stmdsp Guide

v0.1

Clyne Sullivan

Table of Contents

Installation

First, install the arm-none-eabi compiler toolchain from the link below:

https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads

Choose the installation option to add the programs to your PATH or environment variables.

Next, install make:

http://gnuwin32.sourceforge.net/packages/make.htm

Writing algorithms for stmdsp

This tutorial will guide you through writing your own custom DSP algorithms for the `stmdsp` device.

Algorithms are written in the programming language C++. The following C++ resources may be useful:

- C++ tutorial: https://www.cplusplus.com/doc/tutorial/
- C++ language reference: https://en.cppreference.com/w/

When you create a new algorithm, the following template is given:

```
Sample *process_data(Samples samples)
{
    return samples.data();
}
```

The function `process_data` is responsible for transforming the incoming signal. Samples from the input signal are provided in the `samples` parameter; the function is then expected to return a pointer to the transformed sample data. In the above template, the input samples are simply provided as the output, creating a signal "passthrough".

- The amount of samples in the `samples` parameter can be set by the GUI, defaulting to the maximum of 4096.
- Signal samples are 12-bit integers: the lowest sample value is zero (-3.3 volts) while the highest values is 4095 (+3.3 volts).

Here is another example of an algorithm that halves the values of the input samples:

```
Sample *process_data(Samples samples)
{
    // This line is a comment.
    // Half the value of every sample in the buffer:
    for (int i = 0; i < samples.size(); i++) {
        samples[i] = samples[i] / 2;
    }

    return samples.data();
}
```

Some algorithms may require the input data to be preserved. In this case, a buffer for the algorithm's output data can be created and used:

```
Sample *process_data(Samples samples)
{
    // Create an array of samples:
    static Sample output[samples.size()];

    for (int i = 0; i < samples.size(); i++) {
        // Use the output buffer in case we'll need the original input
        // samples later.
        output[i] = samples[i] / 2;
    }

    return output;
}
```

- The `Samples` object is based on C++'s std::span object, which wraps over a regular `Sample` array. `std::span` provides useful utilities for more complex algorithms.
- These algorithms use two different types of memory: stack memory and program memory. By default, variables in your algorithm will be in stack memory; however, the amount of stack memory available is small compared to the available program memory. Larger variables, like sample buffers, should be placed in program memory by declaring them with the **`static`** keyword.

Using the signal generator

The `stmdsp` device provides a single-channel signal generator in case an external signal source is not available. The generator can be initialized in three ways:

1. Sample list: A list of sample values (ranged zero to 4095) separated by any non-digit characters. For example, a list could be generated in a spreadsheet then simply copy/pasted into the GUI.
2. Formula: An "f(x)" function can be supplied to generate a 4096-sample buffer. Remember that the function should provide values in the zero to 4095 sample value range.
3. Audio: A .wav file can be loaded and played through the signal generator. The audio data is sent to the generator when the algorithm is started.

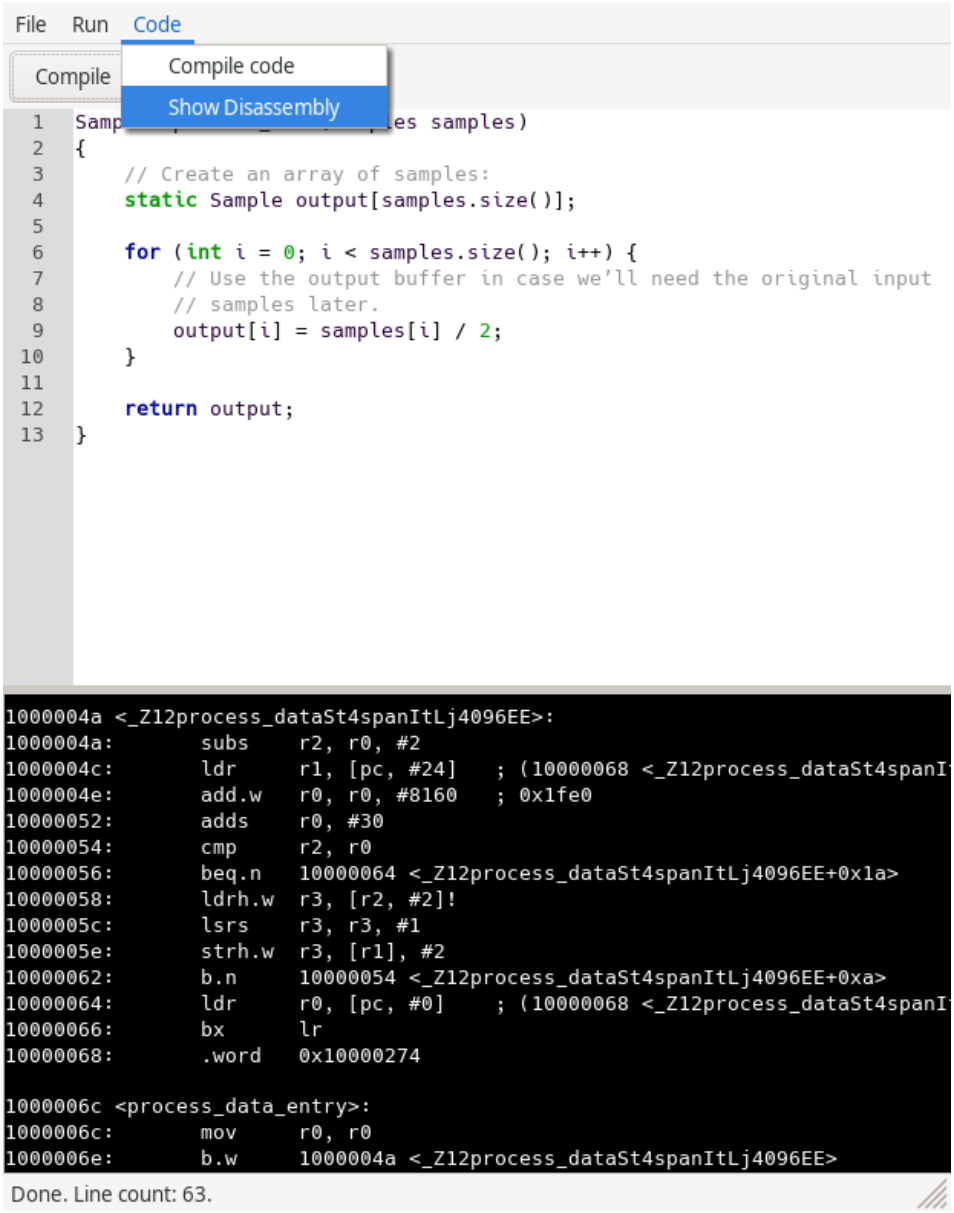
The signal will be produced at the current sample rate.

Testing and analyzing algorithms

The GUI is provides numerous features to study your algorithm's performance:

Viewing algorithm disassembly

After you have successfully compiled your algorithm, you can view your algorithm's disassembly. This allows you to see the processor instructions that make up your algorithm, and can be studied to determine an algorithm's efficiency. To view the disassembly, click the "Show disassembly" option under the Code menu. The disassembly will be displayed in the terminal pane, as shown below:



Measuring algorithm execution time

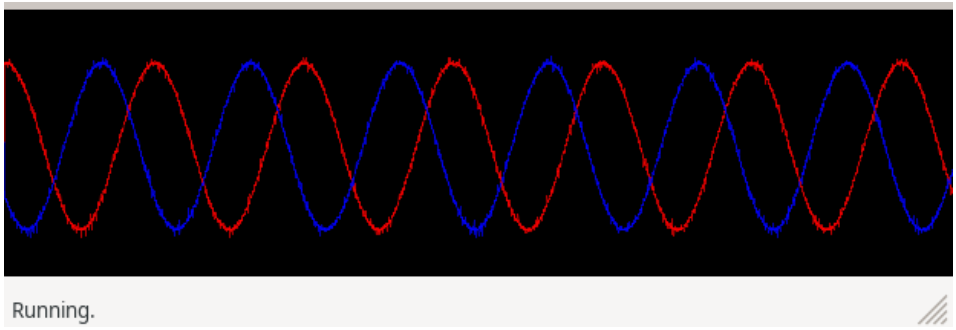
The device is capable of measuring the approximate execution time of your algorithm in processor cycles. This can give a rough estimate of your algorithm's efficiency.

To enable execution time measurement, check the “Measure code time” option in the Run menu. The next time you run your algorithm, the status bar will display how many processor cycles it took to run your algorithm:

Viewing the input and output waveforms

A rough visualization of the input and output waveforms can be displayed over the terminal pane during algorithm execution. To enable this option, check the "Draw samples" option under the Run menu.

The next time you run your algorithm, the terminal pane will display the input waveform in blue and the output waveform in red.



Recording the output signal to a file

The GUI can record the output signal's samples to a comma-separated (.csv) file for further analysis. This feature works at all of the available sample rates.

To enable output recording, simply check the "Log results..." box under the Run menu. You will be prompted to choose where to save the file. Once your algorithm is started, samples will be recorded to the file until the algorithm is stopped.

Device configuration options

The GUI allows for the configuration of the device's sampling rate and algorithm buffer size, which may affect the performance of your algorithm.

The sampling rate determines the frequency at which the device samples the incoming signal. Higher sampling rates have benefits including improved input and output signal detail, as well as the ability to detect higher frequency signals. However, high sample rates come at the cost of less time available for algorithm execution, restricting an algorithm's capabilities.

Setting the sample rate is simply achieved using the drop down box above the code editor:

Note that the selected sampling rate also applies to the signal generator; that is, the generator will create its signal at the same rate.

Changing the algorithm's buffer size results in your algorithm having access to larger (or smaller) chunks of a signal when executed. This may be useful when operations over longer periods of time are needed. Larger buffer sizes result in a longer maximum execution time, but in return may require additional memory usage by your algorithm. Using a smaller buffer size would be necessary for memory-intensive computations; a smaller buffer size also means your algorithm and output will respond more quickly to changing input.

To set the buffer size, simply select the “Set buffer size…” option under the Run menu.