TDS02

Training Digital Signal Processing

Lab Work Handbook

Version 2.2b

J.W. Peltenburg
J.Z.M. Broeders

Version History

Date	Version	Description	Author
27-08-2019[1]	2.2b[2]	Updated for Code Composer Studio version 9. Corrected some errors.	BroJZ
16-10-2018	2.1a	Changed sample frequency for IIR filter from 48 kHz to 8 kHz.	BroJZ
15-05-2018	2.0	Adapted to CC3220 LAUNCHXL and CC3200AUDBOOST development boards.	BroJZ
17-11-2016	1.5	Removed bonus assignments.	BroJZ
26-08-2016	1.4	Added appendix about fixed point arithmetic.	BroJZ
25-09-2016	1.3	Fixed issues #6, #7, #8 and #9.	BroJZ
25-09-2016	1.2	Clarified some assignments.	BroJZ EijTJ
12-09-2016	1.1a	Fixed issues #1, #2, #3 and #5.	BroJZ
26-08-2016	1.1	Corrected some errors. Removed paragraph about Amplitude Modulation.	BroJZ

Continued on next page.

Date	Version	Description	Author
22-06-2015	1.0	First LaTeX version. • Converted to Code Composer Studio version 6. • Adapted to C5505 eZdsp development board. • Added a new assignment to explore architectural features of a DSP and some bonus assignments. • Replaced Von Hann window with Hamming window.	BroJZ
13-11-2013	0.7	Converted to Code Composer Studio version 5. Assuming tools are pre-installed.	PelJH
13-03-2012	0.6	Made union code multiple lines with indenting. Response of von Hann window image scale made consistent.	PelJH
15-06-2011	0.5	Some additions and clarifications to IIR text.	PelJH
01-06-2011	0.4	Minor corrections on student feedback. Added IIR filter structures and final assignment. Added bonus assignments.	PelJH
25-05-2011	0.3	Minor corrections on student feedback. Added IIR BLT Theory.	PelJH
18-05-2011	0.2	Added FIR Assignment.	PelJH

Continued on next page.

Date	Version	Description	Author
01-01-2011	0.1	Initial Version of the new Lab Handbook, large portions from old handbook by E.H.W. van de Logt. • All new assignments instead of the assignments from the old TDS book by Chassaing. • Some corrections and contributions by J.F. Theinert.	PelJH

[1] Dates are formated in the Gregorian way (dd-mm-$yyyy$).

[2] Explanation version coding $A.Bc$: A = major change, B = minor change, c = linguistic or mathematical corrections.

Contents

1

Introduction

Digital Signal Processing (DSP) is an important aspect in the field of Embedded Systems Engineering. For many years the huge interests and developments in the industry signify the importance of DSP techniques. Important applications of DSP can be found in consumer electronics, e.g. media boxes, hearing aids, synthesizers, sound-cards and especially mobile phones. In the industrial and research sector, DSP techniques are extensively used in motor and motion control, and in complex systems such as large sensor networks, machine vision systems, telecommunication systems, control plants and satellite arrays for astronomical purposes (such as the LOFAR[3] in 2012).

The course is meant for final year students of the minor Embedded Systems. In this course, a short introduction (or refreshment) to DSP theory will be given. If you want more background information or detailed information we recommend the books *Digital Signal Processing Using the ARM® Cortex®-M4* [12] and *Real-Time Digital Signal Processing: Fundamentals, Implementations and Applications* [10].

Electrical signals can be directly processed by analog components such as operational amplifier. It is not possible to directly use DSP in an analog environment. To

[3] http://www.lofar.org/.

enable DSP, the analog electrical input signals must first be sampled and digitalized by an Analog to Digital Converter (ADC). After processing the digital output signal can be transfered back to the analog domain by using a Digital to Analog Converter (DAC). A typical digital processing system, used in an analog environment, is shown in Figure 1.1. In which $x(t)$ is the analog input signal which is a function of the time t, $y(t)$ is the analog output signal, $x[n]$ is the digital, discrete input signal indexed by the sample number n, and $y[n]$ is the digital, discrete output signal.

Figure 1.1: Digital signal processing in an analog environment.

The advantages of DSP compared to analog signal processing are [10]:

- Flexibility. The behavior of a DSP system is mainly determined by its software. The behavior of analog systems, on the other hand, is entirely determined by its hardware. This makes digital systems much easier to adapt to changing functional requirements or to enhance their performance.
- Reliability. The characteristics of analog components change when the environment (e.g. the temperature) changes and also deteriorate with age. Therefore, the performance of analog signal processing systems will drift with changing environmental conditions and over time. The performance of DSP systems will not drift.
- Reproducibility. Due to the tolerances of analog components two identically produced analog signal processing units will not have completely the same characteristics. Therefore analog units often need fine-tuning before being taken into use. Two identically produced and programmed DSP units will always have exactly the same characteristics so fine-tuning is not needed.
- Complexity. Using digital processing, complex applications which are not possible with analog techniques are feasible. For example: face and speech recognition, data compression, MRI scanners, and radar tracking.

- Costs. Because many DSP systems can share the same hardware (the behavior is implemented in software), a DSP system almost always costs less than its analog counterpart.

The disadvantages of DSP compared to analog signal processing are:

- Bandwidth. DSP systems have a limited bandwidth determined by the sample rate. The bandwidth is limited to half of the sample frequency. This limit is called the Nyquist frequency or folding frequency. Analog signal processing systems have, in theory, unlimited bandwidth.
- Precision. DSP systems have a limited precision determined by the number of bits used for the ADC and DAC. Analog signal processing systems have, in theory, unlimited precision.

An example of a simple algorithm that can be implemented in the DSP block shown in Figure 1.1 is a so called Finite Impulse Response (FIR) filter. The equation for a Finite Impulse Response (FIR) filter is:

$$y[n] = \sum_{k=0}^{N} b_k \cdot x[n-k] \tag{1.1}$$

In which $y[n]$ represents the output sample with index n and $x[n-k]$ stands for the input sample with index $n-k$. The constant N is the so called order of the filter. The constants, so called coefficients of the filter, b_k determine the characteristics of the filter.

As can be seen in Equation (1.1) the calculation of $y[n]$ uses $N+1$ multiplications and N additions. The accumulation of the results of multiplications is a frequently used operation in many DSP algorithms. Also note that the calculation of $y[n]$ consist of a small loop. Small loops frequently occur in DSP algorithms. We will explore FIR filters and their implementation further in Chapter 3.

There are two types of DSP applications: non-real-time and real-time. For real-time systems the value of output sample $y[n]$ must be calculated before a certain deadline. In most real-time DSP systems, the output samples must be produced at the same rate as the input signal is sampled. Tools which run on a PC like

MATLAB can be used for non-real-time DSP. For real-time DSP we can use specific hardware (e.g. a digital signal processor). For relatively slow sample rates (e.g. audio applications) we can also use a modern generic processor. For example, in this course we will use an ARM® Cortex®-M4 MCU to implement our DSP (audio) algorithms. As we will discover in Section 3.4, MATLAB can also be used to design real-time DSP algorithms that will be executed on an embedded processor.

This course will consist mainly of working on practical assignments within the field of DSP. A DSP application development board is available for the lab. The goals of this course are mainly to teach the students to apply DSP algorithms in practice and to learn to work with the specialized hardware (i.e. a codec) that is available on the market today. The codec (coder-decoder) will be introduced in Section 2.1.1.

1.1 Purpose and Prerequisites

The purpose of this course is to teach you to:

- work with several important components of a DSP system,
- write simple C programs to implement a filter using an ARM processor and a codec,
- design filters in MATLAB and use them with your own C code,
- design, implement and test a FIR and IIR filter, and
- optimize your C code and exploit the specific features of a modern codec.

The prerequisites of this course are:

- know how to program, and
- know how to program microcontrollers.

1.2 Course Planning

The module Training Digital Signal Processing (TDS02) is awarded with 3 ECTS-credits. Passing this module will take about 80 working hours which consist of:

- 8 lab sessions of about 2.5 hours each = 20 hours total.
- about 60 hours of homework (prepare, write code, use MATLAB, write reports) (about 5.5 hours per week).

1.3 Document Organization

This document consists of several parts:

- Chapter 2 is an introduction to working with the course specific development boards.
- Chapter 3 will refresh your knowledge about FIR filters and windowing and introduce you to the MATLAB filter toolbox. It also contains the first of the assignment that counts for your grade.
- Chapter 4 will refresh your knowledge about IIR filter. It introduces several IIR filter structures that can be programmed. This chapter also contains the second assignment that counts for your grade.
- Chapter 5 will teach you how to profile your code and how to take advantage of the specific features provided by a modern codec to speed up your code. This chapter also contains the third, and last, assignment that counts for your grade.
- Appendix A introduces fixed-point arithmetic.

2

Preliminary Assignments

This chapter provides an introduction to working with the course specific DSP development boards and software development environment. It contains:

- An introduction to the most important components of the DSP development boards: ARM® Cortex®-M4 MCU processor and the coder-decoder (codec).
- A tutorial about working with the software development environment: Code Composer Studio.
- An assignment to generate an output signal with the DSP development boards.
- An assignment to capture an input signal with the DSP development boards.
- An assignment to recall some DSP basics and teach you how to create a time delay by using a buffer.

2.1 Assignment 0: Introduction to the CC3200AUDBOOST and CC3220 LAUNCHXL Boards

This lab work handbook uses the CC3200AUDBOOST Audio BoosterPack[4] [3], shown in Figure 2.1, in combination with the CC3220S LaunchPad development board[5] [5], shown in Figure 2.2.

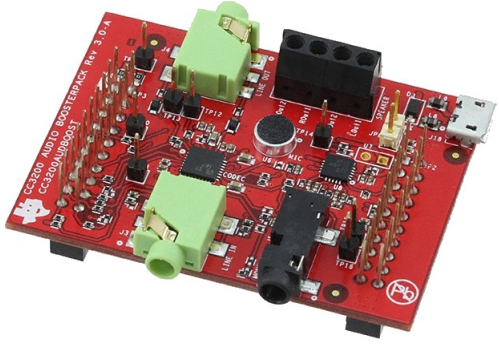

Figure 2.1: The CC3200AUDBOOST Audio BoosterPack.

The two most important components on these boards are the CC3220S SimpleLink™ Wi-Fi® Wireless Microcontroller Unit and the TLV320AIC3254 Codec (coder-decoder). The CC3220S System-on-Chip (SoC) [4, 6] is a single-chip with two separate execution environments: an user application dedicated ARM® Cortex®-M4 MCU and a network processor MCU. The TLV320AIC3254 [14, 15] is a 20-bit stereo audio codec with embedded miniDSP which can operate with a sample rate of up to 192 ksps.

2.1.1 TLV320AIC3254 Codec

The two most important components within a codec are the Analog to Digital Converter (ADC) and the Digital to Analog Converters (DAC). As can be seen in

[4] See: http://www.ti.com/tool/CC3200AUDBOOST.

[5] See: http://www.ti.com/tool/cc3220s-launchxl.

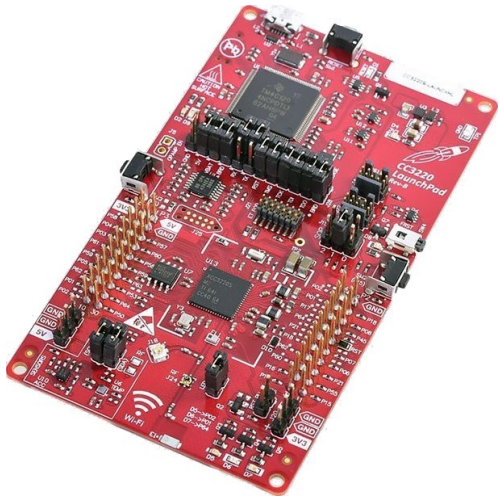

Figure 2.2: The CC3220 LAUNCHXL SimpleLink™ Wi-Fi® LaunchPad™ Development Kit.

Figure 2.3 the TLV320AIC3254 stereo codec includes not only two ADCs and two DACs but also includes several amplifiers and signal processing blocks.

As can be seen in the schematics of the CC3200AUDBOOST [2, Page 1] the stereo LINE IN input of the board are connected to the IN1L and IN1R inputs of the codec, and the HPL (HeadPhone Left) and HPR outputs of the codec are connected to the stereo LINE OUT output of the board.

The codec is connected to the CC3220 Launchpad through two serial buses: an I^2C bus[6] and an I^2S bus[7]. The I^2C bus is used to configure and control the codec and the I^2S bus is used to transfer the audio samples.

The codec contains an analog programmable gain amplifier before the ADC and a digital volume control after the DAC. Despite its name, this digital volume control

[6] I^2C stands for Inter-Integrated Circuit, more information can be found at: https://en.wikipedia.org/wiki/I%C2%B2C.

[7] I^2S stands for Inter-IC Sound, more information can be found at: https://en.wikipedia.org/wiki/I%C2%B2S.

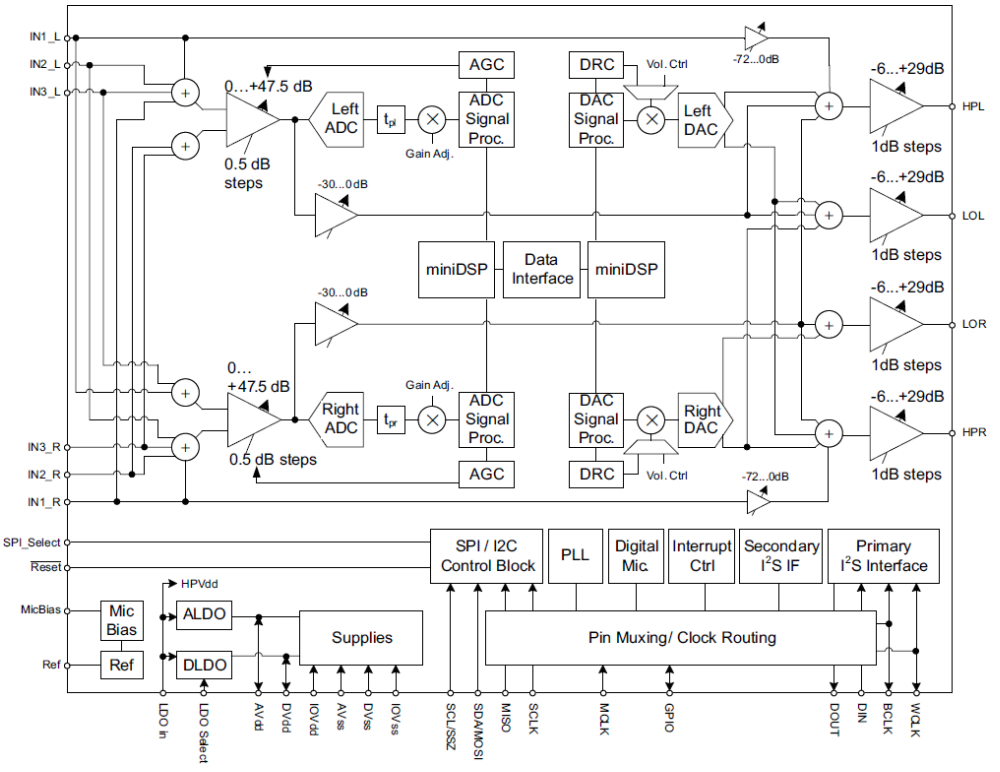

Figure 2.3: Simplified block diagram of the codec [14, Page 3].

is implemented as an analog amplifier with programmable gain. The amplification factors are specified in decibels (dB), as can be seen in Figure 2.3. This is a logarithmic unit which is frequently used in electrical engineering. The gain in dB (G_{dB}) of an amplifier can be calculated as follows:

$$G_{\text{dB}} = 20 \cdot \log_{10} \frac{V_{out}}{V_{in}} \tag{2.1}$$

In which V_{in} and V_{out} are the input respectively the output voltages of the amplifier.

Using a logarithmic scale for the amplification of audio signals makes sense because the sensitivity of the human ear to sound pressure works on a logarithmic scale too.

Besides the ADCs, DACs, and amplifiers the codec also contains:

- Two miniDSP cores. The first miniDSP core is tightly coupled to the ADC, the second miniDSP core is tightly coupled to the DAC. They support application-specific algorithms in the record and playback paths of the device. The miniDSP cores are fully software controlled. Target algorithms, like active noise cancellation, acoustic echo cancellation or advanced DSP filtering can be loaded into the device after power-up.
- ADC and DAC signal-processing blocks for filtering and effects. These processing blocks support different types of digital filtering.
- Automatic Gain Control (AGC). AGC can be used to maintain a nominally-constant output level.
- Dynamic Range Compression (DRC). DRC automatically adjusts the gain of the DAC to prevent hard clipping of peak signals.
- Beep generator. This generator can generate a sine wave signal.
- Digital Auto Mute. This feature switches of the output signal when the input is constant. This eliminates high-frequency noise during silent periods of music or speech.
- Headset Detection. The codec can determine which type of headset is plugged in.

The codec is a complicated digital signal processing component on its own and it's documentation [14, 15] can be quite overwhelming at first.

2.1.2 CC3220S SoC

The functional block diagram of the CC3220S SimpleLink™ Wi-Fi® Wireless and Internet-of-Things Solution, a Single-Chip Wireless MCU, is shown in Figure 2.4.

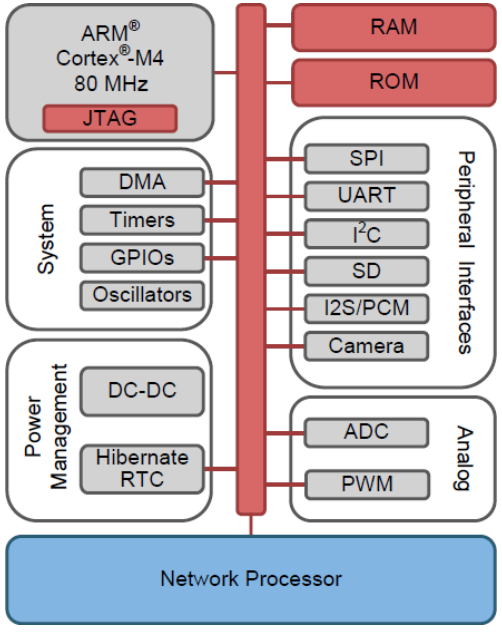

Figure 2.4: Functional block diagram of the CC3220S SoC [6, Page 4].

The CC3220S System-on-Chip (SoC) is a single-chip with two separate execution environments: an user application dedicated ARM® Cortex®-M4 MCU and a network processor MCU.

The ARM Cortex-M4 has no hardware support for floating-point calculations. Therefore, fixed-point calculations will be used to implement the DSP algorithms discussed in this Lab Work Handbook. Floating-point numbers use a constant number of significant bits (the significant) which are scaled by an exponent. The decimal number 1234.56789 can be encoded in decimal floating-point notation as 1.23456789×10^3 and also as 123456789×10^{-5}. As you can see the position of the decimal point can "float" within the number by adjusting the value of the exponent. In computing systems the IEEE754 standard [9] to represent real numbers is almost always used. This standard defines several formats for example single precision (which is used to implement the type `float` in the C programming language)

and double precision (which is used to implement the type **double** in C). Double precision numbers in the IEEE754 standard are 64 bits wide. One bit is used to determine the sign, 52 bits are used for the significant and 11 bits are used for the exponent[8]. The floating-point representation makes it possible to cover a wide, dynamic range of values with a constant number of significant bits. A double precision number has 52 significant bits which corresponds to about 16 significant decimal digits.

Fixed-point numbers use a fixed number of digits after and before the radix point. For example 12 bits before and 4 bits after the binary point. The format of a fixed-point binary number can be specified by using the Qn.m notation. In which n is the number of bits before the binary point (without the sign bit) and m is the number of bits after the binary point. For example, a number in Q0.15 format has one sign bit, zero bits before the binary point (a zero is implied) and 15 bits after the binary point. If the number of bits before the binary point is zero the Q format is sometimes abbreviated by omitting the n. For example Q0.15 can be abbreviated as Q15. Fixed-point numbers can be used to represent a limited range of values with a constant resolution. There is no direct support (build-in types) for fixed-point numbers in the C programming language. MATLAB, on the other hand, does support fixed point numbers. In Appendix A a small introduction into fixed-point arithmetic is given.

The disadvantage of using floating-point numbers is that a significant amount of hardware is needed to perform fast floating-point calculations. This hardware uses a significant amount of power. Fixed-point calculations, on the other hand, only need about the same amount of hardware as integer calculations do. Therefore, in embedded systems where price and or power usage must be minimized, fixed-point numbers are often preferred over floating-point numbers.

The ARM Cortex-M4 CPU has several specific features which enables it to execute digital signal algorithms fast [13]:

[8] See: https://en.wikipedia.org/wiki/Double-precision_floating-point_format.

- As mentioned in the introduction, DSP algorithms frequently use multiply-addition combinations. For example, in a FIR filter implementation input samples are multiplied by coefficients and added together. The Cortex-M4 has specific MAC (Multiply ACcumulate) instructions. For example, it is capable of a 16-bit x 16-bit multiplication and a 32-bit add in a single cycle.
- The Cortex-M4 also has a 16-bit SIMD vector processing unit. With this SIMD (Single Instruction Multiple Data) unit the Cortex-M4 can execute four 8-bit or two 16-bit calculations with only one instruction.
- In a general purpose CPU an overflow occurs when the result of an arithmetic operation on two signed numbers overflows the sign bit. For example, when the largest possible 16-bit signed value ($2^{15} - 1 = 32767 =$ 0x7FFF)[9] is incremented by one the result is 0x8000 $= -32768 = -2^{15}$. In a general purpose CPU this overflow is signaled by a flag in some status register, but it is the responsibility of the programmer to take appropriate actions. The Cortex-M4 has instructions which will signal an overflow but it also has an alternative set of instructions which are called saturating instructions. When these instructions are used, the output of a calculation is clipped to its maximum or minimum value when the sign bit overflows. For example, when the largest possible 16-bit signed value (32767 = 0x7FFF) is incremented by one in a saturating addition instruction the result is 0x7FFF = 32767. In many DSP algorithms saturation is the proper thing to do when the sign bit threatens to overflow. When this is the case the programmer can use the saturating instructions and does not has to check for overflows any more.

Besides these DSP specific features the Cortex-M4 also has features which will speed up the execution of generic algorithms such as pipelining and branch prediction [7].

To execute DSP algorithms even more efficiently Texas Instruments also provides processors which are specialized for this task; so called DSP's (Digital Signal Processors). The C5000 family[10] of DSPs is optimized for fixed-point calculations and

[9] The prefix 0x is used to denote hexadecimal notation.

[10] http://www.ti.com/processors/dsp/c5000-dsp/overview.html

is very energy efficient. They also offers a family of DSPs which are optimized for floating-point calculations: the C6000 family[11]. Other companies which produce DSPs are: Analog Devices[12], NXP Semiconductors[13], and others[14].

2.1.3 Electrostatic Discharge

Before we continue there is one very important thing to know: **The CC3220 LAUNCHXL and CC3200AUDBOOST development boards are sensitive to electrostatic discharge (ESD)!**

Figure 2.5: The CC3220 LAUNCHXL and CC3200AUDBOOST are sensitive to ESD.

Before you actually touch the board, observe the following precautions:

- Ground yourself by using a wrist-strap.
- Always use a shielded bag if you need to transport the board.

If you fail to comply with these precautions you can damage the board beyond repair.

[11] http://www.ti.com/processors/dsp/c6000-dsp/overview.html

[12] http://www.analog.com/en/products/processors-dsp/dsp.html

[13] https://www.nxp.com/products/processors-and-microcontrollers/additional-processors-and-mcus/digital-signal-processors:Digital-Signal-Processors

[14] https://en.wikipedia.org/wiki/Digital_signal_processor#Modern_DSPs

2.2 Assignment 1: Working with Code Composer Studio

In this assignment you will run and test a demo program on your CC3220 LAUNCHXL and CC3200AUDBOOST development boards. You need a source to produce an audio signal (preferable a signal generator) and a way to inspect the output (preferable an oscilloscope). Alternatively you can use your smartphone and a headset to test the demo program.

2.2.1 Installing Software and Configure Hardware

The website http://tds02.bitbucket.io/ explains how to:

- install the software needed for this course:
  - Tera Term. A terminal emulator is needed because a lot of TI's demo programs use a (virtual) terminal connection to report statuses and errors. Tera Term was chosen because it recognizes which (virtual) serial ports are available.
  - UniFlash. This program is used to program the flash memory on the CC3220 LAUNCHXL board.
  - Soundcard Oscilloscope. In the lab you can use a normal oscilloscope but at home you can use this program to generate and measure signals. You can also use this this program to generate frequency response graphs.
  - Code Composer Studio (CCS). This Integrated Development Environment (IDE) will be used to develop software for the CC3220S. This IDE is provided for free by Texas Instruments and is based on the Eclipse open source IDE which is widely used.
  - SimpleLink Software Development Kit (SDK). This SDK contains software libraries which make software development for the CC3220S more easy. It is also contains many example projects.

- reconfigure the CC3220 LAUNCHXL to not start a Wi-Fi access point when the board is powered up.
- recompile the driverlib (a part of the SimpleLink SDK). To properly debug programs which use the driverlib, it must be recompiled.
- connect the CC3200AUDBOOST Audio BoosterPack to the CC3220 LAUNCHXL board.

2.2.2 Running the Demo Program

Follow the description given at: http://tds02.bitbucket.io/ to run the demo program on your CC3220 LAUNCHXL and CC3200AUDBOOST boards.

Code Composer Studio is an Eclipse-based IDE. All Eclipse-base IDEs work from a certain workspace. In this tutorial the workspace directory `C:\workspace_v9\CC3220S` is used.

Please note that after completing an assignment or at the end of the lab, you have to move your complete workspace from the local `C:\` drive to your private network drive so that you will not lose your work when other students use the computer!

Be sure to move, and not copy, your workspace to prevent others from using your work without your consent. Whenever you start with the lab again, you can just copy back your workspace to `C:\`. You may also place your workspace on your private network drive `H:\` and let CCS work from there, but this might slow down the compilation process.

It is recommended to play a little bit with the demo program in the debug perspective. Try to add breakpoints and variables watches. If you think everything is working fine, call your instructor. If everything works as expected, your instructor will sign off assignment 1.

2.3 Assignment 2: Generating Output

For this assignment you need an oscilloscope to view the signal that is generated by the CC3200AUDBOOST board. If you do not have a oscilloscope available then you can use your PC using the program which can be found here: https://www.zeitnitz.eu/scms/scope_en.

2.3.1 Polling-based Output

The most straightforward method for sending samples to the codec is to use polling which is explained in this section.

Open Code Composer Studio and copy the demo project `line_in_2_line_out` to a new project called `audioSine1kHz`. Replace the code in the file `main_nortos.c` with the code from `audioSine1kHz/main_nortos.c` shown in Listing 2.1.

```
6  #include <stdint.h>
7  #include <stddef.h>
8  #include <stdio.h>
9  #include <NoRTOS.h>
10
11 #include <ti/devices/cc32xx/inc/hw_memmap.h>
12 #include <ti/devices/cc32xx/inc/hw_types.h>
13 #include <ti/devices/cc32xx/driverlib/prcm.h>
14 #include <ti/devices/cc32xx/driverlib/i2s.h>
15 #include <ti/drivers/I2C.h>
16
17 #include "Board.h"
18 #include "config.h"
19
20 // You can select the sample rate here
21 #define SAMPLINGFREQUENCY 48000
22 #if SAMPLINGFREQUENCY < 8000 || SAMPLINGFREQUENCY > 48000 ↩
    ↪ || SAMPLINGFREQUENCY % 4000 != 0
23 #error Sampling Frequency must be between 8 kHz and 48 kHz ↩
    ↪ (included) and must be a multiple of 4 kHz.
```

```
24 #endif
25
26 int main(void)
27 {
28     // Init CC3220S LAUNCHXL board.
29     Board_initGeneral();
30     // Prepare to use TI drivers without operating system
31     NoRTOS_start();
32
33     printf("1 kHz sine wave ==> Left HP LINE OUT.\n");
34
35     // Configure an I2C connection which is used to ↩
        ↪ configure the audio codec.
36     I2C_Handle i2cHandle = ConfigureI2C(Board_I2C0, ↩
        ↪ I2C_400kHz);
37     // Configure the audio codec.
38     ConfigureAudioCodec(i2cHandle, SAMPLINGFREQUENCY);
39
40     // Configure an I2S connection which is use to ↩
        ↪ send/receive samples to/from the codec.
41     ConfigureI2S(PRCM_I2S, I2S_BASE, SAMPLINGFREQUENCY);
42
43     /* Pre-generated sine wave data, 16-bit signed fixed ↩
        ↪ point samples Q0.15 */
44     int16_t sinetable[48] = {
45         0x0000, 0x10b4, 0x2120, 0x30fb, 0x3fff, 0x4dea,
46         0x5a81, 0x658b, 0x6ed8, 0x763f, 0x7ba1, 0x7ee5,
47         0x7ffd, 0x7ee5, 0x7ba1, 0x76ef, 0x6ed8, 0x658b,
48         0x5a81, 0x4dea, 0x3fff, 0x30fb, 0x2120, 0x10b4,
49         0x0000, 0xef4c, 0xdee0, 0xcf06, 0xc002, 0xb216,
50         0xa57f, 0x9a75, 0x9128, 0x89c1, 0x845f, 0x811b,
51         0x8002, 0x811b, 0x845f, 0x89c1, 0x9128, 0x9a76,
52         0xa57f, 0xb216, 0xc002, 0xcf06, 0xdee0, 0xef4c
53     };
54
55     int16_t sec, msec, sampleNum;
56     int16_t dataLeft;
```

```
57      size_t n = 0;
58
59      for (sec = 0; sec < 5 * 60; sec++) {
60          for (msec = 0; msec < 1000; msec++) {
61              for (sampleNum = 0; sampleNum < ↩
                 ↪ SAMPLINGFREQUENCY/1000; sampleNum++) {
62                  dataLeft = sinetable[n];
63                  I2SDataPut(I2S_BASE, I2S_DATA_LINE_0, ↩
                     ↪ (unsigned long)dataLeft);
64                  I2SDataPut(I2S_BASE, I2S_DATA_LINE_0, 0);
65                  n++;
66                  if (n == 48) {
67                      n = 0;
68                  }
69              }
70          }
71      }
72
73      printf("\n***Progam ended***\n");
74
75      return 0;
76  }
```

Listing 2.1: Program to generate a 1 kHz sine wave on the left audio output channel.

Connect the LINE OUT output of the CC3200AUDBOOST board to the oscilloscope, compile and run the program, and view the output signals on the scope.

The signal is quite noisy. You should select “HF Rejection” in the trigger menu to properly trigger and measure the signal. First press the “Autoset” button on the scope, then press the “MENU” button in the “TRIGGER” section of the scope. First select “Slope/Coupling” and then press “Rejection Off” twice to select “Rejection HF”. With this setting the scope will reject the high frequency noise in the signal when triggering. Your scope should display something similar to Figure 2.6. The colors are inverted in this figure to save some ink.

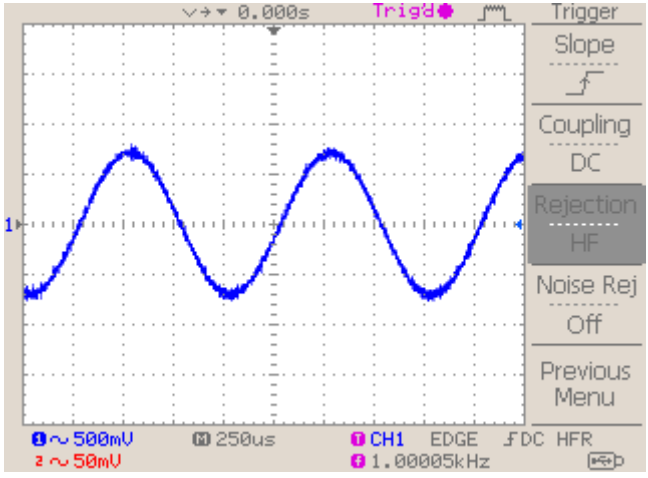

Figure 2.6: 1 kHz sine wave.

Samples are transfered from the CC3220S chip to the TLV320AIC3254 codec by using the I^2S bus[15].

Every time the function `I2SDataPut` [4, Page 382] is called, it will stay in some loop that waits for the codec to be ready to accept new data. The codec expects two new samples every sample time T_s. One sample for the left audio channel and one sample for the right audio channel. In this case the sample frequency f_s is 48 kHz so the sample time $T_s = 1/f_s$ is 20.833 µs. So the function `I2SDataPut` must be called every 10.417 µs. If there is any spare time between two calls to the function `I2SDataPut`, it keeps waiting until a new sample must be sent. When the codec is ready to receive a new sample the function `I2SDataPut` will actually send the sample. When the function `I2SDataPut` is called too late, a so called underflow error is generated by the I^2S hardware and the communication with the codec comes to a halt.

The program in Listing 2.1 repeatedly waits (inside the function `I2SDataPut`) until a new sample can be written to the codec. The implementation of the `I2SDataPut`

[15] More information about the I^2S bus can be found at `https://en.wikipedia.org/wiki/I%C2%B2S`

can be found in the driverlib[16]. The function repeatedly checks the XDATA bit in the XSTAT register [4, Page 423] to check is the I^2S controller is ready to send data. This repeatedly checking is called "polling". If our program has nothing else to do, polling is fine. But if we want our program to perform some other actions we must be very careful to provide the samples on time because when we do not provide a new sample on time the signal communication with the codec will come to a halt. Before we look into a different method to output samples, change the program to generate a square wave of 1 kHz on the right audio channel while keeping the 1 kHZ sine on the left audio channel. The top-top amplitude of the square wave should be equal to the top-top amplitude of the sine wave.

Change the sampling frequency to 8 kHz and explain to your instructor what happens. If everything works as expected, your instructor will sign off assignment 2a.

2.3.2 Interrupt-based Output

Another way to output (and input) samples is interrupt-based. This way, our processor will (instead of constantly waiting for the codec to demand a sample) jump to a dedicated piece of code called an interrupt routine, whenever the codec indicates that it wants a new sample. This has advantages over polling.

Here is a nice analogy. Suppose that you're following a lecture on DSP and, between every sentence, your lecturer will ask you and all of your class-mates: "Do you have a question?" Your lecturer is now working polling-based, spending a lot of time and effort in "polling" the students.

Instead of doing this, an agreement can be made that if the students have a question, they raise their hand so they can "interrupt" the lecturer to ask a question (after the lecturer finishes the current sentence). Then, the lecturer only has to answer when a question arises. Now, the lecturer is working interrupt-based, which is obviously much more efficient, since now the lecturer can keep talking when there are no questions.

[16] `C:\ti\simplelink_cc32xx_sdk_3_20_00_06\source\ti\devices\cc32xx\driverlib\i2s.h`

This is somewhat the same for the processor and the codec. When the codec has a question “Can I get a new sample?”, the processor finishes its current instruction, and then only has to give some sample to the codec in a brief moment. Then the processor can go back to its original task at hand.

To facilitate interrupt handling, we’ll make use of the driverlib [4, Page 383]. In Listing 2.2 a simple interrupt-based program is given. This program can be downloaded from `audioInterrupt/main_nortos.c`.

```
6  #include <stdint.h>
7  #include <stddef.h>
8  #include <stdio.h>
9  #include <NoRTOS.h>
10
11 #include <ti/devices/cc32xx/inc/hw_memmap.h>
12 #include <ti/devices/cc32xx/inc/hw_types.h>
13 #include <ti/devices/cc32xx/driverlib/prcm.h>
14 #include <ti/devices/cc32xx/driverlib/i2s.h>
15 #include <ti/drivers/I2C.h>
16
17 #include "Board.h"
18 #include "config.h"
19
20 // Only define MAXOUTPUT when signal is viewed on a scope ↩
    ↪ (to protect your ears).
21 //#define MAXOUTPUT
22
23 // You can select the sample rate here
24 #define SAMPLINGFREQUENCY 48000
25 #if SAMPLINGFREQUENCY < 8000 || SAMPLINGFREQUENCY > 48000 ↩
    ↪ || SAMPLINGFREQUENCY % 4000 != 0
26 #error Sampling Frequency must be between 8 kHz and 48 kHz ↩
    ↪ (included) and must be a multiple of 4 kHz.
27 #endif
28
```

```
29 // ISR that will be called when I2S is ready to send a sample to the codec.
30
31 void I2S_ISR(void)
32 {
33     static int n = 0;
34 #ifdef MAXOUTPUT
35     static unsigned long data = INT16_MAX;
36 #else
37     static unsigned long data = 100;
38 #endif
39     if (n % 2 == 0)
40     {
41         // write left channel
42         I2SDataPutNonBlocking(I2S_BASE, I2S_DATA_LINE_0, data >> 1);
43     }
44     else
45     {
46         // write right channel
47         I2SDataPutNonBlocking(I2S_BASE, I2S_DATA_LINE_0, -data);
48     }
49     n++;
50     if (n == 48) {
51         data = -data;
52         n = 0;
53     }
54     I2SIntClear(I2S_BASE, I2S_INT_XDATA);
55 }
56
57 int main(void)
58 {
59     // Init CC3220S LAUNCHXL board.
60     Board_initGeneral();
61     // Prepare to use TI drivers without operating system
62     NoRTOS_start();
```

```
63
64      printf("1 kHz sinewave ==> Left HP LINE OUT.\n");
65
66      // Configure an I2C connection which is used to ↩
         ↪ configure the audio codec.
67      I2C_Handle i2cHandle = ConfigureI2C(Board_I2C0, ↩
         ↪ I2C_400kHz);
68      // Configure the audio codec.
69      ConfigureAudioCodec(i2cHandle, SAMPLINGFREQUENCY);
70
71      // Configure an I2S connection which is use to ↩
         ↪ send/receive samples to/from the codec.
72      ConfigureI2S(PRCM_I2S, I2S_BASE, SAMPLINGFREQUENCY);
73
74      // Register I2S_ISR
75      I2SIntRegister(I2S_BASE, I2S_ISR);
76      // Enable interrupt to I2S_ISR when I2S is ready to ↩
         ↪ send a sample to the codec
77      I2SIntEnable(I2S_BASE, I2S_INT_XDATA);
78
79      while (1);
80
81      return 0;
82  }
```

Listing 2.2: A simple interrupt-based program.

As you can see on line 79 the `main` function of this program simply burns clock cycles in a **`while`** `(1)`-loop after initializing the codec and the interrupt. In this case, it is not useful to use an interrupt but in a real world application the `main` function can perform other tasks without worrying about the “feeding” of the codec. Normally, within this while loop there will be function calls to do all kinds of things the application has to do (for example communicate with some network device, or read data from storage). However, calculating and sending a new output sample now happens in the interrupt service routine (ISR) called `I2S_ISR()`. Whenever the codec needs a sample, it will interrupt the processor. The processor will jump

into the ISR, send data to the codec, calculate a new sample, and continue with the normal program. Keep in mind that interrupt service routines should be as small and quick as possible and should not contain any polling themselves, hence it might defeat the whole purpose of an interrupt.

The variables `n` and `data` which are defined inside the ISR, see Listing 2.2 line 33 to 38, are declared by using the **`static`** keyword. If we hadn't done this, these variables would have been freshly created each time the ISR is called. By declaring these variables as static local variables[17] their lifetime is extended to the time the program ends. Although, their scope is still local to the function in which they are declared.

On line 14 of Listing 2.2 the file `i2s.h` is included. This file declares the functions we can use to initialize the interrupt vector table and to enable the interrupts. The call to the function `I2SIntRegister` [4, Page 383] on line 75 defines the function to be called when the I^2S interrupt occurs. The call to `I2SIntEnable` on line 77 enables the `I2S_INT_XDATA` I^2S interrupt source.

Now, first predict the output on the left and right LINE OUT channels and then verify your predictions with the oscilloscope. Explain the output signals to your instructor. If everything works as expected, your instructor will sign off assignment 2b.

The code shown in Listing 2.2 can be used as a base program for all the other interrupt-based programs you will write during this course.

Write and test an interrupt-based program that outputs a sine on the left channel and a cosine on the right channel. Use the XY function of the oscilloscope to display a circle.

Make use of a the sine look-up table used in Listing 2.1. Note that you do not need a separate cosine look-up table. The sine and cosine should have a frequency of

[17] For more information about static local variables see: https://en.wikipedia.org/wiki/Static_variable.

1 kHz and the sample rate should be 48 kHz. The amplitudes should be as high as possible.

Show the result to your instructor. If everything works as expected, your instructor will sign off assignment 2c.

2.4 Assignment 3: Receiving Input

In Section 2.2.2 you already tested the demo project called `line_in_2_line_out` which simply copies the signal from the LINE IN input to the LINE OUT output. This demo program uses polling-based input.

2.4.1 Interrupt-based Input

Copy the project `line_in_2_line_out` to a new project called `line_in_2_line_out_interrupt` and modify the program to work interrupt-based. Use a sampling rate of 48 kHz.

You can find the names of the individual I^2S interrupt sources in Table 12-1 of the *CC3220 SimpleLink™ Wi-Fi® and Internet of Things Technical Reference Manual* [4, Page 385].

When your program is working, use a signal generator to apply a 1 kHz saw-tooth shaped signal with an amplitude of 1 V_{pp} to the left channel of the LINE IN input and verify that the signal on the LINE OUT output is similar. What is the delay between the input and output signals?

Show your program and the result to your instructor. If everything works as expected, your instructor will sign off assignment 3.

2.4.2 Audio Input

Connect an audio output (e.g. your smartphone) to the LINE IN input. Verify that you hear the audio signal from the input on the output by connecting headphones to the LINE OUT output.

2.5 Assignment 4: Delays

When making filters, we will need a buffer. Recall the formulas of the preliminary assignment.

One nice application of a buffer is an echo effect. For this assignment we will create this effect on the CC3220 LAUNCHXL and CC3200AUDBOOST boards. You should bring a headset to test it.

Suppose we have some circular buffer with `N` entries `buffer[N]`. A circular buffer is a buffer that, if we want to fill the buffer at some time `n`, we fill it at index `buffer[n mod N]`. This is a formal way of saying that we just let some variable count up with 1 for every sample which indicates the buffer location, and when the variable reaches the end of the buffer, we just reset that variable to 0 so it "circulates" around the buffer from the end back to the start.

We can use this to create a nice echo effect, see Figure 2.7 At time `n` we want to output the buffer value `buffer[n]` multiplied by some constant `c`, plus our current input sample. After we output this sample, we put the sample in buffer location `buffer[n]`. Now if `c == 1`, there will be an infinite echo which will make the signal louder and louder (don't try this, your ears will get hurt). If we make `c` smaller than 1, say, 0.75, then every time the buffer index passes that entry again, that original sample will become smaller (exponentially over time), and every time a new sample is added to it (which in turn becomes smaller every time after that as well).

Write and test a program that applies an echo effect on the audio input. Use a sample frequency of 48 kHz and choose `N` so that the first echo will appear after 0.5 s. Choose `c` to be 0.5 to start with. Also set `c` to 0.75 and observe the difference. The echo effect is best observed by using an audio fragment of spoken text.

Show the result to your instructor. If everything works as expected, your instructor will sign off assignment 4.

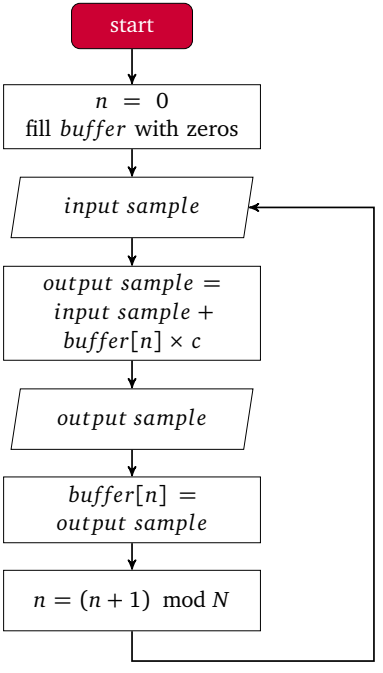

Figure 2.7: Flow diagram to create a simple echo effect.

3

FIR Filters

In this chapter we will focus on designing and implementing a Finite Impulse Response (FIR) filter. The formula for a FIR filter is:

$$y[n] = \sum_{k=0}^{N} b_k \cdot x[n-k] \tag{1.1}$$

Where $y[n]$ are the output samples, b_k are the filter coefficients, $x[n]$ are the input samples, and N is the order of our filter.

3.1 Determination of the Coefficients

Usually we want to filter signals in the time domain, because signals are a function of time in the real world and not of frequency. However, when we speak about filters we often define their response in the frequency domain. We can use some math to transform our filter back to the time domain.

We will now show how this is done. This is a summary of what you may have learned in the DSP01 course (see [11, Chapter 5]).

The Inverse Discrete-Time Fourier Transform (IDTFT) is given by:

$$x[n] = T_s \int_{\frac{-1}{2T_s}}^{\frac{1}{2T_s}} X(f) \cdot e^{j2\pi n f T_s} df \tag{3.1}$$

Where $X(f)$ is the spectrum of our signal, f is the frequency, and T_s is the period of our discrete-time steps which is $\frac{1}{f_s}$, where f_s is the sample frequency.

This is an integral transformation which transforms a signal in the frequency domain ($X(f)$) to the continuous-valued discrete-time domain ($x[n]$).

Since the Discrete-Time Fourier Transform (DTFT) is a linear transformation, we may say convoluting two signals in the time domain is the same as multiplying their spectra in the frequency domain. Therefore we can easily design the frequency response in the frequency domain and then transform it back to the time domain so we can implement it in, for example, a microcontroller.

Let's look at the frequency response magnitudes of a low-pass filter:

$$\left|H_{lp}(f)\right| = \begin{cases} 1 & \frac{-f_s}{a} \leq f \leq \frac{f_s}{a},\ a \leq 2 \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

Note that a must be smaller than 2 because half the sample rate equals the Nyquist-Frequency. The function is shown in Figure 3.1.

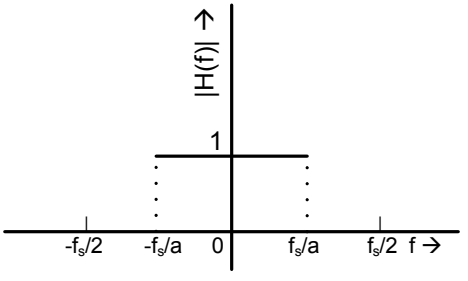

Figure 3.1: Low-pass filter frequency response.

Also recall that the function is symmetric around the origin of the graph due to the complex conjugate properties of the Fourier Transform of some function. We could also have drawn this figure from 0 to $\frac{f_s}{2}$, but this would make the integral we have to solve, for the inverse transform, a bit more cumbersome.

We can transform the above function of frequency $H(f)$ back to a function of time using the IDTFT. Note that f_c is the cut-off frequency of the filter, $a = \frac{f_s}{f_c}$. So for example, if we have a sample rate of 8000 Hz and we want our cut-off frequency to be 1000 Hz, we take $a = 8$.

The IDTFT of a low-pass filter now becomes (substituting $T_s = \frac{1}{f_s}$ and changing the integral limits to confirm with the frequency response of the low-pass filter, since only at that interval our function $H(f) = 1$ and it is 0 anywhere else):

$$h_{lp}[n] = \frac{1}{f_s} \int_{\frac{-f_s}{a}}^{\frac{f_s}{a}} e^{j2\pi n \frac{f}{f_s}} df \tag{3.3}$$

Then by solving the integral we find:

$$h_{lp}[n] = \frac{1}{f_s j2\pi n \frac{1}{f_s}} \left[e^{j2\pi n \frac{f}{f_s}} \right]_{\frac{-f_s}{a}}^{\frac{f_s}{a}} \tag{3.4}$$

Substituting the limits gives:

$$h_{lp}[n] = \frac{1}{j2\pi n} \left[e^{\frac{j2\pi n}{a}} - e^{\frac{-j2\pi n}{a}} \right] \tag{3.5}$$

By using Euler's Formula $e^{\alpha j} = \cos(\alpha) + j\sin(\alpha)$ we find:

$$h_{lp}[n] = \frac{1}{j2\pi n} \left[\cos\left(\frac{2\pi n}{a}\right) + j\sin\left(\frac{2\pi n}{a}\right) - \cos\left(\frac{-2\pi n}{a}\right) - j\sin\left(\frac{-2\pi n}{a}\right) \right] \tag{3.6}$$

Which simplifies to:

$$h_{lp}[n] = \frac{1}{j2\pi n} 2j \sin\left(\frac{2\pi n}{a}\right) \tag{3.7}$$

Further simplification gives:

$$h_{lp}[n] = \frac{\sin\left(\frac{2}{a}\pi n\right)}{\pi n} = \frac{2}{a}\text{sinc}\left(\frac{2n}{a}\right) \tag{3.8}$$

The function $\text{sinc}(n) = \frac{\sin(n\pi)}{n\pi}$ is called the normalized cardinal sine function and is widely used in DSP techniques.

We can now determine any $h_{lp}[n]$ with this formula, except for the non-trivial case of $n = 0$, because dividing by zero is not possible. We can determine the value of $h_{lp}[0]$ by calculating the limit of $h_{lp}[n]$ for $n \to 0$. Because the limit of $n \to 0$ for both the nominator and denominator are zero, L'Hôpital's rule can be applied:

$$h_{lp}[0] = \lim_{n\to 0} \frac{\sin\left(\frac{2}{a}\pi n\right)}{\pi n} = \lim_{n\to 0} \frac{\frac{d\left(\sin\left(\frac{2}{a}\pi n\right)\right)}{dn}}{\frac{d(\pi n)}{dn}} = \lim_{n\to 0} \frac{\frac{2}{a}\pi \cdot \cos\left(\frac{2}{a}\pi n\right)}{\pi} = \frac{2}{a} \tag{3.9}$$

In a similar way we can derive the responses of high-pass, band-pass and band-stop filters as well:

High-pass:

$$h_{hp}[n] = \frac{-\sin\left(\frac{2}{a}\pi n\right)}{\pi n} \tag{3.10}$$

with $H_{hp}(f) = 1$ for $|f| \geq \frac{f_s}{a}$, 0 otherwise.

Band-pass:

$$h_{bp}[n] = \frac{\sin\left(\frac{2}{b}\pi n\right) - \sin\left(\frac{2}{a}\pi n\right)}{\pi n} \tag{3.11}$$

with $H_{bp}(f) = 1$ for $\frac{f_s}{a} \leq |f| \leq \frac{f_s}{b}$, 0 otherwise.

Band-stop:

$$h_{bs}[n] = \frac{\sin\left(\frac{2}{a}\pi n\right) - \sin\left(\frac{2}{b}\pi n\right)}{\pi n} \tag{3.12}$$

with $H_{bs}(f) = 0$ for $\frac{f_s}{a} \leq |f| \leq \frac{f_s}{b}$, 1 otherwise.

3.2 Example

For example, in some digital system with a sample frequency of 8 kHz, we might want to make a low-pass filter with pass-band 0 to 1000 Hz and stop-band 1000 Hz to $F_s/2$ Hz.

For our cut-off frequency of 1000 Hz, first we calculate a, which is:

$$a = \frac{f_s}{f_c} = \frac{8000}{1000} = 8 \tag{3.13}$$

Now we can calculate our first coefficient:

$$h_{lp}[0] = \frac{2}{a} = \frac{1}{4} = 0.25 \tag{3.14}$$

Now for $n \neq 0$:

$$h_{lp}[n] = \frac{\sin\left(\frac{\pi n}{4}\right)}{\pi n} \tag{3.15}$$

The first 10 coefficients on both sides of $n = 0$ of this filter are shown in Table 3.1 and Figure 3.2.

Table 3.1: Coefficients.

n	$h[n]$
0	0.250
1, -1	0.225
2, -2	0.159
3, -3	0.075
4, -4	0.000
5, -5	-0.045
6, -6	-0.053
7, -7	-0.032
8, -8	0,000
9, -9	0.025
10, -10	0.032

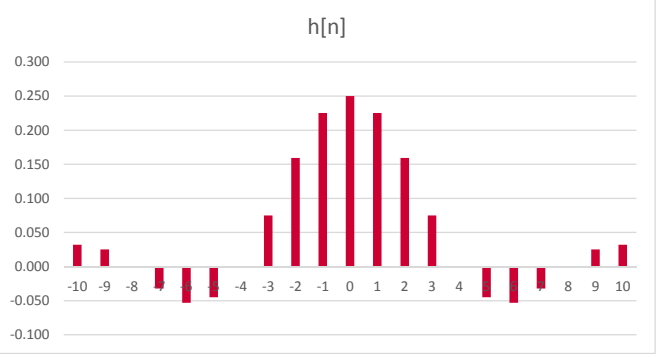

Figure 3.2: Discrete-time transfer function of a low-pass filter.

Note that we should fill in any integer for n, not just -10 to 10. If the number of coefficients gets very large, the delay will be very long. Even so, if we wish to fully replicate the ideal filter response we've used in the previous example, we need to let n go from $-\infty$ to $+\infty$. This would give an infinite delay, so our ideally filtered signal will never appear on the output while the universe lasts, not to mention that we need infinite memory in our DSP system.

3.3 Windowing

Because we cannot allow a real filter to have an infinite number of coefficients, we will need to limit the response of our filter. In the above case, where we cut off all $|n| \geq 11$, we can see what result this has on the frequency response of our filter if we transform it back to the frequency domain (using the DTFT). Because this

is a lot of work, we will do this in MATLAB. We can use the `freqz()` function in MATLAB to calculate the frequency response, see `lpfvbfreqresp.m`. The result is shown in Figure 3.3.

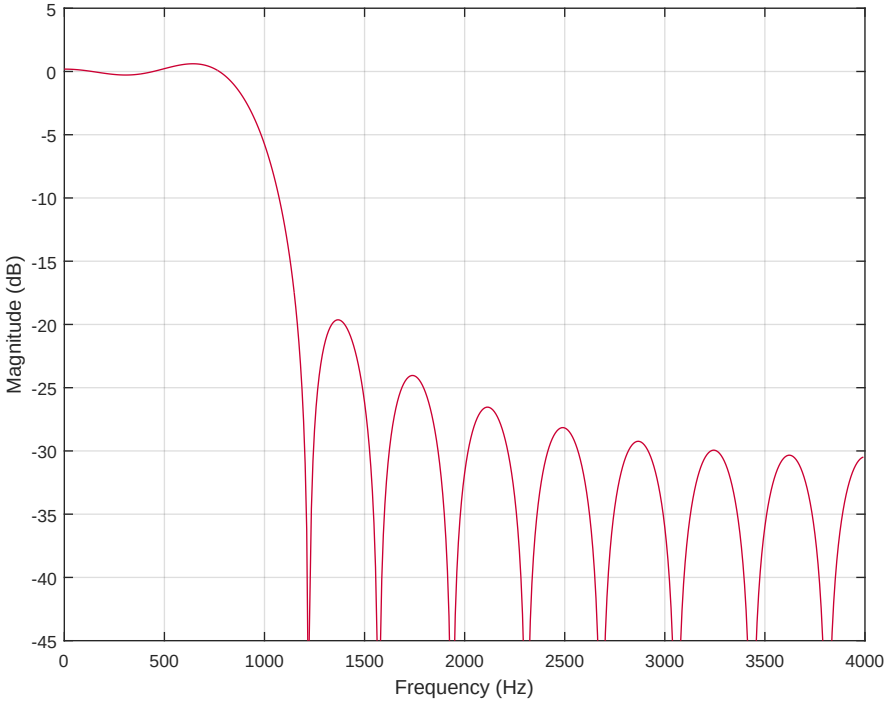

Figure 3.3: Frequency response of an abruptly ended transfer function.

We can now see that our frequency response is not ideal anymore and that so-called side-lobes are introduced where the desired frequency response should be 0. Also the steepness of the filter is not as good as in the ideal case since we have a limited number of coefficients. This is mainly due to the abrupt ending of the desired discrete-time representation of our transfer function $h[n]$.

As you must recall from the DSP01 course we can let the coefficients slowly but more steadily approach to zero near the edges of our "window" (the part of the filter we're interested in), by somehow scaling the coefficients a bit using a method

called *windowing*. This way, the abrupt ending of coefficients (which results in the occurrence of, among other things, the side-lobes) will be somewhat compensated for. This is at the cost of our filter to be less like the ideal filter.

Recall the formula for the output of a non-recursive filter:

$$y[n] = \sum_{k=0}^{N} b_k \cdot x[n-k] \tag{1.1}$$

Note that since our transfer function $h[n]$ (and the window functions which we will discuss later) are non-causal (they depend on values of the future), when implementing the filters, we shift the transfer function $h[n]$ backward in time so that each coefficient $b_k = h\left[k - \frac{N}{2}\right]$ (assuming that N is even). For more information, see [11, page 151].

We can expand this formula by taking the windowing function into account:

$$y[n] = \sum_{k=0}^{N} w_k \cdot b_k \cdot x[n-k] \tag{3.16}$$

Where w_k are the coefficients of our window.

For example, we can take a simple window called the Hamming window (recall DSP01). The formula for a Hamming window is:

$$w[n] = 0.54 + 0.46 \cos\left(\frac{2\pi n}{N-1}\right) \tag{3.17}$$

Where N is the order of the filter.

The Hamming window is one of the most commonly used windows [10]. The window itself is shown in Figure 3.4.

When we apply this window to our desired, but abruptly ended discrete-time transfer function we get the discrete-time transfer function shown in Figure 3.5.

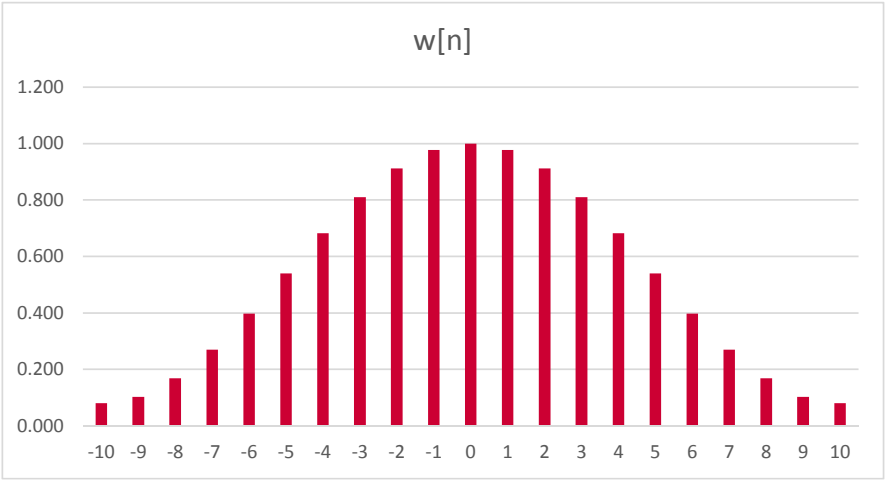

Figure 3.4: The Hamming window.

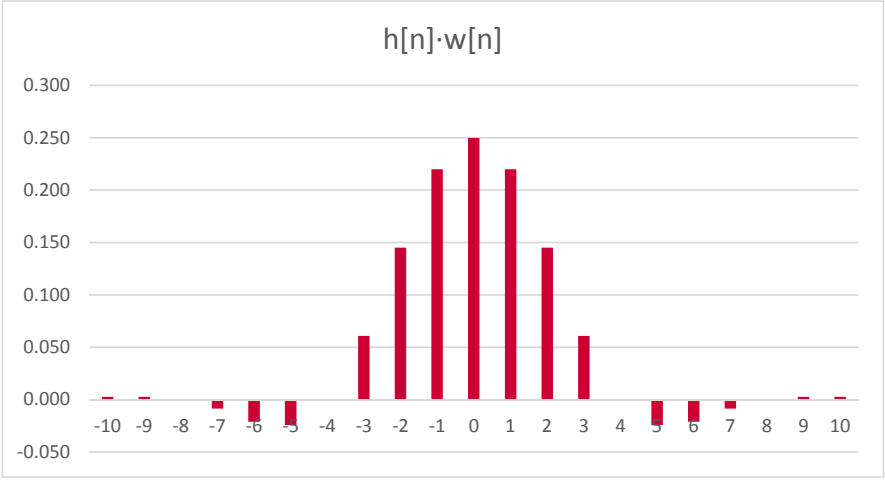

Figure 3.5: Discrete-time transfer function adjusted by the Hamming window.

Compare Figure 3.5 to Figure 3.2. Note that the coefficients and the edge of the function slowly decrease to 0, therefore avoiding the abrupt ending of our transfer function, and thus reducing unwanted effects such as side-lobes, etc. For

comparison, see Figure 3.6 which shows the frequency response of the filter with and without the application of the Hamming window. The frequency response graph shown in red is that of our original filter with the abrupt ending of coefficients (also called a rectangular window), and the graph shown in blue is that of our filter with the Hamming window applied.

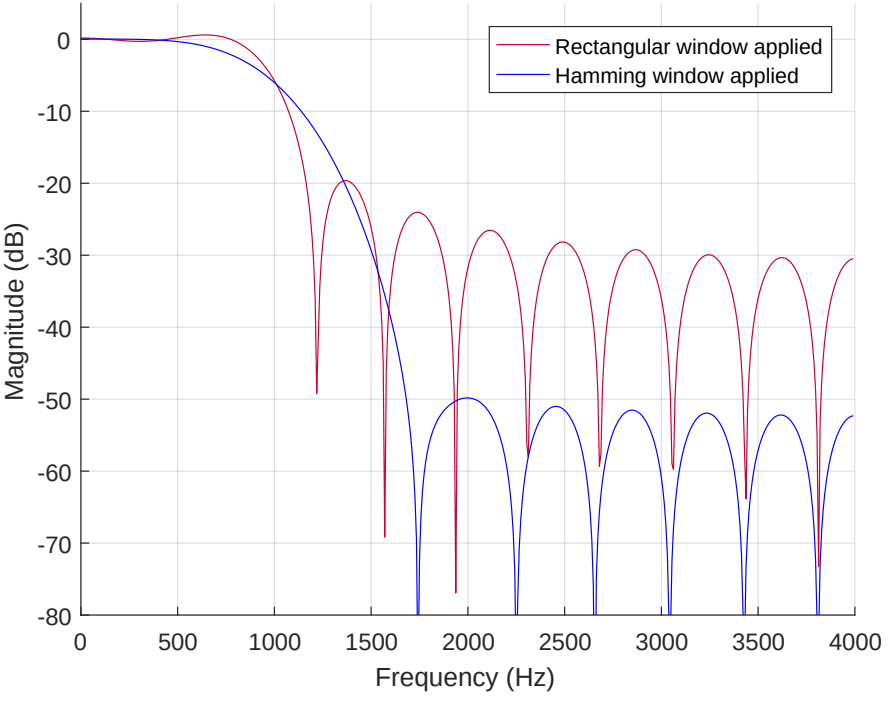

Figure 3.6: The effect of the application of the Hamming window on the frequency response (blue) compared to applying a rectangular window (red).

Although the steepness of the filter is decreased, the side-lobes of our Hamming windowed filter are smaller (note that the first side-lobe for the Hamming windowed filter has a maximum magnitude of -50 dB, and for the rectangular window this maximum magnitude is -20 dB).

There are many other window types, and the Hamming window is certainly not one of the best windows. For your assignment you may use any other window type

as long as you give arguments for choosing a certain type and discuss its properties in your report.

3.4 MATLAB Filter Designer

We have already shown that it takes a lot of work to derive the coefficients, not to mention to analyse what happens with different windows. We can use a special tool in MATLAB to do this easier and faster. We will give an example of the same filter we specified above.

Start MATLAB and type in `filterDesigner` (previously known as FDA Tool). The window shown in Figure 3.7 will open.

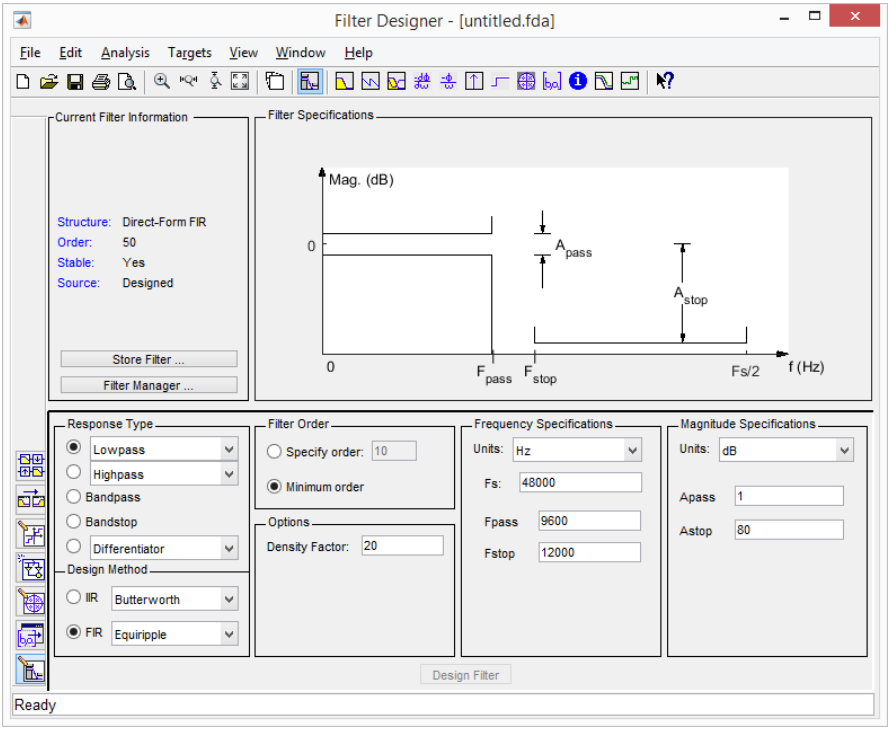

Figure 3.7: The Filter Designer main window.

In MATLAB's Filter Designer we can design filters more easily than when we have to calculate all the values by hand. We will give a short overview on the different sections of this window.

- **Current Filter Information:**
  Here we can see what type of filter implementation we're aiming for (we will discuss these types later). Also the order of the filter can be seen. We can see whether the filter we've designed is stable (always stable for FIR filters) as well. For IIR filters, if you want to change the structure, you can right click on this section to change this option.
- **Response Type:**
  Here we can select the different response types. We will only use low-pass, high-pass, band-pass and band-stop. Also the design method can be selected. For now we will use the FIR method with windowing, but note there are many other ways to design a filter. In Section 4.5 we will design an IIR filter.
- **Filter Order:**
  Here we can specify the order, or we can specify to let MATLAB determine the minimum needed order of our filter based on the other specifications.
- **Options:**
  Here we can select different options and specify parameters for different filter types, windows types, etc.
- **Frequency Specifications:**
  Here we can specify the frequency properties of our filter, like the sample rate, pass frequencies and stop frequencies.
- **Magnitude Specifications:**
  Here we can select the magnitude properties of our filter, like the magnitudes in the pass- and stop-band.
- **Filter Specifications:**
  This section gives a graphical overview of the selected filter and the designed filter. Basically this gives us the frequency response, but we can also show the phase response if we want.

Now use the tool to design a FIR low-pass filter of order 20, using a rectangular window. The cut-off frequency should be 1 kHz and the sample rate should be 8 kHz. Make sure to deselect the "Scale Passband" option in the Options section after you've selected the windowed method. When you push the "Design Filter" you should see the same result as shown in Figure 3.8.

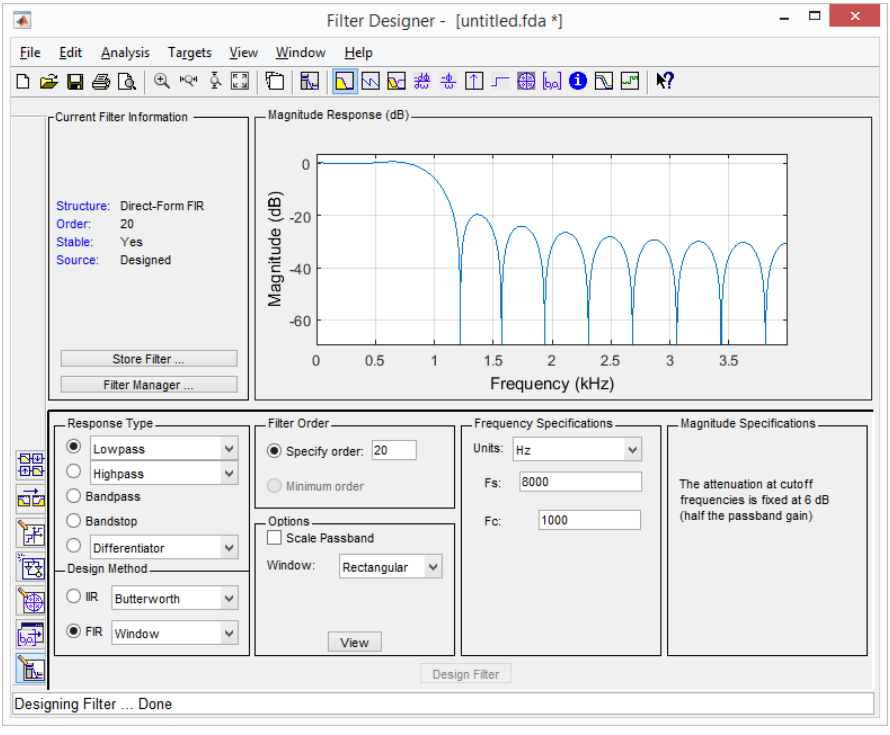

Figure 3.8: 1 kHz low-pass filter in MATLAB's Filter Designer.

Now we are interested in the coefficients of the filter. In the menu, click "Analysis" and then "Filter Coefficients". Verify that they are the same as the coefficients we presented earlier in Table 3.1. We can see the benefit of using a tool like MATLAB's Filter Designer now, since we won't have to calculate all the values by hand.

We can even export the coefficients to a C header that we can use in our programs. In the menu, select “Targets”, “Generate C header...”. Now a new dialog is shown, see Figure 3.9.

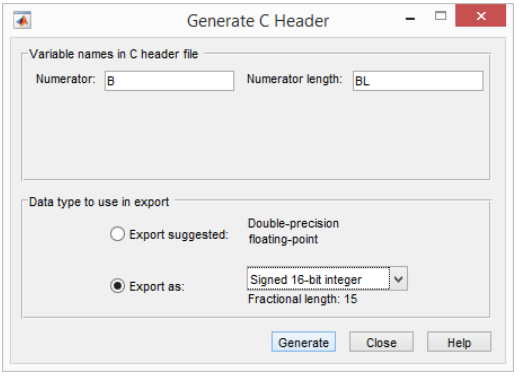

Figure 3.9: Generate C header dialog.

The numerator is the name your coefficients will get later on; the numerator length is a variable which represents the order of your filter + 1. Because we use a Cortex-M4 without floating-point support to implement the filter, we export the coefficients as signed 16-bit integers.

If we open the exported file, we can see the filter coefficients at the bottom, and the variable representing the order + 1 of our filter. Note that the order + 1 is the number of coefficients, and thus the size of our buffer which we will have to use to store delayed samples later on.

The lines that are actually useful are the last 6 ones, which define the variable named `BL` which is initialized with the number of coefficients[18] and the array named `B` which is initialized with the values of the coefficients. Since we will not make use of the header file `tmwtypes.h` from MATLAB, remove everything except those last lines and modify the code so the file `fdacoefs.h` looks like the one shown in Listing 3.1. The variable `BL` is replaced by a define so we can use this identifier to declare the size of the array `B`. We have used the type `int16_t`

[18] Note that the number of coefficients is one more than the order of the filter.

to define the array B. The type int16_t is defined in the standard C include file stdint.h[19].

```
#define BL 21
const int16_t B[BL] = {
     1043,   819,     0, -1054, -1738, -1475,     0,  2458,
     5215,  7375,  8192,  7375,  5215,  2458,     0, -1475,
    -1738, -1054,     0,   819,  1043
};
```

Listing 3.1: 16-bit signed integer coefficients generated by MATLAB.

The **const** keyword actually means that we cannot change the values stored in the array B during run-time.

Note that our coefficients have been scaled now from floating-point numbers in MATLAB, to signed two's complement fixed-point integers. The 16-bit numbers generated by MATLAB have 15 fractional bits and one sign bit. This fixed-point format is often referred to as Q0.15.

For example, the center coefficient B[10] is equal to 8192. This value is obtained by multiplying the floating point value 0.25 by the largest signed 16-bit value + 1 which is 2^{15}.

To make the filter causal, the coefficients have been moved by order/2 samples to the right, since we cannot grab samples in the future. Now the delay is a bit longer but the filter output over a longer time will be the same. As before we define the coefficients $b_k = h\left[k - \frac{N}{2}\right]$ (assuming that N is even) for $k = 0 \ldots N$. So now coefficient b_0 (or B[0] in the C code) is equal to $h[-10]$. The center coefficient $h[0]$ is now referred to as B[10] in the C code.

Now we know how we can calculate the coefficients, a next assignment is given.

[19] stdint.h is a header file in the C standard library introduced in the C99 standard library to allow programmers to write more portable code by providing a set of typedefs that specify exact-width integer types. See: http://en.cppreference.com/w/c/types/integer.

3.5 Assignment 5: Finite Impulse Response Filter

In Figure 3.10 we can see the flow diagram to implement the calculation of a new FIR filter output sample. What is done, basically, is a buffer of size `BL` is filled with a new sample, where `BL` is the number of coefficients and the size of our buffer (so the order of the filter N is `BL-1`). Now, the new sample is calculated by using the formula for a FIR filter:

$$y[n] = \sum_{k=0}^{N} b_k \cdot x[n-k] \tag{1.1}$$

Note that N is the order of the filter here. Suppose we have a filter of order $N = 2$, then our current output $y[n]$ is:

$$y[n] = b_0 \cdot x[n] + b_1 \cdot x[n-1] + b_2 \cdot x[n-2] \tag{3.18}$$

We change this so that the current sample $x[n]$ is always stored in `buffer[0]`, and mirroring time because we will implement this in C, and we cannot use negative offsets in arrays to store previous values. So a delayed sample in the buffer of time $n-5$, will be stored in `buffer[5]` instead of `buffer[-5]`. So to calculate the current output $y[n]$, called `output` in the C code, we may now write:

```
output = B[0]*buffer[0] + B[1]*buffer[1] + ↩
 ↪ B[2]*buffer[2];
```

This is what is done in Figure 3.10 as well. Note that `buffer[k]` in the C code corresponds to $x[n-k]$ in Equation (1.1) and that `B[k]` corresponds to the coefficient b_k in Equation (1.1).

After we've calculated the new output sample, we shift all the entries in the buffer to create the delay on each sample. This is also shown in Figure 3.10. When we're done, we can send the output sample to the codec.

Here is your assignment: Implement a C program that executes a 1 kHz LP FIR Filter with a sample rate of 8 kHz. Use the coefficients from Listing 3.1.

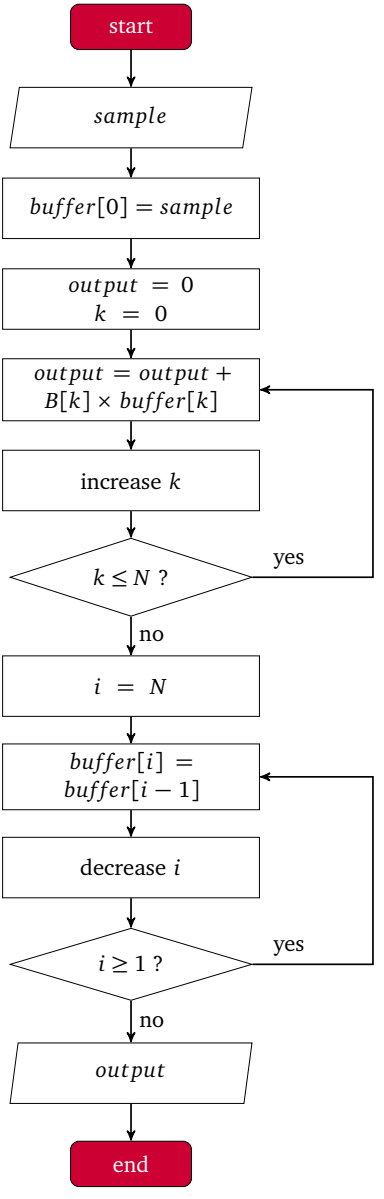

Figure 3.10: Flow diagram for the calculation of a new FIR filter output sample.

After the implementation is complete, put different frequencies on the input and verify the output to correspond with the designed filter in MATLAB using an oscilloscope. You are advised to use the Soundcard Oscilloscope program[20] on a PC to generate a frequency response graph. The best result is obtained by using a frequency sweep input signal generated by a signal generator.

Show the result to your instructor. If everything works as expected, your instructor will sign off assignment 5a.

In the last loop of the flowchart shown in Figure 3.10 each sample in the buffer is moved one place towards the end of the buffer. By using a circular buffer we can make the filter implementation somewhat faster. If we use a circular buffer we just keep track of the position of the oldest sample in the buffer and override this value with the new input sample at the start of the flowchart. Now, change your program to use a circular buffer[21] and show the result to your instructor. Note that you also have to change the code in the first loop shown in Figure 3.10 because the most recent sample is no longer located at index 0 in the buffer. If everything works as expected, your instructor will sign off assignment 5b.

When your code works, you will get a new filter specification from your instructor. Create new coefficients using MATLAB's Filter Designer yourself and write a report about this assignment. The guidelines for the report can be found in the course wiki.

When your new filter is implemented and has the proper characteristics, show the result to your instructor. If everything works as expected, your instructor will sign off assignment 5c.

[20] https://www.zeitnitz.eu/scms/scope_en

[21] More information about circular buffers can be found at https://en.wikipedia.org/wiki/Circular_buffer.

4

IIR Filters

In this chapter we will focus on designing and implementing an Infinite Impulse Response (IIR) filter. These filters are also called recursive filters. A FIR filter, which was introduced in Chapter 3, uses a certain number of preceding input samples to calculate the current output sample. An IIR filter not only uses preceding input samples, but it also uses a certain number of previous output samples. Thus, the formula for an IIR filter is:

$$y[n] = \sum_{k=0}^{N} b_k \cdot x[n-k] - \sum_{i=1}^{M} a_i \cdot y[n-i] \tag{4.1}$$

Or:

$$y[n] = -a_1 \cdot y[n-1] - a_2 \cdot y[n-2] - \cdots - a_M \cdot y[n-M] + b_0 \cdot x[n] + b_1 \cdot x[n-1] + \cdots + b_N \cdot x[n-N] \tag{4.2}$$

The output of an IIR filter depends not only on the current and past inputs, but also on the previous outputs (hence it is recursive).

4.1 Determination of the Coefficients

Digital recursive filters (which we often specify in the z-domain) are relatively young compared to analogue recursive filters (which we often specify in the s-domain). Formulas for analogue filters are well known to designers. These formulas often form the basis for digital recursive filter design as well, since there are methods to transform a formula in the s-domain into a formula in the z-domain. Using such a method, the properties of the filter in the time-domain and frequency-domain are, approximately, preserved. One popular method is called the Bilinear Transform (BLT). We will give a short recap on the BLT (as seen in DSP01) and show a simple example.

The BLT is:

$$s \approx \frac{2}{T_s} \cdot \frac{z-1}{z+1} \tag{4.3}$$

Where T_s is the sample period which is $\frac{1}{f_s}$ where f_s is the sample frequency.

Also, recall that the frequency response of a continuous-time filter can be determined by evaluating the transfer function $H(s)$ at:

$$s = j\omega_c \tag{4.4}$$

Likewise, the frequency response of a discrete-time filter can be determined by evaluating the transfer function $H(z)$ at:

$$z = e^{j\omega_d} \tag{4.5}$$

Where ω_c are the continuous-time domain frequencies and ω_d are the discrete-time domain frequencies.

Since the s-domain analyses a system for $t \to \infty$, but the z-domain only concerns periodic signals, it is interesting to see the relation between ω_c and ω_d.

We can find this by using the BLT:

$$j\omega_c \approx \frac{2}{T_s} \cdot \frac{e^{j\omega_d} - 1}{e^{j\omega_d} + 1} \tag{4.6}$$

This simplifies to:

$$\omega_c \approx \frac{2}{T_s} \tan\left(\frac{\omega_d T_s}{2}\right) \tag{4.7}$$

See [11, page 203] or https://en.wikipedia.org/wiki/Bilinear_transform#Frequency_warping.

Note that this is not a linear relation. If you design a filter in the s-domain and convert it to a filter in the z-domain, especially at the edges of the spectrum of the discrete-time filter, the response will be "warped". This is due to the tan function defined in the relation. This effect is called frequency warping. Before you apply the Z-transform, any frequencies used in the s-domain transfer functions should therefore first be "pre-warped".

Now we have all the tools needed to transform a function of s into a function of z.

4.2 Example of a Simple Recursive Low-Pass Filter

Suppose we want to make an IIR filter with sample rate of 8000 Hz similar to a low-pass RC filter with a cut-off frequency of 1000 Hz. Recall the transfer function of a simple low-pass RC filter:

$$H(s) = \frac{1}{1 + \frac{s}{\omega_c}} = \frac{\omega_c}{\omega_c + s} = \frac{\omega_c}{s + \omega_c} \tag{4.8}$$

We can create this filter in MATLAB using: `hs = tf(1000*2*pi, [1 1000*2*pi])` were `tf` stands for transfer function. We can plot its frequency and phase response

using the bodeplot command as has been done in the program lpfbodelog.m. The result of this program is shown in Figure 4.1.

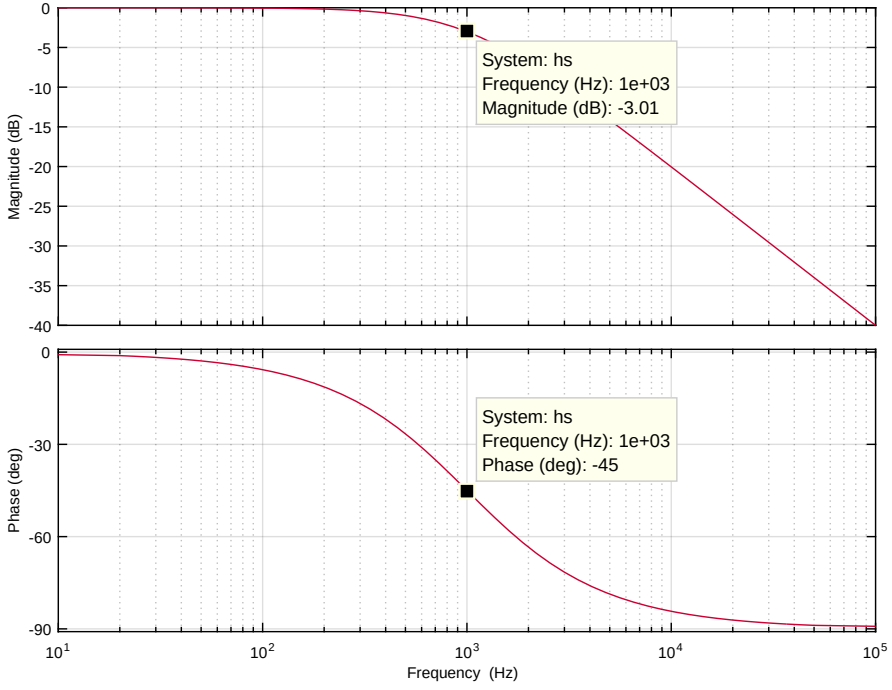

Figure 4.1: Bode plot of a low-pass filter with a cut-off frequency of 1000 Hz drawn with a logarithmic frequency scale.

The filter shown in Figure 4.1 is clearly a low-pass filter with a cut-off frequency of 1000 Hz. As you may know[22] the amplification at the cut-off frequency should be $20\log(1/\sqrt{2}) \approx -3.01$ dB and the phase at the cut-off frequency should be $-45°$. Both facts can be verified in Figure 4.1.

Note that the frequency is plotted with a logarithmic scale as is custom for Bode plots. Also note that the frequency magnitude response of this filter will go to minus infinity as the frequency goes to infinity.

[22] See: https://en.wikipedia.org/wiki/Low-pass_filter

Because we are interested in the frequencies from 0 to 8000 Hz the Bode plot is drawn again in Figure 4.1 for this frequency range with the program `lpfbodelin.m`. This time a linear frequency scale is used.

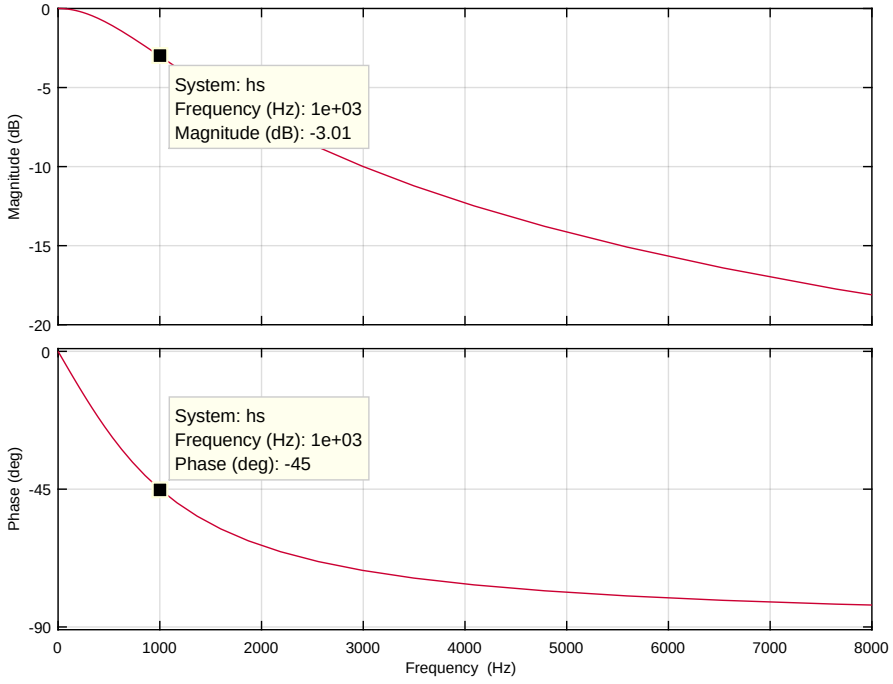

Figure 4.2: Bode plot of a low-pass filter with a cut-off frequency of 1000 Hz drawn with a linear frequency scale.

Now we can apply the BLT to the analog version of the transfer function to make it discrete.

Again:

$$H(s) = \frac{\omega_c}{s + \omega_c} \tag{4.8}$$

Applying the BLT:

$$H(z) \approx \frac{\omega_c}{\frac{2}{T_s}\cdot\frac{z-1}{z+1}+\omega_c} = \frac{\omega_c T_s(z+1)}{2(z-1)+\omega_c T_s(z+1)} = \frac{\omega_c T_s z+\omega_c T_s}{2z-2+\omega_c T_s z+\omega_c T_s}$$

$$= \frac{\omega_c T_s z+\omega_c T_s}{(\omega_c T_s+2)z+\omega_c T_s-2} = \frac{\frac{\omega_c T_s}{\omega_c T_s+2}\cdot z+\frac{\omega_c T_s}{\omega_c T_s+2}}{z+\frac{\omega_c T_s-2}{\omega_c T_s+2}} \tag{4.9}$$

By taking $\omega_c = 1000\cdot 2\pi$ and $T_s = \frac{1}{8000}$ we find:

$$H(z) \approx \frac{0.2820z+0.2820}{z-0.4361} \tag{4.10}$$

We can also do this in MATLAB quickly:

```
>> hs = tf(1000*2*pi, [1 1000*2*pi])

hs =
     6283
  --------
  s + 6283

Continuous-time transfer function.

>> hz = c2d(hs, 1/8000, 'tustin')

hz =
  0.282 z + 0.282
  ---------------
    z - 0.4361

Sample time: 0.000125 seconds
Discrete-time transfer function.
```

Now we can verify if the responses are the same using the program `lpfzbodelin.m`. The result is shown in Figure 4.3. Note that the magnitude scale is extremely large (−400 dB represents an attenuation of 10^{20}).

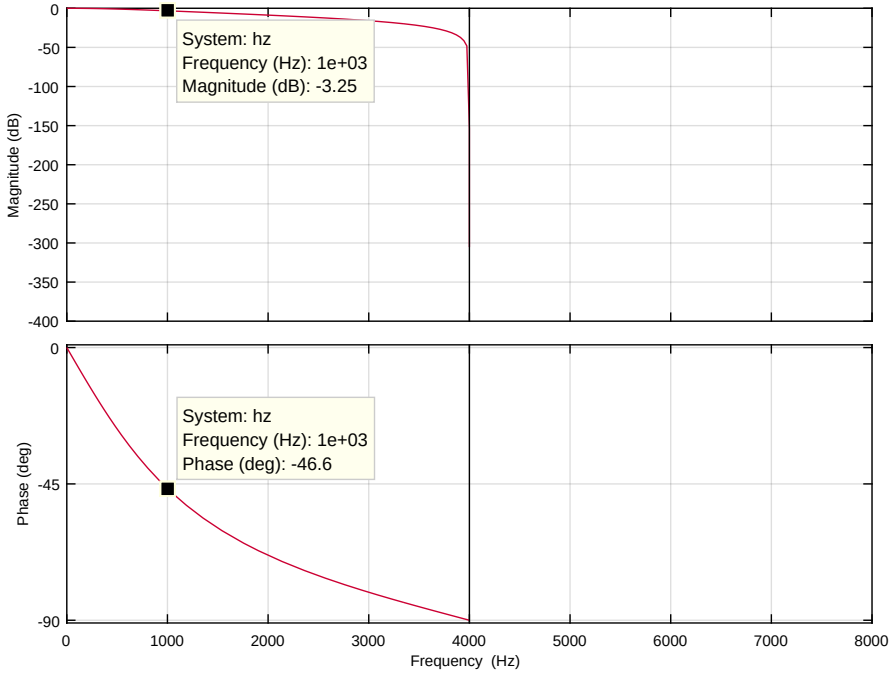

Figure 4.3: Bode plot of a discrete-time low-pass filter with a cut-off frequency of 1000 Hz and a sample frequency of 8000 Hz.

We can see several interesting properties of the BLT transform in Figure 4.3. Due to the frequency warping, the whole frequency response of the analogue filter, for which the frequency goes to infinity, is now "compressed" into the frequency response of the digital filter for which the response only goes to $f_s/2$. The non-linear characteristic of the tangent is seen here. Also, due to this warping effect, the cut-off frequency has somewhat shifted. As we can see, the amplification at the intended cut-off frequency is −3.25 dB but should be −3.01 dB. The phase at the intended cut-off frequency is −46.6° but should be −45°. If we want to correct this, we have to apply the pre-warping.

We know that:

$$\omega_c \approx \frac{2}{T_s}\tan\left(\frac{\omega_d T_s}{2}\right) \tag{4.7}$$

Now, if our desired cut-off frequency in the discrete-time version of our filter $\omega_d = 2\pi \cdot 1000$, considering our sample rate of 8000 Hz, then in the analogue domain, the pre-warped frequency is:

$$\omega_{c_{prewarped}} \approx \frac{2}{\frac{1}{8000}}\tan\left(\frac{2\pi \cdot 1000 \cdot \frac{1}{8000}}{2}\right) = 16000\tan\left(\pi \cdot \frac{1000}{8000}\right) \approx 6627\,\text{rad/s} \tag{4.11}$$

If we use this pre-warped frequency in our derived formula for the low-pass filter, we find:

$$H(z) \approx \frac{\frac{\omega_{c_{prewarped}} T_s}{\omega_{c_{prewarped}} T_s + 2} \cdot z + \frac{\omega_{c_{prewarped}} T_s}{\omega_{c_{prewarped}} T_s + 2}}{z + \frac{\omega_{c_{prewarped}} T_s - 2}{\omega_{c_{prewarped}} T_s + 2}} \approx \frac{0.2929 \cdot z + 0.2929}{z - 0.4142} \tag{4.12}$$

We can do this in MATLAB even quicker:

```
>> hs = tf(1000*2*pi, [1 1000*2*pi])

hs =
     6283
  --------
  s + 6283

Continuous-time transfer function.
```

```
>> hz = c2d(hs, 1/8000, 'prewarp', 1000*2*pi)

hz =
  0.2929 z + 0.2929
  -----------------
     z - 0.4142

Sample time: 0.000125 seconds
Discrete-time transfer function.
```

Now if we look at the Bode plot, shown in Figure 4.4, we see that the new filter has the correct amplitude and phase response at the cut-off frequency.

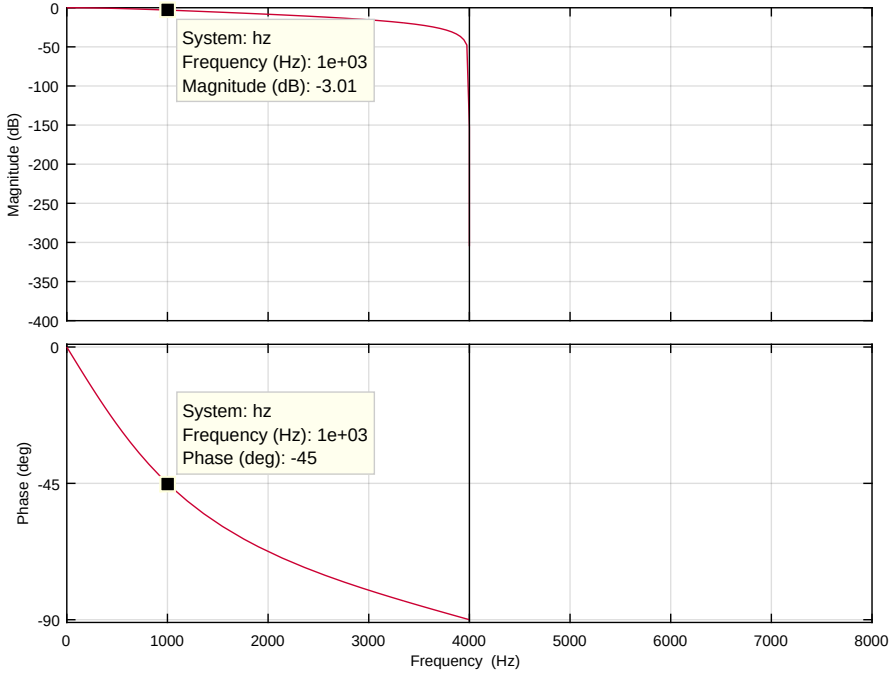

Figure 4.4: Bode plot of a discrete-time low-pass filter with a cut-off frequency of 1000 Hz and a sample frequency of 8000 Hz.

In Figure 4.5 the Bode plot of the analog ($H(s)$) and digital ($H(z)$) filters are shown.

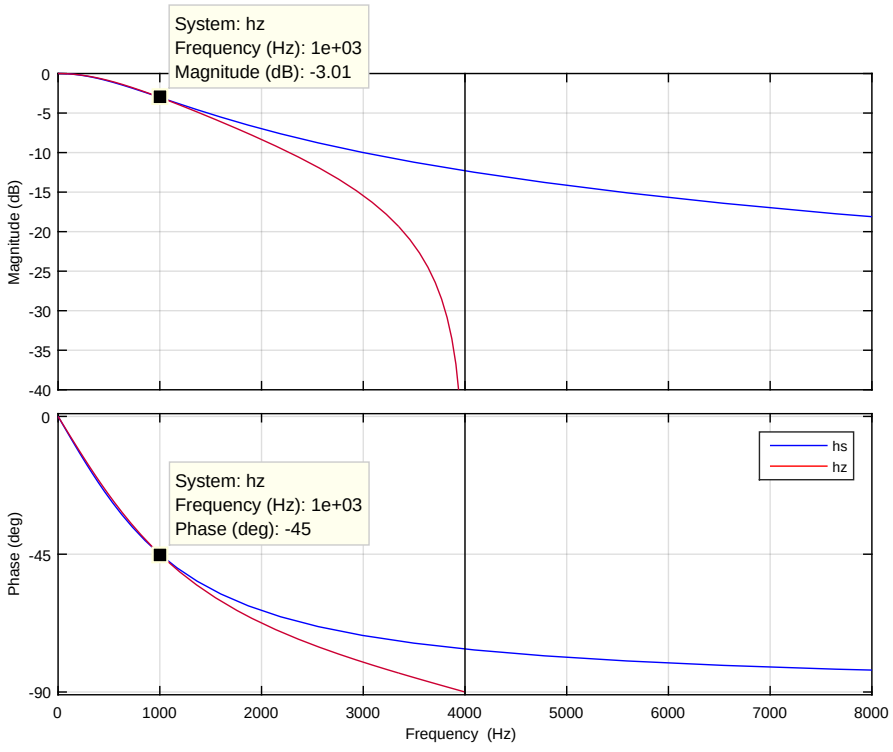

Figure 4.5: Bode plots for $H(s)$ and $H(z)$.

Now that we have the right response, we can easily calculate the coefficients for the filter. Since:

$$H(z) = \frac{Y(z)}{X(z)} \approx \frac{\frac{\omega_{c_{prewarped}} T_s}{\omega_{c_{prewarped}} T_s + 2} \cdot z + \frac{\omega_{c_{prewarped}} T_s}{\omega_{c_{prewarped}} T_s + 2}}{z + \frac{\omega_{c_{prewarped}} T_s - 2}{\omega_{c_{prewarped}} T_s + 2}} \tag{4.13}$$

We can now solve $Y(z)$.

For readability we substitute $u = \frac{\omega_{c_{prewarped}} T_s - 2}{\omega_{c_{prewarped}} T_s + 2}$ and $w = \frac{\omega_{c_{prewarped}} T_s}{\omega_{c_{prewarped}} T_s + 2}$.

$$Y(z) \cdot z + u \cdot Y(z) = w \cdot X(z) \cdot z + w \cdot X(z) \tag{4.14}$$

Now we make the filter causal by multiplying both sides with z^{-1}:

$$Y(z) + u \cdot Y(z) \cdot z^{-1} = w \cdot X(z) + w \cdot X(z) \cdot z^{-1} \tag{4.15}$$

Thus:

$$Y(z) = -u \cdot Y(z) \cdot z^{-1} + w \cdot X(z) + w \cdot X(z) \cdot z^{-1} \tag{4.16}$$

Now we can convert this to the time domain:

$$y[n] = -u \cdot y[n-1] + w \cdot x[n] + w \cdot x[n-1] \tag{4.17}$$

If we substitute back u and w:

$$y[n] = \frac{-\omega_{c_{prewarped}} T_s - 2}{\omega_{c_{prewarped}} T_s + 2} \cdot y[n-1] + \frac{\omega_{c_{prewarped}} T_s}{\omega_{c_{prewarped}} T_s + 2} \cdot x[n] + \frac{\omega_{c_{prewarped}} T_s}{\omega_{c_{prewarped}} T_s + 2} \cdot x[n-1] \tag{4.18}$$

We find the coefficients for an implementable version of the recursive filter.

The design method described so far comes in handy when we want to design a filter dynamically in software. For example when we do not now the cut-off frequency which is required beforehand. If we now the requirements of the filter beforehand it is much easier to use the MATLAB Filter Design and Analysis tool.

4.3 MATLAB's Filter Designer

Calculating the recursive filter coefficients by hand takes a long time. Therefore we will use MATLAB's Filter Designer to calculate the coefficients for our Infinite Impulse Response (IIR) filters.

IIR filters can have many different implementations. Most implementations (or structures) can be selected in the Filter Designer. Your assignment will be to implement one of these structures.

4.4 Filter Structures

The simplest form is the *Direct-Form I Single Sections* structure. If you right click in the "Current Filter Information" section of the Filter Designer design window, you can convert the structure of the filter, as can be seen in Figure 4.6. By default, the structure for an IIR filter is: "Direct-Form II, Second Order Sections".

The structures of the filter can be explored if you click Show Filter Structure. Now, create a simple filter, convert its structure to "Direct-Form I", and convert it to "Single Section". Select "Show Filter Structure" from the pop-up menu shown in Figure 4.6 and verify that the structure which is shown, which is duplicated in Figure 4.7, directly implements the formula for a recursive filter:

$$y[n] = \sum_{k=0}^{N} b_k \cdot x[n-k] - \sum_{i=1}^{M} a_i \cdot y[n-i] \tag{4.1}$$

Note that even though MATLAB will generate coefficient a_1, this coefficient is often 1, and signifies the total output gain of the filter. The direct-form I structure is the most straightforward implementation. Also note that MATLAB begins the index of an array with 1, and not with 0 as in C.

Another filter structure is the direct-form II structure shown in Figure 4.8.

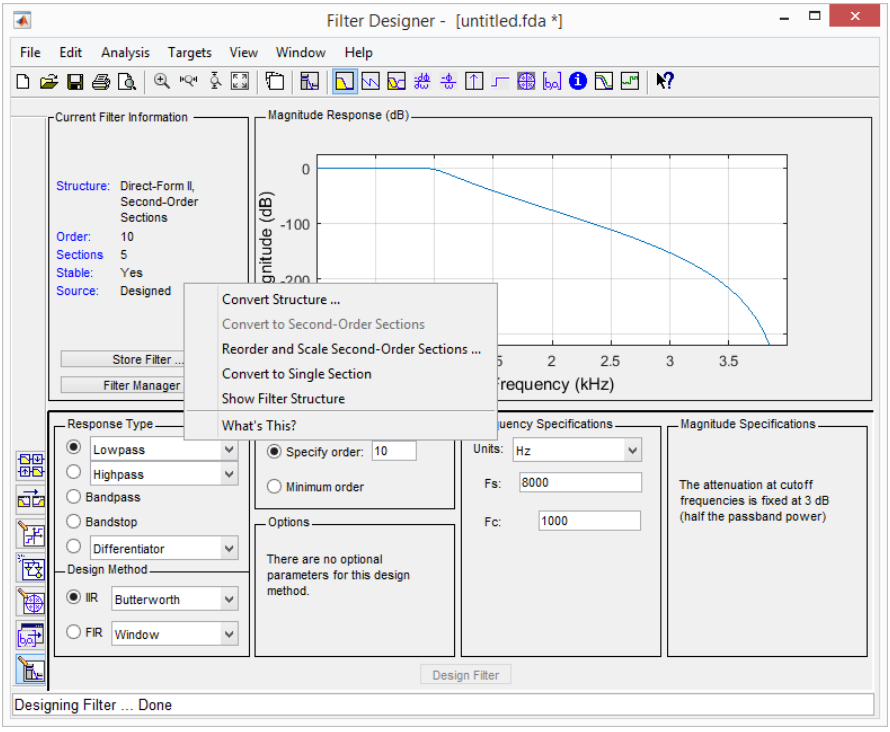

Figure 4.6: Options provided in the "Current Filter Information" section of the Filter Designer design window.

For this structure, the recursive and non-recursive part of the filter is swapped. This has the advantage that the delay elements can be combined. For more information see [8].

A disadvantage of these structures is that if there is only a small (e.g. rounding) error in any of the coefficients the output value will be incorrect due to the recursive nature of these filters. Every (e.g. rounding) error will be recursively applied to the newly calculated samples.

Usually the higher coefficients have a smaller value than the lower coefficients. If this is all stored in, for example, a 16-bit fixed-point number, then the very

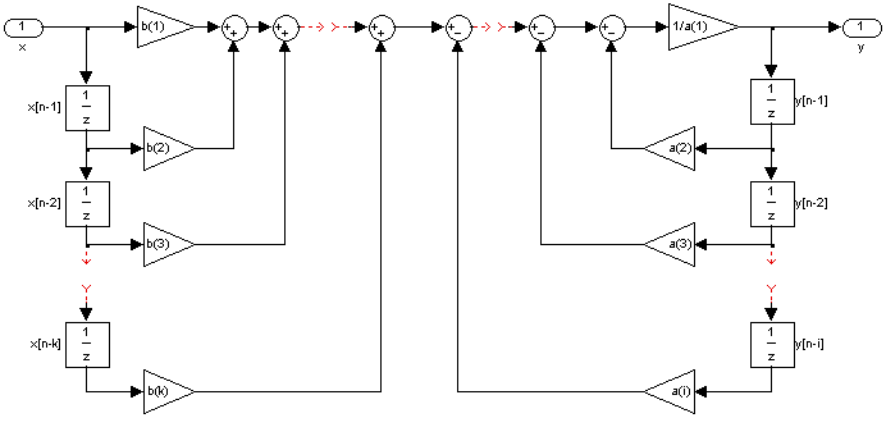

Figure 4.7: Direct-form I single section filter structure.

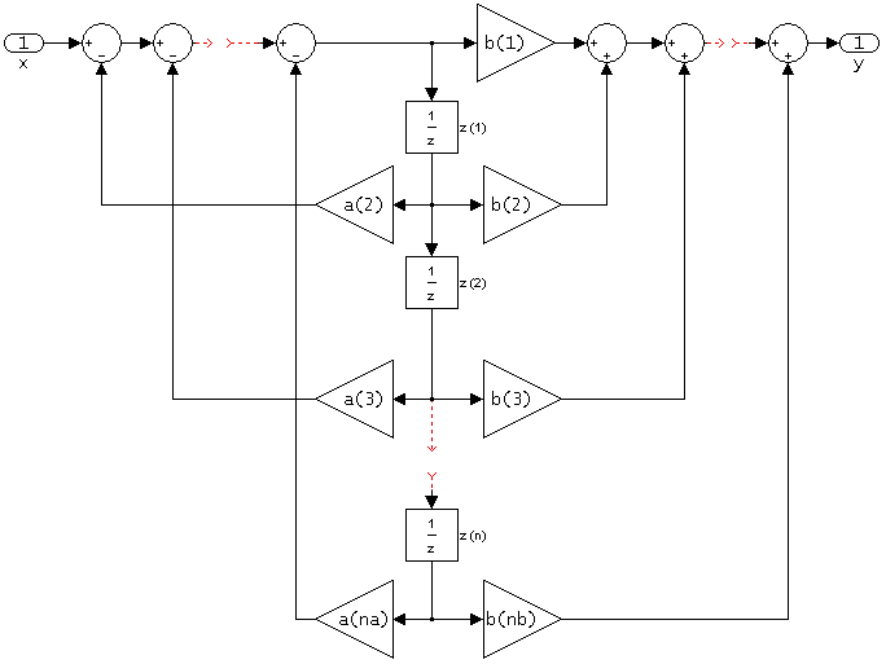

Figure 4.8: Direct-form II single section filter structure.

small values have less significant bits, a lower precision, and thus the round-off error is relatively big. Therefore, with a limited amount of bits, the effective range of a coefficient value is low. Even with floating point numbers this range will decrease exponentially (you might want to look up how floating-point numbers are stored[23]).

A good solution to this is to *cascade* the filters in smaller, second-order (or sometimes called biquadratic) sections. This can be applied to both the direct-form I and direct-form II structures. Figure 4.9 shows an example of an IIR second-order cascaded structure.

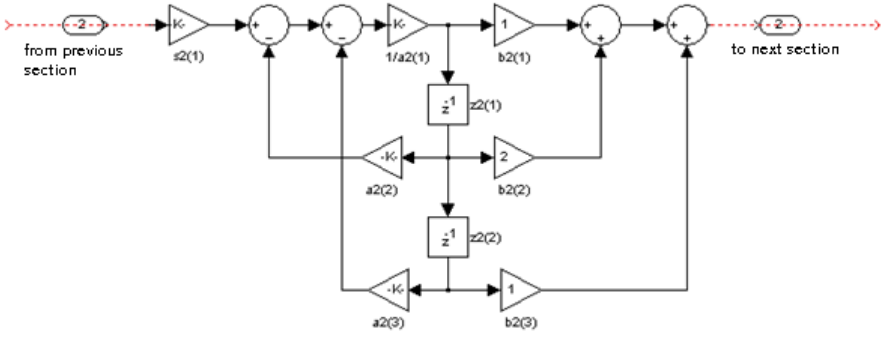

Figure 4.9: Direct-form II cascaded filter structure.

Note that the input of the section that is shown here is the output of a previous section that looks exactly the same (namely like a second order IIR filter), only with different coefficients. Again, the output of this section is the input for the next section until there are no more sections for the sample to pass through. By cascading multiple second-order filters, each filter coefficient will use most of the range of a digitally stored number, thus making round-off errors less pronounced in the final output.

Now that we've seen how the coefficients are calculated and how we can structure IIR filters, a new assignment is given.

[23] https://en.wikipedia.org/wiki/Floating_point

Use MATLAB to calculate the filter coefficients. Don't attempt to derive the coefficients with the BLT yourself. This is outside the scope of the course, only mention the properties of the original analog equivalent of your filter in your report.

4.5 Assignment 6: Infinite Impulse Response Filter

Choose an IIR structure to implement. This can be:

- direct-form I, or
- direct-form II.

The first part of the assignment is to implement a simple second order low-pass filter to test the code for your chosen structure. Design a simple low-pass IIR filter in MATLAB with a cut-off frequency of 1000 Hz and a sample frequency of 8000 Hz. It is important that you first test your code implementation before continuing with higher-order filters or cascaded structures.

Hint: First draw a flow diagram like the one in the FIR filter assignment, see Figure 3.10, before you write your code.

Developing an IIR filter using MATLAB's Filter Designer proves to be more difficult than expected. See these notes: `http://tds02.bitbucket.io/IIRfilter.htm`.

Show the result to your instructor. If everything works as expected, your instructor will sign off assignment 6a.

When your code works, you will get a new filter specification from your instructor.

You are free to choose between an implementation with a cascaded structure (of second order sections) or a single section filter.

Show the result of your new filter to your instructor. If everything works as expected, your instructor will sign off assignment 6b.

Write a report about this assignment. The guidelines for the report can be found in the course repository.

5

Optimizing Your Filter

In this chapter you will learn how to measure the execution time of your code and how to take advantage of the specific features provided by the Cortex-M4 and TLV320AIC3254 codec to speed up the implementation of your filter.

5.1 How to Optimize C Code for the Cortex-M4

In Section 2.1.2 several features which enable the Cortex-M4 to perform digital signal algorithms faster are enumerated.

The DSP capabilities of ARM® Cortex®-M4 and Cortex-M7 Processors [13] describes how you can maximize the performance of your DSP code by using certain instructions, C code idioms and intrinsics, and the CMSIS DSP Library[24]. You can also use the information provided in chapter 3 of *ARM Optimizing C/C++ Compiler v18.1.0.LTS User's Guide* [1, Page 55] to optimize your code.

It is also possible to implement a filter in the TLV320AIC3254 codec itself. This is the most effective and efficient way to implement a FIR or IIR filter. The Cortex-M4

[24] http://arm-software.github.io/CMSIS_5/DSP/html/index.html

is now only used to initialize the codec. How to implement a filter in the codec is explained in the *TLV320AIC3254 Application Reference Guide* [14].

5.2 Assignment 7: Profile and Optimize your Filter

Measure the number of clock cycles which are needed to calculate a new output sample for your implementation of the FIR filter (assignment 5) or IIR filter (assignment 6). You can use the Profile Clock which is provided in CCS[25] to do this.

Use the techniques described in Section 5.1 to optimize your code and make it as fast as you can.

You may also optimize your filter by implementing it inside the codec.

Write a report about this assignment. The guidelines for the report can be found in the course repository.

[25] See http://software-dl.ti.com/ccs/esd/documents/ccs_counting_cycles.html#profile-clock.

Bibliography

[1] *ARM Optimizing C/C++ Compiler v18.1.0.LTS User's Guide*. Texas Instruments. 2018. URL: http://www.ti.com/lit/ug/spnu151r/spnu151r.pdf (cit. on p. 69).

[2] *CC3200AUDBOOST schematics*. Texas Instruments Incorporated. 2014. URL: http://www.tij.co.jp/jp/lit/df/tidra20/tidra20.pdf (cit. on p. 14).

[3] *CC3200AUDBOOST User's Guide*. Texas Instruments Incorporated. 2014. URL: http://www.ti.com/lit/ug/swru383a/swru383a.pdf (cit. on p. 13).

[4] *CC3220 SimpleLink™ Wi-Fi® and Internet of Things Technical Reference Manual*. Texas Instruments Incorporated. 2017. URL: http://www.ti.com/lit/ug/swru465/swru465.pdf (cit. on pp. 13, 26, 27, 28, 31, 32).

[5] *CC3220 SimpleLink™ Wi-Fi® LaunchPad™ Development Kit Hardware User's Guide*. Texas Instruments Incorporated. 2018. URL: http://www.ti.com/lit/ug/swru463b/swru463b.pdf (cit. on p. 13).

[6] *CC3220 SimpleLink™ Wi-Fi® Wireless and Internet-of-Things Solution, a Single-Chip Wireless MCU*. Texas Instruments Incorporated. 2017. URL: http://www.ti.com/lit/ds/symlink/cc3220.pdf (cit. on pp. 13, 17).

[7] *Cortex™-M4 Devices Generic User Guide*. ARM. 2016. URL: https://static.docs.arm.com/dui0553/b/DUI0553.pdf (cit. on p. 19).

[8] A.W.M. van den Enden and N.A.M. Verhoeckx. *Digitale Signaalbewerking*. MK Publishing, 2002. ISBN: 978-90-6674-649-7 (cit. on p. 65).

[9] "IEEE Standard for Floating-Point Arithmetic." In: *IEEE Std 754-2008* (2008), pp. 1–70 (cit. on p. 17).

[10] Sen M. Kuo, Bob H. Lee, and Wenshun Tian. *Real-Time Digital Signal Processing: Fundamentals, Implementations and Applications*. 3rd. MK Publishing, 2013. ISBN: 978-1-118-41432-3 (cit. on pp. 7, 8, 42).

[11] Paul A. Lynn and W. Fuerst. *Inleiding Digitale Signaalbewerking met Maple en Matlab*. Ed. by J.W.M. Andriessen. ThiemeMeulenhof, 2004. ISBN: 978-90-5574-448-0 (cit. on pp. 35, 42, 55).

[12] Donald S. Reay. *Digital Signal Processing Using the ARM® Cortex®-M4*. 1st. John Wiley & Sons, Inc., 2015. ISBN: 978-1-118-85904-9 (cit. on p. 7).

[13] *The DSP capabilities of ARM® Cortex®-M4 and Cortex-M7 Processors*. Thomas Lorenser. 2016. URL: https://community.arm.com/cfs-file/__key/communityserver-blogs-components-weblogfiles/00-00-00-21-42/7563.ARM-white-paper-_2D00_-DSP-capabilities-of-Cortex_2D00_M4-and-Cortex_2D00_M7.pdf (cit. on pp. 18, 69).

[14] *TLV320AIC3254 Application Reference Guide*. Texas Instruments Incorporated. 2012. URL: http://www.ti.com/lit/an/slaa408a/slaa408a.pdf (cit. on pp. 13, 15, 16, 70).

[15] *TLV320AIC3254 Ultra Low Power Stereo Audio Codec with Embedded miniDSP*. Texas Instruments Incorporated. 2014. URL: http://www.ti.com/lit/ds/symlink/tlv320aic3254.pdf (cit. on pp. 13, 16).

A

Fixed-point Arithmetic

This appendix gives a short introduction into fixed-point arithmetic. Floating-point and fixed-point numbers were introduced in Section 2.1.2. The Qn.m notation for fixed-point numbers was also explained in Section 2.1.2. In this appendix we will see how arithmetic operations (add, subtract, multiply, and divide) with fixed-point numbers can be performed. As already explained the processor which we use is optimized for 16-bit fixed-point numbers. In this appendix we will use 8-bit fixed-point numbers instead. A Qn, m fixed-point value will be stored in a $1 + n + m$-bit two's complement integer variable.

A.1 Add and Subtract

If we want to add or subtract two fixed-point numbers we have to align their radix points. For example if a is a Q5.2 encoded fixed point number with value 00001101 ($000011.01_2 = 3.25_{10}$) and b is a Q4.3 encoded fixed point number with value 00100101 ($00100.101_2 = 4.625_{10}$), then $a + b$ can not be calculated by simply adding their values. The radix points must be aligned before the addition as shown in Figure A.1. As can be seen in Figure A.1, the result is a Q5.3 fixed-point number with value 000111110 ($000111.111_2 = 7.875_{10}$).

```
000011.01  +
 00100.101 =
000111.111
```

Figure A.1: Adding a Q4.3 to a Q5.2 fixed-point number.

In general, when we add a Qn_1, m_1 fixed-point number by a Qn_2, m_2 fixed-point number, the result will be a $Q\max(n_1 + n_2 + 1), \max(m_1 + m_2)$ fixed-point number.

When programming the number of bits we use for variables is most of the times fixed (e.g. to 8-bit). In this case we can convert a from Q5.2 to Q4.3 by shifting it one place to the left. Care must be taken, not to generate an overflow by this operation. After the shift we can use an integer addition because both numbers are Q4.3 and the result will also be Q4.3 (when we assume that no overflow occurs when performing the integer addition). The C code is given in Listing A.1.

```
int8_t a = 0x0d; // Q5.2 with decimal value 3.25
int8_t b = 0x25; // Q4.3 with decimal value 4.625
int8_t sum = (a << 1) + b; // sum will be Q4.3 with ↩
 ↪ decimal value 7.875
```

Listing A.1: Adding a Q4.3 to a Q5.2 fixed-point number in C.

Alternatively it is also possible to convert b from Q4.3 to Q5.2 by shifting it one place to the right. This operation can not cause an overflow but some precision is lost. After the shift we can use an integer addition because both numbers are Q5.2 and the result will also be Q5.2 (when we assume that no overflow occurs when performing the integer addition). The C code is given in Listing A.2.

```
int8_t a = 0x0d; // Q5.2 with decimal value 3.25
int8_t b = 0x25; // Q4.3 with decimal value 4.625
int8_t sum = a + (b >> 1); // sum will be Q5.2 with ↩
 ↪ decimal value 7.75
```

Listing A.2: Adding a Q4.3 to a Q5.2 fixed-point number in C. Please note the `sum` is less precise than the `sum` calculated in Listing A.1.

Fixed-point numbers can be subtracted in a similar way. For example we can calculate $a - b$ by converting a to Q4.3 and perform an integer subtraction. The result will be Q4.3 in two's complement notation (when we assume that no overflow occurs when performing the integer subtraction). The operation is shown in Figure A.2. The result is $11110.101_{\text{two's complement}} = -00001.011_2 = -1.375$, which is the correct answer.

```
00011.010 -
00100.101 =
11110.101
```

Figure A.2: Subtracting two Q4.3 fixed-point numbers.

A.2 Multiply and Divide

If we multiply two fixed-point numbers we can just multiply them by using integer multiplication. The only thing left to do, is figuring out where the radix point must be placed in the result. Figure A.3 shows how $a \cdot b$ can be calculated.

```
      000011.01 *
      00100.101 =
      000.01101 +
     0000.0000  +
    00001.101   +
   000000.00    +
  0000000.0     +
 00001101       +
00000000        +
00000000        =
0000001111.00001
```

Figure A.3: Multiplying a Q5.2 with a Q4.3 fixed-point number.

As can be seen in Figure A.3, the result is a Q9.5 fixed-point number with value 0000001111.00001 ($0000001111.00001_2 = 15.03125_{10}$), which is the correct answer. The precision if the result is higher than the precision of the multiplier

and multiplicand. We can convert the result to the same precision as the multiplier (Q4.3) by shifting it two places to the right and truncating it to 8 bits. We obviously will lose some precision and before the truncation we must check if the result can be represented in 8 bits. This will yield a Q4.3 fixed-point number with value 01111000 ($01111.000_2 = 15_{10}$).

In general, when we multiply a Qn_1, m_1 fixed-point number by a Qn_2, m_2 fixed-point number, the result will be a $Qn_1 + n_2, m_1 + m_2$ fixed-point number.

When we program in C it is important to prevent an overflow while calculating a product. Most of the time the multiplier and the multiplicand must be casted to a bigger data type before the multiplication is performed.

Divisions can be performed in a similar manner as multiplications. In general, when we divide a Qn_1, m_1 fixed-point number by a Qn_2, m_2 fixed-point number, the result will be a $Qn_1 + m_2, m_1 - m_2$ fixed-point number.