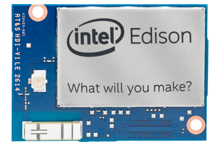

Intel® Edison Tutorial: Liquid-DSP – Signal Processing

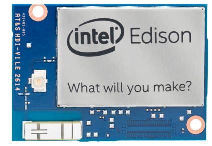

Table of Contents

Introduction

This tutorial will guide users through primary example functions, and data structures available within the liquid-dsp library. To provide essential experience the examples provided include example code constructions. Users will be guided to modify this code to apply low-pass and high-pass filters to a given signal. This tutorial assumes that students are familiar with signal processing concepts such as low-pass filtering, high-pass filtering, and elementary complex number analysis. This document also is intended to help users gain experience in code review and documentation.

The liquid-dsp library is a valuable resource developed to support many applications including that of the development of the software defined radio technology. This is documented here:

www.liquidsdr.org

Prerequisite Tutorials

Users should ensure they are familiar with the documents listed below before proceeding.

1. Intel Edison Tutorial – Introduction, Linux Operating System Shell Access
2. Intel Edison Tutorial – Introduction to Linux
3. Intel Edison Tutorial – Introduction to OPKG
4. Intel Edison Tutorial – Introduction to Vim
5. Intel Edison Tutorial – Liquid-DSP – Installation Guide

List of Required Materials and Equipment

1. 1x Intel Edison Compute Module.
2. 2 x USB 2.0 A-Male to Micro-B Cable (micro USB cable).
3. 1x powered USB hub **OR** 1x external power supply.
4. 1x Personal Computer.

Digital Signal Processing Code

Follow the steps below in order to develop an understanding of liquid-dsp program development.

1. Access the shell on the Intel Edison. For more information, refer to the document labelled ***Intel Edison Tutorial – Introduction, Shell Access and SFTP***.
2. Ensure that the Intel Edison has the Vim editor installed. For more information, refer to the document labelled ***Intel Edison Tutorial – Introduction to Vim***.
3. Ensure that the Intel Edison has Git installed. For more information, refer to the document labelled ***Intel Edison Tutorial – Introduction to OPKG***.
4. Ensure that the Intel Edison has internet access.
5. Navigate to the home directory.

   $ cd

6. Acquire the provided files by issuing the command shown below.

   $ git clone https://github.com/pban1993/edison_dsp_chirp.git

```
root@edison:~# ls
root@edison:~# git clone https://github.com/pban1993/edison_dsp_chirp.git
Cloning into 'edison_dsp_chirp'...
remote: Counting objects: 12, done.
remote: Compressing objects: 100% (9/9), done.
remote: Total 12 (delta 2), reused 10 (delta 2), pack-reused 0
Unpacking objects: 100% (12/12), done.
Checking connectivity... done.
root@edison:~# ls
edison_dsp_chirp
root@edison:~#
```

Figure 1: Successful cloning of edison_dsp_chirp repo.

7. Enter the edison_dsp_chirp directory and read the source code file **filter_chirp.c**.

   $ cd edison_dsp_chirp
   $ vi filter_chirp.c

```
int main(int argc, char **argv)
{
        /* structures for DSP */
        float input_x[BUFF_SIZE], input_y[BUFF_SIZE];
        float complex *x, *y_orig, *y_filt;
        unsigned int i;         /* current sample index */
        unsigned int order;     /* filter order */
        unsigned int n;         /* number of samples in the file */
        float Fc;               /* actual cutoff frequency */
        float Fs;               /* sampling frequency */
        float fc;               /* normalized cutoff frequency Fc/Fs */
        float f0;               /* center frequency for bandpass. Unused. */
        float Ap;               /* pass-band ripple */
        float As;               /* stop-band attenuation */
        /* filters */
        liquid_iirdes_filtertype f_type;
        liquid_iirdes_bandtype b_high_pass, b_low_pass;
        liquid_iirdes_format f_format;
        iirfilt_cccf filter_high_pass, filter_low_pass;

        /* Variables for reading file line by line */
        char *ifile_name, *ofile_name;
```

Figure 2: Snippet from filter_chirp.c.

8. Notice the **float complex** data structure.

- **float complex** is a data structure for storing complex float numbers.

  Examine the below code snippet involving declaration and initialization of a **float complex** data type**.**

  ```
  float complex z;
  …
  z = 3.0f + _Complex_I*4.0f;
  ```

- There are a number of functions to access the data within a **float complex** variable.

  cabsf() returns the absolute value of the complex number.
  cargf() returns the argument of the complex number.
  crealf() returns the real component of the complex number.
  cimagf() returns the imaginary component of the complex number.

  ```
  printf("z: %8.4f + j %8.4f\n", crealf(z), cimagf(z));
  printf("z: %8.4f / _%8.4f\n", cabsf(z), cargf(z));
  ```

- Users may declare arrays of **float complex,** as with any other data structure.

  ```
  float complex z[1024];
  float complex *z1;
  …
  z1 = (float complex *) malloc(sizeof(float complex) * 1024);
  ```

For more information, please open the following links on a web-browser on a personal computer.

http://en.cppreference.com/w/c/numeric/complex
http://man7.org/linux/man-pages/man7/complex.7.html

9. Exit the Vim editor.
10. Use the Makefile to compile the C-code source file into an executable binary file.

$ make

```
root@edison:~/edison_dsp_chirp# make
gcc -o filter_chirp filter_chirp.c -lc -lm -lliquid
root@edison:~/edison_dsp_chirp#
```

Figure 3: Successful compilation of filter_chirp.c.

11. Execute the binary file.

$ ./filter_chirp

```
root@edison:~/edison_dsp_chirp# ./filter_chirp
Attempting to read from file 'chirp_data.csv'.
Attempting to write to file 'output_data.csv'.
root@edison:~/edison_dsp_chirp#
```

Figure 4: Successful execution of ./filter_chirp.

12. Examine the first 10 lines of the file

$ head output_data.csv

Notice:

- **sample** is the 0-indexed sample number for the data.
- **x** is the time series information in seconds.
- **y_orig_r** is real part of the original signal from **chirp_data.csv**.
- **y_orig_i** is 0 because the signal in **chirp_data.csv** has no imaginary component.
- **y_filt_r** is 0 because the code does not currently perform any signal processing.
- **y_filt_i** is 0 because the code does not currently perform any signal processing.

```
root@edison:~/edison_dsp_chirp# head output_data.csv
sample,x,y_orig_r,y_orig_i,y_filt_r,y_filt_i
0,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000
1,  0.0100,  0.1647,  0.0000,  0.0000,  0.0000
2,  0.0200,  0.1989,  0.0000,  0.0000,  0.0000
3,  0.0300,  0.1524,  0.0000,  0.0000,  0.0000
4,  0.0400,  0.1871,  0.0000,  0.0000,  0.0000
5,  0.0500,  0.3530,  0.0000,  0.0000,  0.0000
6,  0.0600,  0.5190,  0.0000,  0.0000,  0.0000
7,  0.0700,  0.5530,  0.0000,  0.0000,  0.0000
8,  0.0800,  0.5041,  0.0000,  0.0000,  0.0000
root@edison:~/edison_dsp_chirp#
```

Figure 5: First 10 lines of output_data.csv.

Examination of the Input Signal

The input signal provided in the comma separated value file labelled **chirp_data.csv** is a summation of three component signals. These signals are listed below.

Chirp Component

1. The chirp signal is defined:

$$y_1(t) = A\sin(\omega(t)t)$$

where $\omega(t)$ is a function of time, such that the frequency linearly increases with time and where amplitude A is constant.

The chirp component in **chirp_data.csv** is defined below.

At sample 0, the frequency of the sine wave is 1Hz (6.28rad/sec).

At sample 1024, the frequency of the sine wave is 10Hz (62.8rad/sec).

The amplitude is a constant value of 1.

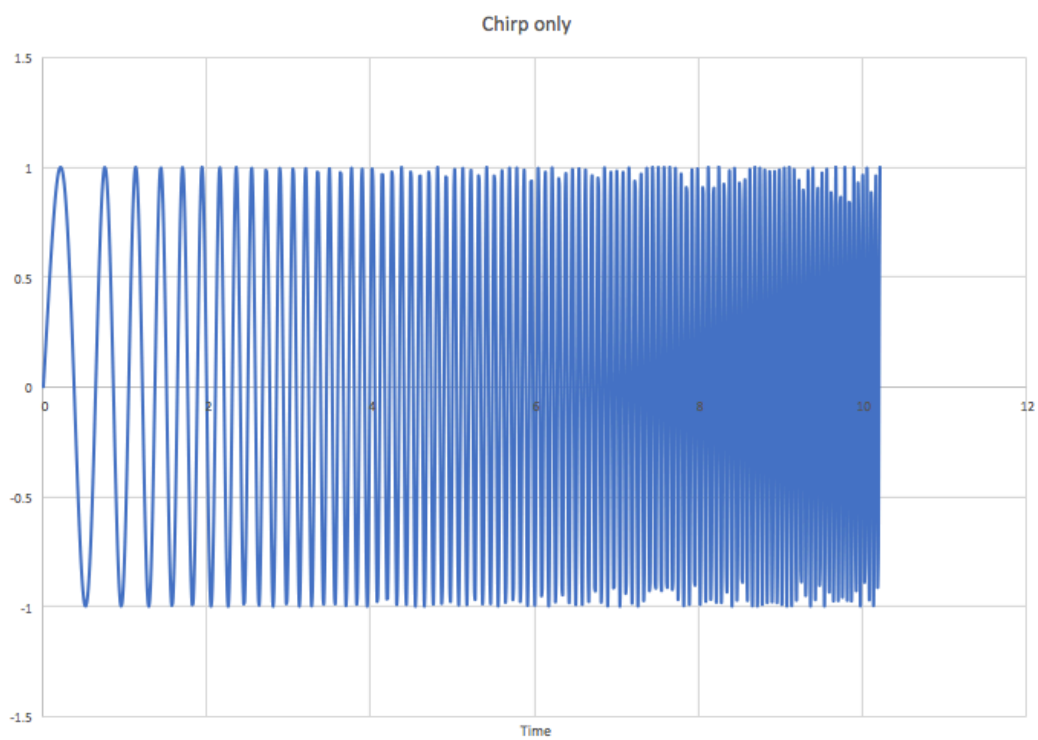

Figure 6: Chirp signal component from chirp_data.csv.

Low Frequency Noise Component

2. A large amplitude, low frequency noise signal is defined:

$$y_2(t) = Bsin(\omega_1 t)$$

where ω_1 is a constant and where amplitude B is a constant.

The low frequency noise component in **chirp_data.csv** is defined below.

The frequency is a constant value of 0.1Hz = 0.628rad/sec.

The amplitude is a constant value of 1.

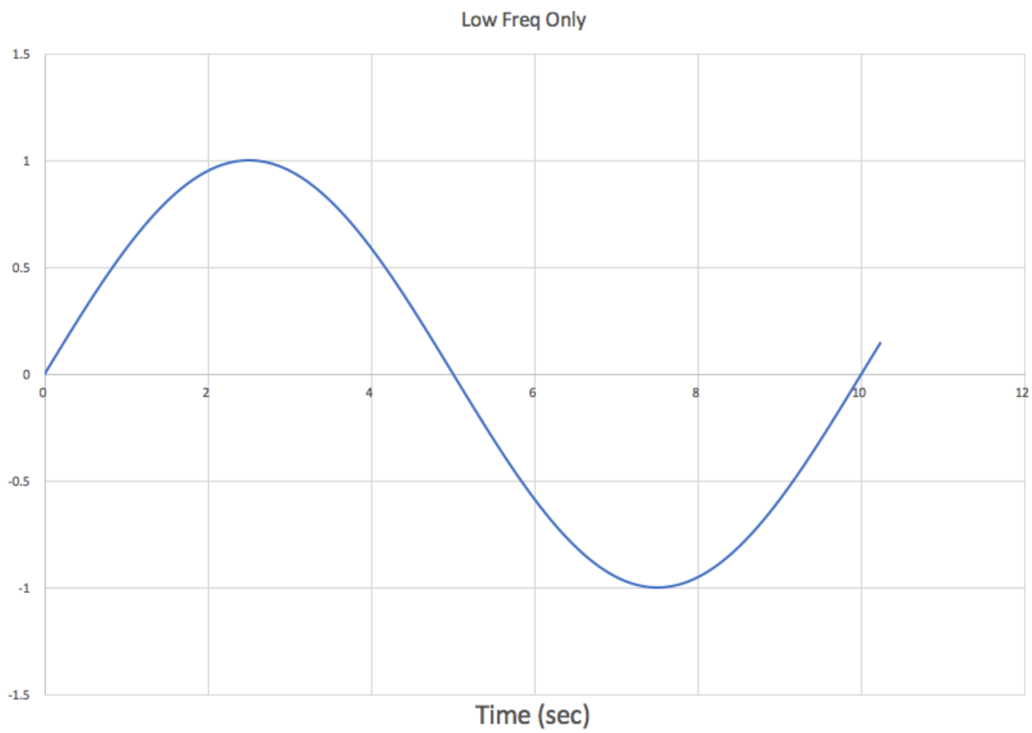

Figure 7: Low frequency component from chirp_data.csv.

High Frequency Noise Component

3. A low amplitude, high frequency noise signal is defined:

$$y_3(t) = Csin(\omega_2 t);$$

where ω_2 is a constant and where amplitude C is a constant.

The high frequency noise component in **chirp_data.csv** is defined below.

The frequency is a constant value of 10Hz = 62.8rad/sec.

The amplitude is a constant value of 0.1.

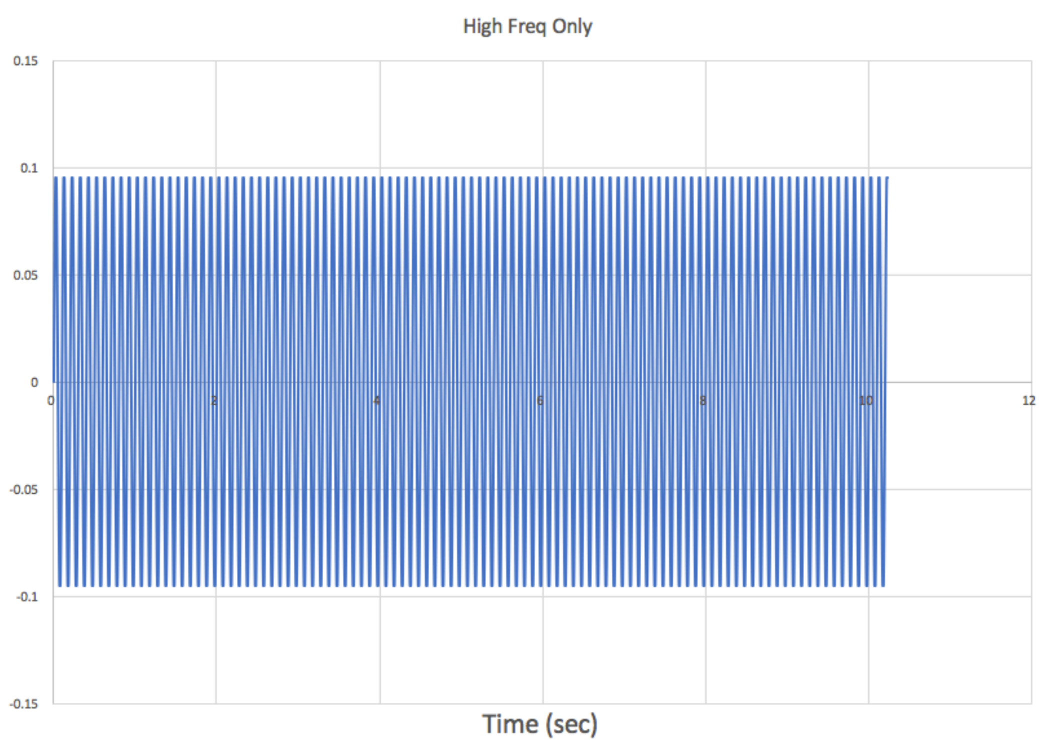

Figure 8: High frequency component from chirp_data.csv.

Summation of Components

The three signals y_1, y_2 and y_3 are summed to produce the overall signal y. This overall signal y is the only signal present in the file **chirp_data.csv**.

$$y = y_1 + y_2 + y_3$$

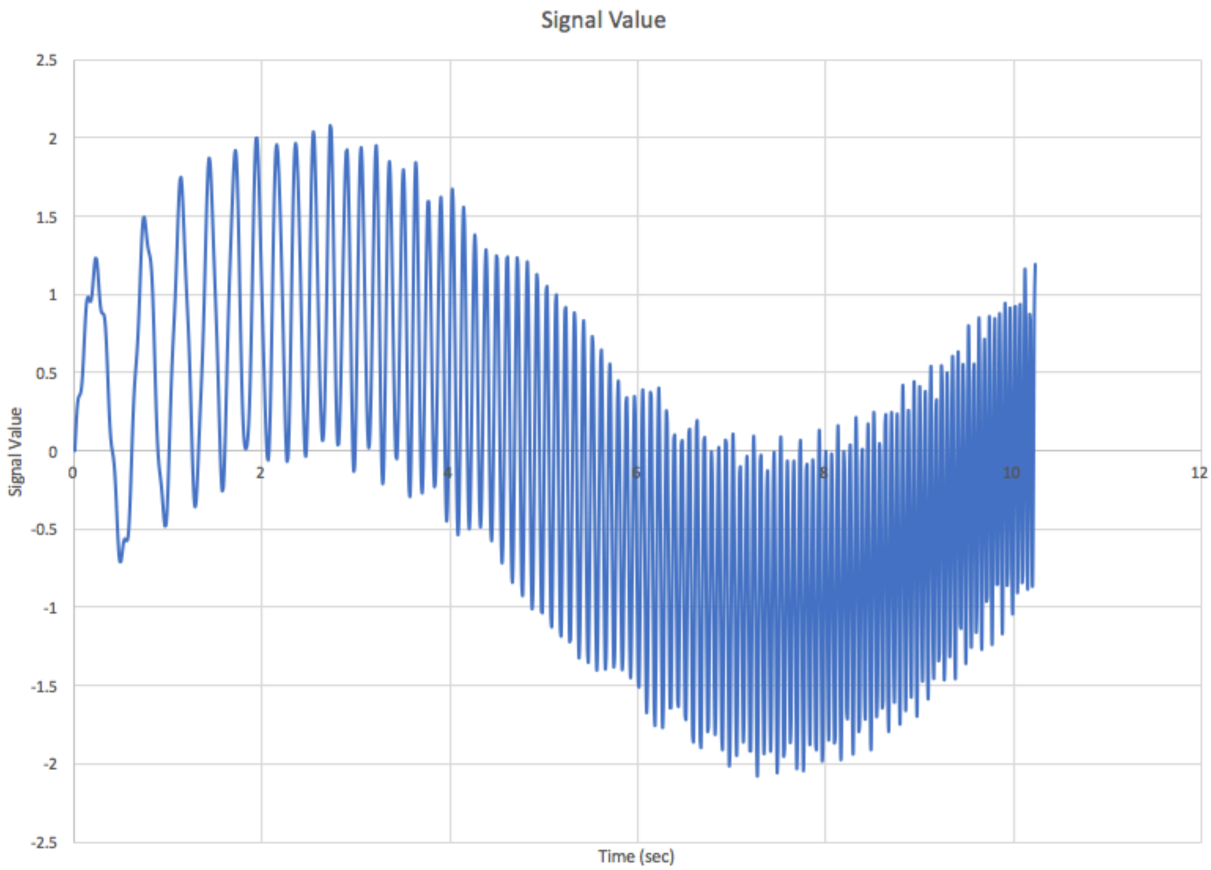

Figure 9: The signal found in chirp_data.csv.

Tasks

The goal of the tasks below is to extract the original chirp signal found in Figure 6. To enable this, users will be guided through using the liquid-dsp library to filter the signal found in **chirp_data.csv**.

1. Create a backup of the C-code source file.

   $ cp filter_chirp.c filter_backup.c

2. Modify the file **filter_chirp.c** as per the instructions below.

   NOTES:
   Do not modify anything before the comment "ONLY MODIFY THIS SECTION"
   Do not modify anything after the comment "DO NOT MODIFY ANYTHING PAST THIS COMMENT".

   Examine the appendix of this document for useful hints regarding the liquid-dsp library.

   Remove the low frequency noise ONLY from the signal stored in **y_orig**. Use a high-pass filter. Store this data into the array labelled **y_filt**.

   Name the output file **high_passed_output.csv**.

   Produce a plot of **y_filt_r** against **x**. Take a screenshot of this plot.

   Experiment with the **order** of the filter. What does increasing the order do? What does decreasing the order do? Produce plots of **y_filt_r** against **x** for three different values of the order.

   Experiment with the **corner** frequency of the filter. What does increasing the corner frequency to? What does decreasing the cutoff frequency do? Produce plots of **y_filt_r** against **x** for three different values of corner frequencies.

3. Modify the file **filter_chirp.c** as per the instructions below.

   NOTES:
   Do not modify anything before the comment "ONLY MODIFY THIS SECTION".
   Do not modify anything after the comment "DO NOT MODIFY ANYTHING PAST THIS COMMENT".

   Examine the appendix of this document for useful hints regarding the liquid-dsp library.

   Remove the high frequency noise ONLY from the signal stored in **y_orig**. Use a low-pass filter. Store this data into the array labelled **y_filt**.

Name the output file **low_passed_output.csv**.

Produce a plot of **y_filt_r** against **x**. Take a screenshot of this plot.

Experiment with the **order** of the filter. What does increasing the order do? What does decreasing the order do? Produce plots of **y_filt_r** against **x** for three different values of the order.

Experiment with the **corner frequency** of the filter. What does increasing the corner frequency to? What does decreasing the corner frequency do? Produce plots of **y_filt_r** against **x** for three different values of the corner frequencies.

4. Modify the file **filter_chirp.c** as per the instructions below.

   NOTES:
   Do not modify anything before the comment "ONLY MODIFY THIS SECTION".
   Do not modify anything after the comment "DO NOT MODIFY ANYTHING PAST THIS COMMENT".

   Examine the appendix of this document for useful hints regarding the liquid-dsp library.

   Remove the high frequency noise from the signal stored in **y_orig**. Use a low-pass filter. Store the result in **y_filt**. Users should select an appropriate corner frequency and order. Remove the low frequency noise from the signal stored in **y_filt**. Use a high-pass filter. Store the result in **y_filt**. Users should select an appropriate corner frequency and order.

   Name the output file **band_passed_output.csv**.

   Produce a plot of **y_filt_r** against **x**. Take a screenshot of this plot.

5. Experiment with the following parameters in order to extract the chirp waveform from the input data.

   - Low-pass filter corner frequency
   - High-pass filter corner frequency
   - Order

   Once the chirp waveform has been extracted from the original input signal, generate a plot of **y_filt_r** against **x**. Take a screenshot of this plot.

   Note: the waveform does not have to perfectly match the chirp signal shown in Figure 6. There may still be some low-frequency and high-frequency noise. This will help users develop an understanding that signal processing may never be able to recover an exact replica of the signal of interest.

   Record the parameters used for obtaining the chirp waveform.

Appendix

Function Prototype

```
// this function returns a filter object that can be used to filter signals.
iirfilt_cccf // return type
iirfilt_cccf_create_prototype( //function name
        liquid_iirdes_filtertype _ftype, // filter type (Butterworth | elliptic | etc)
        liquid_iirdes_bandtype _btype, // band type (highpass | lowpass | etc)
        liquid_iirdes_format _format, // filter format (SOS | TF)
        unsigned int _order, // filter order
        float _fc, // normalized cut-off frequency = Fc (Hz) / Fs (Hz)
        float _f0, // normalized center-frequency, unused for high-pass and low-pass filters
        float _Ap, // dB. pass-band ripple. Ignored for Butterworth filter.
        float _As // dB. top-band ripple. Ignored for Butterworth filter
        )
```

Example Filter Generation

```
//This section provides example code to show users how to build a filter object.

liquid_iirdes_filter ftype;
liquid_iirdes_filter btype_low, btype_high;
liquid_iirdes_filter format;
ftype = LIQUID_IIRDES_BUTTER; // use Butterworth for this tutorial
btype_low = LIQUID_IIRDES_LOWPASS;
btype_high = LIQUID_IIRDES_HIGHPASS;
format = LIQUID_IIRDES_SOS; // use SOS for this tutorial

float fc, Fc, Fs, F0, f0;
F0 = 10.0f; // Hz. This value is ignored for low-pass and high-pass filters.
Fc = 15.0f; // Hz. Corner frequency.
Fs = 45.0f; // Hz. Inspect the file chirp_data.csv to derive this value.
fc = Fc/Fs; // Normalized value. No units.
f0 = F0/Fs; // Normalized value. No units.

// both unused for the ftype LIQUID_IIRDES_BUTTERWORTH
// so just use the following values
float Ap, As;
Ap = 40.0f;
As = 0.1f;

unsigned int order;
// Experiment with this value as per the instructions in the section labelled Tasks.
order = 1 // No units.
```

```
// Example filter generation
iirfilt_cccf iir_filter_object;
iir_filter_object = iirfilt_cccf_create_prototype(ftype, btype_low, format, order, fc, f0, Ap, As);
```

Filter Usage

```
// This section demonstrates how to use a generated filter to process a signal.
float complex input_signal[1024], output_signal[1024];
int i;

// Generate a random signal to process
for(i=0; i<1024; i++) {
	input_signal[i] = randnf() + _Complex_I*randnf();
}

// Perform the filtering
// The function iirfilt_cccf_execute operates on a per-sample basis
// Take careful note, the sample for the output must be passed by reference
for(i=0; i<1024; i++) {
	iirfilt_cccf_execute(iir_filter_object, input_signal[i], &output_signal[i]);
}
```

Further examples

Open the links on a web-browser on a personal computer and inspect the code for additional documentation

https://github.com/jgaeddert/liquid-dsp/blob/master/examples/iirfilt_cccf_example.c
https://github.com/jgaeddert/liquid-dsp/tree/master/examples
https://github.com/jgaeddert/liquid-dsp