle with Apple to keep your software working, you would be better off to develop on older versions of OSX, or just switch to Linux.

Platform Configuration

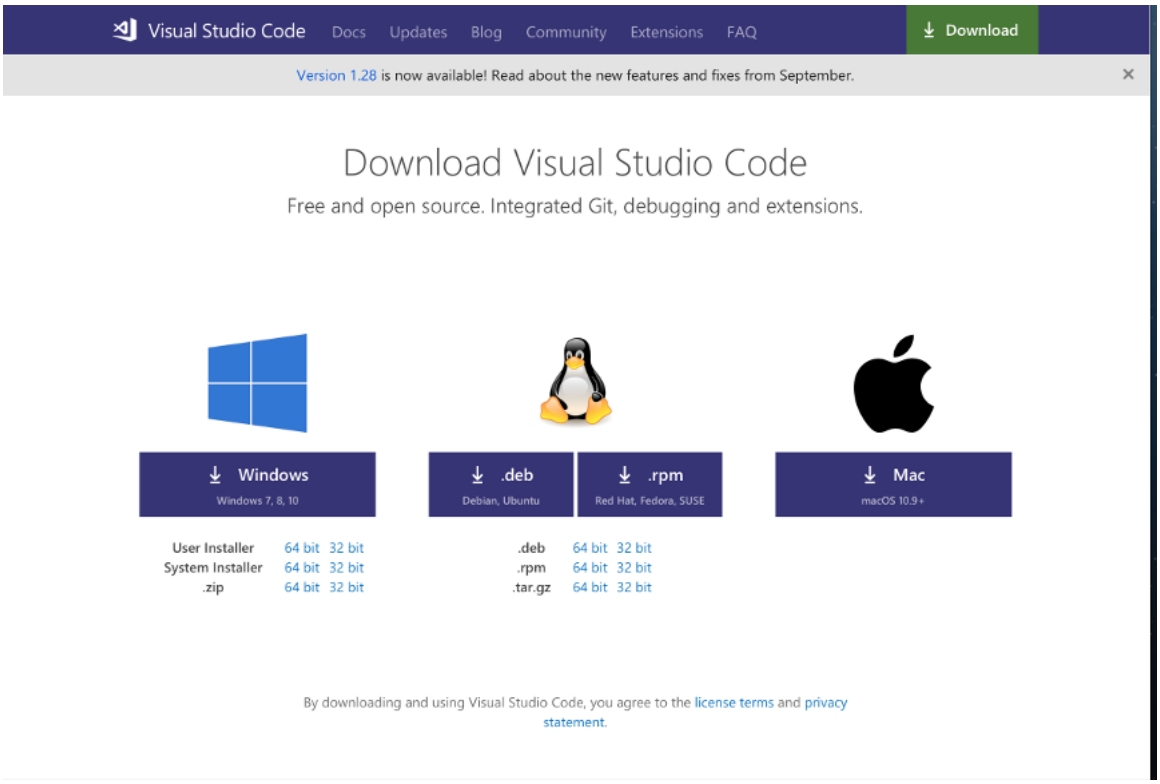
It is possible, and often desirable, to maintain cross-platform compatibility with the build and debug tools for a project. This is possible with VSCode, provided the gnu compiler toolchain is used, and a unix build environment is incorporated in the Windows computer.

This section tells how to obtain and install VSCode, gnu embedded arm toolchain, the Cortex-Debug extension for VSCode, and the JLink or OpenOCD debug server on Linux, macOS, and Windows, as well as unix build tools for Windows.

It often happens that a particular version of a needed program for Mac or Linux, and a different version must be found and used. This is typical of the current state of vendor compatibility.

Download and Install Visual Studio Code

VSCode can be downloaded from https://code.visualstudio.com/Download.



Linux Installation

Choose a download that suits your computer and kind of Linux.

With Ubuntu, a dialog appears that asks what to do with the downloaded file. It is easy to open it with Ubuntu Software Center. After a while another screen appears with a progress bar, and a reference to https://code.visualstudio.com/docs/setup/linux for installation instructions and FAQ. After that the Software center declares the installation complete. A link to the Visual Studio Code executable, called "code" has been installed in **/usr/bin**.

Debian Mate allows you to install the download any way you like, but suggests the use of a cross-Linux installation tool called snap. If you use that, it does everything, and you don't have much to do but watch. It also sets up the mechanism for periodic update of VSCode.

To run VSCode with ubuntu, open a terminal and execute:

/usr/bin/code

You may wish to pin the launcher icon to the launch bar at the left of the Ubuntu screen.

If you installed into Debian mate with snap, a Visual Studio Code entry appears in the Applications menu under Programming.

Other versions of linux may offer other ways to install vscode, but you can always download a tar and extract the program wherever you want it, and execute it with:

/path to code/code

macOS Installation

On the Mac, the download button copies Visual Studio Code.app into the downloads directory. Simply move it to the Applications directory, run it, and pin it's launcher icon to the launch bar if you wish.

Windows Installation

On Windows, it is easiest to choose the 64 or 32 bit user installer. Pressing the download button starts the installer download. Find the installer, whose name begins with "VSCodeUserSetup" which appears in your Downloads folder. Click the installer to set up VSCode and launch it for the first time. Pin the launcher icon to the launch bar at the bottom of the screen if you wish.

The Cygwin website www.cygwin.com allows you to download and use a subsystem of unix commands on your windows computer. The Cygwin website assumes you know what unix commands you need, when it presents an elaborate menu that allows you to select the unix

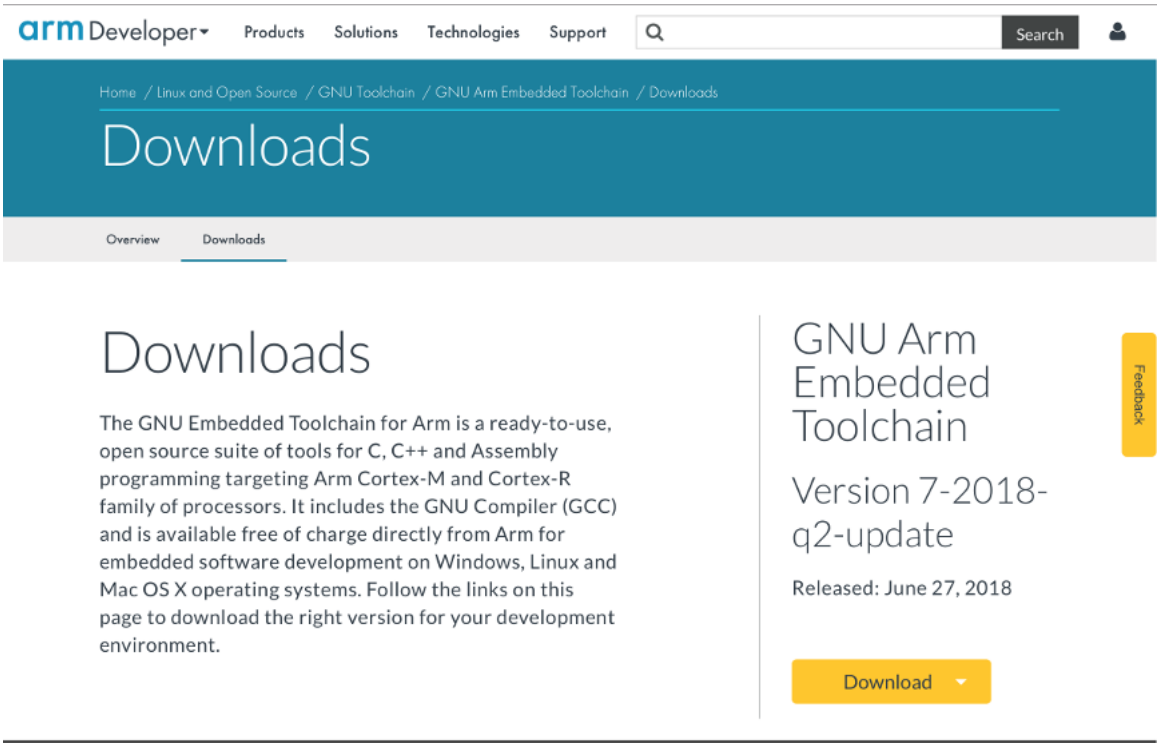
subsystems and commands you want. If you make the wrong choices, you can go back and change them as needed by rerunning the Cygwin installation program and making changes to your choices.

It takes a while to get the hang of the Cygwin installer menu system, but after a while you should be able to download the base Cygwin system, the make tools, and any other convenient unix commands you might want.

I tried at first to substitute the bash shell for the windows terminal in vscode, but it led to multiplying complications. So I ended up with the base system, a few unix commands, and the unix make and make depend programs.

Download and Install Embedded Arm Toolchain

Binaries of The Embedded Arm Toolchain for all three platforms are available at http://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads



The page contains the latest 32 and 64 bit mac, windows, and linux downloads for the embedded arm toolchain.

Downloading macOS Tools

The latest archive of the macOS 64 bit version of the toolchain will have a name similar to gcc-arm-none-eabi-7-2018-q2-update-mac.tar.bz2. Download it to the Mac. Create a tool directory in your home directory, If the Mac bunzipped the archive, move the .tar file there and extract it with a commaned like:

tar -xvf gcc-arm-none-eabi-7-2018-q2-update-mac.tar

If the archive was not bunzipped, move the tar.bz2 archive to the tool directory, and extract it with a command like:

tar -xvjf gcc-arm-none-eabi-7-2018-q2-update-mac.tar.bz2

These download and extract suggestions worked with OSX Sierra. Some Mac versions do the extraction automatically when you click on the downloaded file.

Download Linux Tools

The latest archive of the Linux 64 bit version will have a name similar to gcc-arm-none-eabi-7-2018-q2-update-linux.tar.bz. Download it to your Linux computer. Create a tool directory in your home directory. Move the download to your tool directory. If Linux did not bunzip the file, extract it with the command:

tar -xvjf gcc-arm-none-eabi-7-2018-q2-update-linux.tar.bz2

otherwise, use the command:

tar -xvf gcc-arm-none-eabi-7-2018-q2-update-linux.tar

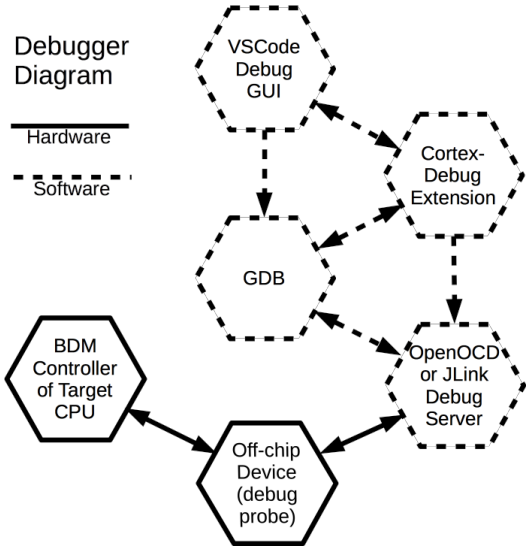
> The toolset named above failed to correctly compile one module in my build, so I deleted that version, and found an older 64 bit linux download on the same webpage: gcc-arm-none-eabi-7-2017-q4-major.tar.bz2. That version was downloaded and extracted in the same way to the tool directory. It worked fine and so it is still there.

Download Windows Tools

Create a tools directory. Download a Win32 tools installer suitable for your computer. The installer name will be something like gcc-arm-none-eabi-7-2018-q2-update-win32.exe. Move it to your tools directory, and execute it there.

> Similar to what happened with the Linux distribution, the latest Win32 distribution contained a defective objcopy program, It was necessary to fall back to an earlier distribution.

Debugger Installation



How the debugger works

Some of this information comes from: https://code.visualstudio.com/docs/extensionAPI/api-debugging

Most microprocessors, including the Arm Cortex-M, have a builtin controller called the Background Debug Mode controller.

Several microprocessor pins are allocated to communication between the BDM controller and an off-chip device, sometimes on the same circuit board as the processor, and sometimes in a separate debug probe. The off-chip device can configure the dedicated microprocessor pins to enter a debug mode, during which time the off-chip device can read and write RAM and FLASH, run or stop the microprocessor, or just execute the next instruction.

Two protocols are in common use for communication between the target processor and the off-chip debug device. They are called JTAG and SWD.

The off-chip debug device communicates, usually over USB or TCP/IP, with a Debug Server running on the programmer's computer. The debug server accepts commands from the GDB debug program, translates them to JTAG or SWD requests, and replies with data received from the off-chip device.

The VSCode Debugger GUI communicates with either the debug server, or directly with the debugger, GDB, through a debugger extension.

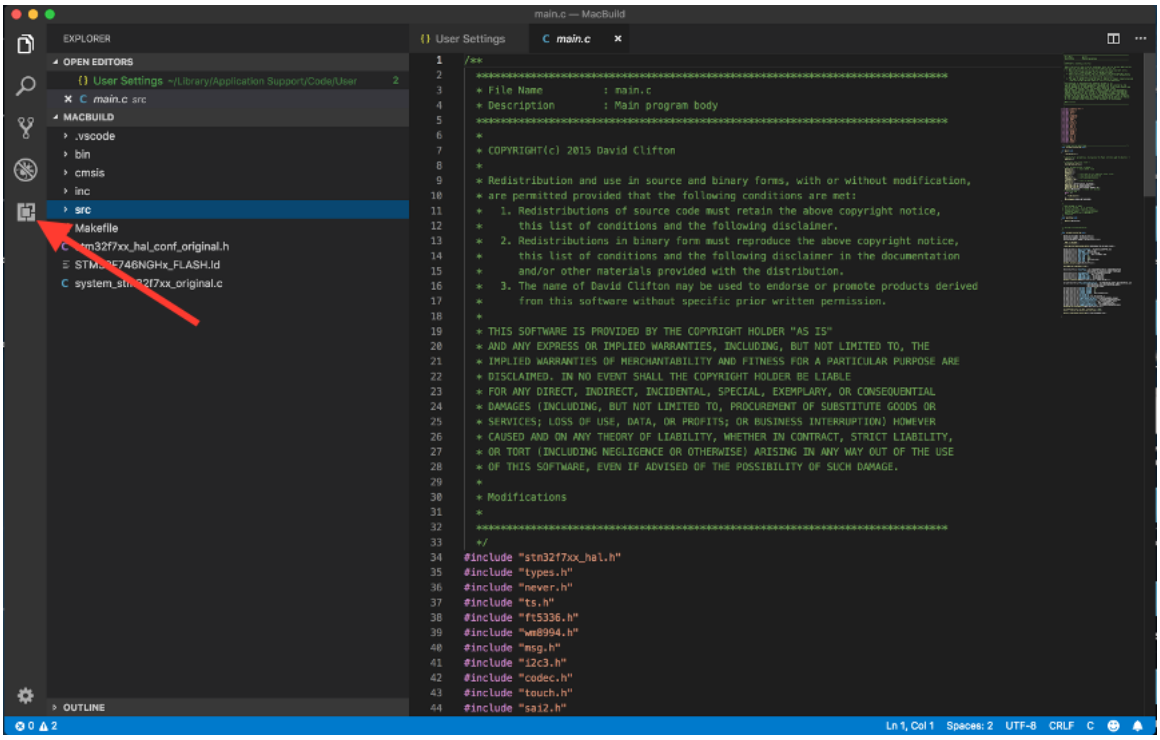
The user can also communicate directly with GDB via a command line and debug console in the Debug GUI.

VSCode Debugger Extensions

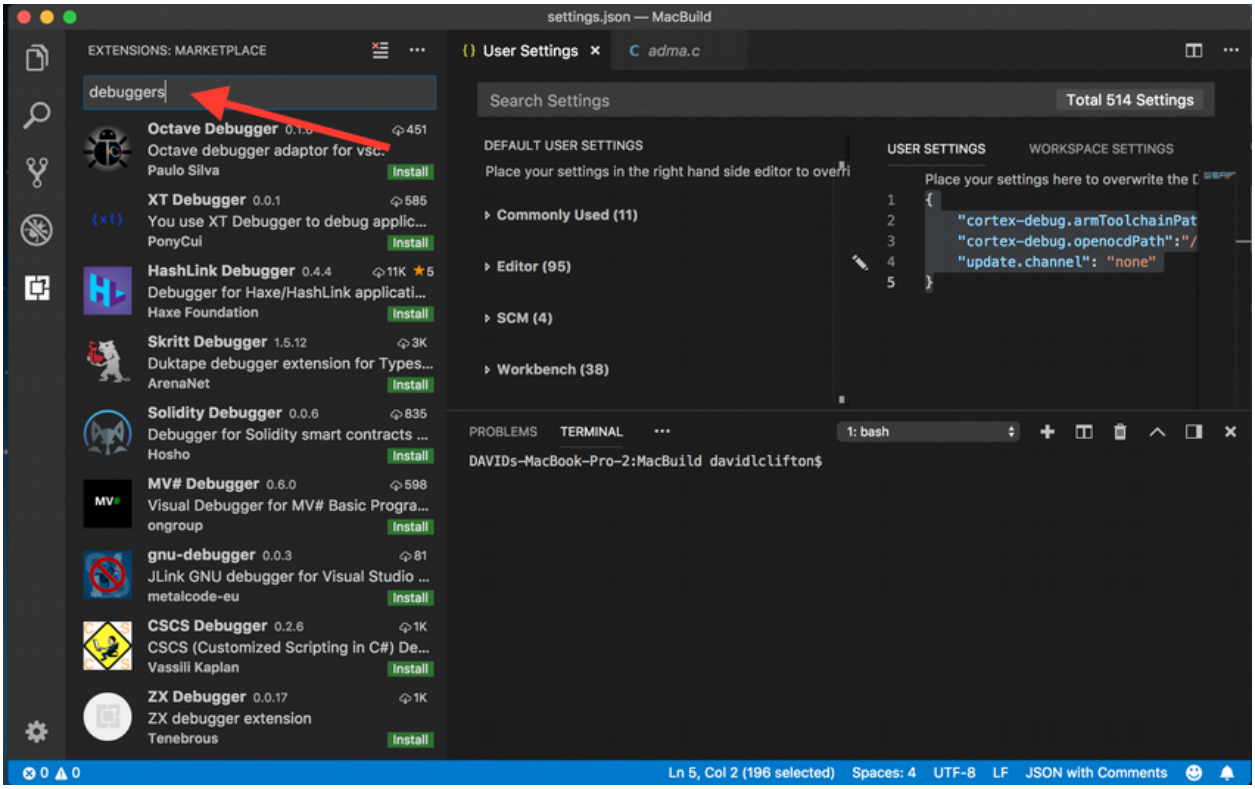
Several websites recommend using the Native Debug VSCode extension from WebFreak as a debugger extension. Other websites revealed that the Cortex-Debug extension was a greater help for Arm Cortex-M development than Native Debug.

For best results, find and install the Cortex-Debug extension into VSCode on each build computer.

How to Find An Extension



At left border of the VSCode window is a square icon that opens an Extensions panel on the left side of the window. At the top of the panel is a text field into which you can enter search terms for extensions.



For example, enter “debuggers” in the search field and hit return. A huge variety of debuggers will appear in the extensions panel. Each entry will have an “Install” button. If you click on an install button, the corresponding extension will be installed to your project. Once installed, the button will change to a gear symbol, which will let you Disable or Uninstall the extension.

Installing the Cortex-Debug Extension

If you type Cortex-Debug into the extension search field and press return, the Cortex-Debug extension should appear as the first entry in the extension panel.

Click it’s Install button. Installation is complete when the install button turns to a gear button.

Skipping Other Recommended Extensions

VSCode recommends several other extensions, including the “C/C++ IntelliSense debugging and code browsing” extension, and the “C/C++ IntelliSense” extension.

Those extensions complain when they cannot find definitions for preprocessor defines and macros that appear in include files, but are never actually used in the build. They also complain about automatically included files in the gcc arm libraries. These issues result in constant interruptions when you are trying to get something else done. Since the example project uses its own makefile, there is no harm in refusing the extensions or deleting them if they somehow get installed.

The IntelliSense extensions use a file in the project folder called .vscode\c_cpp_properties.json. You can delete that as well.

If the code project does not possess it's own makefile, c_cpp_properties.json and IntelliSense extensions may be needed.

You might also avoid the use of the IntelliSense extensions by using a VSCode CMake extension, and defining the build with that tool.

If you need to use IntelliSense extensions and c_cpp_properties.json, you can visit: http://shadetail.com/blog/using-visual-studio-code-for-arm-development-include-paths/ and http://shadetail.com/blog/using-visual-studio-code-for-arm-development-defines/ for further information on working with those extensions.

Obtain and Install JLinkGDBServer

It is relatively easy to setup and Install the JLink debug server. You will need a SEGGER debug probe before you install the debug server software, which is free with the purchase of the debug probe.

Getting the JLink Debug Probe

To obtain a SEGGER debug probe, go the the website www.segger.com, and click on "Products"" in the horizontal menu at the top of the page. Several panels will be displayed, one of which will contain links to six different JLink debug probes.

JLink Base is good enough for use with VSCode. So is JLink EDU. JLink Base sells for $378. JLink-EDU sells for $60. JLink-EDU-mini sells for $18, but you have to search the website more carefully to find it.

Once you have the probe, you'll need to find and install the JLink software package on your development computer.

Installing the JLink Debug Server Software

Click on Downloads at the top of the website. Then click on JLink/JTrace in the first column. A single column list of downloads appears. Choose the JLink Software and Documentation Pack. In the table produced, choose the installer for your platform and version of operating system.

On the Mac, you get a DMG that takes care of installing everything you need. Just make sure that the directory into which you install it is included your OSX PATH variable.

On the Debian mate system, the downloaded debug server could not find the probe, so I tried the beta version of the debug server, and after copying the appropriate files into /etc/udev/rules.d, /opt/SEGGER, and /usr/bin directories, everything worked fine. The Windows version comes with a good installer that takes care of everything including setting the path to the debug server.

Obtain and Install OpenOCD Server

The Cortex-Debug extension also works with the OpenOCD server, ST-Util server, pyOCD server, and Black Magic Probe server.

Setting up the OpenOCD Server

The most informative website on OpenOCD is the one at: http://openocd.org.

There are three ways to get OpenOCD:

- Download the source code for your platform and build it yourself.
- Download via apt-get in debian Linux varieties
- Download a binary installation maintained by other OpenOCD users on GitHub.

Installing OpenOCD on Ubuntu Linux

The easiest installation of OpenOCD is the one for a current Ubuntu Linux.

Simply open a terminal and enter the command:

sudo apt-get install openocd

Depending upon the Ubuntu version, the application openocd is installed in either **/usr/local/bin** or **/usr/bin**. It's data files are installed in directory openocd in either **/usr/local/share** or **/usr/share**.

Installing OpenOCD on Windows 7

There are many ways to do this, including the one described below:

- Download OpenOCD-201807287z from http://gnutoolchains.com/arm-eabi/openocd/
- Copy OpenOCD-201807287z to your tools directory and unzip it, which creates the directory OpenOCD-20180728 in your tools directory.
- The bin subdirectory contains usb and ftdi dlls and the openocd.exe. The drivers subdirectory contains more usb support as well as support for the ST-Link adaptor used by the STM32F7 Discovery Board. The share subdirectory contains chip and board configuration scripts, a universal adaptor hex file, and contributed solutions from users.

Installing OpenOCD on Mac OS

The website http://openocd.org/getting-openocd/ suggests that obtaining the Mac OS version is as easy as installing the Homebrew package manager for macOS, opening a command window, and entering the command:

brew install openocd

The Homebrew package manager can be obtained at https://brew.sh.

> That method installed a version of openocd that did not work well with Visual Studio Code on macOS Sierra. Another way that DID work is described in the apache-newt instructions located at https://mynewt.apache.org/v1_1_0/os/get_started/cross_tools/. Those instructions are included below:

Download the file https://github.com/runtimeco/openocd-binaries/raw/master/openocd-bin-0.10.0-MacOS.tgz for Mac OS.

Change to the root directory:

cd /

Untar the tarball and install into /usr/local/bin. You will need to replace ~/Downloads with the directory that the tarball is downloaded to.

sudo tar -xf ~/Downloads/openocd-bin-0.10.0-MacOS.tar

After downloading and untarring, openocd should be found in /usr/local/bin and the openocd directory containing the scripts subdirectory should be found in /usr/local/share.

Check the OpenOCD version you are using by entering:

which openocd

You should see:

/usr/local/bin/openocd

Check the version number with:

openocd -v

The result should be:

Open On-Chip Debugger 0.10.0

Licensed under GNU GPL v2

For bug reports, read

http://openocd.org/doc/doxygen/bugs.html

If you see one of these errors, take the suggested action:

Library not loaded: /usr/local/lib/libusb-0.1.4.dylib

Suggested Action - execute:

brew install libusb-compat

Library not loaded: /usr/local/opt/libftdi/lib/libftdi1.2.dylib

Suggested Action - execute:

brew install libftdi

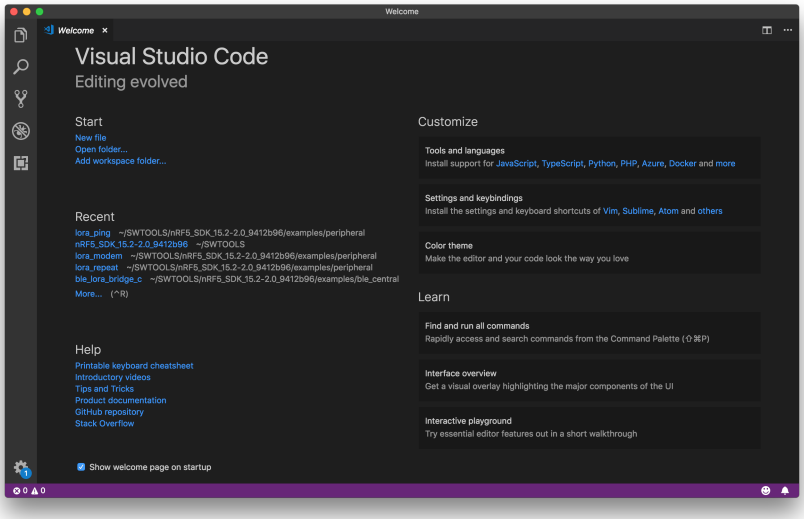
Library not loaded: /usr/local/lib/libhidapi.0.dylib

Suggested Action - execute:

brew install hidapi

The brew versions of these libraries do work correctly

Configure Visual Studio Code



VSCode separates projects by putting them into separate Folders. Each project Folder is setup with configuration and build tools.

Project Configuration

The first step after loading and installing your tools is to configure your project by setting up a means to build it. VSCode supports many build tools; but if you want to support complete cross-platform compatibility, a good approach is to create a standard unix makefile.

Building with a unix make file

Once your project has a valid make file, and builds on macOS or Linux, make can be run immediately from a bash terminal in VSCode. You can open a bash terminal by choosing New Terminal from the Terminal menu at the top of VSCode. Use the cd command, if needed, to position to the same directory as the Makefile. Then just type your make commands.

If you have Cygwin make properly installed in Windows, you can run the same make file from a VSCode Windows cmd.exe terminal in the same way you would from a bash terminal.

In either case, be sure to first enter the same directory as the Makefile.

Install Cygwin make for a Windows project

The Cygwin website www.cygwin.com allows you to download and use a subsystem of unix commands on your windows computer. The Cygwin website presents an elaborate menu that allows you to select the unix subsystems and commands you want. If you make the wrong choices, you can go back and change them as needed by rerunning the Cygwin installation program and making changes to your choices.

It takes a while to get the hang of the Cygwin installer's menu system, but after a while you should be able to download the base Cygwin system, the make tools, and any other convenient unix commands you might need.

I tried at first to substitute the bash shell for the windows terminal in vscode, but it multiplied complications. So I ended up using the cmd.exe shell with the Cygwin base system, a few unix commands, and the make and make depend programs.

Several places on the web recommend using a VSCode, Task.json file to launch your cygwin make. That also leads to a confusing array of choices. The simplest path is to use a the cmd.exe shell with a unix make file, and cygwin make, in your build directory.

> You must support this approach by putting your /cygwin or /cygwin64 directory path, and your arm toolchain path into the Windows PATH variable. Then put the path to your project's VSCode Folder, and the path to your Chip vendor support directories into VARIABLES in your makefile. Those variables can be used by make to locate the source files for including, compiling, and debugging.

When you want to run your make file, you can go to the vscode Folder directory for your project, open a new terminal, enter the same sub-directory as the Makefile, and type make or make clean, (or any other target in the Makefile).

But that is jumping ahead. Next, you need to enter some settings for your compiler and debug tools.

User Settings

A variety of workspace and user settings are possible with VSCode. All the current workspace and user settings can be seen by clicking the gear icon at the lower left of the vscode window. A page will appear which gives you a choice between User Settings and Workspace Settings. Choose UserSettings and scroll down until you find Extensions. After choosing Extensions, click on Cortex-Debug Configuration and then click on **Edit in settings.json*** to bring up the settings.json file which contains your user settings. Enter into settings.json the cortex-debug.armToolchainPath and cortex-debug.JLinkGDBServerPath or cortex-debug.openOcdPath for that debug server.

A sample settings.json file for a Linux installation is shown below:

```
{

    "cortex-debug.armToolchainPath":"/home/dc/SWTOOLS/gcc-arm-none-eabi—8-2018-q4-ma\
jor/bin/",
    "cortex-debug.JLinkGDBServerPath":/usr/bin/JLinkGDBServer",
    "editor.renderControlCharacters":true,
    "editor.renderWhitespace": "all",
    "extensions.ignoreRecommendations": true
}
```

Next is a sample settings.json file for a Windows installation:

```
{

     "git.ignoreMissingGitWarning": true,
     "cortex-debug.armToolchainPath":"path to arm toolchain",
     "cortex-debug.openOcdPath":"path to openocd executable",
     "terminal.integrated.shell.Windows”:”c:\\Windows\\System32\\cmd.exe",
     "update.channel":"none",

}
```

Here are the user settings for my Mac installation:

```
{

    "cortex-debug.armToolchainPath":"/home/dc/SWTOOLS/gcc-arm-none-eabi-7-2018-q2-up\
date/bin/",
    "cortex-debug.JLinkGDBServerPath":/usr/local/bin/JLinkGDBServer",
    “window.zoomLevel”:0,
    “editor.renderControlCharacters”:true”,
    “editor.renderWhitespace”:”all”

}
```

Debug Settings

The launch.jason file, located in a .vscode sub-directory of your project folder, tells the debugger what it needs to know. If launch.json is not already there, you can create it with the text editor and paste it into your .vscode sub-directory. If it is already in your .vscode directory, it probably has some default settings that will not work.

Instructions for showing .hidden folders and files on macOS are available at: https://ianlunn.co.uk/articles/quickly-showhide-hidden-files-mac-os-x-mavericks/ Recent versions of macOS toggle hidden files by pressing shift-command-period.

In Windows 7 you can show extensions and hidden files by clicking the Organize menu in the heading bar near the top of the file explorer window, choosing the Folder and search option choice, and clicking it's View tab. Those options appear under Advanced settings.

An example launch.json file for a windows project is shown below:

```
{

"version": "0.2.0",
"configurations": [

      {
         "request": "launch",
         "type": "cortex-debug",
         "cwd" : "${workspaceRoot}",
         "executable": "c:\\Projects\\lora_ping\\pca10040\\blank\\armgcc\\_build\\nr\
f52832_xxaa,out",
         "servertype": "jlink",
         "device": "nRF52",
         "interface": "SWD”,
         "ipAddress": null,
         "serialNumber":null
      }
  ]
}
```

Example launch.json file for a Mac or Linux project:

```
{
    "version": "0.2.0",
    "configurations": [
{
    "type": "cortex-debug",
    "request": "launch",
    "servertype": “openocd”,
    "cwd": "${workspaceRoot}",
    "executable":"bin/Debug/VSRuntimeMac.elf",
    "name": "Debug STM32F7(OpenOCD)",
    "device": "STM32F7",
```

```
    "configFiles": [
        "board/stm32f7discovery.cfg"
    ]
}
]
}
```

For more information on entries in launch.json for the Cortex-Debug extension, visit the Cortex-Debug website:

https://marcelball.ca/projects/cortex-debug/

It can take a while to configure your first project in VSCode. But after that first one, the rest are relatively easy, and far less work than using a commercial IDE, not to mention the fact that your projects all have cross-platform builds.

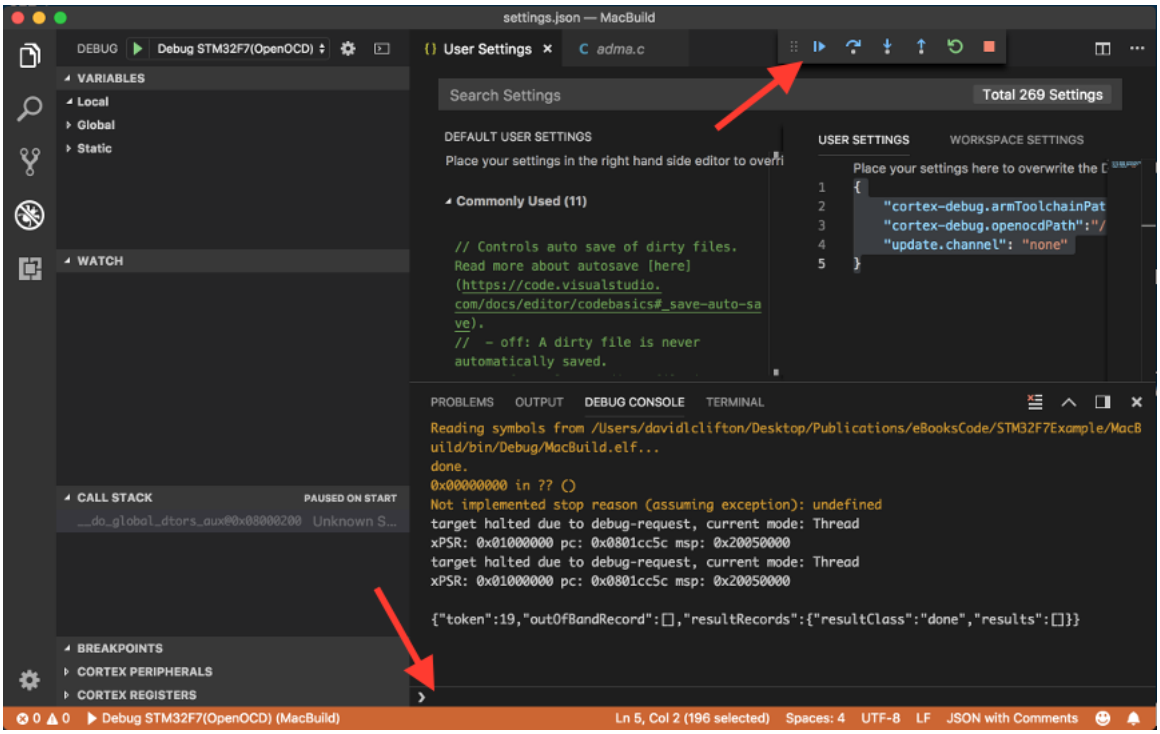
Build and Run the Example Project

The STM32F7 Example Project was derived from an Embitz project for a digital theremin that runs on the STM32F7 Discovery Board.

The information in the EmBitz project files was used to set up the project Makefiles for the Mac, Windows, and Linux builds.

The source code was copied to new Mac, Windows, and Linux project folders, called MacBuild, WindowsBuild, and LinuxBuild. Those project folders were compressed as digital extras for this eBook. The filenames given to the compressed project folders are: **MacBuild.zip**, **LinuxBuild.zip**, and **WindowsBuild.zip**. When you purchased this eBook, **leanpub** sent you an email with a link to a zip file that contains these compressed build directories. You can also download your extras from your **leanpub** Library page at http://www.leanpub.com.

Build and debug the version appropriate for your machine by following the instructions in the appropriate **Build and Run** section after the Debug GUI screenshot below.



The lower red arrow points to the debug command line.

The upper red arrow points to the debug toolbar.

Build and Run the Mac Version

```
Extract the MacBuild.zip file to create a folder called MacBuild.

Open the MacBuild Folder with VSCode.

Select Terminal from the VSCode main menu and New Terminal from the Terminal menu.

At the terminal prompt enter:

   make clean

The terminal should reply:

   rm ./bin/Debug/*.*

or, if the object files were empty to begin with:

   rm: ./bin/Debug/*.*: No such file or directory
   make: *** [clean] Error 1


Next just enter:

   make
```

The build output will appear in the terminal.

If the build does not complete correctly, something probably went wrong in a previous step. Be sure to check that everything in the **Configure Visual Studio Code** section was completed successfully.

Since the example runs on an STM32F7 Discovery Board, you can load the code by opening a finder window to your MacBuild directory and dragging MacBuild/bin/Debug/MacBuild.bin to the Discovery Board icon on the desktop.

Or you can pick Debug from the main menu, and Start Debugging from the Debug menu.

If you started with Debugging, you can click the DEBUG CONSOLE to watch what is happening while you wait for a |> or a || symbol in the debug toolbar on the top right (See the Debug GUI above).

If |> appears in the debug toolbar, click it to start debugging the program.

If || appears in the debug toolbar, position the cursor in the debug command line at the bottom of the screen, and press return. That should make the |> appear in the debug toolbar. Click |> to begin debugging.

Build and Run the Linux Version

```
Extract the LinuxBuild.zip file to create a folder called LinuxBuild.

Open the LinuxBuild Folder with VSCode

Select Terminal from the VSCode main menu and New Terminal from the Terminal menu

At the terminal prompt enter:

    make clean

The terminal should reply:

    rm ./bin/Debug/*.*

or, if the object files were empty to begin with:

  rm: ./bin/Debug/*.*: No such file or directory
  make: *** [clean] Error 1


Next enter:

    make
```

The build output will appear in the terminal.

If the build does not complete correctly, something probably went wrong in a previous step. Be sure to check that everything in the **Configure Visual Studio Code** section was completed successfully.

Since the example project runs on an STM32F7 Discovery Board, you can load the code by opening a Desktop window to your LinuxBuild and dragging LinuxBuild/bin/Debug/LinuxBuild.bin to the Discovery Board icon on the desktop.

Or you can pick Debug from the main menu, and Start Debugging, or Start Without Debugging from the Debug menu.

If you started with Debugging, you can click the DEBUG CONSOLE to watch what is happening while you wait for a |> or a || symbol in the debug toolbar on the top right (See the Debug GUI below).

If |> appears in the debug toolbar, click it to start debugging the program.

If || appears in the debug toolbar, position the cursor in the debug command line at the bottom of the screen, and press return. That should make the |> appear in the debug toolbar. Click |> to begin

debugging.

Build and Run the Windows Version

```
    Extract the WindowsBuild.zip file to create a folder called WindowsBuild.

    Open the WindowsBuild Folder with VSCode

    Select Terminal from the VSCode main menu, and Run Task from the Terminal menu t\
hat appears.

    Choose clean-WindowsBuild and the line:

    rm ./bin/Debug/*.*

    should appear in the terminal.

            rm ./bin/Debug/*.*

    or, if the object files were empty to begin with:

            rm: ./bin/Debug/*.*: No such file or directory
            make: *** [clean] Error 1

    Next, select Terminal from the VSCode main menu, Run Task from the Terminal menu\
, and make-WindowsBuild from the sub-menu that appears.
```

The build output should appear in the terminal.

If the build does not complete correctly, something probably went wrong in a previous step. Be sure to check that everything in the **Configure Visual Studio Code** section was completed successfully.

Since the example runs on an STM32F7 Discovery Board, you can load the code either by copying WindowsBuild/bin/Debug/WindowsBuild.bin to the Discovery Board icon on the desktop, or pick Debug from the main menu, and Start Debugging, or Start Without Debugging from the Debug menu.

If you started with Debugging, you can click the DEBUG CONSOLE to watch what is happening while you wait for a |> or a || symbol in the debug toolbar on the top right (See the Debug GUI below).

If |> appears in the debug toolbar, click it to start debugging the program.

If || appears in the debug toolbar, position the cursor in the debug command line at the bottom of the screen, and press return. That should make the |> appear in the debug toolbar. Click |> to begin debugging.

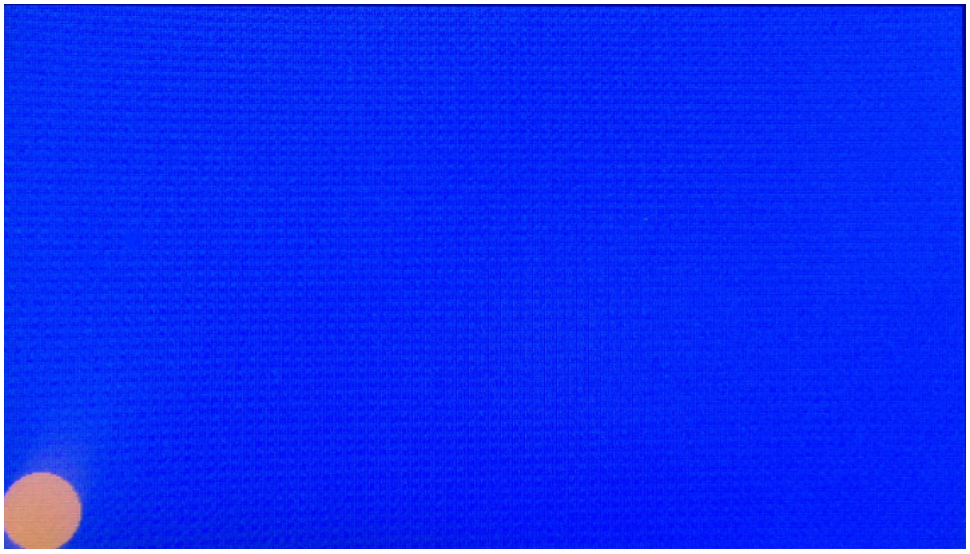
About the Example Project

The example project produces a digital Theremin that is controlled by touching and moving your finger on the STM32F7 Discovery Board Touchscreen.

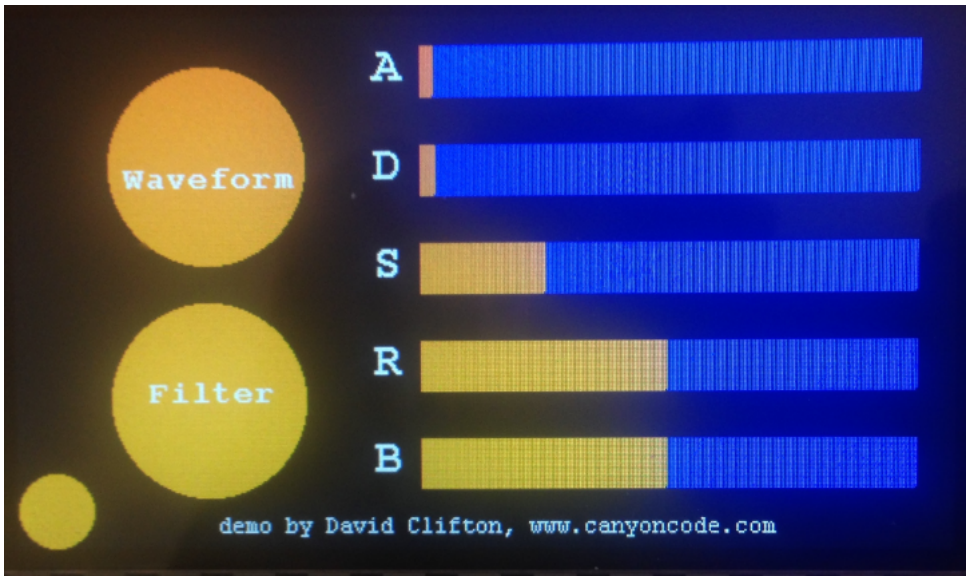
The program has two views, one for playing the Theremin, and the other for setting envelope shape, waveform, and frequency difference between left and right channels. Move between the views by touching the orange dot at the bottom left of the screen.

Screen Shots

XY Controller View



Theremin Settings View



Sound Characteristics

Frequency Range

To hear the output, plug a mini-speaker or headphones into the light green (rightmost) 3.5 mm audio jack attached to the board.

The lowest synthesized frequency is 4 Hz, but that sounds like a sequence of pulses. The lowest recognizable tone is slightly less than 40 Hz or about C2 at the left side of the screen. As the finger moves to the right, the highest fundamental note frequency goes up to 1kHz (about C6) with the finger at the right edge of the screen. Harmonics are present up to about 10 kHz.

Synthesis Settings

Available settings are Waveform, Filter, Attack, Decay, Sustain, Release, and Left-Right channel frequency separation.

Waveform

Press the Waveform button repeatedly to cycle through the available waveforms. Waveforms provided are Saw, Square, Triangle, Sine, and Squaw, which is a combination of Saw and Square Waves.

Filter

Pressing the Filter button toggles a simple smoothing filter on or off.

Attack

The attack slider varies the duration of the attack phase of the note envelope. The attack phase is characterized by the upper half of a logistic function, which is steepest at the onset, and levels out over time.

Decay

The decay slider controls the exponential rate of decay used to reduce signal amplitude from a maximum at the end of the attack phase to the selected Sustain level.

Sustain

The sustain slider controls the level at which the sound is sustained while the finger remains on the touchscreen.

Release

The Release slider controls the rate at which the envelope decays after the userâ€™s finger is removed from the screen.

Frequency Separation

The frequency difference between the left and right channel may be varied from 0-20 Hz with the Beat Frequency slider. This tends to produce a vibrato or slight to strong dissonant effect when played through a speaker. When played through earphones, the binaural beats can produce a pseudo-surround-sound effect, and possible brain wave entrainment.

Access to Project Documentation

The XY Theremin code was written for another eBook available at canyoncode.com: *XY Theremin for STM32F7-DISCO board*. That eBook contains requirements, architecture, design, selected source code, as well as verification and validation tests for the project.

Appendix – When to Use an Open-Source IDE

The Introduction mentions some IDE suppliers who charge a premium, as well as periodic maintenance fees, for the use of their tools.

In my experience there are several reasons why these expensive solutions are often chosen.

1) When a new system is based on an old system, which employs one of the premium IDEs, it normally costs less to start up the new development with the same IDE. This is especially true if the organization uses the same IDE supplier for most of its products.

2) An argument based upon long-term cost of the tool is likely to fall on deaf ears. This is because time-to-market is a more financially relevant consideration than efficient utilization of tools.

3) The code libraries supplied with the premium systems may include an RTOS, ethernet and wireless networking, USB, Graphics, and SD or SDDC card access. If those features are needed, it might be cheaper to go with a premium IDE. A good way to determine if that is the case is to compare the cost of the development seat, with the cost of finding and integrating replacements for the software supplied by the IDE. It pays to be aware that the premium libraries are not guaranteed free of bugs.

4) Even for simple applications, engineers sometimes pick the premium system because it supplies most of the code for system startup, and creating instructions for the linker. If this is the only reason standing between you and choosing open source tools, it is better to go with the open-source solution. Chip makers supply templates to get you started on the startup and linker code. And it is not that hard to figure out how to add what you need.

If market pressures are irrelevant to you, then by all means use the Open-Source IDE. Your learning curve investment will be more than rewarded by repeated reuse of that knowledge, as well as the feeling that comes from accomplishing a useful task without spending more than a few bucks.

Sources Referenced (by URL)

https://blck.mn/2018/01/visual-studio-code-cortex-debug/ Visual Studio Code – Cortex-Debug,Setting up VSCODE to perform the build

https://brew.sh Homebrew, the Missing Package Manager for macOS

https://code.visualstudio.com/docs Visual Studio Code, Documents

https://code.visualstudio.com/docs/extensionAPI/api-debugging Visual Studio Code, Documents

https://code.visualstudio.com/Download Visual Studio Code, Download

https://code.visualstudio.com/license MICROSOFT VISUAL STUDIO CODE, License

http://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads Embedded Arm Compiler Tools, Downloads

https://github.com/runtimeco/openocd-binaries/raw/master/openocd-bin-0.10.0-MacOS.tgz OpenOCD Binary 0.10.0 for macOS

http://gnutoolchains.com/arm-eabi/openocd/ Download OpenOCD with STM32F7, STM32H7 and MSP432 support for Windows

https://www.howtogeek.com/howto/41382/how-to-use-linux-commands-in-windows-with-cygwin How to Use Linux Commands in Windows with Cygwin, JUSTIN GARRISON, JANUARY 25, 2011, 1:23AM EDT

https://ianlunn.co.uk/articles/quickly-showhide-hidden-files-mac-os-x-mavericks/ Quickly Show/Hide Hidden Files on macOS Sierra, OS X El Capitan & Yosemite, Ian Lunn, Ianlunn.co.uk,Jan 6,2014

https://leanpub.com/xythereminforstm32f7-disco XY Theremin for STM32F7-DISCO board - Example for Cross-Platform Cortex-M Development David Clifton, Updated Oct 30,2018

https://marcelball.ca/projects/cortex-debug/cortex-debug-launch-configurations/ Cortex-Debug Launch Configurations Marcel Ball

https://mynewt.apache.org/v1_1_0/os/get_started/cross_tools/ Installing the Cross Tools for ARM Apache Software Foundation, http://www.apache.org

http://openocd.org/doc/doxygen/bugs.html Bug Reporting

http://openocd.org/getting-openocd/ Getting OpenOCD

http://shadetail.com/blog/using-visual-studio-code-for-arm-development-defines/ Using Visual Studio Code for ARM Development – Defines, Michael McAvoy, Embedded Rambling, Rust, September 17, 2017

http://shadetail.com/blog/using-visual-studio-code-for-arm-development-include-paths/ Using Visual Studio Code for ARM Development – Include Paths, Michael McAvoy, Embedded Rambling, Rust, September 17, 2017

https://www.reddit.com/r/embedded/comments/7q1hri/visual_studio_code_extension_for_debugging_arm/ Visual Studio Code Extension for Debugging ARM Cortex-M Microcontrollers, Marus30