

C2: Digital Filter Simulation, Implementation, and Testing on a Digital Signal Processor

1 Aims

This exercise will help you to understand and appreciate a practical filter implementation on a digital signal processor. Specifically, after this lab you should

- understand a convolution / digital filter by hands-on programming;
- become aware of the flexibility of digital filters as opposed to analogue ones;
- make a first step toward using a digital signal processor (DSP) and the addressing of its peripherals;
- become aware of the real-time aspect of signal processing and the trade-off between the computational complexity and the performance of a digital filter.

Please prepare **Actions 1–4** before and **Actions 5–9** during the lab. Additional files and instructions can be found at <http://www.ecs.soton.ac.uk/~sw1/C2/LabC2.html>.

2 Preparation

2.1 Finite Impulse Response Filter

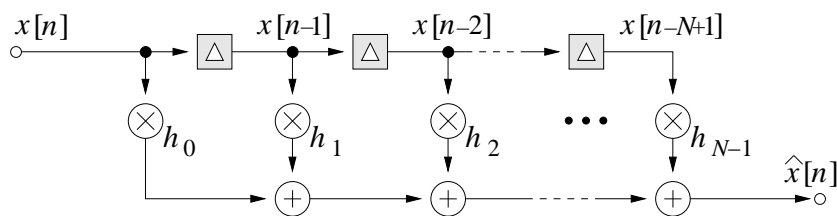


Figure 1: Finite impulse response filter flow graph.

Based on the flow graph of the finite impulse response (FIR) filter in Fig. 1, the output $\hat{x}[n]$ can be calculated as

$$\hat{x}[n] = \mathbf{h}^T \cdot \mathbf{x}_n \quad (1)$$

where \mathbf{h} contains the filter coefficients and \mathbf{x}_n the samples of the input signal held in the tapped delay line (TDL) at time instance n ,

$$\mathbf{h}^T = [h_0 \ h_1 \ \cdots \ h_{N-1}] \quad (2)$$

$$\mathbf{x}_n^T = [x[n] \ x[n-1] \ \cdots \ x[n-N+1]]. \quad (3)$$

In the simplest case, with $h_n = \frac{1}{N}$ for $n = 0(1)N-1$, the filter takes an average of N consecutive samples of the signal $x[n]$. This type of averaging is a lowpass filter operation.

2.2 FIR Filter Design and Implementation in Matlab

Other filter types or improved lowpass filters can be attained by suitable design of the filter coefficients. The filter can be uniquely characterised by its impulse response. Observe from Fig. 1 that for an FIR filter, the filter coefficients $h[n]$ form the impulse response. The frequency response is related to the impulse response by the DFT.

Action 1: Design at least two different filters (e.g. a short and a long one, lowpass or high-pass), for example in Matlab using the function `fir1()` or `remez()`. Plot their impulse responses and magnitude responses, and save the coefficients into a file in ASCII format. In the lab, you can later paste these coefficients into your c-code and implement “your filter(s)”.

Action 2: Implement an FIR filter in Matlab using the coefficient vector \mathbf{h} and the tapped delay line \mathbf{x}_n based on the steps

```
iterate for number of samples in  $x[n]$ 
    update TDL vector  $\mathbf{x}_n$ 
    calculate output  $\hat{x}[n]$  according to (1) .
```

2.3 FIR Filter on a DSP

The operation in (1) requires N multiply-accumulates (MACs), for which a DSP is optimised. Specifically, the TMS320C6711 used in this lab can perform up to 300 million MACs/sec.

Action 3: Calculate for your filter designs, how many MACs/sec are required for real time operation assuming a sampling rate of 8kHz. What would be the maximum number of filter coefficients that could still be realised in a real-time implementation on the C6711?

To implement an FIR filter, two arrays for the tap-delay-line (TDL) samples in \mathbf{x}_n and the coefficients \mathbf{h} are required as shown in Fig. 2. By multiplying corresponding array elements and accumulating the products, one output sample $\hat{x}[n]$ is calculated. The arrays may be addressed by indexing, or by using appropriate pointers as shown in Fig. 2.

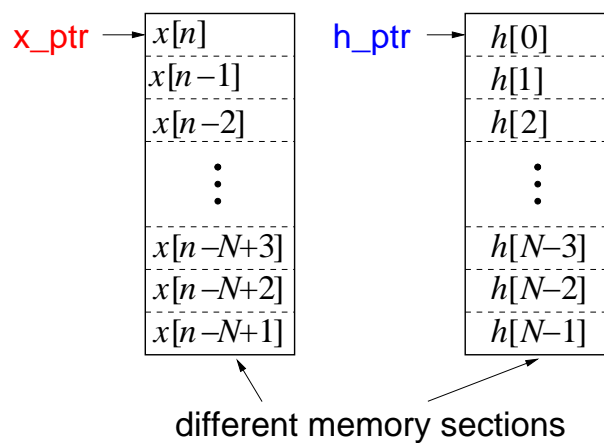


Figure 2: Arrays in memory holding the tap-delay-line (TDL) values \mathbf{x}_n and the coefficients \mathbf{h} .

In the subsequent sampling period with time index $n + 1$, the TDL vector \mathbf{x}_n has to be updated to \mathbf{x}_{n+1} . This can be performed by shifting all entries in the left array of Fig. 2 down by one place and feeding in the latest datum at the top position. This however will require N memory moves, which are significantly more costly than the N multiplications required to calculate the next scalar product $\hat{x}[n + 1] = \mathbf{h}^T \cdot \mathbf{x}_{n+1}$!

In order to save memory moves, pointer addressing may be used to only overwrite the oldest datum when updating the TDL array, as indicated in Fig. 3. During execution of a scalar product or when decrementing the pointer to insert a new datum, care has to be taken that the pointer remains within the bounds of the array. This arrangement is generally referred to as a *circular buffer* and is commonly realised with the aid of modulo- N addressing. With these savings, the resulting complexity of an FIR filter implementation is given by N MACs and one memory move per sampling period.

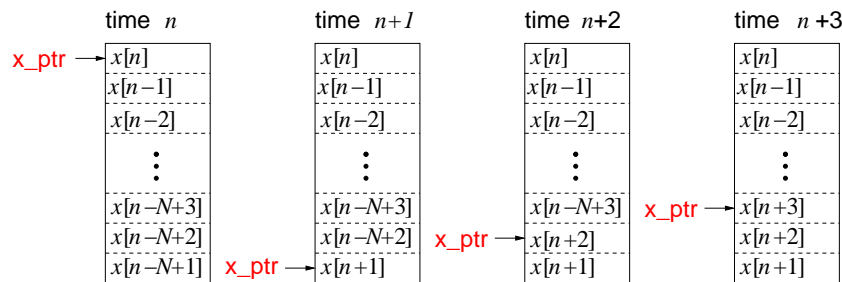


Figure 3: Circular buffer implementation for updating the TDL array.

Action 4: Assuming that the most recent input sample $x[n]$ is held in a C-variable `in_sample`, produce on paper a segment of C-code that will — operated once per sampling period — (i) update an array `x[N]` holding the TDL values and (ii) perform the scalar product with the array `h[N]` holding the filter coefficients, with `x` and `h` as defined in Fig. 2. After execution of your code segment, the output sample should be in a variable `xhat`. Your code segment should use array indexing. Additionally, produce a second example with pointer addressing (no circular buffer required, although you may want to give it some thought!).

2.4 DSP Peripherals

The C6711 can interact with peripherals via a memory expansion port and two serial ports (multichannel buffered serial ports, McBSP0 and McBSP1). On the DSP board used in the Lab, McBSP0 is connected to an ADC / DAC converter AD535 which can be used for audio I/O. The AD535 is a mono device, sampling at 8kHz with a word length of 12 bit.

3 Familiarisation with the C6711 and Code Composer Studio

In this section you learn how to connect the board and how to use the compiler. The example program passes audio samples through the DSP. You will use this file to later include your digital filter functions in order to process the audio signal.

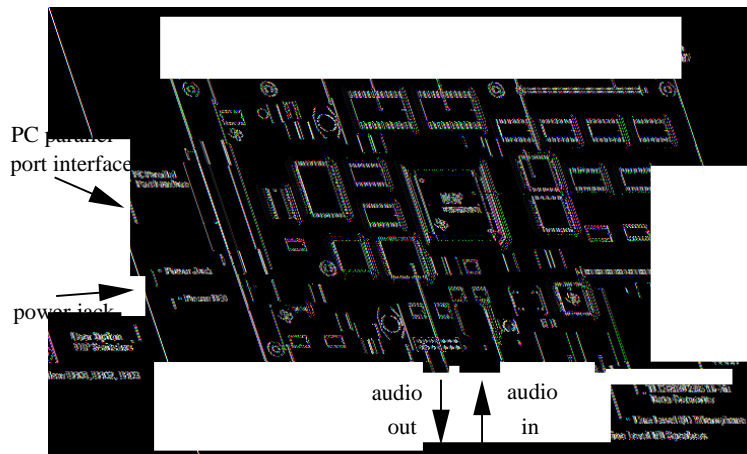


Figure 4: DSP starter kit (DSK) board with a C6711 processor.

3.1 DSP, Board, and Software

We are using a Texas Instruments TMS320C6711 floating point processor, which is placed on a DSP starter kit (DSK) board as shown in Fig. 4 for experimentation. This board has a number of peripheral devices connected to the C6711 processor, such as switches, LEDs, an expansion peripheral interface, an expansion memory interface, and a digital-to-analogue (DAC) and analogue-to-digital (ADC) converter AD535. We will use the AD535 to allow the DSP to process audio signals and play back the result.

1. Connect the DSK to the host computer's parallel port; also connect the power supply to the board — some LEDs should light up when the device is powered up.
2. Start code composer studio (CCS), an integrated software development environment for Texas Instruments DSPs, by double-clicking the C'6000 CCS icon on the desktop.

Note: if you need to reset the DSK while CCS is running, go to **Debug: Reset DSP**. If this does not work, you will have to close CCS, and then perform the reset (probably by power-cycling the DSK) prior to re-opening CCS.

3. To test the correct functioning of the DSK, click on **GEL: Check DSK: QuickTest**. The LEDs on the board should start flashing while the DSP steps through a self-test.

3.2 Project File

1. Create a new project called `LabC2.prj` under

Project: New

and save it in an appropriate subdirectory in your workspace.

2. Create a DSP/BIOS configuration file called `LabC2.cdb`

File: New: DSP/BIOS Configuration

using the `dsk6711.cdb` template. In this configuration file, go to **Systems: Global Settings** and right-click on properties: ensure that the Chip Support Library (CSL) is set to 6711. Save the file `LabC2.cdb` in your project folder.

3. Add the configuration file to the project.

Project: Add Files to Project

This automatically adds the file `LabC2cfg.s62`.

4. Also add the linker command file `LabC2cfg.cmd`.

3.3 Audio Loopback Programme

You are provided with a program `loop.c` which reads a sample from the on-board ADC and writes it back to the DAC. Both operations are performed via McBSP0 and hardware interrupt.

1. Add the C program `loop.c` to the project. Viewing the file, note that
 - I/O ports are initialised by functions `CSL_init()`, `BSL_init()`, `codec_init()`, and `HWI_init()` which are built on functions within the chip support library (CSL, specific for the C6711 processor) and the board support library (BSL, specific for the DSK board).
 - Transmitted and received data is passed serially between the McBSP0 and the AD535 codec chip. The McBSP0 issues a hardware interrupt whenever a newly received 16-bit sample can be picked up from the McBSP0 receive register, or when the McBSP0 transmit register is empty and needs to be filled again.
 - The input datum is transferred from the McBSP0 receive register into a variable `in_sample` by the interrupt service routine (ISR) `RINT0_HWI()`. You can include any processing into this ISR function `RINT0_HWI()` later.
 - The output data is transmitted via the ISR function `XINT0_HWI()`.
2. In the Project Toolbar, click onto the DSP/BIOS configuration file `LabC2.cdb`. Under **Scheduling:** HWI there is a list of hardware interrupts in descending priority. You need to associate `HWI_INT9` with `MCSP_0_TRANSMIT` and the ISR function `XINT0_HWI` (which is defined in the c file `loop.c`). You can do this by right-clicking onto `HWI_INT9` and **Properties**. Be sure that you enter a leading underscore for the function name and check the **Use Dispatcher** box. The later will make sure that register contents are saved prior to execution of the interrupt service routine, which could otherwise overwrite important register values.
3. Associate `HWI_INT11` with `MCSP_0_RECEIVE` and the ISR function `RINT0_HWI`.

Action 5: If processing will be performed in the hardware interrupt ISR function `RINT0_HWI`, will there be a conflict when an `MCSP_0_TRANSMIT` interrupt occurs?

3.4 Building the Project

Building a project consists of two steps: compiling and linking. Each of these steps has various options which determine, for example, how efficient the generated DSP code will be. Two `.txt` files are provided that contain appropriate compiler and linker options.

1. Set the compiler options

Project: Build Options

by pasting the content of `compiler.txt` into the compiler window (any old content in this window should be deleted prior to pasting).

2. Set the linker options by copying the content of `linker.txt` into the linker window.
3. Now build the project:

Project: Rebuild All

Once CCS has successfully compiled and linked, it will create a file `loop.out` that is the DSP executable file.

3.5 Load and Run the Program

Having built the program, it can be downloaded to the DSP and run.

1. Load the program onto the DSP

File: Load

The DSP executable file, here `loop.out`, has a `.out` extension and can usually be found in a subdirectory `Debug` of your current project folder.

2. Start the execution of the program:

Debug: Run

3. Stop the execution of the program:

Debug: Halt

Note: when the program is modified and rebuilt later, it can be reloaded to the DSP using **File: Reload Program**. Note that you should always halt the DSP before loading or reloading a program.

Action 6: Connect an audio source and headphones to the DSK board, and verify the correct working of the loopback system. Why is the audio quality somewhat impaired?

4 Filter Implementation

4.1 Modifying your Project

You will now add an FIR filter into the previous loopback system. If do not want to go through the tedious setup of Sec. 3 for a new project, you can simply rename `loop.c` to a new file name, delete `loop.c` from the Project Toolbar, and add the new `.c` file to the Project via

Project: Add Files to Project.

If you do not want to overwrite your previously produced DSP executable file, you can additionally modify the output file in the linker window

Project: Build Options: Linker

as appropriate.

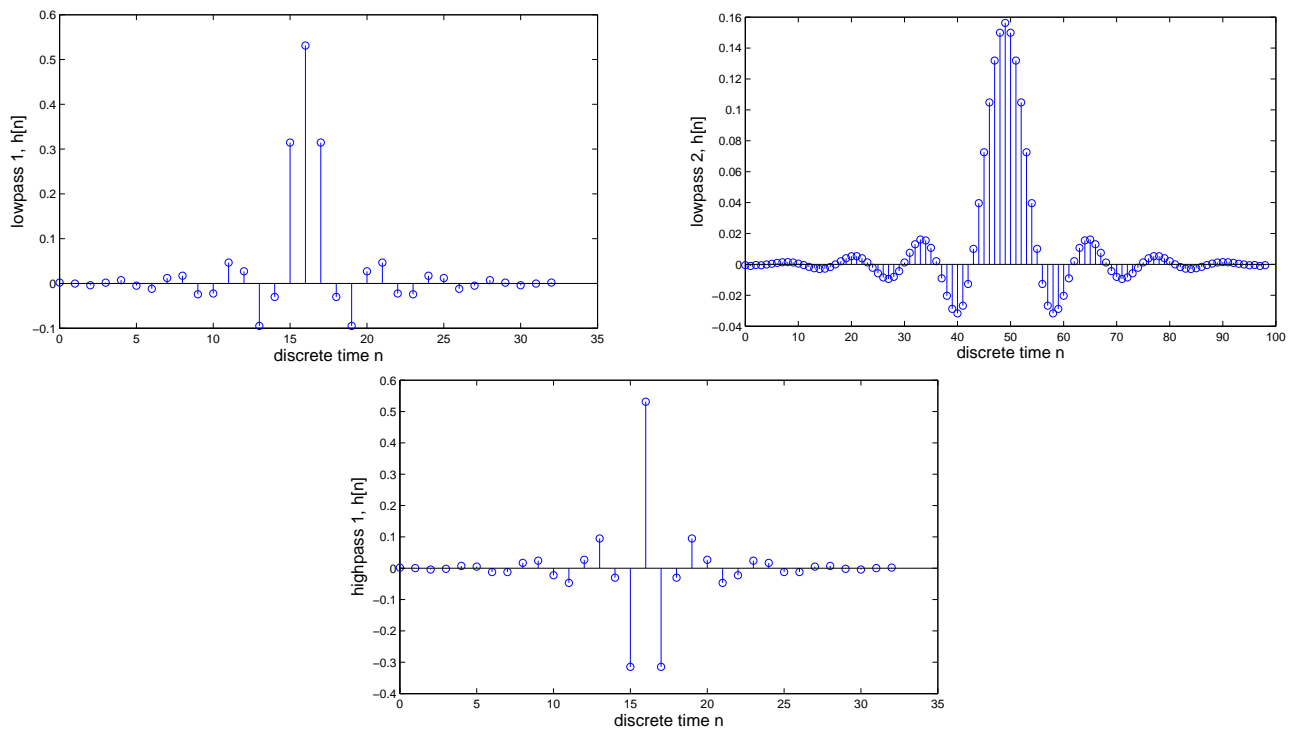


Figure 5: Impulse responses of `lowpass1` (top left), `lowpass2` (top right), and `highpass1` (below).

4.2 Definition of Filter Coefficients

In case you failed to obtain your own filter designs, you can find 3 designs contained in files `lowpass1.txt`, `lowpass2.txt`, and `highpass1.txt`. Simply paste the files into the global variable definition section of your `.c` file. The general format is

```
#define FilterLength 5
float x[FilterLength];
float h[FilterLength] = {0.2, 0.2, 0.2, 0.2, 0.2};
```

whereby `FilterLength` is equivalent to the variable N in Fig. 1, and `x` and `h` are the base addresses of the arrays shown in Fig. 2.

The impulse response with the coefficients $h[n]$ are given in Fig. 5, while the frequency responses are plotted in Fig. 6. Note that the tighter lowpass filter design `lowpass2` requires about 3 times as many coefficients as `lowpass1` to achieve a comparable stopband attenuation!

4.3 Main Filter Function

Action 7: Expand the ISR function `RINT0_HWI()` to include a simple routine updating the TDL vector \mathbf{x}_n with the current sample in `in_sample`, and performing the scalar product $\mathbf{h}^T \cdot \mathbf{x}_n$, with the result written to `out_sample`. The TDL should be initialised with zero values at the beginning of `main()`.

Create a `.out` file for at least two different filter implementations. Note that if you write your filter code flexible enough, you only need to paste a new set of coefficients into the global variable definition section according to Sec. 4.2.

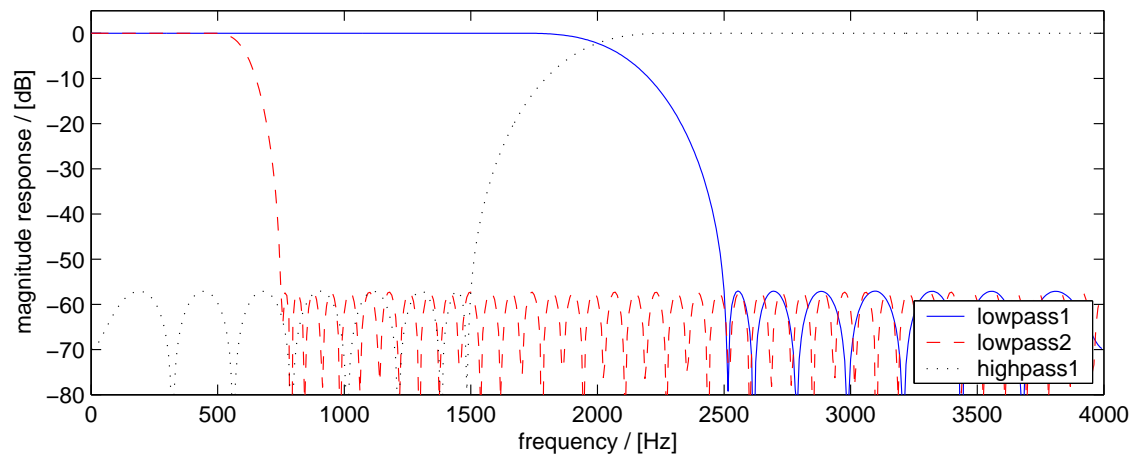


Figure 6: Magnitude responses of the three different filter designs.

4.4 Measurement

Action 8: Connect an audio source and headphones to the DSK board, and compare the sound of the audio output $\hat{x}(t)$ to the original signal $x(t)$. Can you notice the effect of lowpass or highpass filtering?

Action 9: Connect a signal generator to the audio input making sure that the voltage level remains below $\pm 1V$ *without any DC offset*. Display the filtered and unfiltered waveforms $x(t)$ and $\hat{x}(t)$ on an oscilloscope and measure the gain of the filter for a sufficient set of frequencies. Record your measurements and verify the magnitude response of your filter design (or Fig. 6).

4.5 Optional Work

1) You can improve the efficiency of your implementation by using a circular buffer as shown in Fig. 3. You may use a pointer `x_ptr_base` to the base address of the TDL array and apply `*(x_ptr_base + x_ptr)` to access the array elements, whereby `x_ptr` is an integer which you can restrict to the interval between 0 and `FilterLength-1`.

2) Have a bit of fun! Comparing `lowpass1` and `highpass1` in Figs. 5 and 6, note that the modulation with a $f = 4\text{kHz}$ cosine, $\cos(2\pi f/f_s \cdot n) = \cos(\pi \cdot n) = (-1)^n$ effectively negates every second coefficient, leading to an reversion of the spectrum by very simple means. Similarly, by flipping the sign of every second coefficient, you can reverse the frequency spectrum of a signal. Implement this scheme, e.g. based on `loop.c`, and check if you can still recognise a frequency-reversed audio signal.

5 Assessment and Marking

Please document any filter design and code segments in your log book. Your success in the lab will be measured with respect to the above Actions and your understanding in terms of the aims stated in Sec. 1.