

DSP-kit Quick-start Guide

Jonathan Lock, Tomas McKelvey
{lock, tomas.mckelvey}@chalmers.se

September 30, 2019

1 Introduction

This document contains instructions that will help you set up a working development and test environment for the digital-signal-processing kit (DSP-kit) used in the course projects. These projects apply some of the signal-processing methods and algorithms discussed in the course and allow you to test and experiment with them on physical hardware in real-time. The DSP-kit consists of a microcontroller with a microphone and loudspeakers as well as a programming and control interface. You will use a personal computer (PC) to program the DSP-kit to perform a specific task, as well as use the PC's screen and keyboard for interaction (as the DSP-kit does not have an integrated screen or keyboard we will make use of those resources in your PC). The instructions in the following sections will help you set up everything you need to program and interact with the DSP-kit.

To run code on the DSP-kit a *toolchain* is required that compiles, links, and generates assembly code that matches the DSP-kit hardware. The toolchain also manages uploading the assembly code to the DSP-kit, after which your program is run by the DSP-kit hardware. Once the program is uploaded and running a serial *communication channel* is used to control and read status information from the DSP-kit. Here your PC will essentially be used as a glorified keyboard and screen — any keys pressed and characters displayed are directly sent to and received from the DSP-kit respectively without any processing on your PC. We will use PlatformIO <http://platformio.org>, a utility which downloads and installs all the needed programs and utilities for both the toolchain and communications channel. (More on this in Section 2.)

Note for Windows users;

To access the development board a USB driver for the DSP-kit needs to be installed, available at <http://www.st.com/en/development-tools/stsw-link009.html>. You will need to use a valid email address when registering, using e.g. <https://www.mailinator.com/> is a convenient way to avoid future spam. Download, unpack, and run `stlink.winusb.install.bat` in administrator mode *before plugging the DSP-kit into your computer*.

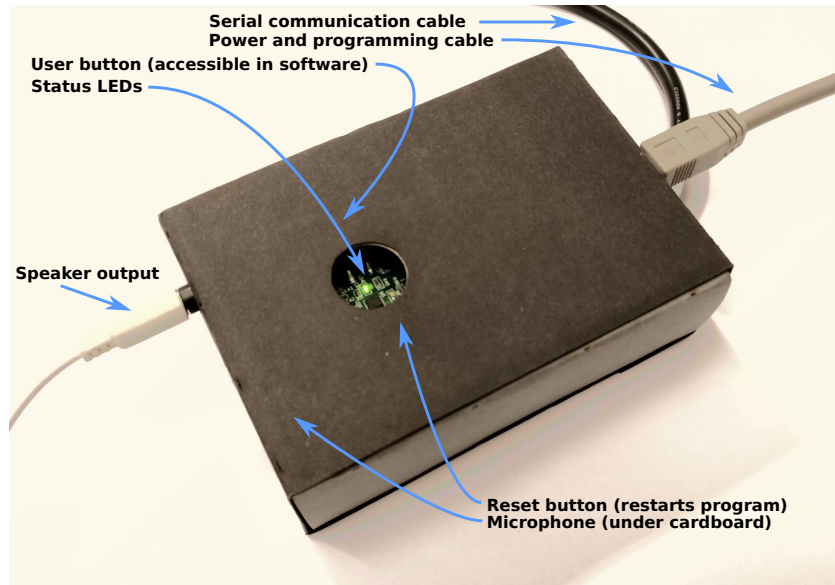


Figure 1: DSP box illustration with input/output, user buttons, and status LED's indicated.

To begin with, ensure you have received a cardboard box containing;

- One black/gray cardboard box with a permanently attached black USB cable. This black/gray box contains all electronics and the microcontroller that will be programmed¹. This should be inside a bubble-wrap bag. *When packed inside the brown cardboard box, disconnect the gray USB cable and keep the DSP hardware inside the bubble-wrap bag!* This will avoid breaking the USB connector and stop loose debris from shorting out the exposed electronics on the next use.
- One gray USB-A to mini-USB cable.
- One pair of small speakers.

To use the DSP-kit you will need a total of three USB ports. Should you only have access to two you can use a cell phone charger with a USB-A port to power the speakers, they do not need to be connected to your computer.

Figure 1 shows the location of all user-accessible input/output on the DSP board. Note that the user and reset buttons are located just outside the circular cutout and are used by lightly pressing on the cardboard case, which is flexible enough to move along with the button.

¹The black/gray box contains an STM32F407G-DISC1, which is a development board with a 168MHz ARM cortex-M4 microcontroller and associated supporting electronics.

Note on problems;

Should you experience issues when using the DSP-kit, first check the current state of the online troubleshooting guide <https://docs.google.com/spreadsheets/d/1fsnSaYTMHcnsjN-a5xMeMxK-9kSq6DsIaacUJoeZyMU/edit?usp=sharing>. Should problems remain contact one of the individuals listed at the start of this document.

2 Install PlatformIO and the code skeleton

Install the toolchain by following PlatformIO's instructions at <http://platformio.org/platformio-ide>. For the tasks we will perform in this course, it makes no difference as to whether Visual Studio Code (VSCode, not to be mistaken with the much heavier and nonfree Visual Studio) or Atom is used. Both environments are free, open-source, cross-platform development environments. For our convenience we recommend you use VSCode unless you have a particular reason to choose Atom.

Note;

You must install both VSCode/Atom and the PlatformIO extension. These are separate steps that are described in the above link. Furthermore, installing the PlatformIO extension will take a while to complete and may require you to restart your PC, possibly multiple times.

The most recent code skeleton can always be found at https://gitlab.com/cth_ssy130/ssy130-files. Download the contents and place the entire project in a folder of your choice.

Note for Linux users;

It is possible that you need superuser privileges to upload the firmware to the DSP-kit and start the serial communications interface.

A quick way of resolving this is to simply run the IDE as root, e.g. for Ubuntu/Debian execute `'sudo code'` in a console application.

Should you not wish to run the entire IDE as root, then follow the instructions for PlatformIO core; <http://docs.platformio.org/en/latest/core.html>. Use the terminal commands `'sudo pio run -t upload'` and `'sudo pio run -t clean'` to upload and clean the source workspace respectively. Use a serial terminal application (e.g. `minicom`) to access the serial device. Be sure to use a baud rate of 921600^a, no flow control, 8 data bits, 1 start bit, and 1 stop bit.

^aIf you have changed the baud rate setting in `config.h`, set the terminal application baud rate to the same value.

4

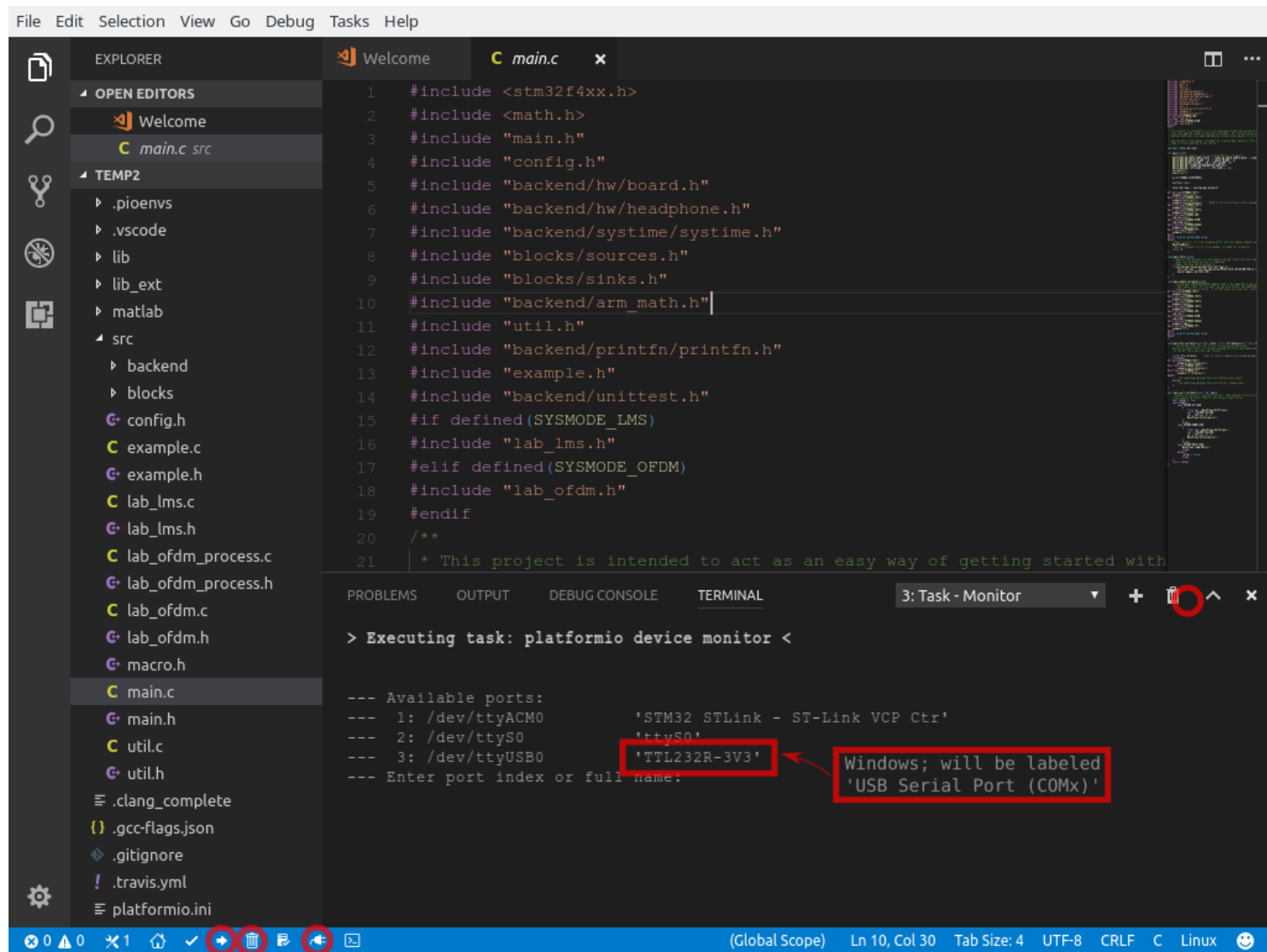


Figure 2: VSCode IDE.

3 Verify toolchain

1. Start VSCode and open the directory you placed the code skeleton in (this should contain a file `'platformio.ini'`, among others).
2. The lower blue bar in VSCode should now contain the icons circled in Figure 2. If not, either the PlatformIO extension has not been completely installed or you have not opened the correct directory.

You will use the circled icons to start all interaction with the DSP-kit. From left to right, use the highlighted icons to

- compile and upload the current source code to the DSP-kit (i.e. use this button to execute your current source code on the DSP-kit),
 - clean the current workspace (use this if you previously aborted a compile midway through, i.e. not needed most of the time),
 - and open the serial monitor (use this to interact with the DSP-kit in real-time).
3. Plug in all three USB cables and connect the speaker's 3.5 mm plug to the DSP-kit.
 4. Compile and upload the current source code (this will take 30 - 50 seconds). On completion (relatively loud) music should start playing from the speakers attached to the DSP-kit.
 5. Open the serial monitor (the rightmost circled icon in Figure 2), select the correct serial device (labeled `'TTL232R-3V3'` (Linux/Mac) or `'Usb Serial Port (COMx)'` (Windows)), and maximize the serial monitor window.
 6. Press the reset button (figure 1). If all is well something like the following text will be displayed in the serial monitor window (you have to have maximized the monitor to see all the output).

```

/---/---\ \ / / |---/ / - \
\---\---\ \ v / | | | \ | | |
---) |---) | | | |---) | | | |
|---/---/ | | | |---/ \---/
| - \ --- | | | | / ( ) | -
| | / --- | ' \ | ' / | | -
| | \ --- | ) | | . \ | | |
|---/|---/ .---/ | | \ \ \ |
      | |

```

Compiled on Oct 13 2017 at 13:00:00

```

System setup;
  Sample rate      (48000) Hz
  Blocksize;      (256) samples
  Mic. lowpass    cutoff; (0) Hz
  Mic. highpass   cutoff; (20) Hz

```

Use the '+' and '-' keys respectively to change the speaker volume during operation.

Use the '0' key to mute/unmute the output.

```

-----
Testing functions with reference input parameters
No unit testing needed for selected project configuration.
Testing complete!
-----

```

```

Usage guide;
Press the user button to switch between audio sources.
Press any of the following keys to change the current effect
  '1' - No effect applied
  '2' - Flanger
  '3' - Ring modulator
  '4' - Bit crusher
  '5' - Increase bit crusher amplitude bit-depth
  '6' - Decrease bit crusher amplitude bit-depth
  '7' - Increase bit crusher sample-rate
  '8' - Decrease bit crusher sample-rate
  '9' - Toggle bit crusher reconstruction mode

```

Most of the above contents will always be shown on startup, giving some useful system information. If the speakers are plugged in, they should now be playing some music. Pressing any of the indicated keys should cause the output audio to change and display a short message on the screen.

Volume control;

Pressing the '+', '-', or '0' keys will always increase, decrease, or mute/unmute the speaker output regardless of the configured project, but will not give a response on the screen.

If all the above steps have succeeded then your computer is now fully configured for use with the DSP-kit and you should be able to proceed with the project coding tasks.

4 Changing the system operation mode for projects with config.h

config.h, located in the src directory, contains many useful settings, some of which you will need to change depending on the current assignment you are working on. Most of the configuration statements are fairly well documented; the following in particular are worth additional attention.

4.1 SYSMODE

The SYSMODE.XXX define statements control the entire system behavior, setting sample-rate, block-size, and which example/lab functions will be called. For example, in order to compile for the OFDM-lab, comment `#define SYSMODE_TEST1` and uncomment `#define SYSMODE_OFDM`. This would give the following SYSMODE section:

```
//Fully functional example system modes
//#define SYSMODE_TEST1
//#define SYSMODE_TEST2
//#define SYSMODE_TEST3
//#define SYSMODE_TEST4
//#define SYSMODE_RADAR
//#define SYSMODE_FFT

//Student project system modes
#define SYSMODE_OFDM
//#define SYSMODE_LMS
```

By default, the application will be compiled for SYSMODE_TEST1. Any of the other example modes can be activated simply by uncommenting the relevant line. Details on what each test mode does and how to use them is listed in comments in config.h. All source code for the test modes can be found in example.h/.c.

4.2 Audio waveform

For SYSMODEs where an audio waveform is played back (e.g. SYSMODE_TEST1) the audio waveform can be changed by changing the

```
#define MP3_DATA_XYZ
```

statement. Permissible values are listed in /src/blocks/waveforms/mp3_data.h. For example, one permissible value is

```
#define MP3_DATA_FUR_ELISE
```

which will configure the system to load the associated waveform.

Should you wish, you can generate your own music tracks for playback. To do this, start with an MP3 file with a sample rate of 16 kHz, mono audio, and no larger than approximately 900KiB (a compiler error will be generated if your track is too

large). A bitrate of 24-32 kb/s is a good tradeoff between size and quality for this application but is not critical. You can use the free application Audacity <https://www.audacityteam.org/> to resample, mix to mono, and trim the track length as needed. Now, place the generated file in the `/src/blocks/waveforms/raw` directory and (assuming python 3 is available on your computer) execute `python3 converter.py` in the `/src/blocks/waveforms` directory. If all has gone well you should see that `mp3_data.h` and `mp3_data.h` have been updated to include your music track. Finally, select your music track by modifying the `MP3_DATA_XYZ` statement as applicable.

4.3 Serial baud-rate

If you get an error message along the lines of an invalid baud rate, or you only receive garbled characters in the serial monitor window, then your PC probably does not support the baud-rate (communication speed for serial interface) we have chosen. You can change the baud rate to a value supported by your machine by modifying `config.h` and `platformio.ini`. Suppose your PC supports a baud-rate of 1000000, then change

```
#define DEBUG_USART_BAUDRATE          (921600ULL)
```

to

```
#define DEBUG_USART_BAUDRATE          (1000000ULL)
```

also, in the file `platformio.ini`, change the baud rate setting

```
[env:proj_master]  
monitor_baud = 921600
```

to

```
[env:proj_master]  
monitor_baud = 1000000
```

Note that selecting lower baud-rates decreases DSP-kits performance, as more time is spent shuffling data to the PC. Generally speaking, higher baud-rates will always be better as long as the output remains valid.

5 The DSP-kit in day-to-day use

At this stage you should have a working toolchain that allows you to program the DSP-kit as well as a working serial communication channel. A typical workflow for the DSP-kit for implementing/testing a project task is;

1. Connect all three USB cables (using a separate USB power supply for the loudspeakers if needed) and connect the 3.5 mm speaker plug to the DSP-kit.
2. Open VSCode (opening the project directory if needed).

3. Modify the source code as needed.
4. Use the upload button, lower leftmost in Figure 2, to compile, link, and upload the current source code to the DSP-kit.
5. Open/switch to and maximize the serial communication interface, lower and upper rightmost respectively in Figure 2, and interact with the DSP-kit as needed.
6. If you find an issue with the DSP-kit behavior, go to step 3. Otherwise, great success; start on the project report.

In general, there is no need to connect/disconnect any hardware during development, i.e. you may keep the speakers and serial communication USB cable plugged into your computer during all stages of development.

Avoid disconnecting or resetting the DSP-kit using the reset button while uploading new software to the device, as this may brick the DSP-kit (i.e. turn the DSP-kit into a paperweight that you cannot program or use in any meaningful way). Though this will not damage the DSP-kit hardware it is somewhat time-consuming to resolve. Should this occur either install the ST-link utility <http://www.st.com/en/development-tools/stsw-link004.html> (Windows-only) and perform a full chip erase, or come to one of the authors of the document where we will do this for you.

6 DSP-kit software back-end

In order to understand the system behavior it is important to have some basic knowledge of the DSP-kit software back-end.

6.1 Microphone and speaker data streams

Figure 3 illustrates the structure of the back-end that handles supplying the application code (i.e. your code) with microphone samples while also shuffling the requested speaker output to the physical speakers. Note that there is a single microphone input, but two (identical in structure) speaker outputs.

The source for `SYSMODE.TEST2` illustrates a minimal example of the application code needed to read from the microphone and write to the speakers, as the relevant lines in `/src/example.c` contain only;

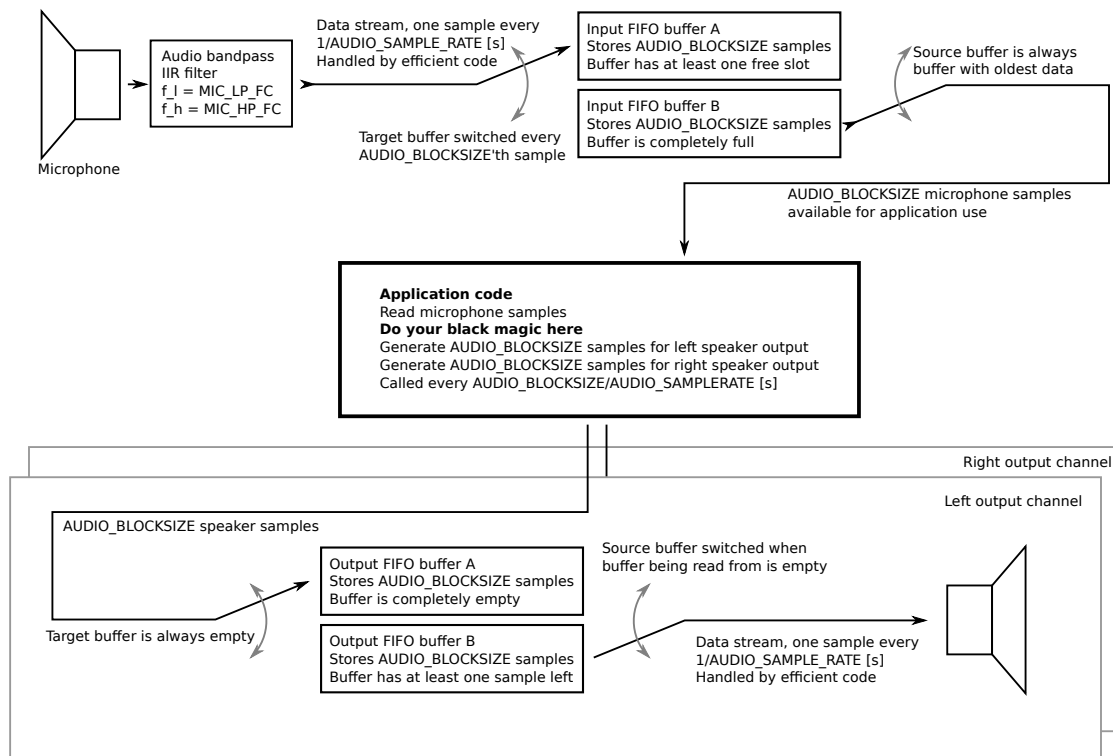


Figure 3: Data stream overview.

```

#if defined(SYSMODE_TEST2)
void example_test2_init(void){
    //No setup needed
}

void example_test2(void){
    //Allocate space for the microphone and waveform samples.
    float micdata[AUDIO_BLOCKSIZE];
    float wavedata[AUDIO_BLOCKSIZE];

    //Write AUDIO_BLOCKSIZE samples to the microphone and waveform buffer
    blocks_sources_microphone(micdata);
    blocks_sources_waveform(wavedata);

    //Send the waveform data to the left output and the microphone data to
    //the right output without any processing
    blocks_sinks_leftout(wavedata);
    blocks_sinks_rightout(micdata);
}
#endif

```

Here, `example_test2_init` (which does nothing) is called once on startup by `main.c`. After this, `example_test2` is called by `main.c` each time the next microphone and speaker buffers are full and empty respectively. With the default settings of `AUDIO_SAMPLERATE = 16000` and `AUDIO_BLOCKSIZE = 256` this implies `example_test2` is called once every $\frac{256}{16000} \approx 16$ ms.

The back-end data stream diagram also gives the following insights,

- There is no requirement to read the microphone data — it can be ignored — which results in the microphone buffer being overwritten with new data.
- The application code is called once an entire block (i.e. a contiguous group `AUDIO_BLOCKSIZE` samples) have been buffered. This structure helps reduce function call overhead, leaving more execution time available for useful signal processing in the code you will write.
- The total round-trip latency from the microphone to the loudspeaker will always be at least $3 \cdot \text{AUDIO_BLOCKSIZE}$ (as `AUDIO_BLOCKSIZE` samples are first buffered in the microphone buffer, then in a buffer for the application code, and then finally in the speaker buffers). This implies that any microphone-to-speaker filtering you perform will consist of the physical channel behavior, followed by some additional latency that can be viewed as a linear system with response;

$$y[n] = x[n - 3 \cdot \text{AUDIO_BLOCKSIZE}]$$

i.e. a simple propagation delay. In practice, the actual delay is slightly longer as the microphone IIR filter also adds some latency, though for most purposes this is small enough to ignore.

- It is crucial that the application code finishes execution within $\text{AUDIO_BLOCKSIZE} / \text{AUDIO_SAMPLERATE}$ seconds. If it does not, the loudspeaker buffer will run out of

queued samples and (by design) the system will stop operating. This condition is tested for in the back-end and an error message will be printed should this occur².

6.2 Self-testing

During start-up, the DSP-kit will perform a self-test of all the functions you need to write code for when the relevant `SYSMODE` configuration is selected.

The system self-test will apply known inputs to your function(s) and compare their output to known-good (i.e. assumed correct) output. If the functions you write code for do not return values identical or close enough to the reference output execution will stop and an error message will be displayed.

This functionality is useful as it adds some degree confidence that your code works correctly and that the subsequent system behavior you see is expected. However, it is possible for your code to not pass the self-testing despite being correct (false-negative), and conversely, it is possible that your code passes the self-test while in fact generating incorrect results (false-positive). In summary, though the self-test functionality is useful, do not blindly place faith its verdict — *trust, but verify*.

See Algorithms 1, 2, and 3 for a simple example of some types of errors that can and cannot be detected respectively.

Undetectable erroneous code will usually manifest with one or more of the following behaviors;

- spontaneous crashes (noticeable as no response to key presses through the serial monitor and the green status LED will stop blinking),
- garbage or mangled data printed to serial monitor, and/or
- otherwise generally erratic and variable behavior.

Should you find a case where you are certain the self-test-functionality returns either a false-negative (i.e. an error message prints and execution halts, despite you being sure of the validity of your code) or a detectable false-positive (i.e. code execution commences and you are sure the displayed behavior is incorrect and you are not incorrectly addressing memory) please let one of the authors listed at the top of this document know so a patch may be issued.

²If the execution time is too large and does not increase with `AUDIO_BLOCKSIZE` (e.g. you wish a print a long status message, that is not made longer when increasing `AUDIO_BLOCKSIZE`), the permissible time can be increased both by increasing `AUDIO_BLOCKSIZE` and by decreasing `AUDIO_SAMPLERATE`.

Algorithm 1 Reference code.

Adds two vectors **a** and **b** of length **length**, storing result in **res**.

```
void vec_add_ref(float * a, float * b, float * res, int length){
    int i;
    for(i = 0; i < length; i++){
        res[i] = a[i] + b[i];
    }
}
```

Algorithm 2 Detectable erroneous code.

res[0] not set (will contain undefined (i.e. seemingly random) data) and **'*** operation used instead of **'+** operation.

```
void vec_add_detect(float * a, float * b, float * res, int length){
    int i;
    for(i = 1; i < length; i++){
        res[i] = a[i] * b[i];
    }
}
```

Algorithm 3 Undetectable erroneous code.

In the for loop the last iteration will be for **i=length**, so we're addressing **a[length]** and **b[length]**, i.e. the **length+1**'th element. This will not cause a crash, but will give undefined (i.e. seemingly random) values for **a** and **b**. More critically, we're writing to **res[length]**. This memory location is most likely used by some other variable, so we're probably corrupting other parts of the program. Note that calling this function, if it doesn't crash, will generate a vector **res** that is identical to the output from **vec_add_ref**, i.e. *if the DSP-kit doesn't crash, the self-testing function cannot determine that this function is broken!* (For reference, **a[0]** and **a[length-1]** are the first and last elements of **a** respectively).

Furthermore, **a_big_array** will use more memory than available (10^6 floating point values, 4 bytes each, requiring a total of ≈ 3.9 MiB of memory, while the DSP kit has 192 KiB of memory).

If we're very lucky the compiler will detect these faults, but don't count on it!

```
void vec_add_undetected(float * a, float * b, float * res, int length){
    float a_big_array[1000000];
    int i;
    for(i = 0; i <= length; i++){
        res[i] = a[i] + b[i];
    }
}
```
