

Digital Signal Processing  
Application on the Texas Instrument C6748 Processor

VERSION 1.2

JULIEN OSMALSKYJ and JEAN-JACQUES EMBRECHTS

October 15, 2014

# Contents

<b>1</b>	<b>Introduction to Digital Signal Processors</b>	<b>3</b>
1.1	Introduction . . . . .	3
1.1.1	Processors architecture . . . . .	4
1.2	C6748 Processor . . . . .	5
1.2.1	Architecture . . . . .	5
1.2.2	Interrupts . . . . .	8
1.3	OMAP L-138 Experimenter Board . . . . .	10
<b>2</b>	<b>Development environment: Code Composer Studio</b>	<b>12</b>
2.1	First connection to the EVM . . . . .	12
2.2	Create a project . . . . .	14
<b>3</b>	<b>Input / Output</b>	<b>17</b>
3.1	Data format . . . . .	18
3.2	Polling . . . . .	19
3.3	Interrupts . . . . .	20
3.4	Direct Memory Access (DMA) . . . . .	20
3.5	Volatile variables . . . . .	21
3.5.1	Syntax . . . . .	21
3.5.2	Use . . . . .	21
3.5.2.1	Peripheral registers . . . . .	22
3.5.2.2	Interrupt service routines . . . . .	22
<b>4</b>	<b>Filtering</b>	<b>24</b>
4.1	Finite Impulse Response (FIR) Filters . . . . .	24
4.1.1	Windowed-sinc filters . . . . .	24
4.1.2	Windowed-sinc filters design . . . . .	26

4.2	Recursive Filters . . . . .	27
4.2.1	Narrow-band filters . . . . .	27
4.2.2	IIR filters structures . . . . .	29
4.2.2.1	Direct Form I Structure . . . . .	29
4.2.2.2	Direct Form II Structure . . . . .	29
4.2.2.3	Cascade structure . . . . .	30
<b>5</b>	<b>Frame-based signal processing</b>	<b>32</b>
5.1	Frame-based vs Sample-based processing . . . . .	32
5.2	Direct Memory Access . . . . .	33
5.2.1	Architecture . . . . .	33
5.2.2	DMA transfer . . . . .	34
5.3	DMA in practice . . . . .	37
5.3.1	Ping-Pong buffering . . . . .	37
5.3.2	Project organization . . . . .	37
5.4	Frame-based convolution . . . . .	40
5.4.1	Overlap-Add . . . . .	40
<b>6</b>	<b>Fast Fourier Transform</b>	<b>42</b>
6.1	Discrete Fourier Transform . . . . .	42
6.2	Fast Fourier Transform . . . . .	43
6.2.1	Forward FFT . . . . .	43
6.2.2	Inverse FFT . . . . .	44
6.2.3	FFT in practice . . . . .	44
6.3	FFT Convolution . . . . .	45

# Chapter 1

## Introduction to Digital Signal Processors

### 1.1 Introduction

Digital Signal Processors (DSP) are specific processors used for signal processing. Digital signal processing is used in many areas such as sound, video, computer vision, speech analysis and synthesis, etc. Each of these areas need digital signal processing and can therefore use these specific processors. DSPs are found in cellular phones, disk drives, radios, printers, MP3 players, HDTV, digital cameras and so on. DSPs take real world signal like voice, audio, video, temperature, pressure, position, etc. that have been digitized using an analog-to-digital converter and then manipulate them using mathematical operations. DSPs are usually optimized to process the signal very quickly and many instructions are built such that they require a minimum amount of CPU clock cycles.

Typical optimization of the processors include hardware multiply accumulate (MAC) capability, hardware circular and bit-reversed capabilities, for efficient implementation of data buffers and fast Fourier transforms (FFT), and Harvard architecture (explained in Section 1.1.1). One specificity of DSPs are their MAC operation (Multiply ACcumulate). Once a signal has been digitized, it is available to the processor as a serie of numerical values. Digital signal processing often involves the convolution operation, which correspond to a serie of sums and products. On a general CPU, each of these instruction takes more than one clock cycle. However, DSP's MAC operation performs a product and a sum in one clock cycle, allowing thus faster signal processing. Another specificity of DSPs is their ability to access more than one memory address in one cycle. This allows the processor to read an instruction and the corresponding data simultaneously. This is an improvement over general processors.

DSPs are typically very similar to microcontrollers. They usually provide single chip computer solutions integrating on-board volatile and non-volatile memory as well as a range of peripheral interfaces. They moreover have a small footprint, making them ideal for embedded applications. In addition, they tend to have low power consumption requirements, which makes them suitable for portable applications, for example in cell phones, or in a car.

In general, digital signal processors are used primarily in real-time applications. Real-time processing means that the processing must be done within a fixed time period, usually

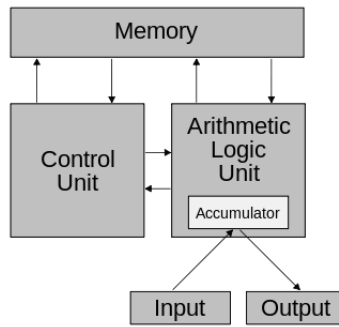


Figure 1.1: Von Neumann architecture

synchronized with an external peripheral, for example an analog-to-digital converter. This course focuses on real-time digital signal processing and each examples used in the course deal with that real-time constraint.

This document is organized as follows. This first chapter introduces DSPs architectures, and in particular the Texas Instrument C6748 DSP architecture, which is used in this course. Chapter 2 describes the Texas Instrument integrated development environment (IDE) and the setup of Code Composer Studio to start programming the DSP. In Chapter 3, several input / output schemes are described to access the audio samples. Chapter 4 introduces notions of filtering and presents the two most used filters, which are the Finite Impulse Response (FIR) filters and the Infinite Impulse Response (IIR) filters. The chapter also describes some useful implementations of those filters. Direct Memory Access (DMA) is explained in Chapter 5. DMA is used to transfer blocks of samples between the codec and the memory without any CPU intervention, which is useful for many frame-based signal processing algorithm. Finally, Chapter 6 details the use of the Discrete Fourier Transform (DFT) in the DSP, as well as the Fast Fourier Transform (FFT) algorithm.

Two reference books are used for this course. The first one is *Digital Signal Processing and Applications with the OMAP-L138 eXperimenter* [8], by Donald Reay, and the second one is *Real-Time Digital Signal Processing from MATLAB to C with the TMS320C6x DSPs* [10], by Thad B. Welch, Cameron H.G. Wright and Michal G. Morrow.

### 1.1.1 Processors architecture

There are two types of architectures used in processors: **Von Neuman** architecture, and **Harvard** architecture. In the first one, programs and data are stored in the same memory zone, as showed in Figure 1.1. In that kind of architecture, an instruction contains the operating code (opcode) and the addresses of the operands.

In the second type of architecture, there are two memories: one for the program, and one for the data. Both memories are separated and are accessed via separated buses, as shown in Figure 1.2. That organization allows to read an instruction and the data simultaneously, which improves the overall performance of the processor.

General microprocessors usually make use of the Von Neuman architecture. However, DSPs tend to use the Harvard architecture which is more suited for real-time applications. Some DSPs use a modified Harvard architecture, in which there is one external bus for the data, and one external bus for the addresses, like the Von Neuman architecture. However, the DSP contains two internal distinct data buses and two distinct addresses buses. The

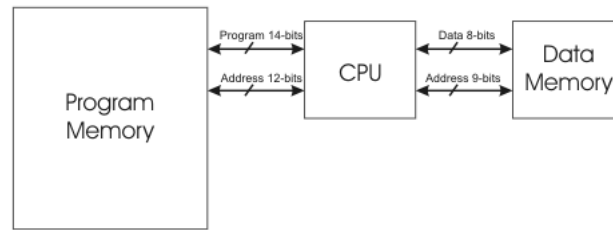


Figure 1.2: Harvard architecture

transfer between the external and internal buses are performed using time multiplexing. This is the case in some Texas Instrument processors (C6000 serie), in particular, this is the case of the C6748 processor used in this course.

## 1.2 C6748 Processor

This course uses a Texas Instrument OMAP-L138 processor, containing a C6748 DSP processor and an ARM9 processor in the same chip. We only make use of the DSP part of the chip. The C6748 is based on Texas Instruments very long instruction word (VLIW) architecture. This processor is well suited for numerically intensive algorithms. The internal program memory is structured so that a total of eight instructions can be fetched every cycle. The C6748 has a clock rate of 375 MHz and is capable of fetching eight 32-bit instructions every  $1/375$  MHz or 2.67 ns. The processor includes both floating-point and fixed-point architectures in one core.

The C6748 includes 326 kB of internal memory (32 kB of L1P program RAM/cache, 32 kB of L1D data RAM/cache and 256 kB of L2 RAM/cache), eight functional units composed of six ALUs and two multiplier units, an external memory interface addressing 256 MB of 16-bit mDDR SDRAM, and 64 32-bit general purpose registers. In addition, the OMAP-L138 features 128 kB of on-chip RAM shared by its C6748 and ARM9 processor cores.

### 1.2.1 Architecture

In this section we give an brief overview of the architecture of the C6748 processor. This section is an excerpt of the C6748 Datasheet [3]. Detailed documentation can be found in [3]. The architecture of the CPU can be seen in Figure 1.3.

The C674x CPU consists of eight functional units, two register files, and two data paths, as shown in Figure 1.4. The two general-purpose register files (A and B) each contain 32 32-bit registers for a total of 64 registers. The general-purpose registers can be used for data or can be data address pointers. The data types supported include packed 8-bit data, packed 16-bit data, 32-bit data, 40-bit data, and 64-bit data. Values larger than 32 bits, such as 40-bit-long or 64-bit-long values are stored in register pairs, with the 32 LSBs of data placed in an even register and the remaining 8 or 32 MSBs in the next upper register (which is always an odd-numbered register). The eight functional units (.M1, .L1, .D1, .S1, .M2, .L2, .D2, and .S2) are each capable of executing one instruction every clock cycle. The .M functional units perform all multiply operations. The .S and .L units perform a general set of arithmetic, logical, and branch functions. The .D units primarily load data from memory to the register file and store results from the register file into memory. The C674x

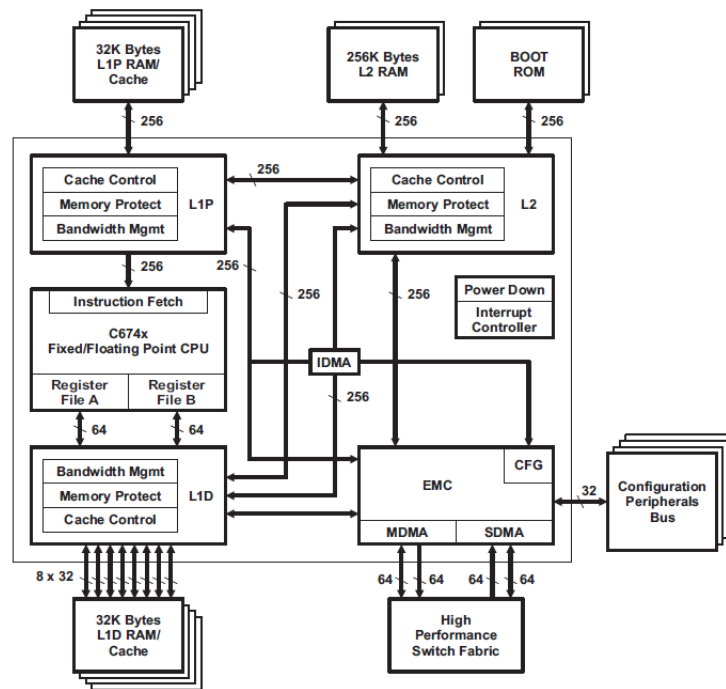


Figure 1.3: C674x Megamodule Block Diagram. Source: [3]

CPU combines the performance of the C64x+ core with the floating-point capabilities of the C67x+ core.

Each C674x .M unit can perform one of the following each clock cycle: one 32 x 32 bit multiply, one 16 x 32 bit multiply, two 16 x 16 bit multiplies, two 16 x 32 bit multiplies, two 16 x 16 bit multiplies with add/subtract capabilities, four 8 x 8 bit multiplies, four 8 x 8 bit multiplies with add operations, and four 16 x 16 multiplies with add/subtract capabilities (including a complex multiply). There is also support for Galois field multiplication for 8-bit and 32-bit data. Many communications algorithms such as FFTs and modems require complex multiplication. The complex multiply (CMPY) instruction takes for 16-bit inputs and produces a 32-bit real and a 32-bit imaginary output. There are also complex multiplies with rounding capability that produces one 32-bit packed output that contain 16-bit real and 16-bit imaginary values. The 32 x 32 bit multiply instructions provide the extended precision necessary for high-precision algorithms on a variety of signed and unsigned 32-bit data types. The .L or (Arithmetic Logic Unit) now incorporates the ability to do parallel add/subtract operations on a pair of common inputs. Versions of this instruction exist to work on 32-bit data or on pairs of 16-bit data performing dual 16-bit add and subtracts in parallel. There are also saturated forms of these instructions. The C674x core enhances the .S unit in several ways. On the previous cores, dual 16-bit MIN2 and MAX2 comparisons were only available on the .L units. On the C674x core they are also available on the .S unit which increases the performance of algorithms that do searching and sorting. Finally, to increase data packing and unpacking throughput, the .S unit allows sustained high performance for the quad 8-bit/16-bit and dual 16-bit instructions. Unpack instructions prepare 8-bit data for parallel 16-bit operations. Pack instructions return parallel results to output precision including saturation support.

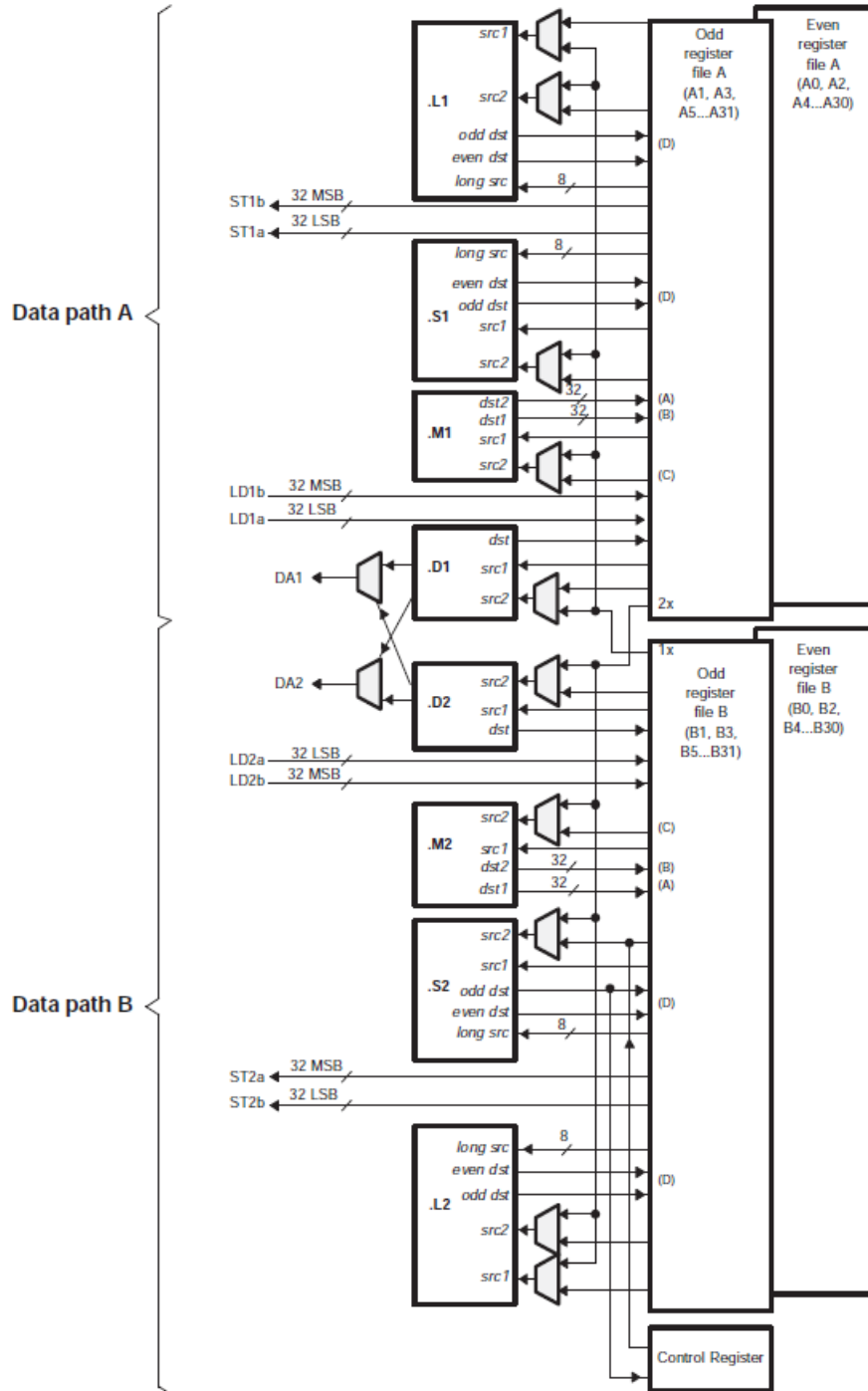


Figure 1.4: C674x CPU Data Paths. Source: [3]



Priority	Interrupt Name	Interrupt Type
Highest	Reset	Reset
	NMI	Nonmaskable
	INT4	Maskable
	INT5	Maskable
	INT6	Maskable
	INT7	Maskable
	INT8	Maskable
	INT9	Maskable
	INT10	Maskable
	INT11	Maskable
	INT12	Maskable
	INT13	Maskable
	INT14	Maskable
	INT15	Maskable
Lowest		

Figure 1.5: Interrupts on the C674x processor. Source: [4]

## 1.2.2 Interrupts

This section introduces the interrupt mechanism of the C6478 processor. This is just an introduction. Complete documentation about the interrupt mechanism can be found in the *TMS320C674x CPU and Instruction Set Reference Guide* [4] and the *TMS320C674x DSP Megamodule Reference Guide* [5].

There are four types of interrupts on the CPU (Figure 1.5): reset, maskable, nonmaskable and exception. The reset interrupt has the highest priority and is used to halt the processor and return it to a known state. The nonmaskable interrupts (NMI) have the second highest priority and are in general used to indicate hardware problems. The lowest priority interrupts are interrupts 4-15 corresponding to the INT4-INT15 signals. These interrupts can be associated with external devices, on-chip peripherals or software control.

### Interrupt selector

The C674x system provides a large set of system events. The interrupt selector (IS) provides a way to select the necessary events and route them to the appropriate CPU interrupt. The interrupt controller supports up to 128 system events. Most of the examples used in this course make explicit use of the following two events for interrupt and DMA-based I/O respectively.

- Event #61: MCASP0\_INT, MCASP0 combined RX/TX interrupt ;
- Event #8: EDMA3\_0\_CC0\_INT1, EDMA3 channel controller 0 shadow region 1 transfer completion interrupt.

The interrupt selector in the C674x routes any of the 128 events to one of the 12 maskable CPU interrupts, as shown in Figure 1.6. The routing implemented by the selector is determined by the content of three Interrupt Mux Registers (INTMUX1 to INTMUX3).

The IS contains interrupt multiplexing registers, INTMUX[3:1] that allow the user to program the source of each of the 12 available CPU interrupts. Each of the events that are presented to the IS has an event number that is used to program these registers. The

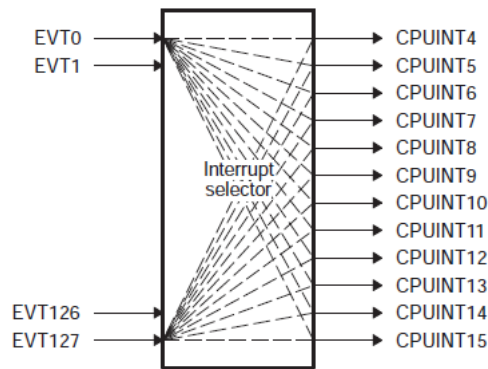


Figure 1.6: Interrupt selector block diagram. Source: [5]

order of the CPU interrupts (CPUINT4 to CPUINT15) determines the priority for pending interrupts.

In this course, the `MCASP0_INT` event is routed to CPUINT4. This is achieved by writing the event number into the seven LSBs of register `INMUX1` using the statement

```
INTC_INTMUX1 = 0x3D.
```

In addition, interrupts must of course be generated and enabled. This means that we need to setup the McASP controller to generate an interrupt on transmit and receive, that is, when a sample is ready to be read or sent to the ADC or the DAC. For the event #61, this is achieved by setting bit 5 in the McASP0 transmitter interrupt control register `XINTCTL`. More details about the configuration of the McASP can be found in the *TMS320C6000 DSP Multichannel Audio Serial Port (McASP) Reference Guide* [2]. This is achieved by the statement

```
MCASP->XINTCTL = 0x00000020; // interrupt on transmit.
```

Finally, INT4 has to be enabled in the CPU by setting bit 4 in the Interrupt Enable Register (IER). In order to interrupt the CPU when INT4 happens, the Global Interrupt Enable (GIE) bit in the Control Status Register (CSR) must also be set. These two configurations are done using the following statements.

```
IER |= 0x12; // enable INT4 and NMI
CSR |= 0x01; // enable interrupt globally
```

### Interrupt Service Table (IST)

The interrupt service routines executed when interrupts occur are determined by the content of the IST. When the CPU begins to process an interrupt, it references the IST which is a table of fetch packets that contain code for servicing the interrupts. The IST consists of 16 consecutive fetch packets (shown in Figure 1.7). Each interrupt service fetch packet (ISFP) contains up to 14 instructions. Thus, a simple interrupt service routine may fit in an individual fetch packet. In general, however, an ISFP contains a branch instruction to a interrupt function in the code of the program. In the examples of this course, the ISFP for INT4 causes a branch to function `interrupt4()`. The content of the IST are set up in assembly language in the file `vectors_intr.asm`.

xxxx 00h	RESET ISFP
xxxx 02h	NMI ISFP
xxxx 04h	Reserved
xxxx 06h	Reserved
xxxx 08h	INT4 ISFP
xxxx 0A0h	INT5 ISFP
xxxx 0C0h	INT6 ISFP
xxxx 0E0h	INT7 ISFP
xxxx 100h	INT8 ISFP
xxxx 120h	INT9 ISFP
xxxx 140h	INT10 ISFP
xxxx 160h	INT11 ISFP
xxxx 180h	INT12 ISFP
xxxx 1A0h	INT13 ISFP
xxxx 1C0h	INT14 ISFP
xxxx 1E0h	INT15 ISFP

Program memory

Figure 1.7: Interrupt Service Table (IST). Source: [4]

### 1.3 OMAP L-138 Experimenter Board

This course is built around an experimenter board using a Texas Instrument OMAP-L138 processor. The Logic PD Zoom OMAP-L138 eXperimenter kit (Figure 1.8) is a low-cost development platform for the OMAP-L138 processor. This device is a dual-core system on chip comprising a TMS320C6748 digital signal processor and an ARM926EJ-S general purpose processor (GPP). In addition, a number of peripherals and interfaces are built into the OMAP-L138, as shown in Figure 1.9.

As this course is concerned with the development of real-time digital signal processing applications, we only make use of the DSP (C6748) side of the device and of the TLC320AIC3106 (AIC3106) analog circuit (codec) connected to the OMAP-L138's multichannel audio serial port (McASP). The ARM side of the device is not used in this course.

The onboard codec (AIC3106) uses sigma-delta technology that provides analog-to-digital conversion (ADC) and digital-to-analog conversion (DAC). It uses a 24.576 MHz system clock and its sampling rate can be selected from a range of settings from 8 kHz to 96 kHz. The board also includes 128 MB of synchronous dynamic RAM and 8 MB of NOR flash memory. Two 3.5 mm jack connectors provide analog input and output. There are also eight user DIP switches that can be read within a program running on the processor, providing thus a simple mean of user interaction. Two LEDs can also be controlled within a program, providing basic output interaction.

The board is connected to the host PC running the Code Composer Studio IDE via an XDS100v1 JTAG emulation built into the experimenter. The software can be written in C or assembly language to be compiled / assembled, linked and downloaded to the C6748. Usage of Code Composer Studio is described in Chapter 2.

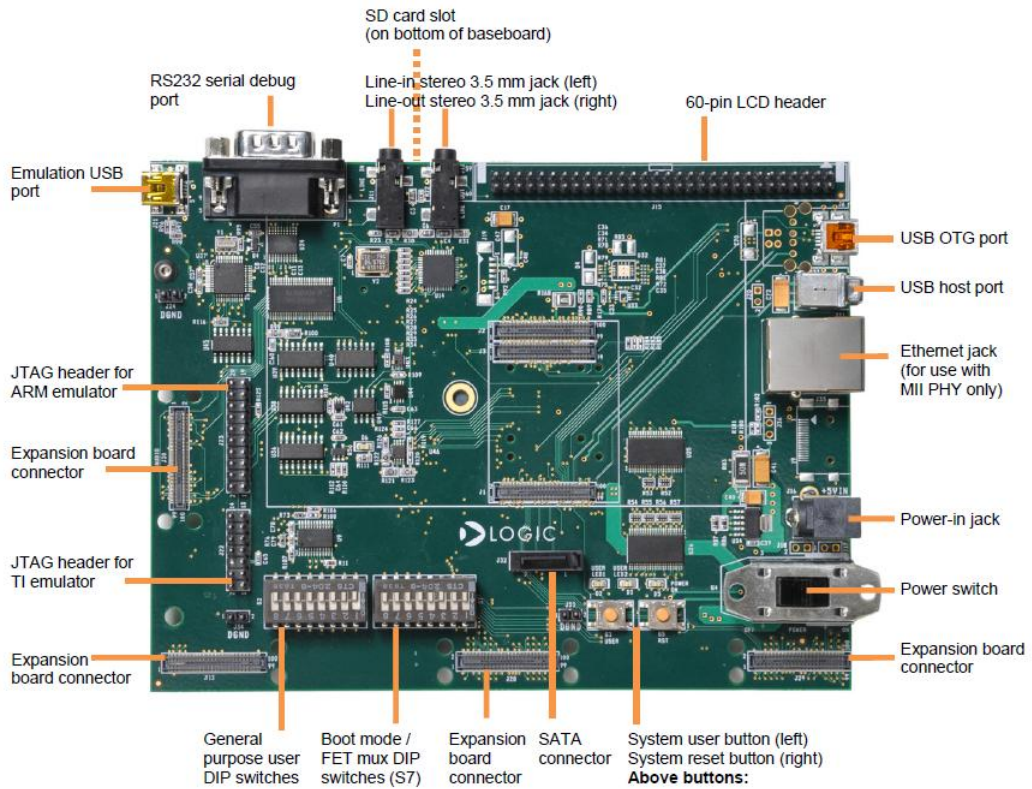


Figure 1.8: OMAP-L138 Experimenter Board. Source: Texas Instrument website.

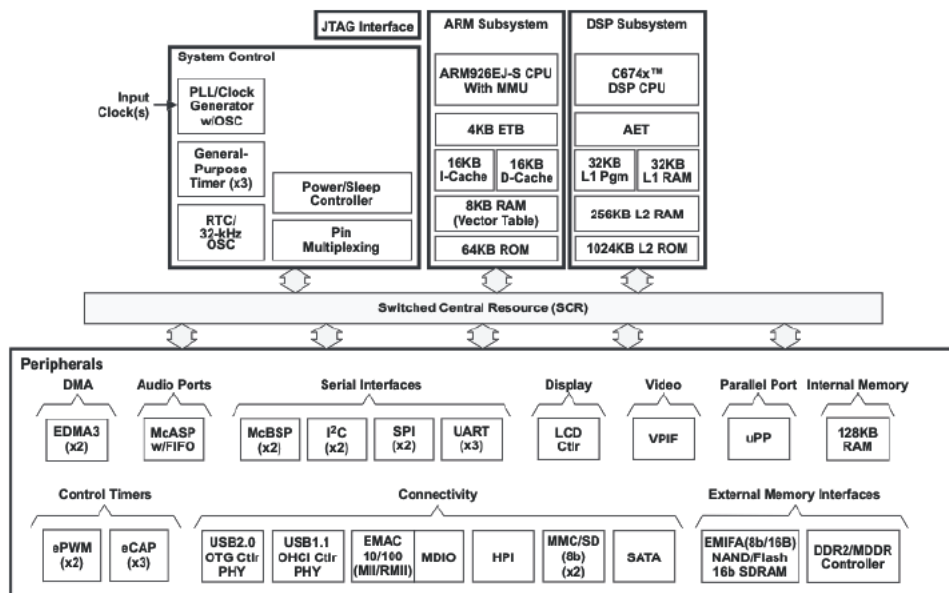


Figure 1.9: OMAP-L138 Experimenter board architecture. Source: [7]

## Chapter 2

# Development environment: Code Composer Studio

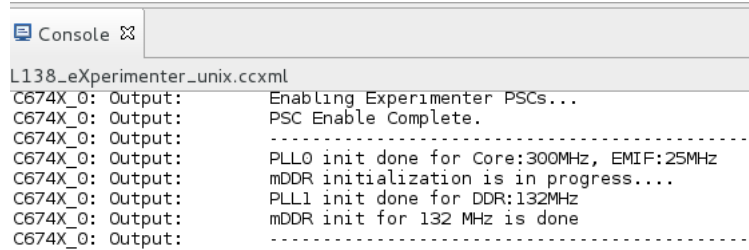
Code Composer Studio (CCS) is the integrated development environment (IDE) provided by Texas Instrument. It is based on the Eclipse framework and therefore requires a Java Runtime Environment (JRE). This chapter details the setup of CCS in order to be able to program the EVM board OMAP-L138 and to run programs on the DSP. Some specific files are needed to communicate with the DSP and to program it. These files are stored in the folder `C:\DSP` in your workstation. This folder contains configuration files and libraries useful to program the EVM board.

- The *bsl* folder contains the *board support library* which proposes functions specific to the evaluation board.
- The *gel* contains configuration files which allow you to connect to the board.
- The *omap* folder contains C libraries and assembly code to program audio applications on the EVM.
- The *dsplib* folder contains a DSP library offering numerous useful functions for DSP applications.
- The *doc* folder contains documentation files and datasheets related to the DSP and its peripherals.

### 2.1 First connection to the EVM

The following procedure allows you to connect to the OMAP-L138 experimenter using Code Composer Studio.

1. Connect the board to the computer using the provided USB connector. The USB programming port is placed near the RS232 port on the board.
2. Each DIP switch must be set to off, except switches #5 and #8 on the second dip switch (Closer to the on/off button).



```

L138_eXperimenter_unix.ccxml
C674X_0: Output:      Enabling Experimenter PSCs...
C674X_0: Output:      PSC Enable Complete.
C674X_0: Output:      -----
C674X_0: Output:      PLL0 init done for Core:300MHz, EMIF:25MHz
C674X_0: Output:      mDDR initialization is in progress....
C674X_0: Output:      PLL1 init done for DDR:132MHz
C674X_0: Output:      mDDR init for 132 MHz is done
C674X_0: Output:      -----

```

Figure 2.1: Connection to the OMAP-L138 EVM.

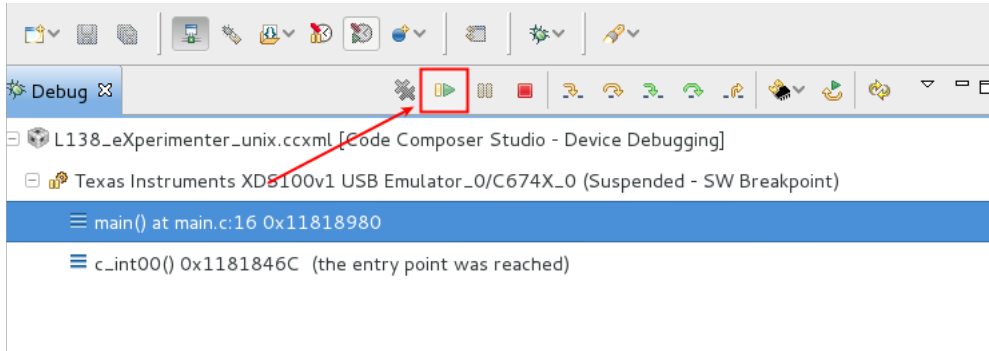


Figure 2.2: Start a program on the DSP.

3. Connect the analog audio input and output to a source and a receiver and turn on the DSP.
4. Start CCS and chose a workspace corresponding to your student group.
5. Select the panel *Target Configurations* in the menu *View* → *Target Configurations*.
6. In that panel, if it is not already the case, import the file *L138\_eXperimenter.ccxml* stored in the folder *C:\DSP\omap* and set it to default by right-clicking on it.
7. Right click on the imported target configuration in the *User Defined* folder and start the connection (*Launch Selected Configuration*). The debugger window should show up.
8. Right click on *Texas Instruments XDS100v1 USB Emulator\_0/C674X\_0 (Disconnected : Unknown)* connect a target (*Connect Target*). A console should appear (see Figure 2.1).
9. Select the menu *Run* → *Load* → *Load Program* and load the file *C:\DSP\pass\Debug\AllPass.out*.
10. Start the program by clicking on the green arrow (see Figure 2.2).
11. Start the audio source. The signal should pass through the DSP without any processing and should be heard on the receiver.
12. Stop the program by clicking on the red square.

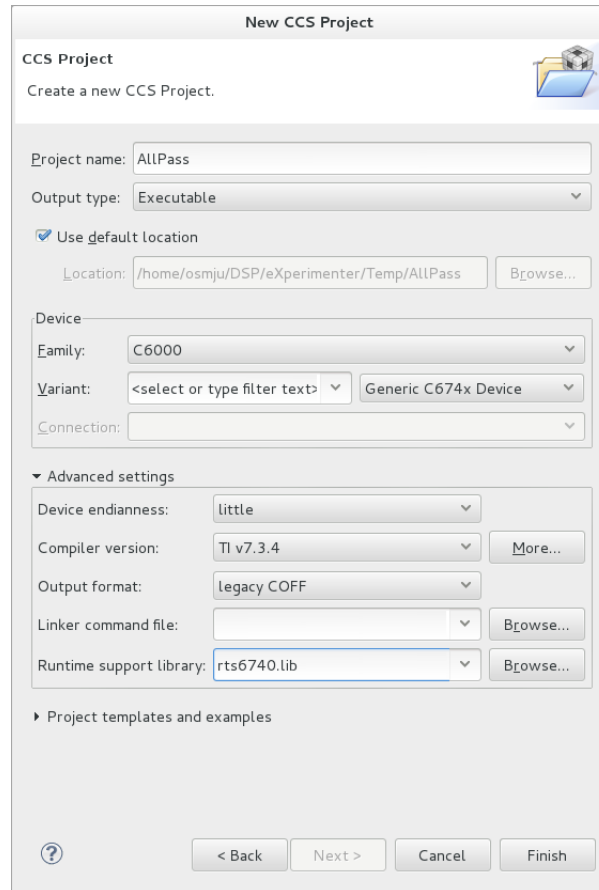


Figure 2.3: Create a new CCS project.

## 2.2 Create a project

Now that the connection to the DSP is effective, it is possible to create projects to program the DSP in C language. To configure a project, we need to include some specific files in order to read the audio samples and to manage CPU interrupts. It is also necessary to set the parameters of the project to include the headers and libraries. This section explains all the necessary steps.

1. Activate the C/C++ perspective using the button at the top right of the screen (*CCS Edit*).
2. Create a new project in *File* → *New* → *CCS Project*. The figure 2.3 should appear.
3. Configure the parameters as showed in the Figure 2.3.
4. Open the project properties in the menu *Project* → *Properties*.
5. In the include options (*Include Options*), add the folder *C:\DSP\omap* and the folder *C:\DSP\bsl\inc*, as showed in the Figure 2.4.
6. In the linker options, add the library file *bsl\lib\evmomapl138\_bsl.lib* in *File Search Path*.
7. Add the following files to the project by right clicking on the project (*Add Files*) :

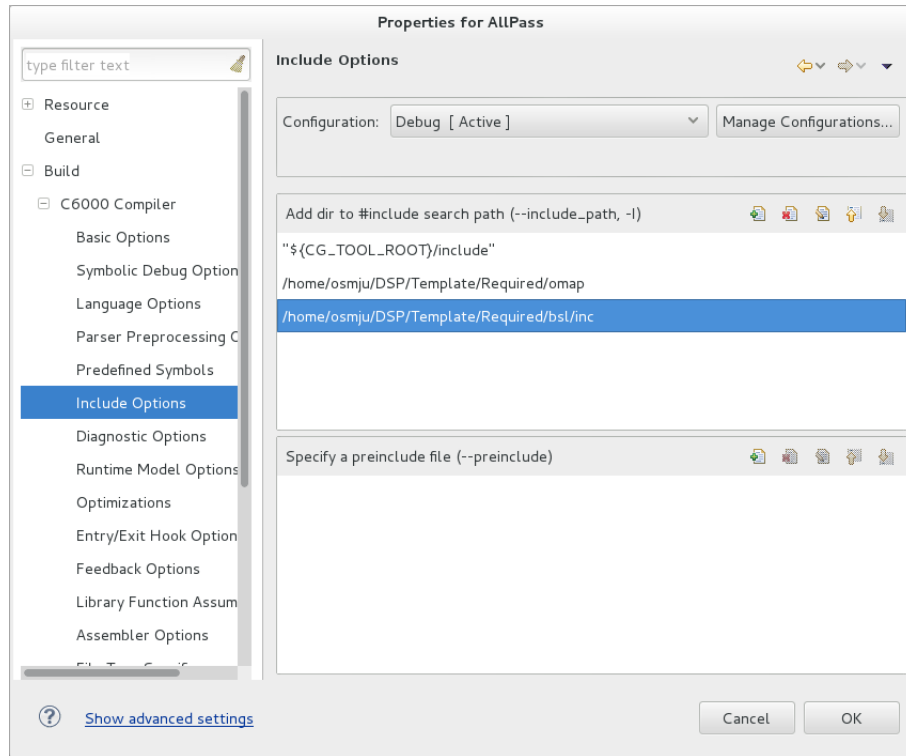


Figure 2.4: Properties of a CCS project.

- (a) C:\DSP\omap\L138\_aic3106\_init.c
  - (b) C:\DSP\omap\L138\_aic3106\_init.h
  - (c) C:\DSP\omap\linker\_dsp.cmd
  - (d) C:\DSP\omap\vectors\_intr.asm
8. Replace the content of *main.c* by the example code of *AllPass.c* (Figure 2.5). This code will be explained later.
  9. Compile the code by clicking on the appropriate button (hammer button) and run it by starting the debugger as showed in the Figure 2.6.

```

1 #include "L138_aic3106_init.h"
2
3 interrupt void interrupt4(void) {
4     uint32_t sample;
5     sample = input_sample();
6     output_sample(sample);
7     return;
8 }
9
10 void main(void) {
11     L138_initialise_intr(FS_48000_HZ, ADC_GAIN_0DB, DAC_ATTEN_0DB);
12     while (1);
13 }

```

Figure 2.5: Example code : AllPass.c





Figure 2.6: Compile and run the program.

10. In the debugger view, start the program by clicking on the green arrow, as showed in the Figure 2.2.

## Chapter 3

# Input / Output

In general, a DSP is composed of a processor and analog interfaces, as showed in the Figure 3.1. The OMAP-L138 eXperimenter proposes such a system with a Texas Instrument C6748 floating point processor and a TLV320AIC3106 (AIC3106) codec. The codec allows to convert the analog signal to a digital signal (ADC: Analog to Digital Converter) and to convert a digital signal back to an analog one (DAC: Digital to Analog Converter). The codec communicates with the DSP via a multichannel serial port (McASP) and I2C interfaces. The latter are available to a user on the board, thus allowing the user to connect a different codec rather than the default one.

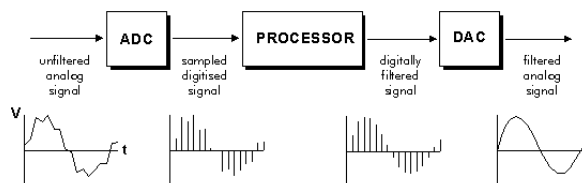


Figure 3.1: General DSP system.

The AIC3106 codec is a low-power device designed for embedded applications. It contains a number of configurable microphone and line inputs, as well as a number of outputs. The codec can sample the analog signal at frequencies between 8 and 96 kHz. The analog-to-digital converter converts an analog signal to a sequence of 16, 24 or 32-bit signed samples (integers), which can next be processed by the CPU. The digital-to-analog converter does the invert operation. In practice, on the evaluation board used in this course, there is one stereo input line and one stereo output line.

There are three possible input / output schemes to access the audio samples into the OMAP-L138 experimenter. Each of these allows to transfer samples from the AIC3106 codec to the DSP C6748. Two techniques, using polling, or interrupts bring the data sample by sample. In the former case, it is the responsibility of the program to test whether a new sample is available at the input. In the latter case, an interrupt is triggered each time a new sample is available, that is, at each sampling period. Of course, the processing of the sample has to occur within a sampling period, otherwise, there will be an overrun and the output will be set to zero. A third method, based on DMA, transfers blocks of samples directly from the codec to the memory without CPU intervention. This

allows to transfer a large amount of samples while the CPU processes another block of samples. That method will be discussed in details in Chapter 5. That last method is particularly useful for frame-based signal processing, for example the FFT algorithm.

### 3.1 Data format

The AIC3106 codec is configured to convert samples of both left and right channels to 16-bit signed integers. Both channels are then combined together to form 32-bit samples which are next sent to the processor via the McASP interface. Access to the ADC and the DAC is made through the following functions.

- `int32_t input_sample()`
- `int16_t input_left_sample(), int16_t input_right_sample()`
- `void output_sample(int32_t out_data)`
- `void output_left_sample(int16_t out_data)`
- `void output_right_sample(int16_t out_data)`

Fonctions `input_sample()` and `output_sample()` read and write both left and right channels. The other functions allows to manipulate each channel separately. The samples are received in a union structure. This allows to access either one 32-bit integer containing both left and right channels, or to access one channel at a time.

```

1 typedef union {
2     uint32_t uint;
3     short channel[2];
4 } AIC31_data_type;
```

This union defines a type `AIC31_data_type` which can be used to store 32-bits values read from the codec. That union is defined in the file `L138_aic3106_init.h`. A variable of that type can access each channel using the constants `LEFT` and `RIGHT` defined in the same file. The following code excerpt reads the samples corresponding to the left and right channels.

```

1 AIC31_data_type codec_data;
2 codec_data.uint = input_sample();
3 short left_sample = codec_data.channel[LEFT];
4 short right_sample = codec_data.channel[RIGHT];
```

Note that the samples can be converted to floating point values using a cast operator:  
`float sample = (float)left_sample;`

```

1 #include "L138_aic3106_init.h"
2
3 void main(void) {
4     uint32_t sample;
5
6     L138_initialise_poll(FS_48000_HZ, ADC_GAIN_0DB, DAC_ATTEN_0DB);
7     while (1) {
8         sample = input_sample();
9         output_sample(sample);
10    }
11 }

```

Figure 3.2: Main loop of a polling DSP program.

## 3.2 Polling

In that communication scheme, the processor has to query the codec at regular time intervals to check whether a new sample is available or not. Thus, at each sampling period, the DSP queries the codec and if a sample is available, it transfers it to its internal registers via the McASP serial port. Figure 3.2 shows the main loop of a program using polling to read the samples.

Functions `L138_initialise_poll()`, `input_sample()` et `output_sample()` are defined in the file `L138_aic3106_init.c`. The initialization function sets the DSP registers and configures the multichannel serial port McASP. This function calls some other low-level functions which will not be described in this course. Some of these functions are defined in the board support library of the EVM board (`evmomapl138_bsl.lib`). The initialization function `L138_initialise_poll()` takes three parameters: the sampling frequency, the ADC gain, and the DAC attenuation. The functions `input_sample()` and `output_sample()` allow to transfer pairs of samples (left and right channels) between the C6748 and the AIC3106. In the polling mode, the `input_sample()` function tests the ready bit (RRDY) of the McASP control register 12 (SRCTL12) until the latter notifies that new samples are ready to be read and to be received in the register XRBUF12. The `output_sample()` tests the transmission bit (XRDY) of the McASP control register 11 (SRCTL11) until it notifies that the codec is ready to receive a new output sample. The output sample will next be sent to the XRBUF11 register. Note that in this mode, the processing has to be done within a sampling period, otherwise some samples are missed and the output becomes unpredictable.

To use the polling mode, you need to include the following files to your project :

- `L138_aic3106_init.h`
- `L138_aic3106_init.c`
- `linker_dsp.cmd`
- `vectors_poll.asm`

```

1 #include "L138_aic3106_init.h"
2
3 interrupt void interrupt4(void) { // interrupt service routine
4     uint32_t sample;
5
6     sample = input_sample();
7     output_sample(sample);
8     return;
9 }
10
11 void main(void) {
12
13     L138_initialise_intr(FS_48000_HZ, ADC_GAIN_0DB, DAC_ATTEN_0DB);
14     while (1) ;
15 }

```

Figure 3.3: Main loop of a interrupt-based DSP program.

### 3.3 Interrupts

In that scheme, an CPU interrupt is triggered when a new sample is ready to be read. Most of the examples used in this course use that interrupt-based technique for real-time I/O. Figure 3.3 shows an example of interrupt-driven program.

As explained in Section 1.2.2, when a new sample arrives to the McASP serial port, event #61 MCASP0\_INT is triggered. The DSP is configured to route this event to CPUINT4, which is one of the 12 non-maskable interrupts. The Interrupt Service Fetch Packet (ISFP) corresponding to INT4 contains one single branch instruction to the function `interrupt4()`. Therefore, each time a new sample is available, the CPU is interrupted and the function `interrupt4()` is executed. This function is shown in Figure 3.3. The function has to retrieve the samples and can process them. In this example, it only reads the input samples and writes them back to the output. If you execute this code, the signal will simply pass through the DSP without any modification.

Since the CPU is interrupted at each sampling period, the main function is very simple. Indeed, it contains a single call to the initialization function, and then it starts an infinite loop. The loop will next be interrupted at each sampling period. Note that if the processing algorithm is too much time consuming, it will be interrupted at the next sampling period, leading again to unpredictable behavior.

To use the polling mode, you need to include the following files to your project :

- L138\_aic3106\_init.h
- L138\_aic3106\_init.c
- linker\_dsp.cmd
- vectors\_intr.asm

### 3.4 Direct Memory Access (DMA)

The C6748 processors has a Direct Memory Access module (EDMA3, for Enhanced Direct Memory Access) that transfers blocks of samples between memory zones. The main ad-

vantage of using DMA against polling or interrupt driven processing, is that it can transfer data without any intervention of the CPU. Therefore, the DMA module can be configured to be synchronized with the codec and transfer the samples directly in a buffer in the memory. While the DMA module transfers the data, the CPU can process another buffer. When the DMA finishes its transfer of N samples, it triggers an interrupt so that the CPU knows that new data is available. Thus, instead of triggering an interrupt at each new sample, the CPU is interrupted only when a block of N samples has been transferred.

DMA-based processing will be covered in detail in Chapter 5.

## 3.5 Volatile variables

When programming embedded hardware such as a DSP, it is common to have global variables that can be modified by an interrupt service routine. At any time, the value of such a variable may change without any action being taken by the code the compiler finds nearby. For example, the `interrupt4` routine used in the above examples could modify a global variable that is later used by the main program. Such specific global variables must be declared with the keyword `volatile`.

### 3.5.1 Syntax

To declare a variable volatile, just include the keyword `volatile` before or after the data type of the variable definition. For example, the following declarations will declare the variable `foo` to be a volatile integer. Note that it is also possible to declare a pointer to a volatile variable.

```
1 volatile int foo; // variable
2 int volatile foo;
3
4 volatile int* foo; // pointer
5 int volatile* foo;
```

### 3.5.2 Use

A variable should be declared volatile whenever its value could change unexpectedly. In practice, only three types of variables could change.

- Memory-mapped peripheral registers;
- Global variables modified by an interrupt service routine;
- Global variables within a multi-threaded application.

Only the two first cases are relevant to this course. In practice, you will only have to deal with the second type of variables. The first case is handled by the library used in the course. Both cases are described below.

### 3.5.2.1 Peripheral registers

Embedded systems such as a DSP board contain hardware, usually with sophisticated peripherals. These peripherals contain registers whose values can change asynchronously to the program flow. Consider for example a 8-bit status register at address 0x1234. An typical example would be to poll the status register until it becomes non-zero. An incorrect implementation is given below.

```

1 int* ptr = (int*)0x1234;
2
3 // wait for register to become non-zero.
4 while (*ptr == 0);
5 // do something else.
```

This code will fail as soon as you turn the optimizer on. The compiler will generate assembly language that looks like this:

```

1 mov ptr, #0x1234    mov a,@ptr    loop    bz loop
```

Having already read the variable's value into the accumulator, there is no need to read it again since the value will always be the same, according to the written code. Therefore, the loop will end up infinite because the compiler does not know that the variable can be modified by an external peripheral. To make that code work properly, we need to modify it as follows.

```

1 volatile int* ptr = (volatile int*)0x1234;
2
3 // wait for register to become non-zero.
4 while (*ptr == 0);
5 // do something else.
```

And the assembly language will look like this.

```

1     mov ptr, #0x1234
2 loop mov a,@ptr
3     bz loop
```

### 3.5.2.2 Interrupt service routines

Interrupt service routines often set variables that are tested in the main code. For example, the McASP serial port generates an interruption when a new sample is ready to be read. The interrupt service routine might set a global flag to be tested by the main code.

```
1 int sample_ready = FALSE;
2
3 interrupt void interrupt4(void) {
4     ...
5     if (new_sample_arrived) {
6         sample_ready = TRUE;
7     }
8     ...
9 }
10
11 void main() {
12     ...
13     while (!sample_ready) {
14         // wait
15     }
16     ...
17 }
```

With the optimization turned off, this code might work correctly. However, with the optimization set, the compiler has no idea that **sample\_ready** can be changed within an ISR. As far as the compiler is concerned, the expression **!sample\_ready** is always true and therefore, the loop can never be exited. Consequently, the code after the while loop may be removed by the optimizer. The solution to that problem is to declare the variable **sample\_ready** to be **volatile**. Then, the optimizer will take that into account and will not discard the loop code.



# Chapter 4

## Filtering

Filtering is one of the most common DSP operations. Filtering can be used for noise suppression, signal enhancement, removal or attenuation of a specific frequency, or to perform special operations such as differentiation, integration, etc. Filters can be thought of, designed and implemented in either the time domain or the frequency domain. There are two main types of filtering techniques. The first kind of filters are the Finite Impulse Response (FIR) filters. Their non-recursive version directly implements the convolution operation with a finite impulse response. The second kind of filters are the Infinite Impulse Response Filters (IIR). These filters are always recursive as they use feedback information to compute their output.

This chapter is a small introduction to filtering. This is not an exhaustive course on filter design. It gives some information on how to implement digital filters in an efficient way on a DSP.

### 4.1 Finite Impulse Response (FIR) Filters

FIR filters are filters whose impulse response is of finite duration. These filters are very stable, in contrast to IIR filters (See Section 4.2). Most FIR filters are non-recursive and are carried-out by convolution. Thus, the output  $y$  of a linear time invariant (LTI) system is determined by convolving its input signal  $x$  with its impulse response  $h$ . For a discrete-time filter, the output is therefore a weighted sum of the current sample and a finite number of previous input samples. Equation 4.1 describes this operation. The impulse response of a  $N$ th-order FIR filter has a length of  $N + 1$  samples.

$$y[n] = \sum_{i=0}^N h[i]x[n-i] = h[0]x[n] + h[1]x[n-1] + \dots + h[N]x[n-N] \quad (4.1)$$

Figure 4.1 shows the block diagram of a non-recursive FIR filter. The diagram is a graphical representation of Equation 4.1.

#### 4.1.1 Windowed-sinc filters

A common category of FIR filters are windowed-sinc filters. These filters are used to separate one band of frequencies from another. They are very stable and can be pushed

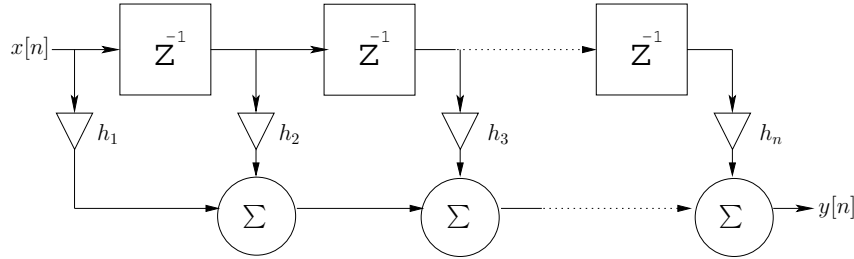


Figure 4.1: Block diagram of a non-recursive FIR filter.

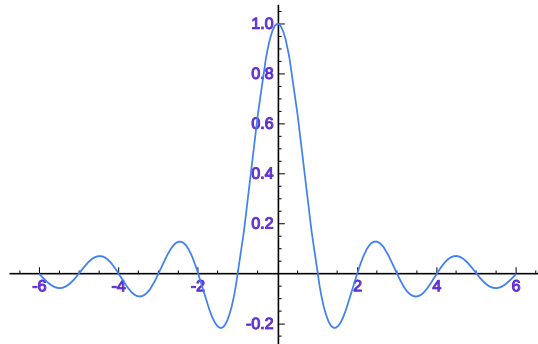


Figure 4.2: Windowed sinc filter kernel.

to great performance levels. The ideal filter kernel (impulse response) of a windowed-sinc filter is shown in Figure 4.2. The curve is of the general form  $\sin(x)/x$ , called the sinc function. More precisely, Equation 4.2 is interesting. This function is obtained by applying the inverse Discrete Fourier Transform (DFT) to an ideal low-pass or high-pass filter. For a low-pass filter, use the + sign and for a high pass filter, use the - sign.  $f_c$  corresponds to the cutoff frequency, expressed as a fraction of the sampling rate.

$$h[i] = \pm \frac{\sin(2\pi f_c i)}{i\pi} \quad (4.2)$$

However, the impulse response described by Equation 4.2 has 3 problems.

- It involves a division by 0 for  $i = 0$ . Therefore, the following values must be adopted:

$$\begin{cases} h[0] = 2f_c & \text{for a low-pass filter} \\ h[0] = 1 - 2f_c & \text{for a high-pass filter} \end{cases}$$

- It continues to both  $+\infty$  and  $-\infty$  without dropping to zero (infinite length).
- It has non-zero values  $h[i]$  for  $i < 0$  (non-causal filter).

Therefore, to be able to use the sinc filter on a computer, we first need to truncate the filter to  $M + 1$  points, symmetrically chosen around the main lobe, where  $M$  is an even number. All samples outside these  $M + 1$  points are set to zero, or simply ignored. Next, the entire sequence of samples is shifted to the right so that it runs from 0 to  $M$ . This allows the filter kernel to be represented using only positive indexes. This modified kernel however does not have an ideal frequency response. Indeed, there is excessive ripple in the

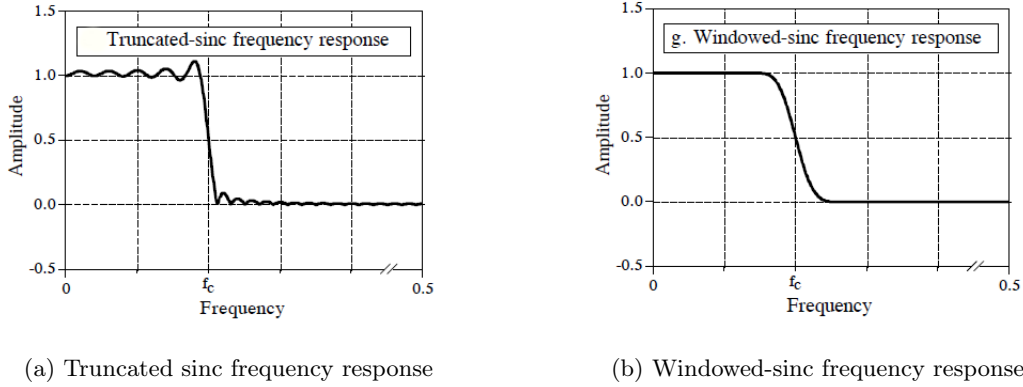


Figure 4.3: Windowed sinc frequency responses. Source: [9]

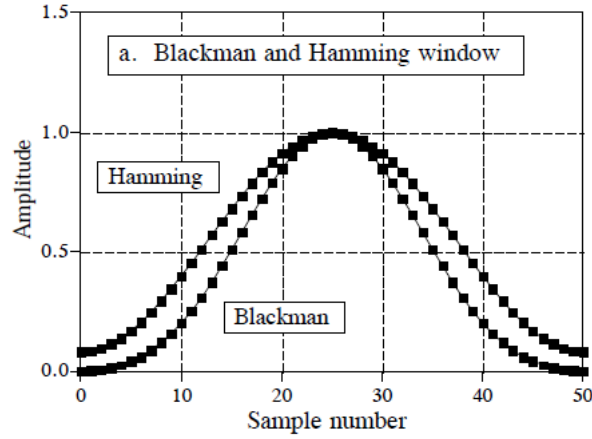


Figure 4.4: Blackman and Hamming windows. Source: [9]

passband and poor attenuation in the stopband, as shown in Figure 4.3a. These problems result from the abrupt discontinuity at the ends of the truncated sinc function.

To improve the filter's frequency response, the kernel can be multiplied by a window in the time domain, for example, the **Blackman window**. This actually results in the **windowed-sinc** filter kernel. The windowing allows to reduce the abruptness of the truncated ends and therefore improve the frequency response. Figure 4.3b shows the resulting frequency response. The passband is flat and the stopband attenuation is good. Other windows exist. Another common window is the **Hamming window**. Both Blackman and Hamming windows are shown in Figure 4.4. The equations of the Hamming and Blackman windows are respectively given by equation 4.3 and equation 4.4.

$$w[i] = 0.54 - 0.46\cos(2\pi i/M) \quad (4.3)$$

$$w[i] = 0.42 - 0.5\cos(2\pi i/M) + 0.08\cos(4\pi i/M) \quad (4.4)$$

#### 4.1.2 Windowed-sinc filters design

To design a windowed-sinc filter, two parameters must be selected: the cutoff frequency,  $f_c$ , and the length of the filter kernel,  $M$ . The cutoff frequency is expressed as a fraction of

the sampling rate, and must therefore be between 0 and 0.5. The value  $M$  sets the roll-off according to the approximation

$$M \approx \frac{4}{BW} \quad (4.5)$$

where  $BW$  is the width of the transition band, which is also expressed as a fraction of the sampling frequency. Note that the time required for a convolution is proportional to the length of the impulse response. Therefore, Equation 4.5 expresses a trade-off between computation time and filter sharpness.

Once the value of  $M$  and  $BW$  are computed, the final equation of a low-pass filter, multiplied by the Blackman window is given by the following equation.

$$h[i] = \frac{\sin(2\pi f_c(i - M/2))}{\pi(i - M/2)} \left[ 0.42 - 0.5\cos\left(\frac{2\pi i}{M}\right) + 0.08\cos\left(\frac{4\pi i}{M}\right) \right] \quad (4.6)$$

To design a *band-pass* filter, we need to convolve a low-pass filter with a high-pass filter, which corresponds to the multiplication of both frequency transfer functions. To design a *band-reject* filter, we need to add a low-pass filter to a high-pass filter, which corresponds to the addition of both frequency transfer functions.

## 4.2 Recursive Filters

Recursive filters are an efficient way of achieving a long impulse response, without having to perform a long convolution. They execute very rapidly, but have less performance and flexibility than other digital filters. Recursive filters are also called *Infinite Impulse Response* (IIR) filters, since their impulse responses are composed of decaying exponentials. This distinguishes them from digital filters carried out by convolution, described in Section 4.1.

Recursive filters have an equation of the following general form.

$$y[n] = a_0x[n] + a_1x[n - 1] + a_2x[n - 2] + \dots + b_1y[n - 1] + b_2y[n - 2] + \dots \quad (4.7)$$

or

$$y[n] = \sum_{k=0}^M a_k x[n - k] - \sum_{l=1}^N b_l y[n - l] \quad (4.8)$$

Each sample in the output signal is found by multiplying the values from the input signal by the “a” coefficients, multiplying the previously calculated values from the output signal by the “b” coefficients, and adding the products together. Note that there isn’t a value for  $b_0$  because it corresponds to the sample being calculated. Equation 4.7 is called the **recursion equation**. The “a” and “b” values that define the filter are called the **recursion coefficients**. Recursive filters are useful because they bypass a longer convolution.

### 4.2.1 Narrow-band filters

A common need in electronics and DSP is to isolate a narrow band of frequencies from a wider bandwidth signal. For example, we may want to eliminate 50 hertz interference in an instrumentation system, or isolate signaling tones in a telephone network. Two types of frequency responses are available: the band-pass and the band-reject (also called

$$\begin{aligned}
 a_0 &= 1 - K \\
 a_1 &= 2(K - R)\cos(2\pi f) \\
 a_2 &= R^2 - K \\
 b_1 &= 2R\cos(2\pi f) \\
 b_2 &= -R^2
 \end{aligned}$$

Figure 4.5: Band-pass 2nd order filter coefficients. Source: [9]

$$\begin{aligned}
 a_0 &= K \\
 a_1 &= -2K\cos(2\pi f) \\
 a_2 &= K \\
 b_1 &= 2R\cos(2\pi f) \\
 b_2 &= -R^2
 \end{aligned}$$

Figure 4.6: Band-reject 2nd order filter coefficients. Source: [9]

notch) filters. Figure 4.7 shows the frequency response of these filters, with the recursion coefficients provided by the following equations. Figure 4.5 gives the equations of a second order band-pass filter, and Figure 4.6 gives the equations of a second order notch filter. In each case, we have

$$K = \frac{1 - 2R\cos(2\pi f) + R^2}{2 - 2\cos(2\pi f)} \quad (4.9)$$

and

$$R = 1 - 3BW \quad (4.10)$$

Two parameters must be selected before using these equations:  $f$ , the center frequency, and  $BW$ , the bandwidth. Both of these must be expressed as a fraction of the sampling frequency, and therefore must be between 0 and 0.5. From these values, we can calculate

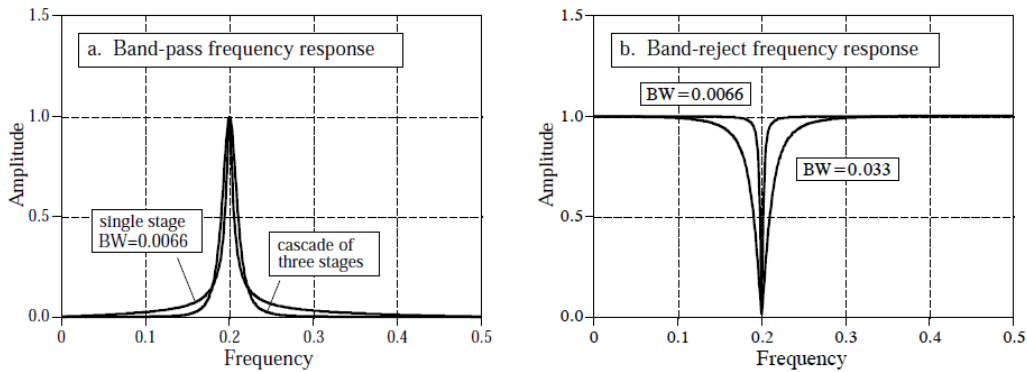


Figure 4.7: Band-pass and band-reject filters frequency responses. Source: [9]

the intermediate variables  $K$  and  $R$  and then the recursion coefficients. Figure 4.7 shows the performances of filters designed using these coefficients. The band-pass has relatively large tails extending from the main peaks. This can be improved by cascading several stages.

## 4.2.2 IIR filters structures

Several different filter structures can be used to represent an IIR filter in a DSP.

### 4.2.2.1 Direct Form I Structure

Direct Form I structure is shown in Figure 4.8. With this structure, the filter described by equation 4.7 can be realized. For an  $N$ th-order filter, this structure contains  $2N$  delay elements, each represented by  $Z^{-1}$ . For example, a second-order filter with  $N = 2$  will contain four delay elements. Direct Form I is the most straightforward implementation of the recursive equation of an IIR filter.

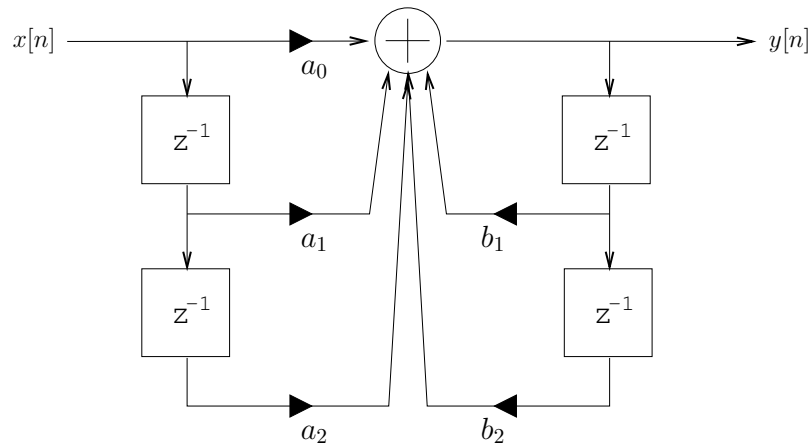


Figure 4.8: IIR Filter Direct Form I

### 4.2.2.2 Direct Form II Structure

The Direct Form II structure is shown in Figure 4.9. It is one of the most commonly used structures. It requires half as many delay elements as in the direct form I. For example, a second-order filter requires two delay elements  $Z^{-1}$ , as opposed to four with the direct form I.

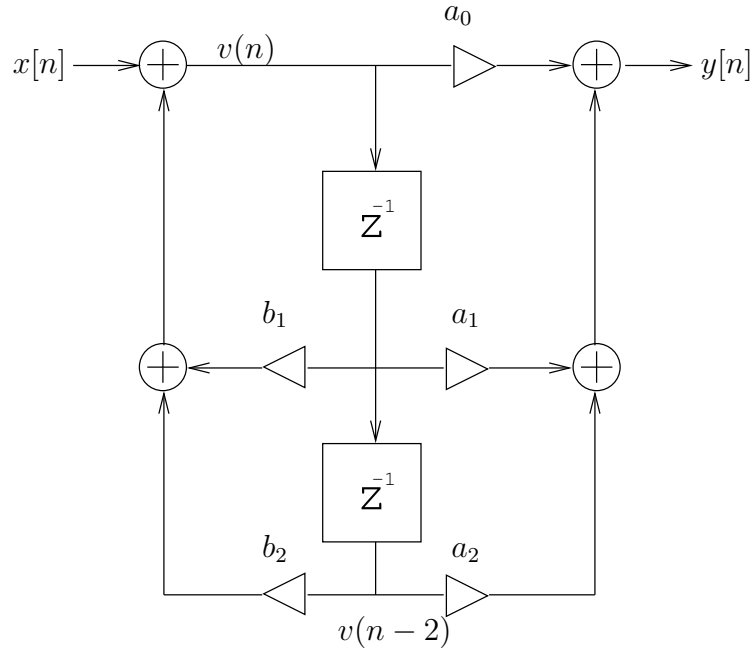


Figure 4.9: IIR Filter Direct Form II

From the block-diagram, it can be seen that

$$v[n] = x[n] + b_1v[n-1] + b_2v[n-2] + \cdots + b_Nv[n-N] \quad (4.11)$$

and that

$$y[n] = a_0v[n] + a_1v[n-1] + a_2v[n-2] + \cdots + a_Nv[n-N] \quad (4.12)$$

### 4.2.2.3 Cascade structure

A transfer function  $H(z)$  can be factorized as

$$H(z) = CH_1(z)H_2(z)\cdots H_r(z) \quad (4.13)$$

An overall transfer function can be represented with cascaded transfer functions. For each section, the direct form II structure or its transpose version can be used. Typically, an  $N$ th order filter is decomposed in cascaded second-order sections. The transfer function  $H(z)$  in terms of cascaded second-order transfer functions can be written as

$$H(z) = \prod_{i=1}^{N/2} \frac{b_{0i} + b_{1i}z^{-1} + b_{2i}z^{-2}}{1 + a_{1i}z^{-1} + a_{2i}z^{-2}} \quad (4.14)$$

For example for a 4th order filter, we have

$$H(z) = \frac{(b_{01} + b_{11}z^{-1} + b_{21}z^{-2})(b_{02} + b_{12}z^{-1} + b_{22}z^{-2})}{(1 + a_{11}z^{-1} + a_{21}z^{-2})(1 + a_{12}z^{-1} + a_{22}z^{-2})} \quad (4.15)$$

The quantization error associated with the coefficients of an IIR filter depends on the amount of shift in the position of its poles and zeros in the complex plane. This implies that the shift in the position of a particular pole depends on the positions of all the other poles. To minimize this dependency of the poles, an  $N$ th order IIR filter is typically implemented as cascaded second-order sections.

**In practice** Matlab's `fdatool` function is capable of designing an IIR filter and outputs its coefficients in second-order sections formats (SOS). Matlab outputs the coefficients in two matrices. The first one contains one row per second-order section and the second matrix contains gains for each stage. To output the coefficients in a suitable format, the function `L138_iirsos_coeffs.m` provided with this course creates a file that you can include in your C programs. The function automatically multiplies the coefficients by the provided gains.

When programming the DSP, keep in mind that the output of each section is the input of the next section. To compute the output of one section, simply use the direct form II structure, as explained in Section 4.2.2.2.



## Chapter 5

# Frame-based signal processing

### 5.1 Frame-based vs Sample-based processing

Until now, the real-time processing of the signal was made on a sample-by-sample basis. Thus, an input sample  $x(t)$  was converted to its digital form  $x[n]$  by the ADC and transferred to the CPU for processing. Then the output sample  $y[n]$  was computed and then converted by the DAC to its analog form  $y(t)$ . Although this processing scheme has its advantages, as it minimizes the system latency by acting on each sample as soon as it is available, it also has serious drawbacks.

One implication of real-time sample-based DSP is that all the processing must be completed in the time between two samples. For some DSP algorithms, it becomes difficult, especially for fast sampling rates. For example, for a sampling frequency  $FS = 48\text{ kHz}$ , we only have  $T_s = 1/(48 \times 10^3) = 20.83\mu\text{s}$  to process both the left and the right channel samples, or  $10.42\mu\text{s}$  per sample. In the OMAP-L138 experimenter, the processor has a frequency of 300 MHz, which gives about 3126 clock cycles per sample. Note that there is also an overhead time associated with codec transfers, memory access, instruction and data cache latency and other factors. Thus, complex algorithms with many memory transfers could easily exceed this maximum number of clock cycles.

Another implication is that only one sample is available for processing. Some class of DSP algorithms, such as the Fast Fourier Transform (FFT) for example, require more than one sample for processing. Indeed, a number of contiguous samples has to be available at any given time, which is impossible when using a sample-by-sample scheme. Moreover, processing each sample individually interrupts the CPU at each sampling period. Thus, for each sample, the state of the processor must be saved in order to be restored, control state must also be preserved and then execution must be restarted at the interrupted point. This interrupt process greatly reduces the performance of the DSP. To remove this limitation, hardware components, referred to as direct memory access (DMA) controllers are usually included as peripherals of DSP systems. Once the DMA controller is programmed to respond to a device that is sourcing or sinking the data, it will automatically perform the required transfers to or from a memory buffer without any processor intervention. When a buffer has been filled or emptied, the DMA controllers interrupts the DSP. Thus, the DSP is freed from the task of data transfers and it can use all its resources to be focused on the computationally-intense processing once a buffer of data is available.

The buffer typically contain hundreds or thousands of samples. Such a block of samples is called a *frame*. Note that in order to keep a real-time constraint, the size of the frames

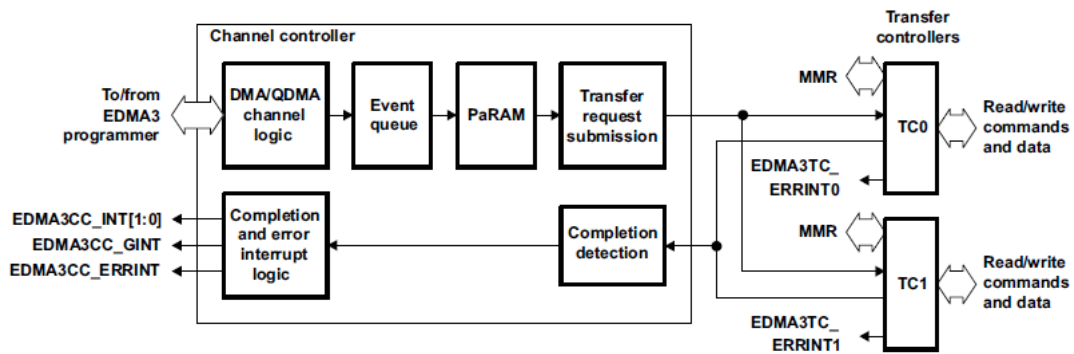


Figure 5.1: Architecture of EDMA3 controller. Source: [6]

should be carefully selected. Indeed, as the frame length increase, the latency of the system will also increase. This depends of course on the sampling frequency. By default, in the examples of this course, the frame size is set to 2048 at a sampling frequency  $FS = 48 kHz$ . The latency using these settings is minimal, while allowing the DSP to process a great number of samples. Therefore, the latency should be computed according to the application.

## 5.2 Direct Memory Access

Frame-based signal processing can be easily enhanced by using Direct Memory Access on the OMAP-L138 processor. Direct Memory Access (DMA) allows to transfer data from one area of memory to another without CPU intervention, and therefore without any CPU instruction, nor interruption. Therefore, during a DMA transfer, the CPU can perform other processing tasks while data is being moved between memory areas. This improves the computational efficiency of a DSP application.

In the OMAP-L138 processor, DMA is implemented using the enhanced direct memory access controller (EDMA3). Detailed information about the EDMA3 controller can be found in the *TMS320C6000 Peripheral Guide* [1]. This section explains how the DMA transfers are being handled for signal processing.

### 5.2.1 Architecture

The EDMA3 controller of the OMAP-L138 processor is made of a channel controller and a transfer controller. The channel controller is just a user interface to the transfer controller, which is responsible for reading and writing data from / to memory. The channel controller keeps track of the number of concurrent DMA transfers and schedules reads and writes to be carried by the transfer controller. The EDMA3 controller contains two channel controllers, EDMA3\_CC0 and EDMA3\_CC1, each capable of handling up to 32 DMA transfers concurrently. A DMA transfer is associated to an EDMA3 channel in the C6748 processor. Thus, there are 32 EDMA3 channels available. The architecture of the EDMA3 controller is summarized in Figure 5.1.

EDMA3 Channel Controller 0			
Event	Event Name / Source	Event	Event Name / Source
0	McASP0 Receive	16	MMCS0 Receive
1	McASP0 Transmit	17	MMCS0 Transmit
2	McBSP0 Receive	18	SPI1 Receive
3	McBSP0 Transmit	19	SPI1 Transmit
4	McBSP1 Receive	20	PRU_EVTOUT6
5	McBSP1 Transmit	21	PRU_EVTOUT7
6	GPIO Bank 0 Interrupt	22	GPIO Bank 2 Interrupt
7	GPIO Bank 1 Interrupt	23	GPIO Bank 3 Interrupt
8	UART0 Receive	24	I2C0 Receive
9	UART0 Transmit	25	I2C0 Transmit
10	Timer64P0 Event Out 12	26	I2C1 Receive
11	Timer64P0 Event Out 34	27	I2C1 Transmit
12	UART1 Receive	28	GPIO Bank 4 Interrupt
13	UART1 Transmit	29	GPIO Bank 5 Interrupt
14	SPI0 Receive	30	UART2 Receive
15	SPI0 Transmit	31	UART2 Transmit
EDMA3 Channel Controller 1			
Event	Event Name / Source	Event	Event Name / Source
0	Timer64P2 Compare Event 0	16	GPIO Bank 6 Interrupt
1	Timer64P2 Compare Event 1	17	GPIO Bank 7 Interrupt
2	Timer64P2 Compare Event 2	18	GPIO Bank 8 Interrupt
3	Timer64P2 Compare Event 3	19	Reserved
4	Timer64P2 Compare Event 4	20	Reserved
5	Timer64P2 Compare Event 5	21	Reserved
6	Timer64P2 Compare Event 6	22	Reserved
7	Timer64P2 Compare Event 7	23	Reserved
8	Timer64P3 Compare Event 0	24	Timer64P2 Event Out 12
9	Timer64P3 Compare Event 1	25	Timer64P2 Event Out 34
10	Timer64P3 Compare Event 2	26	Timer64P3 Event Out 12
11	Timer64P3 Compare Event 3	27	Timer64P3 Event Out 34
12	Timer64P3 Compare Event 4	28	MMCS1 Receive
13	Timer64P3 Compare Event 5	29	MMCS1 Transmit
14	Timer64P3 Compare Event 6	30	Reserved
15	Timer64P3 Compare Event 7	31	Reserved

Figure 5.2: List of synchronization events associated with EDMA3 channels. Source: [3]

### 5.2.2 DMA transfer

To perform a DMA transfer, a number of parameters must be set into the EDMA3 controller. For a transfer to happen, at least a source address and a destination address as well as the number of bytes to be transferred should be specified. Moreover, the event triggering a DMA transfer must be programmed. In addition, it is necessary to determine how the completion of the transfer will be signaled to a program running in the CPU. DMA transfers can either be event-triggered, meaning that the transfers are initiated by peripheral devices or external hardware, or manually-triggered, meaning that the transfers are initiated by the CPU writing to the 32-bit EDMA3 event set register (ESR).

**Event-triggered transfers** In this case, a peripheral initiates the DMA transfers. Each transfer is associated with an EDMA3 channel and each channel is in turn associated with a synchronization event. The list of synchronization events is shown in Figure 5.2.

In this course, the exercises and projects make use of event #0 McASP0 Receive and event #1 McASP0 Transmit, which are synchronized with the operations of the AIC3106 codec.

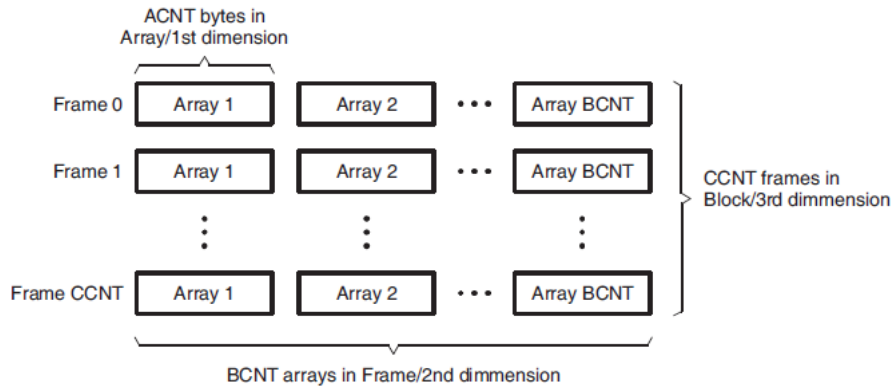


Figure 5.3: Three-dimensional data transferred by the EDMA3 controller. Source: [6]

Synchronization events cause bits in the EDMA3 Event Register (ER) to be set and then, DMA transfers will be triggered if the corresponding bit in the EDMA3 Event Enable Register (EER) is set.

**Manually-triggered transfers** In this case, the CPU writes to the Event Set register (ESR) to manually trigger transfers, regardless of the bits set in the EER. Each bit of the ESR corresponds to one of the 32 EDMA3 channels in the corresponding channel controller. For example, setting bit 5 in the ESR will manually trigger a transfer on channel #5.

**Data transfer** The data transferred by the EDMA3 controller are a three-dimensional block of *CCNT* frames of *BCNT* arrays containing each *ACNT* bytes, as shown in Figure 5.3. Transfers can be configured as either A-synchronized, in which case each an event triggers the transfer of a single array of *ACNT* bytes, or AB-synchronized, in which case each event triggers the transfer of a single frame of *BCNT* arrays of *ACNT* bytes. In this course, the examples use A-synchronized transfers with  $ACNT = 4$ , which corresponds to two 16-bit samples (left and right channels) read from or written to the AIC3106 codec,  $BCNT = Bufcount/2$ , where *Bufcount* corresponds to a constant, and  $CCNT = 1$ . Therefore, each EDMA3 transfer moves a block of *Bufcount* 16-bit samples (left and right) per sampling period. Indeed, each McASP0 event starts the transfer of two 16-bit samples ( $ACNT = 4$  bytes = 32 bits) and  $Bufcount/2$  McASP0 events are required to complete one block transfer. Note that the McASP0 events occur at the AIC3106 codec sampling rate. For realtime I/O, two concurrent DMA transfers are needed: one from the ADC to memory and another onf from memory to the DAC.

**EDMA3 parameter RAM** Each DMA transfer is specified by a set of parameters stored in a 32-byte block of parameter RAM (PaRAM). There is one block of PaRAM per EDMA3 channel. There is one block for each of the 32 channels and 96 blocks which can be used for parameter set linkage (explained later), for a total of 128 PaRAM sets. Figure 5.4 shows the format of the parameter RAM of the EDMA3 controller. The fields of a parameter RAM set are described below.

- **Channel options (OPT)** This word of 32 bits specifies the configuration of a transfer. The relevant bits are bit 20 TCINTEN, bits 17-12 TCC and bit 2 SYNCDIM.

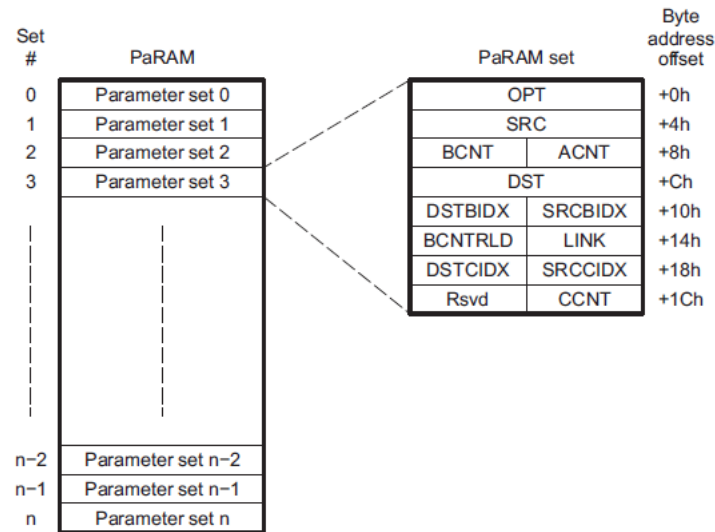


Figure 5.4: Format of the parameter RAM. Source: [6]

- **Channel source (SRC) and desination (DST)** This word specifies the addresses of the source and destination of the data to be transferred.
- **ACNT, BCNT, CCNT** Unsigned 16-bit values representing the number of bytes in an array, the number of arrays and the number of frames.
- **SRCBIDX** Address offset between the arrays in a source frame.
- **DSTBIDX** Address offset between the arrays in a destination frame.
- **Link Address (LINK)** This is a 16-bit address of the PaRAM block from which the parameters will be copied when the current parameter set is exhausted. This field is used for Ping-Pong buffering (explained later).
- **BCNTRLD** Reload value of the BCNT field when BCNT has decremented to 0.
- **SRCCIDX** Address offset between the frames in a source block.
- **DSTCIDX** Address offset between the frames in a destination block.

**Starting a transfer** In this course, McASP0 receive or transmit event is used to trigger an EDMA3 transfer. Therefore the EDMA3 is synchronized with the McASP0 and the AIC3106 codec. Therefore, according to Figure 5.2, channels #0 and #1 are used and therefore, PaRAM sets #0 and #1 need to be configured. In addition, it is necessary to set the bits 0 and 1 in the Event Enable Register (EER) of the EDMA3 controller.

In the case of an A-synchronized transfer, after a triggering event, BCNT is decremented by 1, SRC is incremented by SRCBIDX and DST is incremented by DSTBIDX. When BCNT reaches 0, the A-synchronized transfer is completed. If TCINTEN bit in OPT field is set, then a CPU event is caused and the bit TCC in the EDMA3 event pending register (EPR) is set. A CPU interrupt is triggered only if the system event caused by a transfer completion is routed to a CPU interrupt through the interrupt selector, specified in INTMUXn. The interrupt must be enabled in the register IER. For example, in this

course system event #8 EDMA3\_0\_CC0\_INT1 is routed to CPU interrupt 4 by writing the value 8 in INTMUX1.

Once a transfer is completed, if the LINK field is not set to null, then the PaRAM set is refreshed by copying into it the content of the PaRAM set specified in the LINK field. This allows to switch buffers very easily as explained below.

## 5.3 DMA in practice

### 5.3.1 Ping-Pong buffering

For real-time signal processing, we use a ping-pong buffering scheme in this course. This allows to enhance the performance of a program. In that kind of scheme, two sets of input and output buffers are used. While the ping input buffer is being filled with data coming from the codec, and the ping output buffer is sent to the DAC, the content of the pong input buffer are processed and the results are stored in the pong output buffer. Thus, while the EDMA3 controller transfers data to some buffers, the CPU processes other buffers, maximizing the performance. Once a transfer is completed, the buffers are swapped.

This scheme is implemented in the OMAP-L138 using the EDMA3 linking mechanism. Two parameter sets PaRAM #0 and #1 are configured and two more sets are configured to update each channel. Thus, initially, the SRC and DST fields for PaRAM set #0 are set to McASP0 input and the ping input buffer, and the LINK field is set to another PaRAM set (for example PaRAM #36), in which the SRC and DST fields are set to the McASP0 input and pong input buffer. Therefore, on completion of one transfer from the ADC to the ping input buffer, the parameters are updated to transfer data from ADC to pong input buffer instead of the ping one. This allows to implement easily the ping-pong buffering scheme.

### 5.3.2 Project organization

The files needed to create a DMA based project are stored in the folder C:\DSP\DMA. To use DMA, just create a project as explained in Chapter 2, then add the files `isr.c`, `prototypes.h`, `L138_aic3106_init_edma.c`, `L138_aic3106_init_edma.h` and the linker and interrupt vector files as in Chapter 2.

The DMA initialization function, `EDMA3_PaRAM_setup()`, is defined in the file `isr.c`. This file also contains the declaration of the PING and PONG buffers and the definition of initialization functions, as well as the interrupt function called each time a buffer is full. It also contains the function `process_buffer()` in which the processing must be implemented. The size of the buffer is defined in the constant `BUFCOUNT`, in the file `isr.c`. By default, the size is set to 1024. Note that the real size of a stereo buffer is twice the size defined in `BUFCOUNT`. Thus, a ping or pong buffer has a real size of  $BUFLNGTH = BUFCOUNT \times 2$ . The buffers are stored in external SDRAM, allowing a maximum of 128 Mb of data storage. Keep in mind that a bigger buffer will create more latency and thus decrease the real-time efficiency of the program.

The `interrupt4()` function simply switches the pointers of the Ping / Pong buffers. It also sets a flag `buffer_full = 1`. When this flag is set to 1, the function `process_buffer()`

```

1 int main(void) {
2     L138_initialise_edma(FS_48000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
3     zero_buffers();
4
5     while(1) {
6         while (is_buffer_full());
7         process_buffer();
8     }
9 }

```

Figure 5.5: Main function in `main.c`.

```

1 #define BUFCOUNT 1024
2 #define BUFLNGTH (BUFCOUNT * 2)
3
4 #pragma DATA_SECTION(ping_IN, ".EXT_RAM")
5 #pragma DATA_SECTION(pong_IN, ".EXT_RAM")
6 #pragma DATA_SECTION(ping_OUT, ".EXT_RAM")
7 #pragma DATA_SECTION(pong_OUT, ".EXT_RAM")
8
9 int16_t ping_IN[BUFLNGTH];
10 int16_t pong_IN[BUFLNGTH];
11 int16_t ping_OUT[BUFLNGTH];
12 int16_t pong_OUT[BUFLNGTH];
13
14 int16_t *pingIN = &ping_IN[0];
15 int16_t *pingOUT = &ping_OUT[0];
16 int16_t *pongIN = &pong_IN[0];
17 int16_t *pongOUT = &pong_OUT[0];
18
19 volatile int buffer_full = 0;
20 int procBuffer;

```

Figure 5.6: DMA buffers declarations in `isr.c`.

is executed and the processing is actually done. Do not forget to reset the flag at the end of the processing function to avoid unpredictable behavior.

The `process_buffer()` function defines two local pointers for the Ping or Pong buffers. It then checks which buffer to use. Next it processes the signal. By default, it simply transfer the samples from the input buffer to the output buffer. The code of any filter must be programmed in that function.

Figure 5.6 shows the initialization of the buffers in the file `isr.c`. The `pragma` directives inform the compiler that the ping and pong buffers should be created in external memory.

Figure 5.7 shows the interrupt routine which checks whether the Ping or Pong buffer should be used next. This routine then sets the buffer full flag to 1 so that the processing can be done. Finally, Figure 5.8 shows the processing function.

This file organization allows to have a very simple `main.c` file, which only initializes the buffers and the DSP and starts the main loop. At each loop iteration, we check the `buffer_full` flag. When it is set to 1, the `process_buffer` is executed. The main function is shown in Figure 5.5.

```

1 interrupt void interrupt4(void) { // interrupt service routine
2
3     switch(EDMA_3CC_IPR) {
4
5         case 1:                // TCC = 0
6             procBuffer = PING; // process ping
7             EDMA_3CC_ICR = 0x0001; // clear EDMA3 IPR bit TCC
8             break;
9
10        case 2:                // TCC = 1
11            procBuffer = PONG; // process pong
12            EDMA_3CC_ICR = 0x0002; // clear EDMA3 IPR bit TCC
13            break;
14
15        default:                // may have missed an interrupt
16            EDMA_3CC_ICR = 0x0003; // clear EDMA3 IPR bits 0 and 1
17            break;
18    }
19
20    EVTCLR0 = 0x00000100;
21    buffer_full = 1; // flag EDMA3 transfer
22    return;
23 }

```

Figure 5.7: DMA Interrupt routine in `isr.c`.

```

1 void process_buffer(void) {
2
3     int16_t *inBuf, *outBuf; // pointers to process buffers
4     int16_t left_sample, right_sample;
5     int i;
6
7     if (procBuffer == PING) { // use ping or pong buffers
8         inBuf = pingIN;
9         outBuf = pingOUT;
10    }
11
12    if (procBuffer == PONG) {
13        inBuf = pongIN;
14        outBuf = pongOUT;
15    }
16
17    /* process buffer here */
18    for (i = 0; i < (BUFCOUNT) ; i++) { // simple pass through
19
20        left_sample = *inBuf++;
21        right_sample = *inBuf++;
22
23        *outBuf++ = left_sample;
24        *outBuf++ = right_sample;
25    }
26
27    buffer_full = 0; // indicate that buffer has been processed
28    return;
29 }

```

Figure 5.8: Process buffer function in `isr.c`.



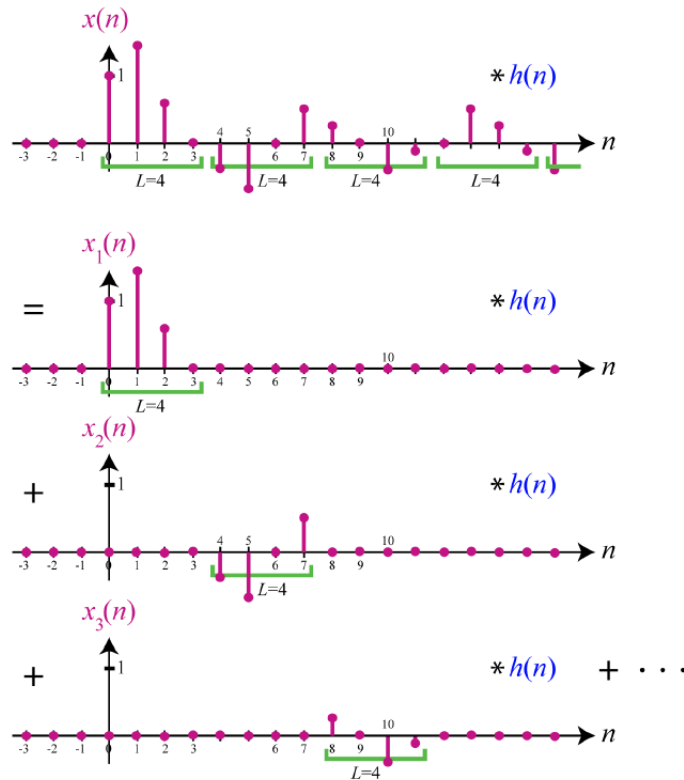


Figure 5.9: Basic idea of the Overlap-Add method.

## 5.4 Frame-based convolution

This section explains the *overlap-add* algorithm to compute convolutions by frames in a DSP. In real-time signal processing, an input signal is constantly fed into the DSP to be processed, and then the associated output is ready without delay (at least, not perceptible delay). If we have a long input signal that continues in time indefinitely, we do not want to wait until the entire input signal has terminated before filtering it. This would be equivalent to doing something “off-line” after all the input is available. Moreover, the DSP does not have an infinite memory to store the signal. So instead, we would like to filter the blocks of input as they come in.

To achieve such a block based filtering, we need to use an adapted version of the convolution algorithm, named the overlap-add algorithm. This process can then be enhanced by using the Fast Fourier Transform, which will be explained in Chapter 6.

### 5.4.1 Overlap-Add

The Overlap-Add method is based on the observation that when we consider two discrete-time signals  $x_k[n]$  and  $h[n]$ , with lengths  $L$  and  $M$  respectively, the resulting convolution  $y_k[n] = x_k[n] * h[n]$  has a length of  $L + M - 1$ . Using this idea, we can divide the input stream  $x[n]$  into  $L$ -length blocks and convolve each block with  $h[n]$ , and then sum all the convolution outputs along the  $L$ -boundaries, as shown in Figure 5.9.

Using the EDMA controller described in the previous sections, we have an easy way of splitting the input stream into fixed-length input blocks. Therefore, we can process each

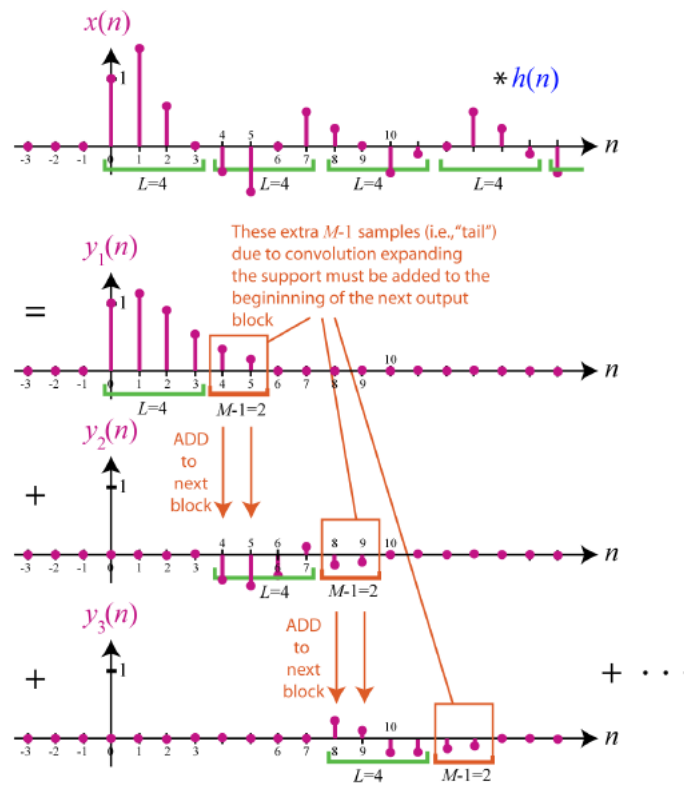


Figure 5.10: Tail resulting from convolution must be added to the next convolved block.

block separately using the overlap-add algorithm.

In Figure 5.9, the operation of convolving a very long signal  $x[n]$  with  $h[n]$  is equivalent to the operation of convolving each  $L$ -block denoted  $x_k[n]$  with  $h[n]$  and then conducting addition judiciously to deal with the “tail” region from each block convolution, as it will be explained next. An important aspect is that after convolving each block with  $h[n]$ , the resulting intermediate signal is  $L + M - 1$  samples in length, and therefore, we have  $M - 1$  extra samples at the end due to the convolution. These  $M - 1$  extra samples must be added to the first  $M - 1$  samples of the *next* convolved block, as shown in Figure 5.10<sup>1</sup>.

One main challenge in a real-time processing scenario is that the timing of completing a block convolution needs to be approximately synchronized with the overall output speed so that tail region may be added to the next block at the right time. If the process of convolving each block is slower than outputting the samples of blocks already convolved, then the tail region will not have the opportunity to be added to the next block, resulting in an erroneous output. One way to deal with this is to slow down the rate of the output. However, another much more attractive way is to make use of the Fast Fourier Transform for convolutions. This will be described in the next chapter.

<sup>1</sup>Images from this section come from <http://www.comm.utoronto.ca/~dkundur/course/real-time-digital-signal-processing>.

## Chapter 6

# Fast Fourier Transform

Fourier analysis describes the transformations between time and frequency domain representations of signals. Four different forms of Fourier transformations (the Fourier transform (FT), the Fourier Series (FS), the discrete-time Fourier transform (DTFT) and the discrete Fourier transform (DFT)) are applicable to different classes of signals according to whether they are discrete or continuous and whether they are periodic and aperiodic.

The DFT is the form of Fourier Transform analysis applicable to signals that are discrete and periodic in both domains (time and frequency). Thus, it transforms a discrete, periodic time domain sequence into a discrete, periodic frequency domain representation. A periodic signal may be characterized entirely by just one cycle. If that signal is discrete, then one cycle comprises a finite number of samples. Therefore, both forward and inverse DFTs are described by finite summations as opposed to infinite summations or integrals. This is very important in digital signal processing since it means that it is practical to compute the DFT using a digital signal processor, or digital hardware.

The Fast Fourier transform is a computationally efficient algorithm for computing the DFT. It requires fewer multiplications than a more straightforward programming implementation of the DFT and its relative advantage in this respect increases with the length of the sample sequences involved. The FFT makes use of the periodic nature of twiddle factors and of symmetries in the structure of the DFT expression. Applicable to spectrum analysis and to filtering, the FFT is one of the most commonly used operations in digital signal processing.

### 6.1 Discrete Fourier Transform

The Discrete Fourier Transform for a finite segment of a digital signal containing  $N$  samples  $x[0], \dots, x[N-1]$  is defined as follows.

$$X[k] = \langle x, e^{jtk2\pi/N} \rangle = \sum_{t=0}^{N-1} x[t] e^{-jtk2\pi/N} \quad (6.1)$$

which corresponds to the inner product of the input signal with periodic basis functions. Each basis function represents a frequency in the range 0 to the sampling frequency. The inverse DFT expresses the time domain signal as a complex weighted sum of  $N$  phasors with spectrum  $X[k]$ . It is defined as follows.

$$x[t] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] e^{jtk2\pi/N} \quad (6.2)$$

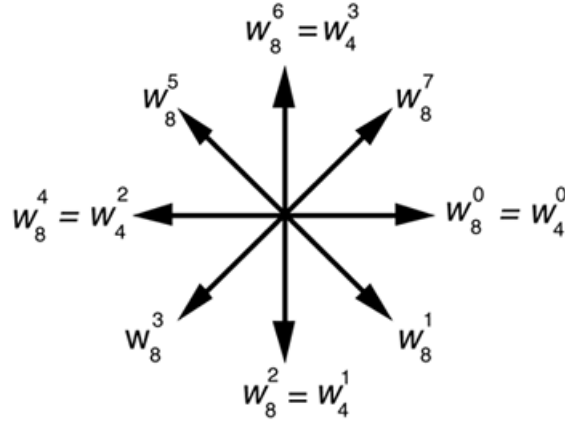


Figure 6.1: Twiddle factors  $W_N^{kn}$  for  $N = 8$  represented as vectors in the complex plane.

The bins of the DFT are numbered  $0, N - 1$  but correspond to frequencies between  $[-\omega_s/2, +\omega_s/2]$ , where  $\omega_s$  corresponds to the sampling frequency. Since  $N$  corresponds to the sampling rate, we need to divide the bin numbers by  $N$  to get the frequencies in terms of fractions of the sampling rate. Thus, to compute the frequency corresponding to a given bin number, we can use the following relation.

$$f = \frac{b_i}{N} f_s$$

where  $b_i$  corresponds to the index of the bin, and  $f_s$  is the sampling frequency.

## 6.2 Fast Fourier Transform

### 6.2.1 Forward FFT

The  $N$ -point complex DFT of a discrete-time signal  $x[n]$  can be written

$$X_N[k] = \sum_{n=0}^{N-1} x[n] W_N^{kn} \quad (6.3)$$

where the constants  $W$  are referred to as *twiddle constants* or *twiddle factors*, where

$$W_N = e^{-j(2\pi/N)} \quad (6.4)$$

Computing all  $N$  values of  $X_N[k]$  involves the evaluation of  $N^2$  terms of the form  $x[n]W_N^{kn}$ , each of which requires a complex multiplication. For larger  $N$ , the computational requirements of the DFT can be very great ( $N^2$  complex multiplications).

The FFT algorithm takes advantage of the periodicity

$$W^{k+N} = W^k \quad (6.5)$$

and symmetry

$$W^{k+N/2} = -W^k \quad (6.6)$$

of  $W_N$  (where  $N$  is even).

```

1 typedef struct {
2     float real;
3     float imag;
4 } COMPLEX;

```

Figure 6.2: Complex structure in `fft.h`.

Figure 6.1 illustrates the twiddle factors  $W_N^{kn}$  for  $N = 8$  plotted as vectors in the complex plane. Due to the periodicity of  $W_N^{kn}$ , the 64 different combinations of  $n$  and  $k$  used in evaluation of Equation 6.3 result in only 8 distinct values. The FFT makes use of this small number of precomputed and stored values rather than computing each one as it is required. Another in which the FFT saves computational effort is by decomposing  $N$ -point DFTs into combinations of  $N/2$ -point DFTs. The decompositions are carried out until 2-point DFTs have been reached, which do not involve any multiplication. This allows to speed up the process a lot.

The FFT is much faster than the classical DFT as its complexity is  $\mathcal{O}(N \log(N))$ , versus  $\mathcal{O}(N^2)$  for the classical DFT.

### 6.2.2 Inverse FFT

The inverse discrete Fourier transform (IDFT) converts a discrete frequency domain sequence  $X_N[k]$  into a corresponding discrete-time domain sequence  $x[n]$ . It is defined as

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X_N[k] W_N^{-kn} \quad (6.7)$$

Comparing this with the forward DFT Equation 6.3, we can see that the forward FFT algorithm described previously can be used to compute the inverse DFT if the following changes are made:

1. Scale the result by  $1/N$ .
2. Replace twiddle factors  $W_N^{kn}$  by their complex conjugates  $W_N^{-kn}$ .

### 6.2.3 FFT in practice

In this course, we make use of a small FFT library provided with the reference book [8]. The library is composed of one `fft.h` available to you in the `omap` folder of your workstation.

The library first defines a complex structure in order to represent the signals as complex vectors. The structure is declared at the beginning of the file `fft.h` as shown in Figure 6.2. Therefore, the first thing to do when using the FFT is to copy the real samples to a complex vector in order to use the library.

Figure 6.3 gives an example of a simple program computing the FFT of a cosine wave form stored in a lookup table. The first step is to set the twiddle factors in a complex array. Then, we can call the function `fft` which takes three parameters: (1) the input vector, (2) the size of the FFT and (3) the twiddle factors vector. Note that the result of the FFT is stored in the input vector. Thus the input data is overwritten with the spectrum of the input signal.

```

1 //
2 // N-point FFT of sequence stored in a lookup table
3 //
4
5 #include <stdio.h>
6 #include <math.h>
7 #include "fft.h"
8
9 #define PI    3.14159265358979
10 #define N     128
11 #define FREQ  800.0
12 #define FS    8000.0
13
14 COMPLEX samples[N];
15 COMPLEX twiddle[N];
16
17 void main() {
18     int n;
19
20     for (n = 0; n < N; ++n) { // set up twiddle factors
21         twiddle[n].real = cos(PI*n/N);
22         twiddle[n].imag = -sin(PI*n/N);
23     }
24
25     for (n = 0; n < N; ++n) { // test frequency
26         samples[n].real = cos(2*PI*FREQ*n/FS);
27         samples[n].imag = 0.0;
28     }
29
30     fft(samples, N, twiddle);
31 }

```

Figure 6.3: Example of the FFT library usage.

In that example, to restore the original signal, we should compute the inverse FFT in order to retrieve the time-domain signal. To do that, we must compute another set of twiddle factors (as explained in the previous section), and then call again the `fft` function. Then it is important to scale down the signal by dividing each time-domain sample by  $N$ .

### 6.3 FFT Convolution

In Section 5.4, we detailed a method to perform convolutions by block rather than waiting for the entire signal. However, computing convolution using the standard equation results in a lot of computational effort. The process can be improved through the use of the FFT in order to speed up the algorithm. Indeed, performing a convolution in the time domain results in a simple *multiplication* in the frequency domain. The procedure to perform a convolution with the FFT is described below.

1. Zero-pad the filter  $h[n]$  with  $K - M$  zeros, where  $K$  is the first power of two greater than  $L + M - 1$ , where  $L$  is the length of one block of data, and  $M$  is the length of the filter  $h$ .
2. Compute the  $K$ -point FFT of the zero-padded filter  $h[n]$  and save it.

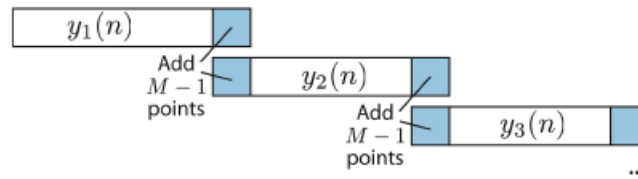


Figure 6.4: FFT overlap-add method.

3. Zero-pad each input segment  $x_k[n]$  of length  $L$  with  $K - L$  zeros to make it the same size as the filter.
4. Compute the  $K$ -point FFT of the segment.
5. Multiply sample-by-sample the two FFT results from Steps 2 and 4.
6. Take the inverse FFT of the resulting product to produce  $y_k[n]$ .

Next, we need to use the same overlap-add algorithm than for the classical convolution. Indeed, the output blocks  $y_k[n]$  need to only be of length  $L$ . Therefore, we need to save the  $K - L$  remaining samples in order to add them to the output of the next processed block. The process is illustrated by Figure 6.4.

# Bibliography

- [1] Texas Instruments. TMS320C6000 Peripherals Reference Guide, February 2001.
- [2] Texas Instruments. TMS320C6000 DSP Multichannel Audio Serial Port (McASP) Reference Guide, November 2008.
- [3] Texas Instruments. TMS320C6748 Fixed- and Floating-Point DSP, June 2009.
- [4] Texas Instruments. TMS320C674x DSP CPU and Instruction Set Reference Guide, July 2010.
- [5] Texas Instruments. TMS320C674x DSP Megamodule Reference Guide, August 2010.
- [6] Texas Instruments. TMS320C6743 DSP Technical Reference Manual, March 2013.
- [7] Logic PD. OMAP-L138 SOM-M1 Hardware Specification, November 2013.
- [8] D. Reay. *Digital Signal Processing and Applications with the OMAP-L138 EXperimenter*. Wiley, 2012.
- [9] S. Smith. *Digital Signal Processing: A practical guide for engineers and scientists*. Elsevier, 2003.
- [10] T. Welch, C. Wright, and M. Morrow. *Real-Time Digital Signal Processing from MATLAB to C with the TMS320C6x DSPs*. CRC Press, 2012.



# List of Figures

1.1	Von Neumann architecture . . . . .	4
1.2	Harvard architecture . . . . .	5
1.3	C674x Megamodule Block Diagram. Source: [3] . . . . .	6
1.4	C674x CPU Data Paths. Source: [3] . . . . .	7
1.5	Interrupts on the C674x processor. Source: [4] . . . . .	8
1.6	Interrupt selector block diagram. Source: [5] . . . . .	9
1.7	Interrupt Service Table (IST). Source: [4] . . . . .	10
1.8	OMAP-L138 Experimenter Board. Source: Texas Instrument website. . . . .	11
1.9	OMAP-L138 Experimenter board architecture. Source: [7] . . . . .	11
2.1	Connection to the OMAP-L138 EVM. . . . .	13
2.2	Start a program on the DSP. . . . .	13
2.3	Create a new CCS project. . . . .	14
2.4	Properties of a CCS project. . . . .	15
2.5	Example code : AllPass.c . . . . .	15
2.6	Compile and run the program. . . . .	16
3.1	General DSP system. . . . .	17
3.2	Main loop of a polling DSP program. . . . .	19
3.3	Main loop of a interrupt-based DSP program. . . . .	20
4.1	Block diagram of a non-recursive FIR filter. . . . .	25
4.2	Windowed sinc filter kernel. . . . .	25
4.3	Windowed sinc frequency responses. Source: [9] . . . . .	26
4.4	Blackman and Hamming windows. Source: [9] . . . . .	26
4.5	Band-pass 2nd order filter coefficients. Source: [9] . . . . .	28
4.6	Band-reject 2nd order filter coefficients. Source: [9] . . . . .	28
4.7	Band-pass and band-reject filters frequency responses. Source: [9] . . . . .	28

4.8	IIR Filter Direct Form I . . . . .	29
4.9	IIR Filter Direct Form II . . . . .	30
5.1	Architecture of EDMA3 controller. Source: [6] . . . . .	33
5.2	List of synchronization events associated with EDMA3 channels. Source: [3] . . . . .	34
5.3	Three-dimensional data transferred by the EDMA3 controller. Source: [6] . . . . .	35
5.4	Format of the parameter RAM. Source: [6] . . . . .	36
5.5	Main function in <code>main.c</code> . . . . .	38
5.6	DMA buffers declarations in <code>isr.c</code> . . . . .	38
5.7	DMA Interrupt routine in <code>isr.c</code> . . . . .	39
5.8	Process buffer function in <code>isr.c</code> . . . . .	39
5.9	Basic idea of the Overlap-Add method. . . . .	40
5.10	Tail resulting from convolution must be added to the next convolved block. . . . .	41
6.1	Twiddle factors $W_N^{kn}$ for $N = 8$ represented as vectors in the complex plane. . . . .	43
6.2	Complex structure in <code>fft.h</code> . . . . .	44
6.3	Example of the FFT library usage. . . . .	45
6.4	FFT overlap-add method. . . . .	46