DSP Lab with TI C6x DSP and C6713 DSK

Collection Editor:
David Waldo

DSP Lab with TI C6x DSP and C6713 DSK

Collection Editor:
David Waldo

Authors:
Douglas L. Jones
David Waldo

Online:
< http://cnx.org/content/col11264/1.6/ >

C O N N E X I O N S

Rice University, Houston, Texas

Collection structure revised: February 18, 2013
PDF generated: August 27, 2013
For copyright and attribution information for the modules contained in this collection, see p. 107.

Table of Contents

1 Assembly Programming Lab

2 C Programming Lab

3 Echo Lab

4 Cosine Generator Lab

5 FIR Lab

6 Noise Cancellation Lab

Chapter 1

Assembly Programming Lab

1.1 C6x Assembly Programming[1]

1.1.1 Introduction

This module contains details on how to program the TI C6000 family of processors in assembly. The C6000 family of processors has many variants. Therefore, it would not be possible to describe how to program all the processors here. However, the basic architecture and instructions are similar from one processor to another. They differ by the number of registers, the size of the registers, peripherals on the device, etc. This module will assume a device that has 32 general-purpose 32-bit registers and eight functional units, like the C6713 processor.

1.1.2 References

- SPRU198: TMS320C6000 Programmer's Guide
- SPRU186: TMS320C6000 Assembly Language Tools User's Guide
- SPRU733: TMS320C67x/C67x+ DSP CPU and Instruction Set Reference Guide

1.1.3 Overview of C6000 Architecture

The C6000 consists of internal memory, peripherals (serial port, external memory interface, etc), and most importantly, the CPU that has the registers and the functional units for execution of instructions. Although you don't need to care about the internal architecture of the CPU for compiling and running programs, it is necessary to understand how the CPU fetches and executes the assembly instructions to write a highly optimized assembly program.

1.1.4 Core DSP Operation

In many DSP algorithms, the Sum of Products or Multiply-Accumulate (MAC) operations are very common. A DSP CPU is designed to handle the math-intensive calculations necessary for common DSP algorithms. For efficient implementation of the MAC operation, the C6000 CPU has two multipliers and each of them can perform a 16-bit multiplication in each clock cycle. For example, if we want to compute the dot product of two length-40 vectors a[n] and x[n], we need to compute:

$$y = \sum_{n=1}^{N} a\left[n\right] x\left[n\right] \tag{1.1}$$

[1] This content is available online at <http://cnx.org/content/m36319/1.1/>.

(For example, the FIR filtering algorithm is exactly the same as this dot product operation.) When an a[n] and x[n] are stored in memory, starting from n=1, we need to compute a[n]x[n] and add it to y (y is initially 0) and repeat this up to n=40. In the C6000 assembly, this MAC operation can be written as:

```
MPY .M a,x,prod
ADD .L y,prod,y
```

Ignore `.M` and `.L` for now. Here, `a`, `x`, `prod` and `y` are numbers stored in memory and the instruction `MPY` multiplies two numbers `a` and `x` together and stores the result in `prod`. The `ADD` instruction adds two numbers `y` and `prod` together storing the result back to `y`.

1.1.4.1 Instructions

Below is the structure of a line of assembly code.

Label:	Parallel bars (||)	[Condition]	Instruction	Unit	Operands	;Comments

Table 1.1

Labels identify a line of code or a variable and represent a memory address that contains either an instruction or data. The first character of a label must be must be in the first column and must be a letter or an underscore (_) followed by a letter. Labels can include up to 32 alphanumeric characters.

An instruction that executes in parallel with the previous instruction signifies this with **parallel bars** (||). This field is left blank for an instruction that does not execute in parallel with the previous instruction.

Every instruction in the C6x can execute conditionally. There are five registers available for conditions (on the C67x processors): A1, A2, B0, B1, and B2. If blank, the instruction always executes. Conditions can take a form such as [A1] where the instruction will execute if A1 is not zero. This can be handy for making loops were the counter is put in a register like A1 and is counted down to zero. The condition is put on the branch instruction that branches back to the beginning of the loop.

1.1.4.2 Register Files

Where are the numbers stored in the CPU? In the C6000, the numbers used in operations are stored in register. Because the registers are directly accessible though the data path of the CPU, accessing the registers is much faster than accessing data in the external memory.

The C6000 CPU has two register files (A and B). Each of these files consists of sixteen 32-bit registers (A0-A15 for file A and B0-B15 for file B). The general-purpose registers can be used for data, data address pointers, or condition registers. The following figure shows a block diagram of the C67x processors. This basic structure is similar to other processors in the C6000 family.

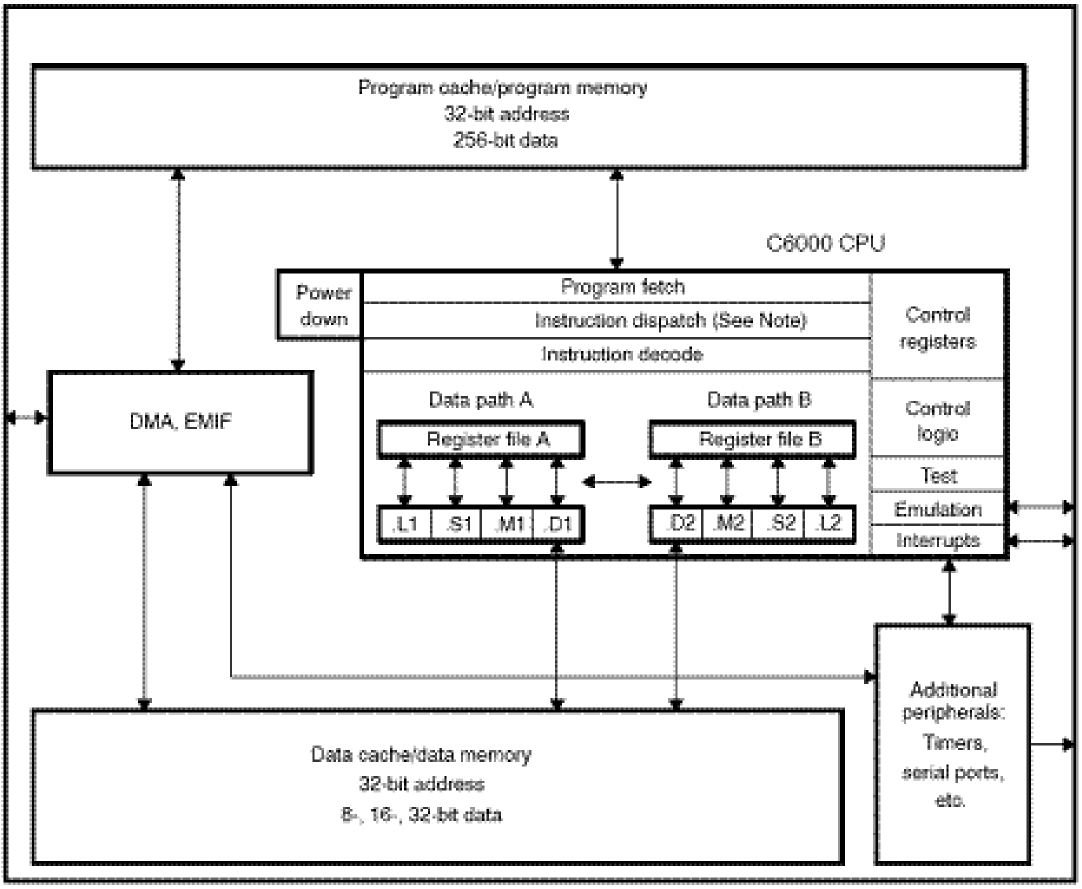


Figure 1.1: TMS320C67x DSP Block Diagram taken from SPRU733: TMS320C67x/C67x+ DSP CPU and Instruction Set Reference Guide

The general-purpose register files support data ranging in size from 16-bit data through 40-bit fixed-point. Values larger than 32 bits, such as 40-bit long quantities, are stored in register pairs. In a register pair, the 32 LSB's of data are placed in an even-numbered register and the remaining 8 MSB's in the next upper register (which is always an odd-numbered register). In assembly language syntax, a colon between two register names denotes the register pairs, and the odd-numbered register is specified first. For example, A1:A0 represents the register pair consisting of A0 and A1.

Let's for now focus on file A only. The registers in the register file A are named A0 to A15. Each register can store a 32-bit binary number. Then numbers such as `a`, `x`, `prod` and `y` above are stored in these registers. For example, register A0 stores `a`. For now, let's assume we interpret all 32-bit numbers stored in registers as unsigned integer. Therefore the range of values we can represent is 0 to $2^{32}-1$. Let's assume the numbers `a`, `x`, `prod` and `y` are in the registers A0, A1, A3, A4, respectively. Then the above assembly instructions can be written specifically:

```
MPY .M1 A0,A1,A3
ADD .L1 A4,A3,A4
```

The TI C6000 CPU has a load/store architecture. This means that all the numbers must be stored in the registers before being used as operands of the operations for instructions such as `MPY` and `ADD`. The numbers can be read from a memory location to a register (using, for example, `LDW`, `LDB` instructions) or a register can be loaded with a constant value. The content of a register can be stored to a memory location (using, for example, `STW`, `STB` instructions).

In addition to the general-purpose register files, the CPU has a separate register file for the control registers. The control registers are used to control various CPU functions such as addressing mode, interrupts, etc.

1.1.4.3 Functional Units

Where do the actual operations such as multiplication and addition take place? The C6000 CPU has several **functional units** that perform the actual operations. Each register file has 4 functional units named `.M`, `.L`, `.S`, and `.D`. The 4 functional units connected to the register file A are named `.M1`, `.L1`, `.S1`, and `.D1`. Those connected to the register file B are named `.M2`, `.L2`, `.S2`, and `.D2`. For example, the functional unit `.M1` performs multiplication on the operands that are in register file A. When the CPU executes the `MPY .M1 A0, A1, A3` above, the functional unit `.M1` takes the value stored in `A0` and `A1`, multiply them together and stores the result to `A3`. The `.M1` in `MPY .M1 A0, A1, A3` indicates that this operation is performed in the `.M1` unit. The `.M1` unit has a 16 bit multiplier and all the multiplications are performed by the `.M1` (or `.M2`) unit. The following diagram shows the basic architecture of the C6000 family and functional units.

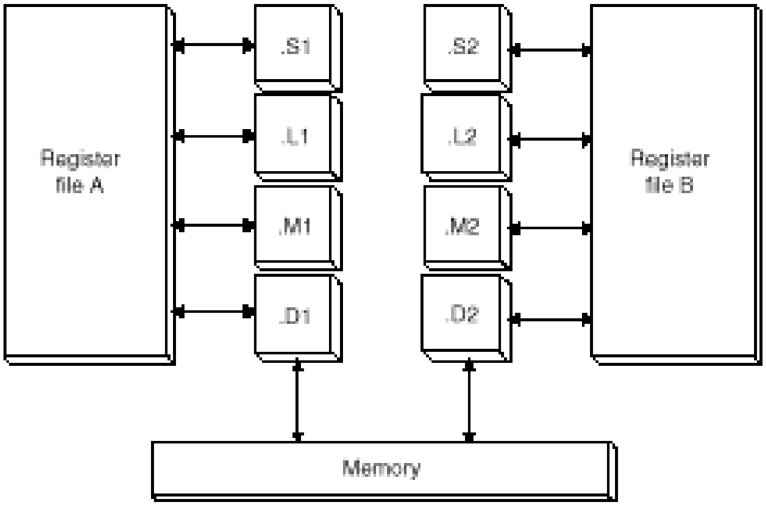


Figure 1.2: Functional Units of the 'C6x taken from SPRU198: TMS320C6000 Programmers' Guide

Similarly, the `ADD` operation can be executed by the `.L1` unit. The `.L1` can perform all the logical operations such as bitwise AND operation (`AND` instruction) as well as basic addition (`ADD` instruction) and subtraction (`SUB` instruction).

Exercise 1.1.1

Read the description of the `ADD` and `MPY` instructions in SPRU733 or similar document for the processor you are using. Write an assembly program that computes `A0*(A1+A2)+A3`.

1.1.5 Typical Assembly Operations

1.1.5.1 Loading constants to registers

Quite often you need to load a register with a constant. The C6x instructions you can use for this task are `MVK`, `MVKL`, and `MVKH`. Each of these instructions can load a 16-bit constant to a register. The `MVKL` instruction loads the LOWER 16-bits and the `MVKH` instruction loads the HIGH 16-bits into the register. In order to load 32-bit values into a register, both instructions are needed.

Exercise 1.1.2

(Loading constants): Write assembly instructions to do the following:

1. Load the 16-bit constant `0xff12` to `A1`.
2. Load the 32-bit constant `0xabcd45ef` to `B0`.

1.1.5.2 Register moves, zeroing

Contents of one register can be copied to another register by using the `MV` instruction. There is also the `ZERO` instruction to set a register to zero.

1.1.5.3 Loading from memory to registers

Because the C6x processor has the so-called load/store architecture, you must first load up the content of memory to a register to be able to manipulate it. The basic assembly instructions you use for loading are `LDB`, `LDH`, and `LDW` for loading up 8-, 16-, and 32-bit data from memory. (There are some variations to these instructions for different handling of the signs of the loaded values.)

However, to specify the address of the memory location to load from, you need to load up another register (used as an address index) and you can use various **addressing modes** to specify the memory locations in many different ways. The addressing mode is the method by which an instruction calculates the location of an object in memory. The table below lists all the possible different ways to handle the address pointers in the C6x CPU. Note the similarity with the C pointer manipulation.

Syntax	**Memory address accessed**	**Pointer modification**
`*R`	`R`	None
`*++R`	`R`	Preincrement
`*-R`	`R`	Predecrement
`*R++`	`R`	Postincrement
`*R-`	`R`	Postdecrement
`*+R[disp]`	`R+disp`	None

continued on next page

*-R[disp]	R+disp	None
*++R[disp]	R+disp	Preincrement
*-R[disp]	R+disp	Predecrement
*R++[disp]	R+disp	Postincrement
*R-[disp]	R+disp	Postdecrement

Table 1.2: C6x addressing modes.

The [disp] specifies the number of elements in word, halfword, or byte, depending on the instruction type and it can be either **5-bit constant** or a **register**. The increment/decrement of the index registers are also in terms of the number of bytes in word, halfword or byte. The addressing modes with displacements are useful when a block of memory locations is accessed. Those with automatic increment/decrement are useful when a block is accessed consecutively to implement a buffer, for example, to store signal samples to implement a digital filter.

Exercise 1.1.3

(Load from memory): Assume the following values are stored in memory addresses:

```
   Loc   32-bit value
100h  fe54  7834h
104h  3459  f34dh
108h  2ef5  7ee4h
10ch  2345  6789h
110h  ffff  eeddh
114h  3456  787eh
118h  3f4d  7ab3h
```

Suppose A10 = 0000 0108h. Find the contents of A1 and A10 after executing the each of the following instructions.

1. LDW .D1 *A10, A1
2. LDH .D1 *A10, A1
3. LDB .D1 *A10, A1
4. LDW .D1 *-A10[1], A1
5. LDW .D1 *+A10[1], A1
6. LDW .D1 *+A10[2], A1
7. LDB .D1 *+A10[2], A1
8. LDW .D1 *++A10[1], A1
9. LDW .D1 *-A10[1], A1
10. LDB .D1 *++A10[1], A1
11. LDB .D1 *-A10[1], A1
12. LDW .D1 *A10++[1], A1
13. LDW .D1 *A10-[1], A1

1.1.5.4 Storing data to memory

Storing the register contents uses the same addressing modes. The assembly instructions used for storing are STB, STH, and STW.

Exercise 1.1.4

(Storing to memory): Write assembly instructions to store 32-bit constant 53fe 23e4h to memory address 0000 0123h.

Sometimes, it becomes necessary to access part of the data stored in memory. For example, if you store the 32-bit word 0x11223344 at memory location 0x8000, the four bytes having addresses location 0x8000, location 0x8001, location 0x8002, and location 0x8003 contain the value 0x11223344. Then, if I read the byte data at memory location 0x8000, what would be the byte value to be read?

The answer depends on the **endian mode** of the memory system. In the **little endian mode**, the lower memory addresses contain the LSB part of the data. Thus, the bytes stored in the four byte addresses will be as shown in Table 1.3.

0x8000	0x44
0x8001	0x33
0x8002	0x22
0x8003	0x11

Table 1.3: Little endian storage mode.

In the **big endian mode**, the lower memory addresses contain the MSB part of the data. Thus, we have

0x8000	0x11
0x8001	0x22
0x8002	0x33
0x8003	0x44

Table 1.4: Big endian storage mode.

In the C6x CPU, it takes exactly one CPU clock cycle to execute each instruction. However, the instructions such as LDW need to access the slow external memory and the results of the load are not available immediately at the end of the execution. This **delay** of the execution results is called **delay slots**.

Example 1.1

For example, let's consider loading up the content of memory content at address pointed by A10 to A1 and then moving the loaded data to A2. You might be tempted to write simple 2 line assembly code as follows:

```
LDW   .D1    *A10, A1
MV    .D1    A1,A2
```

What is wrong with the above code? The result of the LDW instruction is not available immediately after LDW is executed. As a consequence, the MV instruction does not copy the desired value of A1 to A2. To prevent this undesirable execution, we need to make the CPU wait until the result of the LDW instruction is correctly loaded to A1 before executing the MV instruction. For load instructions, we need extra 4 clock cycles until the load results are valid. To make the CPU wait for 4 clock cycles, we need to insert 4 NOP (no operations) instructions between LDW and MV. Each NOP instruction makes the CPU idle for one clock cycle. The resulting code will be like this:

```
LDW    .D1    *A10, A1
NOP
NOP
NOP
NOP
MV     .D1    A1,A2
```

or simply you can write

```
LDW    .D1    *A10, A1
NOP  4
MV     .D1    A1,A2
```

Then, why didn't the designer of the CPU make such that LDW instruction takes 5 clock cycles to begin with, rather than let the programmer insert 4 NOPs? The answer is that you can insert other instructions other than NOPs as far as those instructions do not use the result of the LDW instruction above. By doing this, the CPU can execute additional instructions while waiting for the result of the LDW instruction to be valid, greatly reducing the total execution time of the entire program.

1.1.5.5 Delay slots

In the C6x CPU, it takes exactly one CPU clock cycle to execute each instruction. However, the instructions such as LDW need to access the slow external memory and the results of the load are not available immediately at the end of the execution. This **delay** of the execution results is called **delay slots**.

Example 1.2

For example, let's consider loading up the content of memory content at address pointed by A10 to A1 and then moving the loaded data to A2. You might be tempted to write simple 2 line assembly code as follows:

```
LDW   .D1    *A10, A1
MV    .D1    A1,A2
```

What is wrong with the above code? The result of the LDW instruction is not available immediately after LDW is executed. As a consequence, the MV instruction does not copy the desired value of A1 to A2. To prevent this undesirable execution, we need to make the CPU wait until the result of the LDW instruction is correctly loaded to A1 before executing the MV instruction. For load instructions, we need extra 4 clock cycles until the load results are valid. To make the CPU wait for 4 clock cycles, we need to insert 4 NOP (no operations) instructions between LDW and MV. Each NOP instruction makes the CPU idle for one clock cycle. The resulting code will be like this:

```
LDW    .D1    *A10, A1
NOP
NOP
NOP
NOP
MV     .D1    A1,A2
```

or simply you can write

```
LDW    .D1    *A10, A1
NOP  4
MV     .D1    A1,A2
```

Why didn't the designer of the CPU make such that LDW instruction takes 5 clock cycles to begin with, rather than let the programmer insert 4 NOPs? The answer is that you can insert other instructions other than NOPs as far as those instructions do not use the result of the LDW instruction above. By doing this, the CPU can execute additional instructions while waiting for the result of the LDW instruction to be valid, greatly reducing the total execution time of the entire program.

Description	Instructions	Delay slots
Single Cycle	All instructions except following	0
Multiply	MPY, SMPY etc.	1
Load	LDB, LDH, LDW	4
Branch	B	5

Table 1.5: Delay slots

The **functional unit latency** indicates how many clock cycles each instruction actually uses a functional unit. All C6x instructions have 1 functional unit latency, meaning that each functional unit is ready to execute the next instruction after 1 clock cycle regardless of the delay slots of the instructions. Therefore, the following instructions are valid:

```
LDW    .D1    *A10, A4
ADD    .D1    A1,A2,A3
```

Although the first LDW instruction do not load the A4 register correctly while the ADD is executed, the D1 functional unit becomes available in the clock cycle right after the one in which LDW is executed.

To clarify the execution of instructions with delay slots, let's think of the following example of the LDW instruction. Let's assume A10 = 0x0100A2=1, and your intent is loading A9 with the 32-bit word at the address 0x0104. The 3 MV instructions are not related to the LDW instruction. They do something else.

```
LDW    .D1    *A10++[A2], A9
MV     .L1    A10, A8
MV     .L1    A1, A10
MV     .L1    A1, A2
...
```

We can ask several interesting questions at this point:

1. What is the value loaded to A8? That is, in which clock cycle, the address pointer is updated?
2. Can we load the address offset register A2 before the LDW instruction finishes the actual loading?
3. Is it legal to load to A10 before the first LDW finishes loading the memory content to A9? That is, can we change the address pointer before the 4 delay slots elapse?

Here are the answers:

1. Although it takes an extra 4 clock cycles for the LDW instruction to load the memory content to A9, the address pointer and offset registers (A10 and A2) are read and updated in the clock cycle the LDW instruction is issued. Therefore, in line 2, A8 is loaded with the updated A10, that is A10 = A8 = 0x104.
2. Because the LDW reads the A10 and A2 registers in the first clock cycle, you are free to change these registers and do not affect the operation of the first LDW.
3. This was already answered above.

Similar theory holds for MPY and B (when using a register as a branch address) instructions. The MPY reads in the source values in the first clock cycle and loads the multiplication result after the 2nd clock cycle. For B, the address pointer is read in the first clock cycle, and the actual branching occurs after the 5th clock cycle. Thus, after the first clock cycle, you are free to modify the source or the address pointer registers. For more details, refer Table 3-5 in the instruction set description or read the description of the individual instruction.

1.1.5.6 Addition, Subtraction and Multiplication

There are several instructions for addition, subtraction and multiplication on the C6x CPU. The basic instructions are ADD, SUB, and MPY. ADD and SUB have 0 delay slots (meaning the results of the operation are immediately available), but the MPY has 1 delay slot (the result of the multiplication is valid after an additional 1 clock cycle).

Exercise 1.1.5

(Add, subtract, and multiply): Write an assembly program to compute (0000 ef35h + 0000 33dch - 0000 1234h) * 0000 0007h

1.1.5.7 Branching and conditional operations

Often you need to control the flow of the program execution by branching to another block of code. The B instruction does the job in the C6x CPU. The address of the branch can be specified either by displacement or stored in a register to be used by the B instruction. The B instruction has 5 delay slots, meaning that the actual branch occurs in the 5th clock cycle after the instruction is executed.

In many cases, depending on the result of previous operations, you execute the branch instruction conditionally. For example, to implement a loop, you decrement the loop counter by 1 each time you run a set of instructions and whenever the loop counter is not zero, you need to branch to the beginning of the code block to iterate the loop operations. In the C6x CPU, this conditional branching is implemented using the **conditional operations**. Although B may be the instruction implemented using conditional operations most often, all instructions in C6x can be conditional.

Conditional instructions are represented in code by using square brackets, [ ], surrounding the condition register name. For example, the following B instruction is executed only if B0 is nonzero:

```
[B0]    B     .L1    A0
```

To execute an instruction conditionally when the condition register is zero, we use ! in front of the register. For example, the B instruction is executed when B0 is zero.

```
[!B0]    B     .L1    A0
```

Not all registers can be used as the condition registers. In the C62x and C67x devices, the registers that can be tested in conditional operations are B0, B1, B2, A1, A2.

Exercise 1.1.6

(Simple loop): Write an assembly program computing the summation $\sum_{n=1}^{100} n$ by implementing a simple loop.

1.1.5.8 Logical operations and bit manipulation

The logical operations and bit manipulations are accomplished by the `AND`, `OR`, `XOR`, `CLR`, `SET`, `SHL`, and `SHR` instructions.

1.1.5.9 Other assembly instructions

Other useful instructions include `IDLE` and compare instructions such as `CMPEQ`*etc.*

1.1.5.10 C62x instruction set summary

The set of instructions that can be performed in each functional unit is as follows (See Table 1.6: .S Unit, Table 1.7: .L Unit, Table 1.8: .D Unit and Table 1.9: .M Unit). Please refer to *TMS320C62x/C67x CPU and Instruction Set Reference Guide* for detailed description of each instruction.

.S Unit

Instruction	**Description**
`ADD(U)`	signed or unsigned integer addition without saturation
`ADDK`	integer addition using signed 16-bit constant
`ADD2`	two 16-bit integer adds on upper and lower register halves
`B`	branch using a register
`CLR`	clear a bit field
`EXT`	extract and sign-extend a bit field
`MV`	move from register to register
`MVC`	move between the control file and the register file
`MVK`	move a 16-bit constant into a register and sign extend
`MVKH`	move 16-bit constant into the upper bits of a register

continued on next page

`NEG`	negate (pseudo-operation)
`NOT`	bitwise NOT
`OR`	bitwise OR
`SET`	set a bit field
`SHL`	arithmetic shift left
`SHR`	arithmetic shift right
`SSHL`	shift left with saturation
`SUB(U)`	signed or unsigned integer subtraction without saturation
`SUB2`	two 16-bit integer integer subs on upper and lower register halves
`XOR`	exclusive OR
`ZERO`	zero a register (pseudo-operation)

Table 1.6

.L Unit

Instruction	**Description**
`ABS`	integer absolute value with saturation
`ADD(U)`	signed or unsigned integer addition without saturation
`AND`	bitwise AND
`CMPEQ`	integer compare for equality
`CMPGT(U)`	signed or unsigned integer compare for greater than
`CMPLT(U)`	signed or unsigned integer compare for less than
`LMBD`	leftmost bit detection
`MV`	move from register to register
`NEG`	negate (pseudo-operation)
`NORM`	normalize integer
`NOT`	bitwise NOT
`+OR`	bitwise OR
`SADD`	integer addition with saturation to result size

continued on next page

SAT	saturate a 40-bit integer to a 32-bit integer
SSUB	integer subtraction with saturation to result size
SUBC	conditional integer subtraction and shift - used for division
XOR	exclusive OR
ZERO	zero a register (pseudo-operation)

Table 1.7

.D Unit

Instruction	**Description**
ADD(U)	signed or unsigned integer addition without saturation
ADDAB (B/H/W)	integer addition using addressing mode
LDB (B/H/W)	load from memory with a 15-bit constant offset
MV	move from register to register
STB (B/H/W)	store to memory with a register offset or 5-bit unsigned constant offset
SUB(U)	signed or unsigned integer subtraction without saturation
SUBAB (B/H/W)	integer subtraction using addressing mode
ZERO	zero a register (pseudo-operation)

Table 1.8

.M Unit

Instruction	**Description**
MPY (U/US/SU)	signed or unsigned integer multiply 16lsb*16lsb
MPYH (U/US/SU)	signed or unsigned integer multiply 16msb*16msb
MPYLH	signed or unsigned integer multiply 16lsb*16msb
MPYHL	signed or unsigned integer multiply 16msb*16lsb
SMPY (HL/LH/H)	integer multiply with left shift and saturation

Table 1.9

1.1.6 Useful assembler directives

Other than the CPU instruction set, there are special commands to the assembler that direct the assembler to do various jobs when assembling the code. There are useful **assembler directives**you can use to let the assembler know various settings, such as `.set`, `.macro`, `.endm`, `.ref`, `.align`, `.word`, `.byte` `.include`.

The `.set` directive defines a symbolic name. For example, you can have

```
count    .set    40
```

The assembler replaces each occurrence of `count` with `40`.

The `.ref` directive is used to declare symbolic names defined in another file. It is similar to the `extern` declaration in C.

The `.space` directive reserves a memory space with specified number of bytes. For example, you can have

```
buffer    .space    128
```

to define a buffer of size 128 bytes. The symbol `buffer` has the address of the first byte reserved by `.space`. The `.bes` directive is similar to `.space`, but the label has the address of the last byte reserved.

To put a constant value in the memory, you can use `.byte`, `.word`, *etc.* If you have

```
const1    .word    0x1234
```

the assembler places the word constant `0x1234` at a memory location and `const1` has the address of the memory location. `.byte`*etc.* works similarly.

Sometimes you need to place your data or code at specific memory address boundaries such as word, halfword, *etc.* You can use the `.align` directive to do this. For example, if you have

```
            .align    4
  buffer    .space    128
            ...
```

the first address of the reserved 128 bytes is at the word boundary in memory, that is the 2 LSBs of the address (in binary) are 0. Similarly, for half-word alignment, you should have `.align` directive to do this. For example, if you have

```
            .align    2
  buffer    .space    128
            ...
```

the `.include` directive is used to read the source lines from another file. The instruction

```
            .include    ''other.asm''
```

will input the lines in `other.asm` at this location. This is useful when working with multiple files. Instead of making a project having multiple files, you can simply include these different files in one file.

How do you write comments in your assembly program? Anything that follows `;` is considered a comment and ignored by the assembler. For example,

```
  ; this is a comment
          ADD     .L1     A1,A2,A3      ;add a1 and a2
```

1.1.7 Assigning functional units

Each instruction has particular functional units that can execute it. Note that some instructions can be executed by several different functional units.

The following figure shows how data and addresses can be transfered between the registers, functional units and the external memory. If you observe carefully, the destination path (marked as **dst**) going out of the `.L1, .S1, .M1` and `D1` units are connected to the register file A.

> NOTE: This means that any instruction with one of the A registers as destination (the result of operation is stored in one of A registers) should be executed in one of these 4 functional units.

For the same reason, if the instructions have B registers as destination, the `.L2, .S2, .M2` and `D2` units should be used.

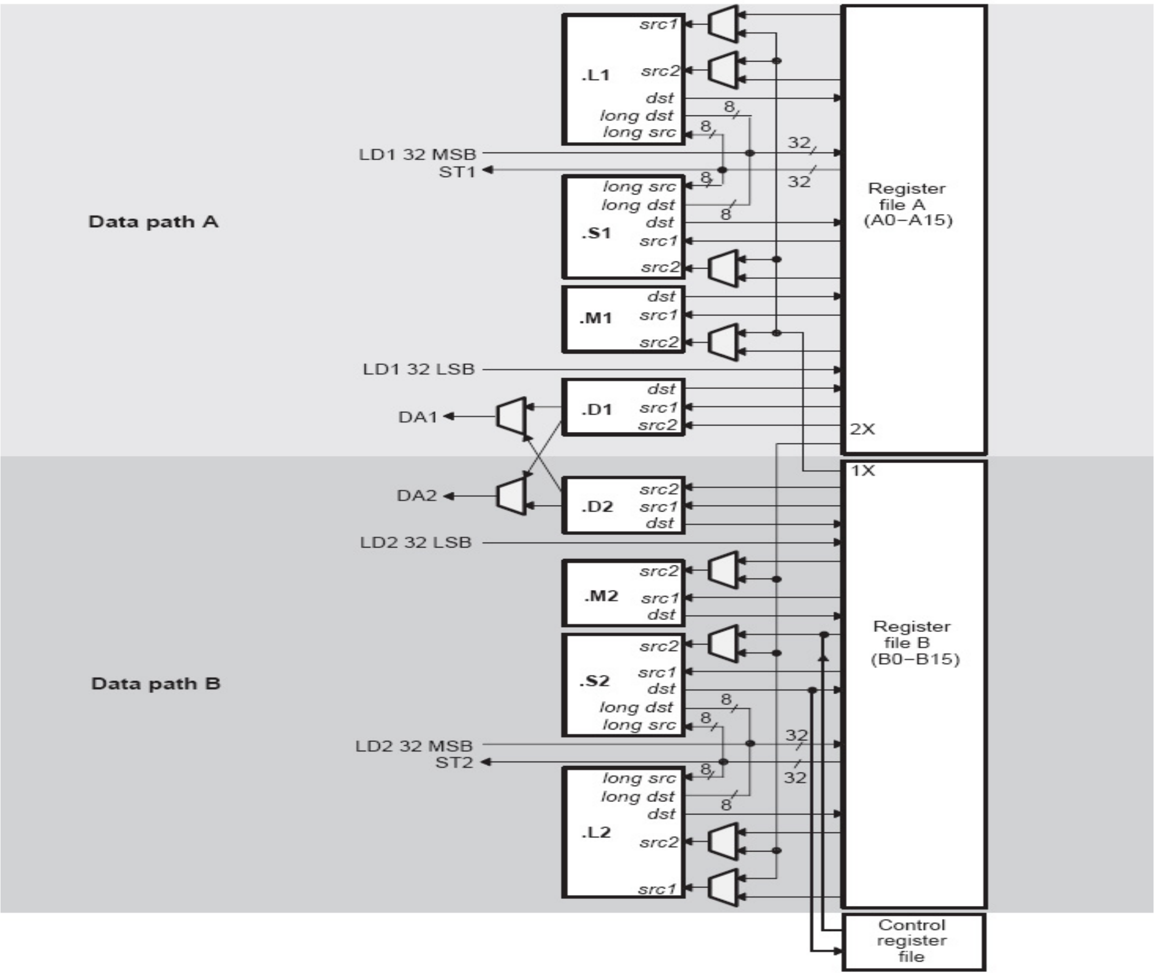


Figure 1.3: TMS320C67x DSP Block Diagram taken from SPRU733: TMS320C67x/C67x+ DSP CPU and Instruction Set Reference Guide

Therefore if you know the instruction and the destination register, you should be able to assign the functional unit to it.

Exercise 1.1.7

(Functional units): List all the functional units you can assign to each of these instructions:

1. `ADD .??  A0,A1,A2`
2. `B .??  A1`
3. `MVKL .??  000023feh, B0`
4. `LDW .??  *A10, A3`

If you look at the figure again, each functional unit must receive one of the source data from the corresponding register file. For example, look at the following assembly instruction:

```
      ADD   .L1    A0,B0,A1
```

The `.L1` unit gets data from `A0` (this is natural) and `B0` (this is not) and stores the result in `A1` (this is a must). The data path through which the content of `B0` is conveyed to the `.L1` unit is called `1X`**cross path**. When this happens, we add `x` to the functional unit to designate the cross path:

```
     ADD    .L1x    A0,B0,A1
```

Similarly the data path from register file `B` to the `.M2, .S2` and `.L2` units are called `2X` cross path.

Exercise 1.1.8

(Cross path): List all the functional units that can be assigned to each of the instruction:

1. `ADD .???  B0,A1,B2`
2. `MPY .???  A1,B2,A4`

In fact, when you write an assembly program, you can omit the functional unit assignment altogether. The assembler figures out the available functional units and properly assigns them. However, manually assigned functional units help you to figure out where the actual execution takes place and how the data move around between register files and functional units. This is particularly useful when you put multiple instructions in parallel. We will learn about the parallel instructions later on.

1.1.8 Writing the inner product program

Now you should know enough about C6x assembly to implement the inner product algorithm to compute

$$y = \sum_{n=1}^{10} a_n \times x_n$$

Exercise 1.1.9

(Inner product): Write the complete inner product assembly program to compute

$$y = \sum_{n=1}^{10} a_n \times x_n$$

where a_n and x_n take the following values:

```
a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, a }
x[] = { f, e, d, c, b, a, 9, 8, 7, 6 }
```

The a_n and x_n values must be stored in memory and the inner product is computed by reading the memory contents.

1.1.9 Pipeline, Delay slots and Parallel instructions

When an instruction is executed, it takes several steps, which are fetching, decoding, and execution. If these steps are done one at a time for each instruction, the CPU resources are not fully utilized. To increase the throughput, CPUs are designed to be pipelined, meaning that the foregoing steps are carried out at the same time.

On the C6x processor, the instruction fetch consists of 4 phases; generate fetch address (F1), send address to memory (F2), wait for data (F3), and read opcode from memory (F4). Decoding consists of 2 phases; dispatching to functional units (D1) and decoding (D2). The execution step may consist of up to 6 phases (E1 to E6) depending on the instructions. For example, the multiply (MPY) instructions has 1 delay resulting in 2 execution phases. Similarly, load (LDx) and branch (B) instructions have 4 and 5 delays respectively.

When the outcome of an instruction is used by the next instruction, an appropriate number of NOPs (no operation or delay) must be added after multiply (one NOP), load (four NOPs, or NOP 4), and branch (five NOPs, or NOP 5) instructions in order to allow the pipeline to operate properly. Otherwise, before the outcome of the current instruction is available (which is to be used by the next instruction), the next instructions are executed by the pipeline, generating undesired results. The following code is an example of pipelined code with NOPs inserted:

```
 1          MVK    40,A2
 2   loop:  LDH    *A5++,A0
 3          LDH    *A6++,A1
 4          NOP    4
 5          MPY    A0,A1,A3
 6          NOP
 7          ADD    A3,A4,A4
 8          SUB    A2,1,A2
 9   [A2]   B      loop
10          NOP    5
11          STH    A4,*A7
```

In line 4, we need 4 NOPs because the A1 is loaded by the LDH instruction in line 3 with 4 delays. After 4 delays, the value of A1 is available to be used in the MPY A0,A1,A3 in line 5. Similarly, we need 5 delays after the [A2] B loop instruction in line 9 to prevent the execution of STH A4,*A7 before branching occurs.

The C6x Very Large Instruction Word (VLIW) architecture, several instructions are captured and processed simultaneously. This is referred to as a Fetch Packet (FP). This Fetch Packet allows C6x to fetch eight instructions simultaneously from on-chip memory. Among the 8 instructions fetched at the same time, multiple of them can be executed at the same time if they do not use same CPU resources at the same time. Because the CPU has 8 separate functional units, maximum 8 instructions can be executed in parallel, although the type of parallel instructions are limited because they must not conflict each other in using CPU resources. In assembly listing, parallel instructions are indicated by double pipe symbols (||). When writing assembly code, by designing code to maximize parallel execution of instructions (through proper functional unit assignments, *etc.*) the execution cycle of the code can be reduced.

1.1.10 Parallel instructions and constraints

We have seen that C62x CPU has 8 functional units. Each assembly instruction is executed in one of these 8 functional units, and it takes exactly one clock cycle for the execution. Then, while one instruction is being executed in one of the functional units, what are other 7 functional units doing? Can other functional units execute other instructions at the same time?

The answer is YES. Thus, the CPU can execute maximum 8 instructions in each clock cycle. The instructions executed in the same clock cycle are called **parallel instructions**. Then, what instructions can be executed in parallel? A short answer is: as far as the parallel instructions do not use the same resource of the CPU, they can be put in parallel. For example, the following two instructions do not use the same CPU resource and they can be executed in parallel.

```
        ADD    .L1    A0,A1,A2
   ||   ADD    .L2    B0,B1,B2
```

1.1.10.1 Resource constraints

Then, what are the constraints on the parallel instructions? Let's look at the resource constraints in more detail.

1.1.10.1.1 Functional unit constraints

This is simple. Each functional unit can execute only one instruction per each clock cycle. In other words, instructions using the same functional unit cannot be put in parallel.

1.1.10.1.2 Cross paths constraints

If you look at the data path diagram of the C62x CPU, there exists only one cross path from B register file to the `L1`, `M1` and `S1` functional units. This means the cross path can be used only once per each clock cycle. Thus, the following parallel instructions are invalid because the 1x cross path is used for both instructions.

```
         ADD     .L1x    A0,B1,A2
    ||   MPY     .M1x    A5,B0,A3
```

The same rule holds for the 2x cross path from the A register file to the `L2`, `M2` and `S2` functional units.

1.1.10.1.3 Loads and Stores constraints

The `D` units are used for load and store instructions. If you examine the C62x data path diagram, the addresses for load/store can be obtained from either A or B side using the multiplexers connecting crisscross to generate the addresses `DA1` and `DA2`. Thus, the instructions such as

```
         LDW     .D2     *B0, A1
```

is valid. **The functional unit must be on the same side as the address source register** (address index in `B0` and therefore `D2` above), because `D1` and `D2` units must receive the addresses from A and B sides, respectively.

Another constraint is that while loading a register in one register file from memory, you cannot simultaneously store a register in the same register file to memory. For example, the following parallel instructions are invalid:

```
        LDW     .D1     *A0, A1
||      STW     .D2     A2, *B0
```

1.1.10.1.4 Constraints on register reads

You cannot have more than **four** reads from the same register in each clock cycle. Thus, the following is invalid:

```
        ADD     .L1     A1, A1, A2
||      MPY     .M1     A1, A1, A3
||      SUB     .D1     A1, A4, A5
```

1.1.10.1.5 Constraints on register writes

A register cannot be written to more than once in a single clock cycle. However, note that the actual writing to registers may not occur in the same clock cycle during which the instruction is executed. For example, the MPY instruction writes to the destination register in the next clock cycle. Thus, the following is valid:

```
       ADD     .L1     A1, A1, A2
||      MPY     .M1     A1, A1, A2
```

The following two instructions (not parallel) are invalid (why?):

```
        MPY     .M1     A1, A1, A2
        ADD     .L1     A3, A4, A2
```

Some of these write conflicts are very hard to detect and not detected by the assembler. Extra caution should be exercised with the instructions having nonzero delay slots.

1.1.11 Ad-Hoc software pipelining

At this point, you might have wondered why the C6x CPU allows parallel instructions and generate so much headache with the resource constraints, especially with the instructions with delay slots. And, why not just make the MPY instruction take 2 clock cycles to execute so that we can always use the multiplied result after issuing it?

The reason is that by executing instructions in parallel, we can reduce the total execution time of the program. A well-written assembly program executes as many instructions as possible in each clock cycle to implement the desired algorithm.

The reason for allowing delay slots is that although it takes 2 clock cycles for an MPY instruction generate the result, we can execute another instruction while waiting for the result. This way, you can reduce the clock cycles wasted while waiting for the result from slow instructions, thus increasing the overall execution speed.

However, how can we put instructions in parallel? Although there's a systematic way of doing it (we will learn a bit later), at this point you can try to restructure your assembly code to execute as many instructions as possible in parallel. And, you should try to execute other instructions in the delay slots of those instructions such as MPY, LDW, *etc.*, instead of inserting NOPs to wait the instructions produce the results.

1.2 Code Composer Studio v4 Assembly Project[2]

1.2.1 Introduction

This module describes how to create a Code Composer Studio (CCS) v4 project that executes a simple assembly program. The module does not explain the code in the project files. The project does not use DSP/BIOS and runs on the TI simulator. In this project the processor used is the TMS320C67xx but the process should work for other processors as well.

1.2.2 Create a CCS project

To create a CCS project select **File->New->CCS Project**.

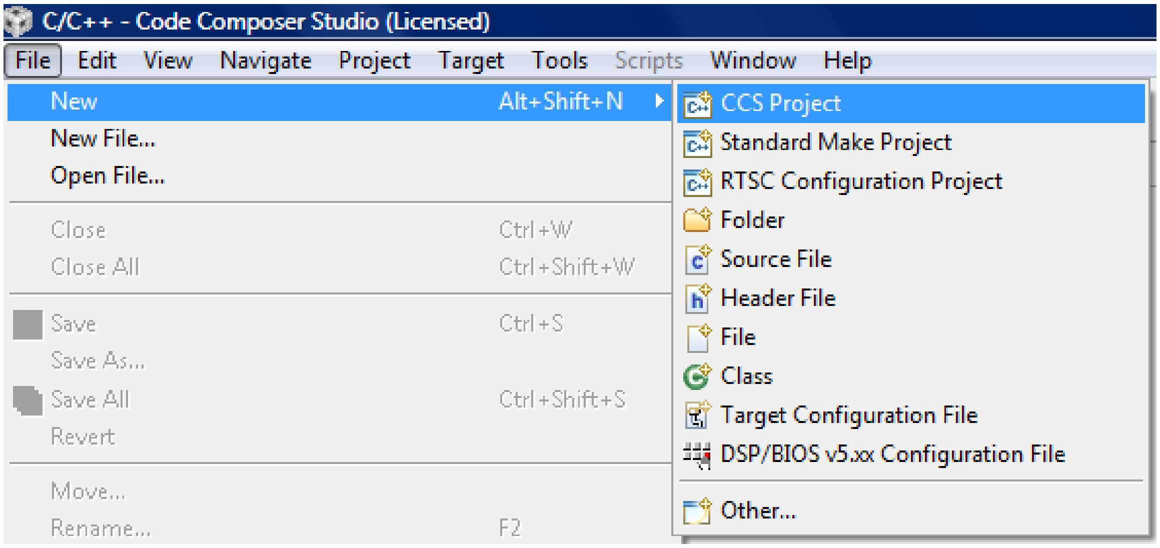
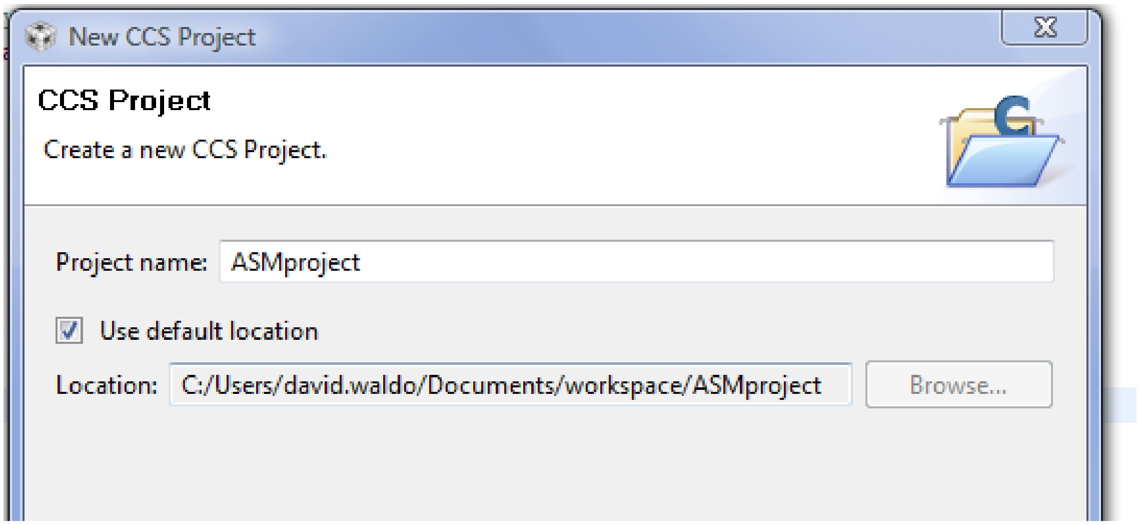


Figure 1.4: Screenshot of CCS Project menu

This will bring up a window where you can enter the name of the project. The location selected is the default location for project files. Press **Next**.

[2] This content is available online at <http://cnx.org/content/m36321/1.3/>.



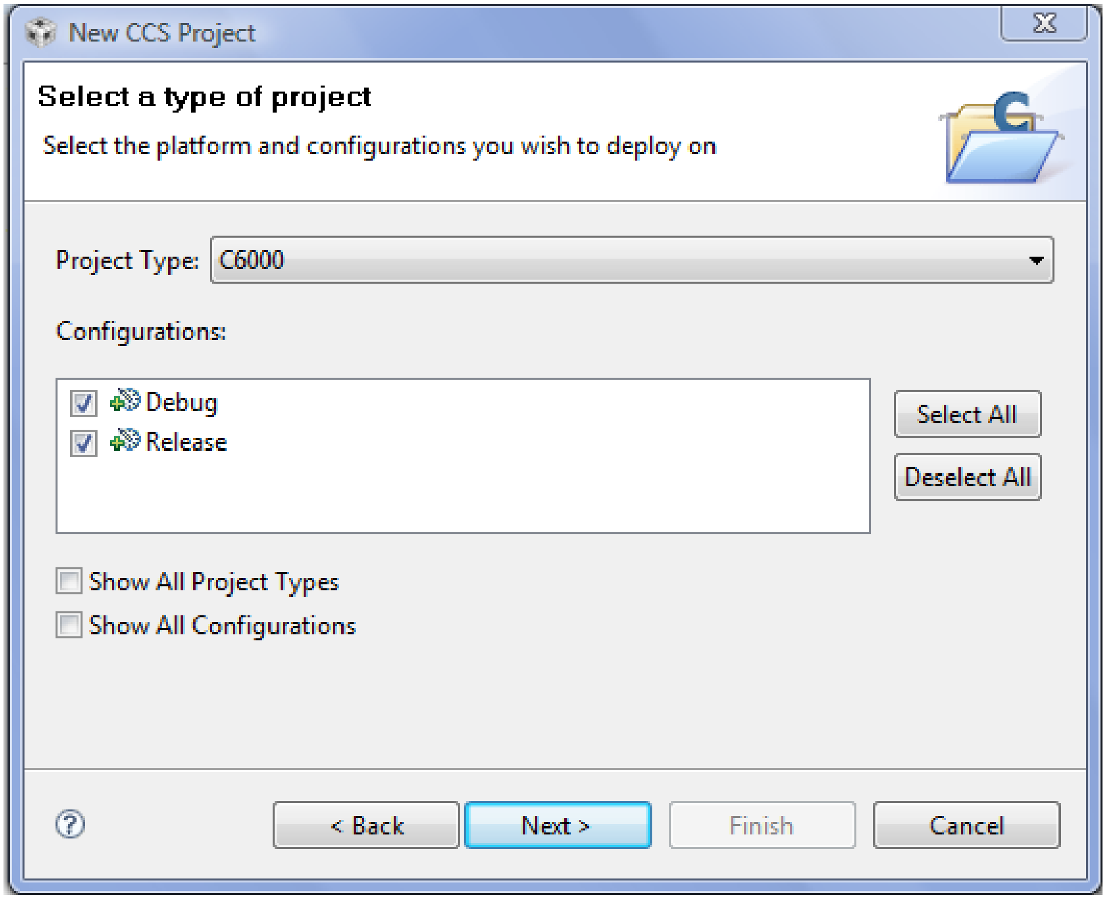
Project name: ASMproject

☑ Use default location

Location: C:/Users/david.waldo/Documents/workspace/ASMproject

Figure 1.5: CCS Project name and location

Since the example uses the TMS320C67xx processor the project type selected is **C6000**. The project configurations are **Debug** and **Release**. Select the **Next** button.



Select a type of project

Select the platform and configurations you wish to deploy on

Project Type: C6000

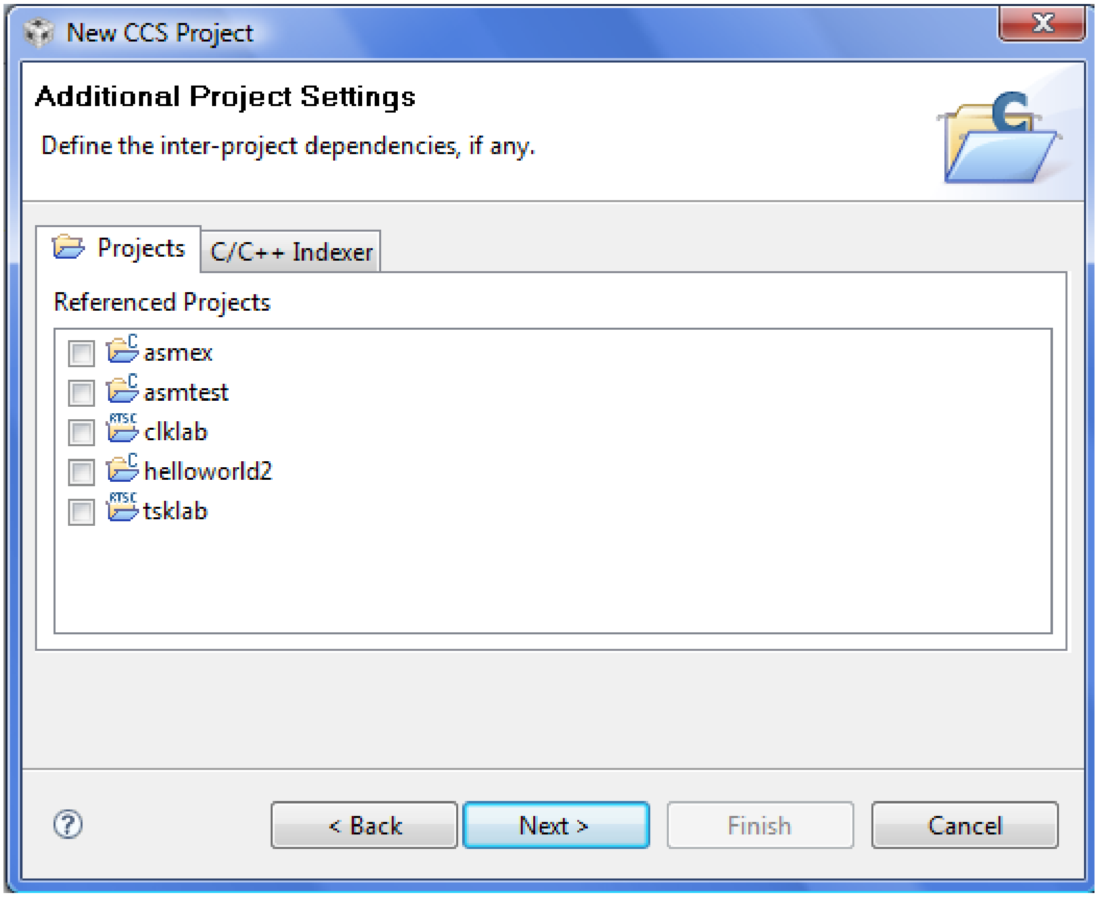
Configurations:

☑ Debug
☑ Release

□ Show All Project Types
□ Show All Configurations

Figure 1.6: Type of CCS project

If there are any project dependencies they are selected on the next screen. Select the **Next** button.



New CCS Project

Additional Project Settings

Define the inter-project dependencies, if any.

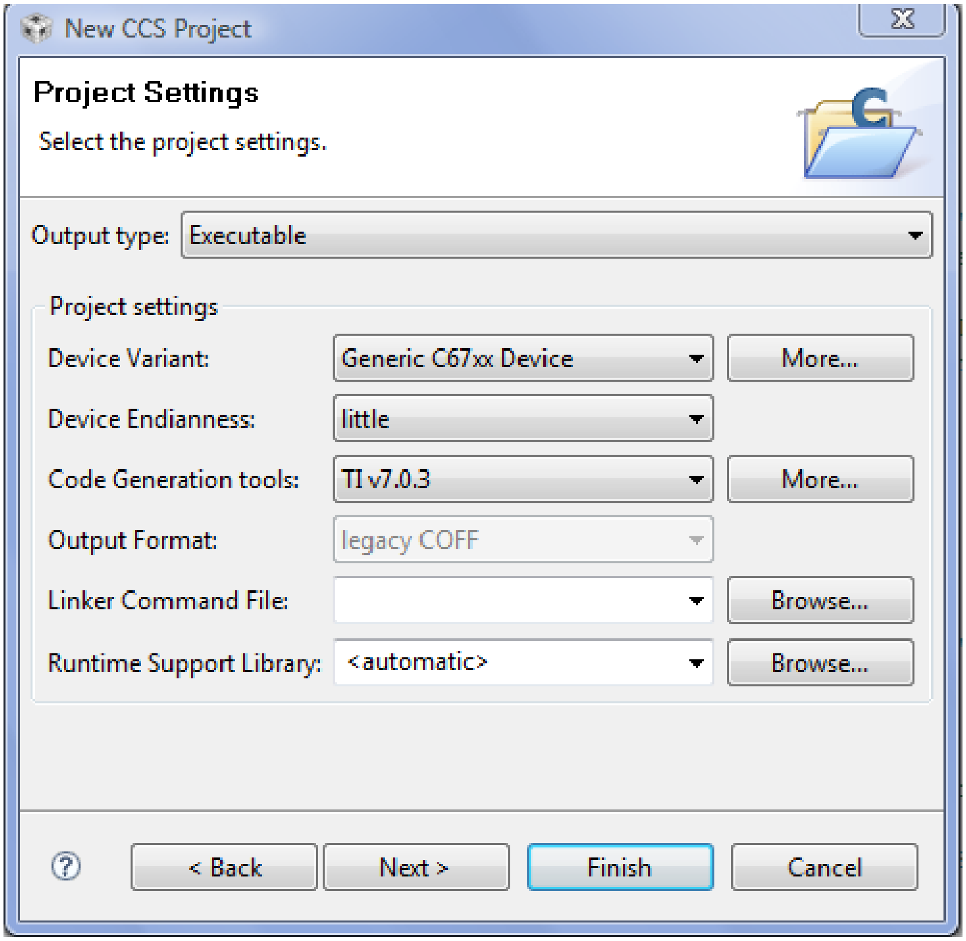
Referenced Projects

□ asmex
□ asmtest
□ clklab
□ helloworld2
□ tsklab

Figure 1.7: CCS Project dependencies

On the **Project Settings Screen**, select the items that pertain to the type of project to be created. Since the project will be executed select **Output Type: Executable**. The processor type is TMS320C67xx so the **Device Variant** is **Generic C67xx Device**. This project will use **Little Endian**. The code generation tools are the latest version (in this case TI v7.0.3). The runtime support library is selected to match the device variant and the endianness selected. The library can be selected automatically. Press **Next**.

Since this project will have assembly programs only, select the **Empty Assembly-only Project**. Select **Finish** and the project will show up in the **C/C++ Projects** view.



New CCS Project

Project Settings

Select the project settings.

Output type: Executable

Project settings

Device Variant: Generic C67xx Device

Device Endianness: little

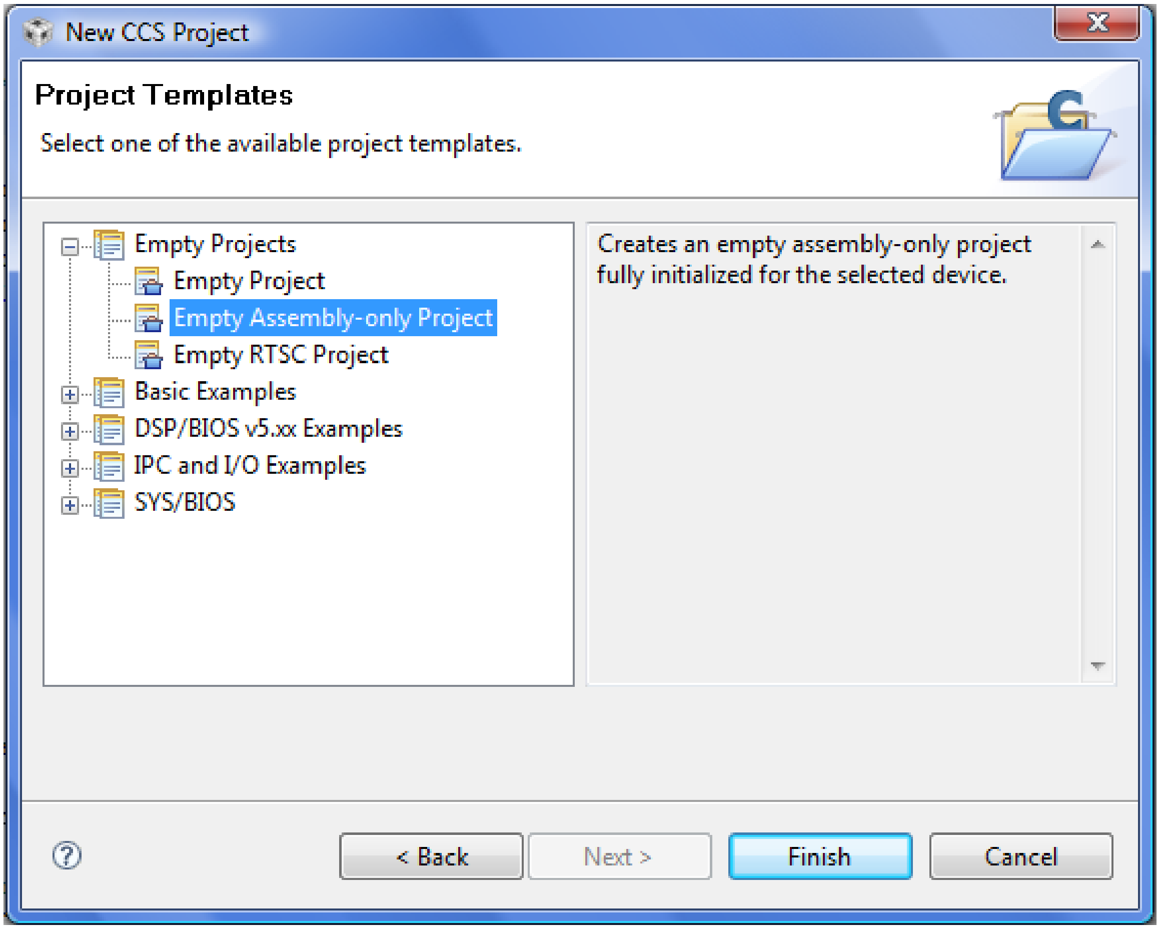
Code Generation tools: TI v7.0.3

Output Format: legacy COFF

Linker Command File:

Runtime Support Library: <automatic>

Figure 1.8: Project settings window



New CCS Project

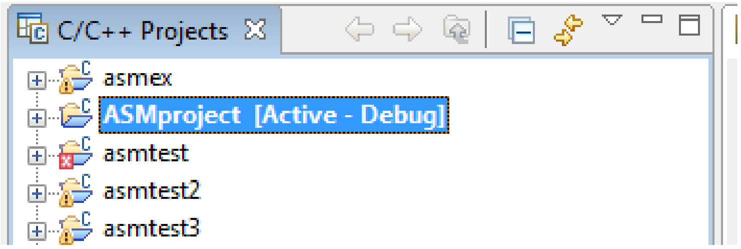
Project Templates

Select one of the available project templates.

- Empty Projects
  - Empty Project
  - Empty Assembly-only Project
  - Empty RTSC Project
- Basic Examples
- DSP/BIOS v5.xx Examples
- IPC and I/O Examples
- SYS/BIOS

Creates an empty assembly-only project fully initialized for the selected device.

Figure 1.9: Project Templates



C/C++ Projects

- asmex
- ASMproject [Active - Debug]
- asmtest
- asmtest2
- asmtest3

Figure 1.10: C/C++ projects view

1.2.3 Add code files

There are three files that will be added to the project. One, `main.asm`, will contain the assembly program we want to execute. The file `vectors.asm` will contain the code that gets executed when the reset interrupt occurs. The file `link.cmd` will be the linker command file that tells the linker where section of memory are located and where to put the code in the memory map.

1.2.3.1 Main program

Create the main.asm file that conains the assembly program to execute. Select **File->New->Source File**.

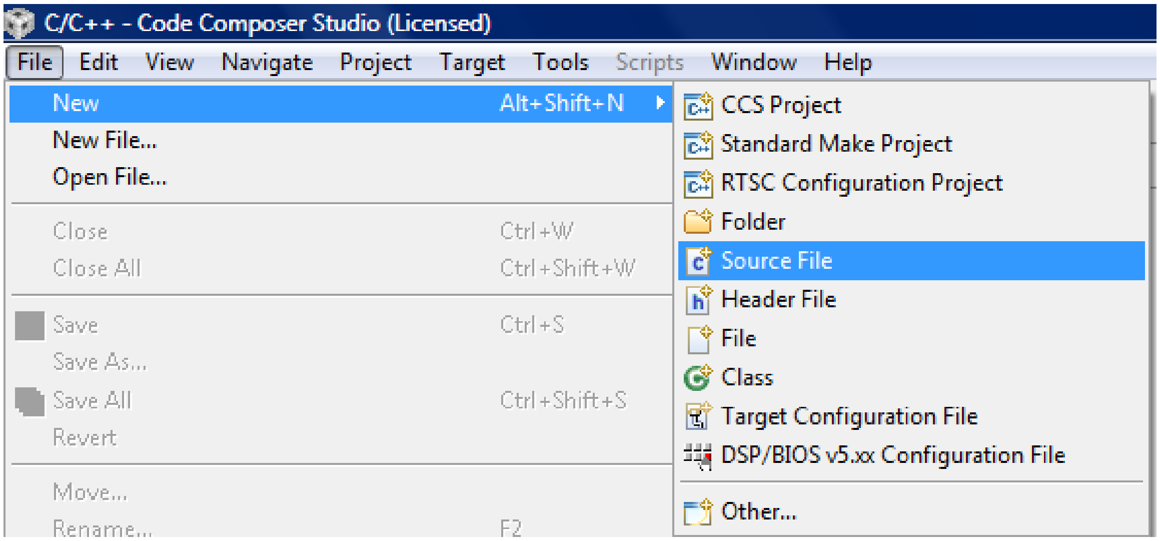
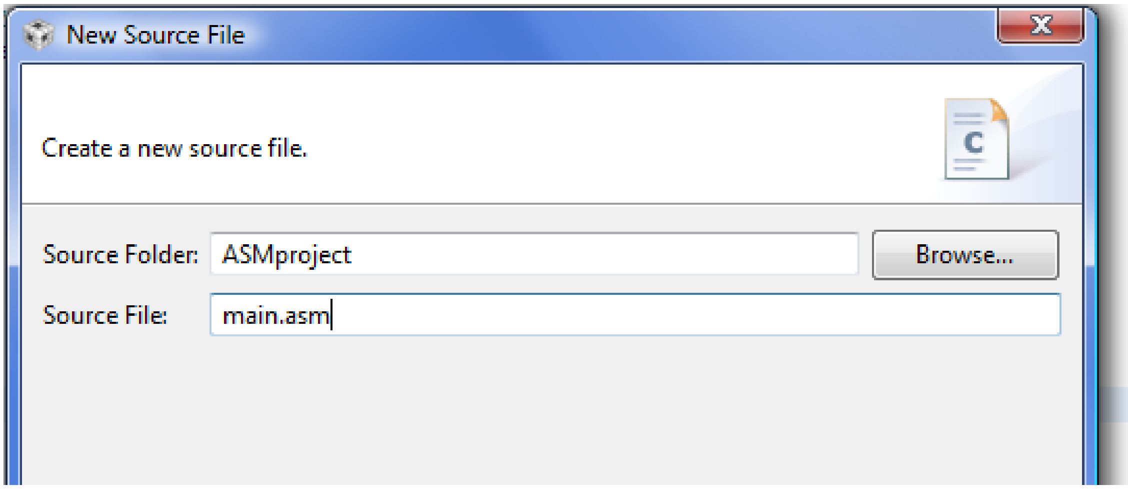


Figure 1.11: New source file

The **New Source File** dialog opens and you can enter the source file name. Add a **.asm** extension for an assembly program file. The source folder should be the folder of your project. Select **Finish** and a new file is opened.



New Source File

Create a new source file.

Source Folder: ASMproject

Source File: main.asm

Figure 1.12: New source file dialog

Enter the assembly code shown below in `main.asm`.

```
; Example assembly file
; Multiply two numbers
.text ; Indicates that what follows is code
.def _c_int00; This symbol is defined in this file
; but used in another file
_c_int00: MVK .S1 0x34,A1 ; Put a number in register A1
MVK .S1 0x25,A2 ; Put a number in register A2
MPY .M1 A1,A2,A1 ; Multiply two numbers in A1 and A2
NOP
NOP
NOP
NOP
NOP
```

1.2.3.2 Reset vector file

When a reset interrupt occurs in a C6000 processor the system will start executing the code located at the interrupt memory location. There are only 8 instructions at each interrupt vector. If a long program is to be executed then the interrupt vector needs to branch to the program location. This is done by using a resect vector program.

Create a new file named `vectors.asm` and enter the following code.

```
.ref _c_int00 ; name of the label where
; the program is located
.sect"vectors" ; this will put this code
; in the vectors memeory section
reset: mvkl.s2 _c_int00,b0 ; put the address in b0
```

```
mvkh.s2_c_int00,b0
b .s2 b0 ; branch to the program
nop
nop
nop
nop
nop
nop
```

1.2.3.3 Command file

The linker command file tells the linker information about the memory map and where to put parts of the code like data and program instructions. Create a new file named `link.cmd` and enter the following code.

```
MEMORY
{
VECS : origin = 0x00000000 length = 0x00000220
IPRAM : origin = 0x00000240 length = 0x0000FDC0
}
SECTIONS
{
vectors > VECS
.text > IPRAM
}
```

1.2.4 Create a target configuration

In order to build and run your project you need to have a target configuration. This file will contain information about the emulator that will be used and the target processor. In this module the target configuration will be the TI simulator for the C6713 processor. First select **File->New->Target Configuration File**.

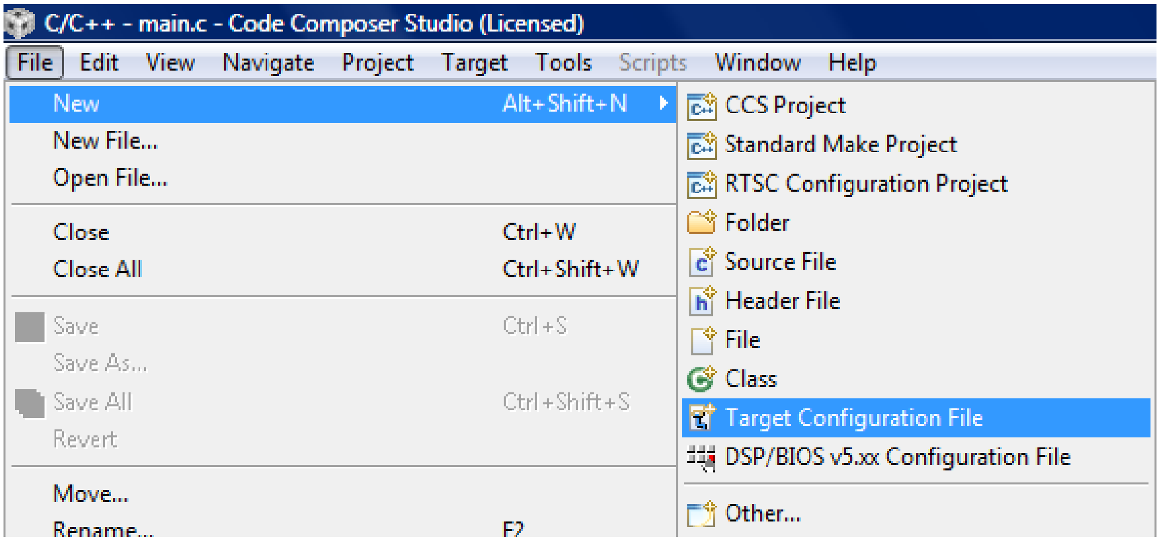
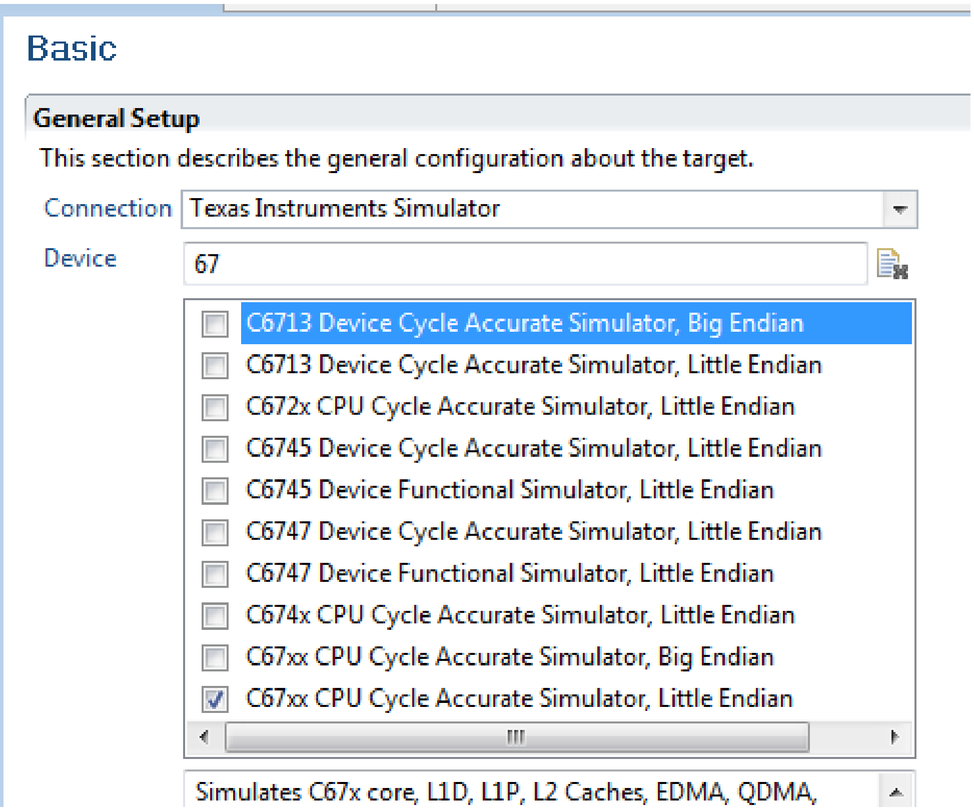


Figure 1.13: Open new target configuration file

In the New Target Configuration dialog enter the configuration file name ending in the **.ccxml** file extension. Put the file in the project directory. Select **Finish** and a new target configuration file opens.

In the configuration, select the connection needed (simulator or some type of emulator, etc.) and the type of device. In this example the TI simulator is used with the C67xx little endian simulator device. Select **Save** when done.



Basic

General Setup

This section describes the general configuration about the target.

Connection: Texas Instruments Simulator

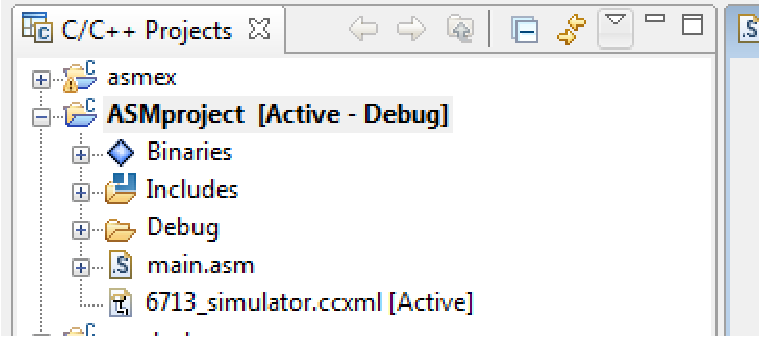
Device: 67

- □ C6713 Device Cycle Accurate Simulator, Big Endian
- □ C6713 Device Cycle Accurate Simulator, Little Endian
- □ C672x CPU Cycle Accurate Simulator, Little Endian
- □ C6745 Device Cycle Accurate Simulator, Little Endian
- □ C6745 Device Functional Simulator, Little Endian
- □ C6747 Device Cycle Accurate Simulator, Little Endian
- □ C6747 Device Functional Simulator, Little Endian
- □ C674x CPU Cycle Accurate Simulator, Little Endian
- □ C67xx CPU Cycle Accurate Simulator, Big Endian
- ☑ C67xx CPU Cycle Accurate Simulator, Little Endian

Simulates C67x core, L1D, L1P, L2 Caches, EDMA, QDMA,

Figure 1.14

Now the target configuration is in the project.



C/C++ Projects

- asmex
- ASMproject [Active - Debug]
  - Binaries
  - Includes
  - Debug
  - main.asm
  - 6713_simulator.ccxml [Active]

Figure 1.15: Project files

1.2.5 Debug the project

To start a debug session either select **Target->Debug Active Project** or press the debug icon in the menu bar.

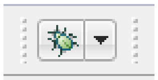
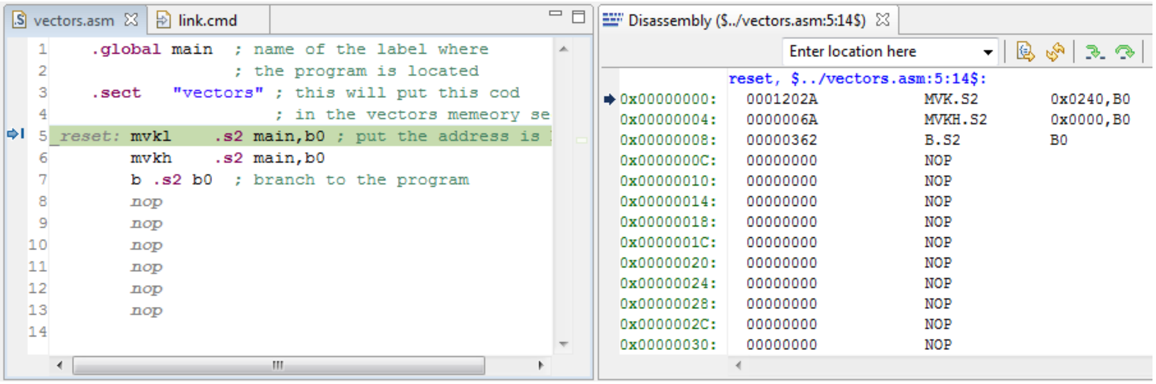


Figure 1.16: Debug icon

When the debug session starts the code files are compiled and linked into an executable file and the file is loaded onto the target. The reset vector code should be located at memroy location 0x0.

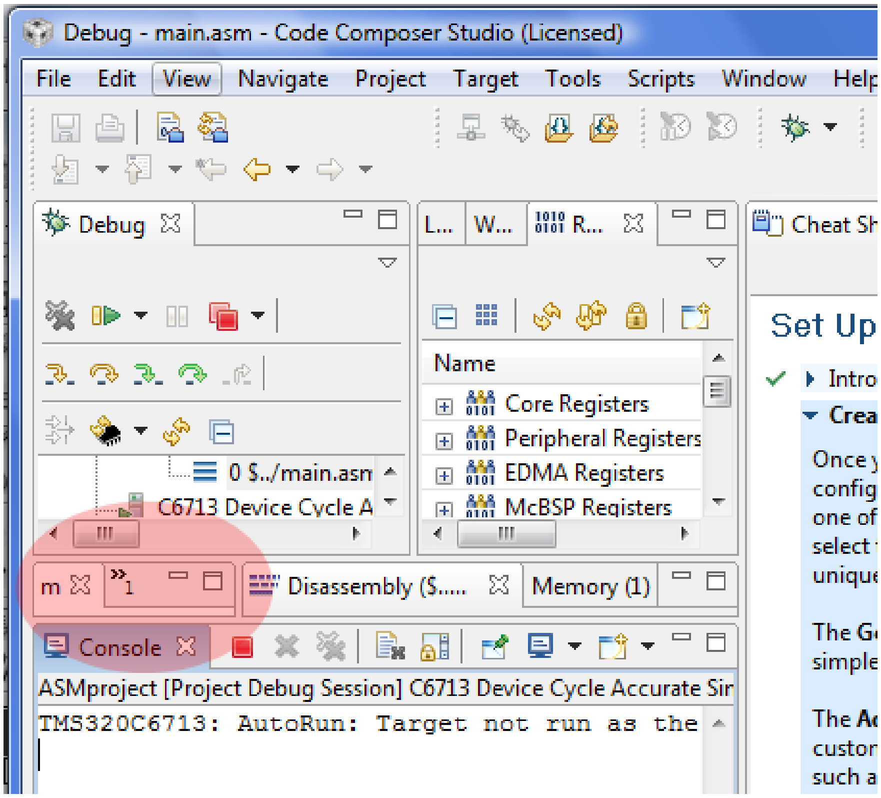


```
    .global main  ; name of the label where
                  ; the program is located
    .sect   "vectors" ; this will put this cod
                       ; in the vectors memeory se
_reset: mvkl     .s2 main,b0 ; put the address is
        mvkh     .s2 main,b0
        b .s2 b0  ; branch to the program
        nop
        nop
        nop
        nop
        nop
        nop
```

```
                 reset, $../vectors.asm:5:14$:
0x00000000:  0001202A    MVK.S2     0x0240,B0
0x00000004:  0000006A    MVKH.S2    0x0000,B0
0x00000008:  00000362    B.S2       B0
0x0000000C:  00000000    NOP
0x00000010:  00000000    NOP
0x00000014:  00000000    NOP
0x00000018:  00000000    NOP
0x0000001C:  00000000    NOP
0x00000020:  00000000    NOP
0x00000024:  00000000    NOP
0x00000028:  00000000    NOP
0x0000002C:  00000000    NOP
0x00000030:  00000000    NOP
```

Figure 1.17: Debugging session

If your window is small, CCS may not have room to open all windows. In this case your debugging window may look like the following figure.



```
ASMproject [Project Debug Session] C6713 Device Cycle Accurate Sir
TMS320C6713: AutoRun: Target not run as the
```

Figure 1.18: Crowded debug window

Note that the window where the program files are located may not show much (circled in red). In this case you will need to expand the window.

Begin executing the code by stepping through the code using the **Assembly Step Into** button.

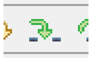
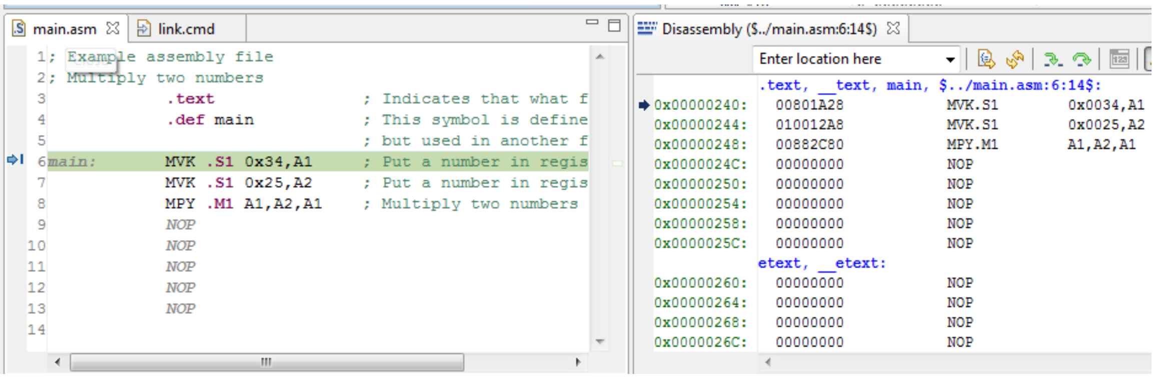


Figure 1.19: Assembly Step Into button

The code should execute until the branch instruction finishes executing and then it should jump to the main function located at a different memory location.



```
; Example assembly file
; Multiply two numbers
                .text                    ; Indicates that what f
                .def main                ; This symbol is define
                                         ; but used in another f
main:           MVK .S1 0x34,A1          ; Put a number in regis
                MVK .S1 0x25,A2          ; Put a number in regis
                MPY .M1 A1,A2,A1         ; Multiply two numbers
                NOP
                NOP
                NOP
                NOP
                NOP
```

```
            .text, __text, main, $../main.asm:6:14$:
0x00000240:   00801A28        MVK.S1      0x0034,A1
0x00000244:   010012A8        MVK.S1      0x0025,A2
0x00000248:   00882C80        MPY.M1      A1,A2,A1
0x0000024C:   00000000        NOP
0x00000250:   00000000        NOP
0x00000254:   00000000        NOP
0x00000258:   00000000        NOP
0x0000025C:   00000000        NOP
            etext, __etext:
0x00000260:   00000000        NOP
0x00000264:   00000000        NOP
0x00000268:   00000000        NOP
0x0000026C:   00000000        NOP
```

Figure 1.20: Debugging session after jump to main function

1.3 Creating a TI Code Composer Studio Simulator Project[3]

1.3.1 Introduction

Become familiar with how to create a Code Composer Studio v3.3 project using the simulator and DSP/BIOS.

1.3.2 Reading

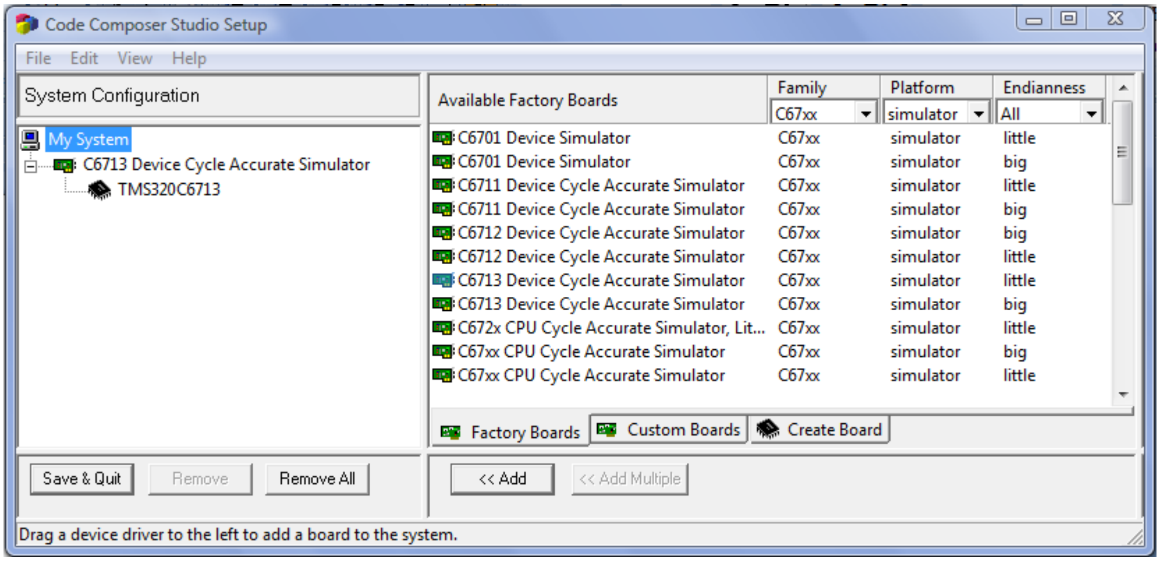
- SPRU509: Code Composer Studio Getting Started Guide

1.3.3 Description

Before creating a project in Code Composer Studio (CCS), you will need to setup CCS for the board or processor you are going to be using. This module will describe how to make a project using the C6713 simulator.

- Run Setup Code Composer Studio. On the right select the C6713 Device Cycle Accurate Simulator and add it to the left. This is the configuration that will be used when CCS is started. Save and quit. Start CCS.

[3] This content is available online at <http://cnx.org/content/m33263/1.5/>.



System Configuration

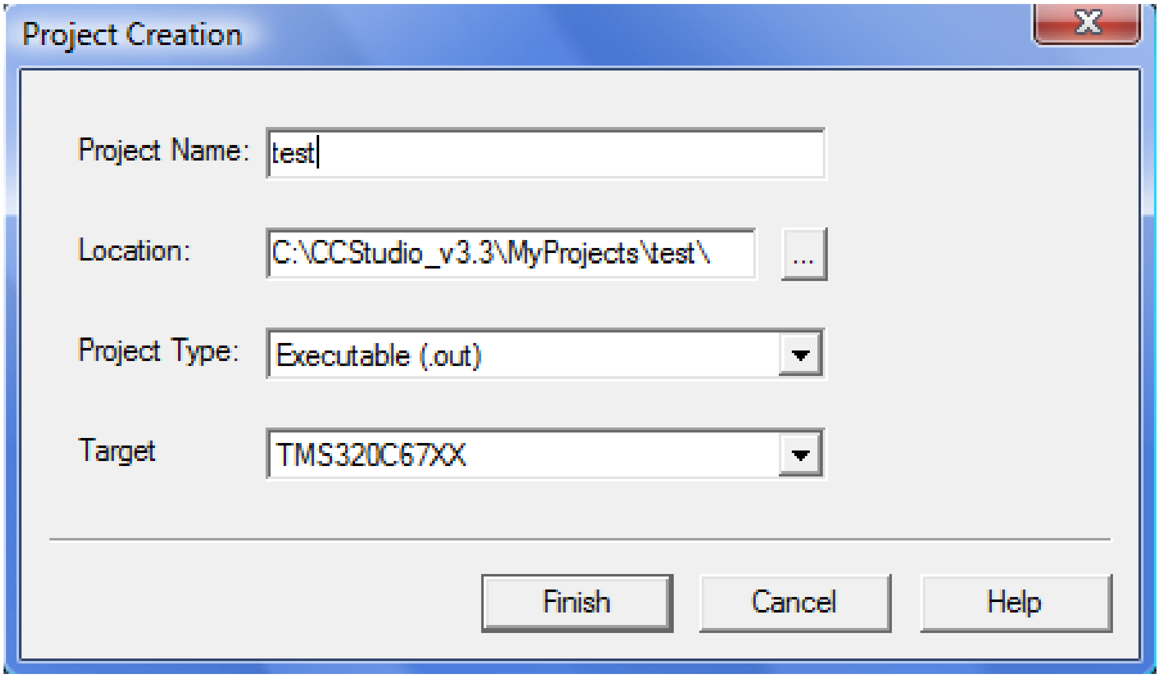
- My System
  - C6713 Device Cycle Accurate Simulator
    - TMS320C6713

Available Factory Boards	Family	Platform	Endianness
	C67xx	simulator	All
C6701 Device Simulator	C67xx	simulator	little
C6701 Device Simulator	C67xx	simulator	big
C6711 Device Cycle Accurate Simulator	C67xx	simulator	little
C6711 Device Cycle Accurate Simulator	C67xx	simulator	big
C6712 Device Cycle Accurate Simulator	C67xx	simulator	big
C6712 Device Cycle Accurate Simulator	C67xx	simulator	little
C6713 Device Cycle Accurate Simulator	C67xx	simulator	little
C6713 Device Cycle Accurate Simulator	C67xx	simulator	big
C672x CPU Cycle Accurate Simulator, Lit...	C67xx	simulator	little
C67xx CPU Cycle Accurate Simulator	C67xx	simulator	big
C67xx CPU Cycle Accurate Simulator	C67xx	simulator	little

Drag a device driver to the left to add a board to the system.

Figure 1.21: CCS Setup for the C6713 Device Cycle Accurate Simulator.

- Create a new project in CCS by selecting Project->New... This will bring up a window that will look like Figure 2. Type the name of your project in the Project Name field and it will add a folder with the same name in the Location field. Leave everything else the same. Click Finish and CCS will open your new project. This will look like Figure 3.



Project Creation

Project Name: test

Location: C:\CCStudio_v3.3\MyProjects\test\

Project Type: Executable (.out)

Target TMS320C67XX

Figure 1.22: Project creation window

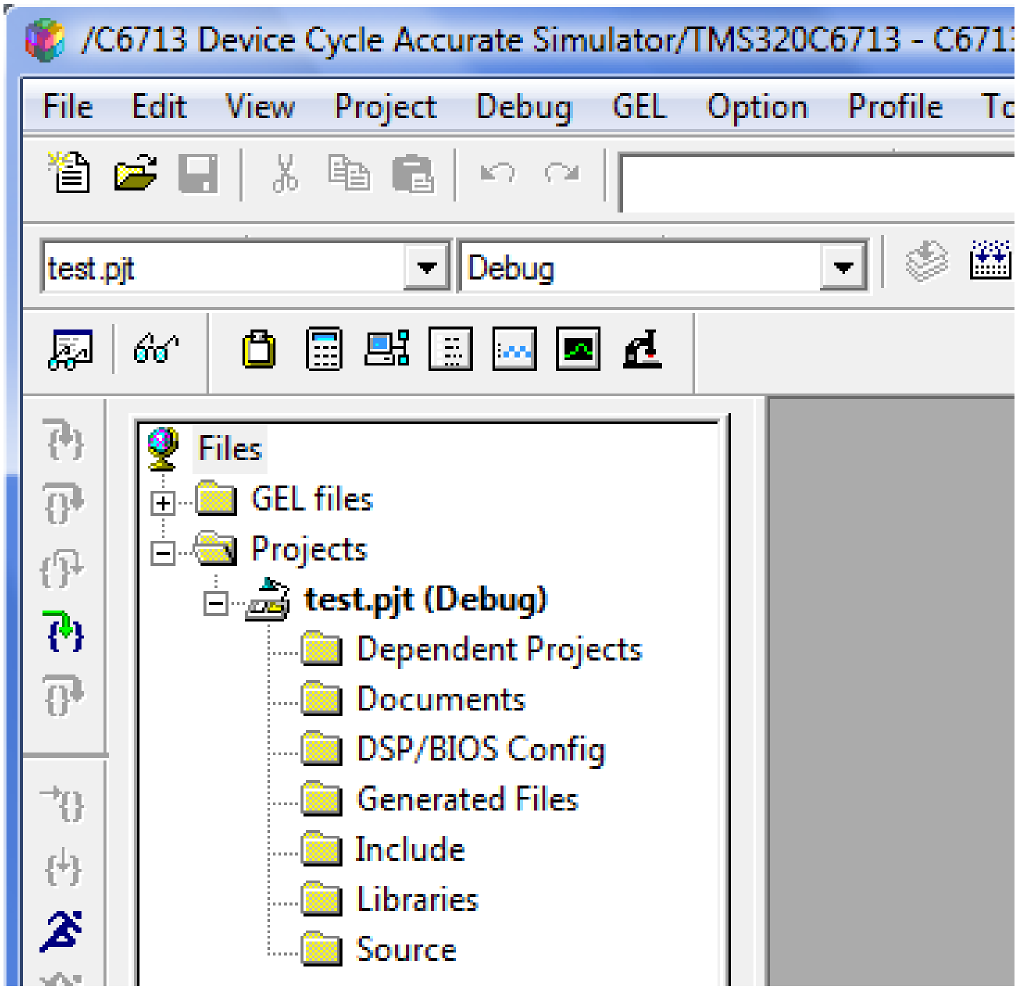
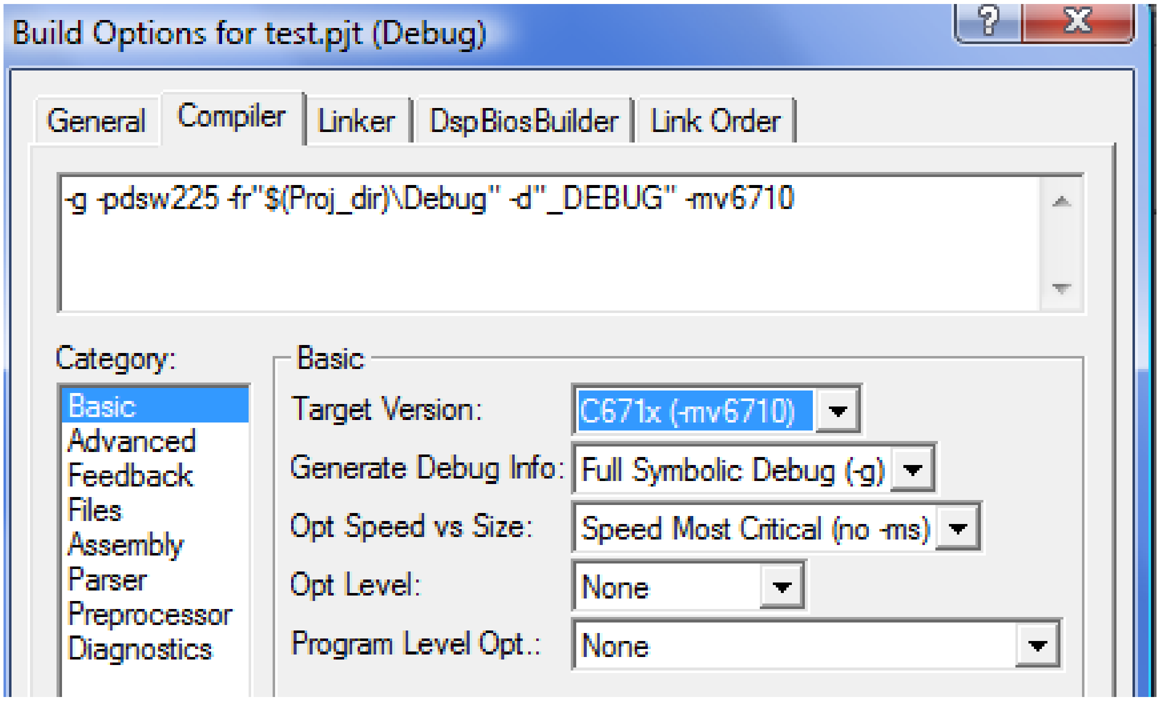


Figure 1.23: Project window

- After the project is created, make sure the correct processor is chosen in the build option. Select Project->Build Options... and select the processor that applies to the the configuration you set up in the Code Composer Setup tool. Here the C671x is selected to match the C6713 Device Cycle Accurate Simulator.



Build Options for test.pjt (Debug)

-g -pdsw225 -fr"$(Proj_dir)\Debug" -d"_DEBUG" -mv6710

Target Version: C671x (-mv6710)

Generate Debug Info: Full Symbolic Debug (-g)

Opt Speed vs Size: Speed Most Critical (no -ms)

Opt Level: None

Program Level Opt.: None

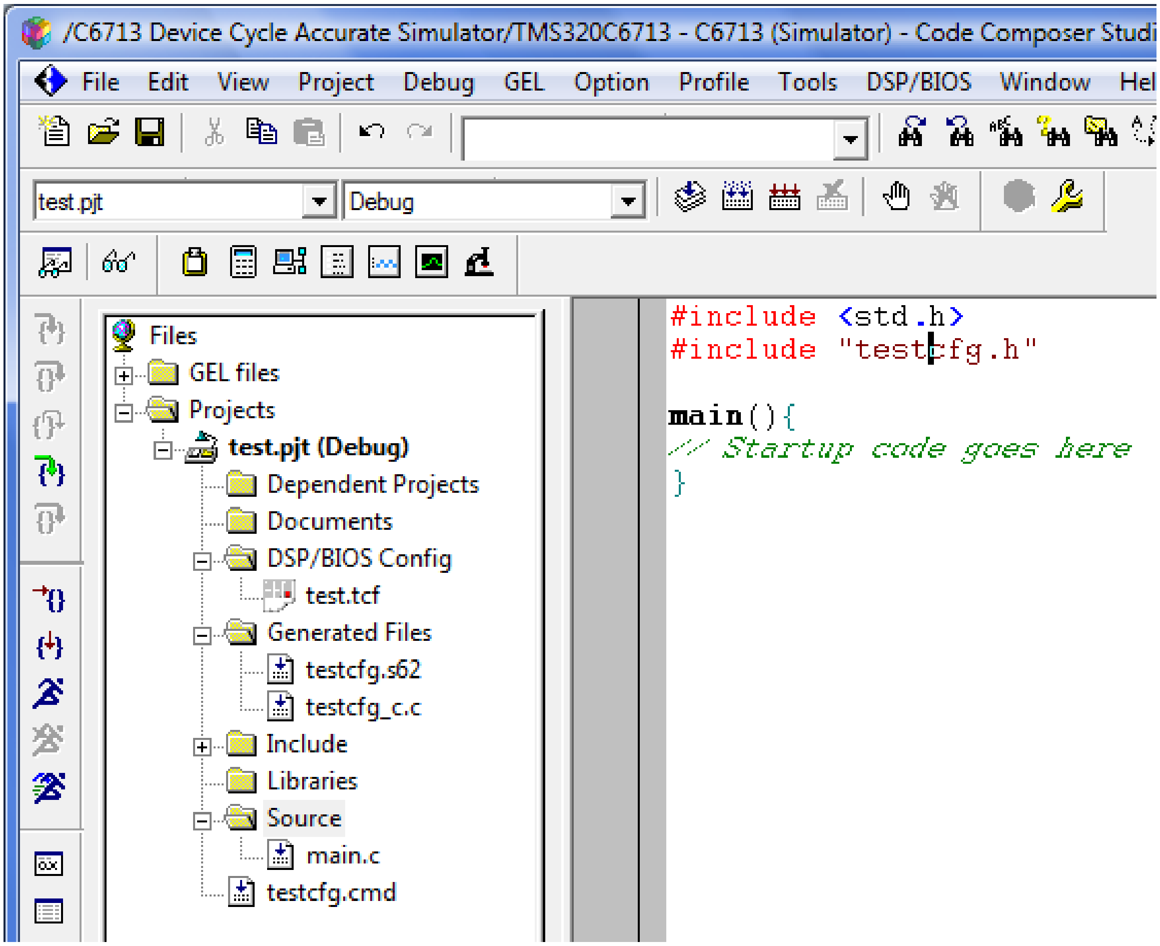
Figure 1.24: Processor selection in Build Options

- In order to build a project you must have files in it. To add files you must first create the files and save them. Create a new assembly code file (*.asm), C code file (*.c) or command file (*.cmd) by selecting File->New->Source File. When you save the file give it the correct extension. Add the files to the project by selecting Project->Add Files To Project... and selecting the files. You can see the files in your project by expanding the "+" signs. All projects will need source files and a command file.
- For projects that use DSP/BIOS, create a new DSP/BIOS Configuration file for the hardware you are using. Select File->New->DSP/BIOS Configuration ... If you are using the C6713 simulator then select the ti.platforms.sim67xx template.
- If you use the template there may be an error in the default settings. If there is no heep set up in any memory bank this must be done first. Click on System->MEM and check the Segment for DSP/BIOS Objects and Segment for malloc()/free(). If they say MEM_NULL you need to set up a memory heep. Go to the SDRAM under MEM, select the properties and check Create a heep in this memory. The size should be 0x00008000. Then go back to System->MEM and change the MEM_NULLs to SDRAM.
- Save the file as configuration file test.tcf and add it to your project. Also add the command file test.cmd file to your project.
- When developing your main C program for your project it should start with the following code example. Note that the `testcfg.h` file is created when the `test.tcf` file is compiled. Save the following code in `main.c` and add it to your project.

```
#include <std.h>
#include "testcfg.h"

main(){
```

```
// Startup code goes here
}
```



```
#include <std.h>
#include "testcfg.h"

main(){
// Startup code goes here
}
```

Figure 1.25: Project with tcf and cmd files added

- Set up the project for building your assembly programs by selecting Project->Build Options, clicking on the Linker tab and selecting No Autoinitialization for the Autoinit Model. When you build your C programs you will want to select Load-time Initialization.
- Once you have all your files in your project you can build the project to produce the object file (*.out). Select Project->Build to incrementally build the project or Project->Build All to rebuild all project files. The Project->Build will only perform functions that are needed to bring the project up to date. This option will usually be quicker than the Project->Build All option.
- Now that the project has been built you must load the object file onto the target board or simulator. Do this by selecting File->Load Program and selecting the *.out file. The file may be located in a Debug subdirectory.

- With the project loaded you can step through the code (Debug->Step Into), view the registers (View->CPU Registers->Core Registers) and debug the results. There are buttons on toolbars that you should become familiar with to aid in the debugging process.
- Since you will be often times loading a program immediately after building it you can set up an option so that the load will occur after you build your project. Select Option->Customize... and click on the Program/Project/CIO tab. Make sure the Load Program After Build box is checked.

1.4 Assembly Programming Lab[4]

1.4.1 Introduction

The purpose of this lab is to help you become familiar with the internal CPU of the 'C6713 processor. Programs are written in assembly. The first program is very simple so that you can make sure you know how to write an assembly program and debug it. The other two programs are a little more involved and help you understand the multiply and sum algorithm, delay slots and parallel instructions.

1.4.2 Part 1

- Make a new project titled lab1 using the 'C6713 Simulator.
- Write a program in assembly that multiplies the two constants 0x35 and 0x10. You will want to review the `MVK` and `MPY` instructions. `MVK` is used to put the constants in a register and the `MPY` is used to multiply them. Save this in a file called `lab1p1.asm`.
- Setup the program in Code Composer Studio and run it, recording the values in the registers at each step as you single step through the program. Be sure to note delay slots.

1.4.3 Part 2

- Write an assembly language program that performs a dot product of the following vectors **using a loop** that incorporates a conditional branch (The data for `a` and `b` should be stored in memory).

a = [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30]
b = [30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1]

$$y = \sum_k a_k \cdot b_k \tag{1.2}$$

Use the `ADD` command to add the data, `SUB` to update a counter and a conditional `B` to perform the loop.

- Save the program as `lab1p2.asm` and swap it for the other file in your project.
- Record the results of the run. Also record the number of clock cycles to execute the program. You do not need to optimize your program in this part.

1.4.4 Part 3

- Write a third program that performs the same operation as in the previous part but uses **parallel instructions** and the **register file cross path**. With the use of parallel instructions and the use of both register files the second program should take fewer instruction cycles. **Be sure to use a loop in your optimized program.**
- Save the program as `lab1p3.asm` and swap it for the other file in your project.
- Record the results of the run. Also record the number of clock cycles to execute the program.

[4] This content is available online at <http://cnx.org/content/m33373/1.4/>.

Chapter 2

C Programming Lab

2.1 Code Composer Studio v4 C Project[1]

2.1.1 Introduction

This module describes how to create a Code Composer Studio (CCS) project that executes a simple C program. The project does not use DSP/BIOS and runs on the CCS simulator. The code will be a simple "hello world" program. In this project the processor used is the TMS320C67xx but the process works for other processors as well.

2.1.2 Create a CCS project

To create a CCS project select **File->New->CCS Project**.

[1] This content is available online at <http://cnx.org/content/m36322/1.4/>.

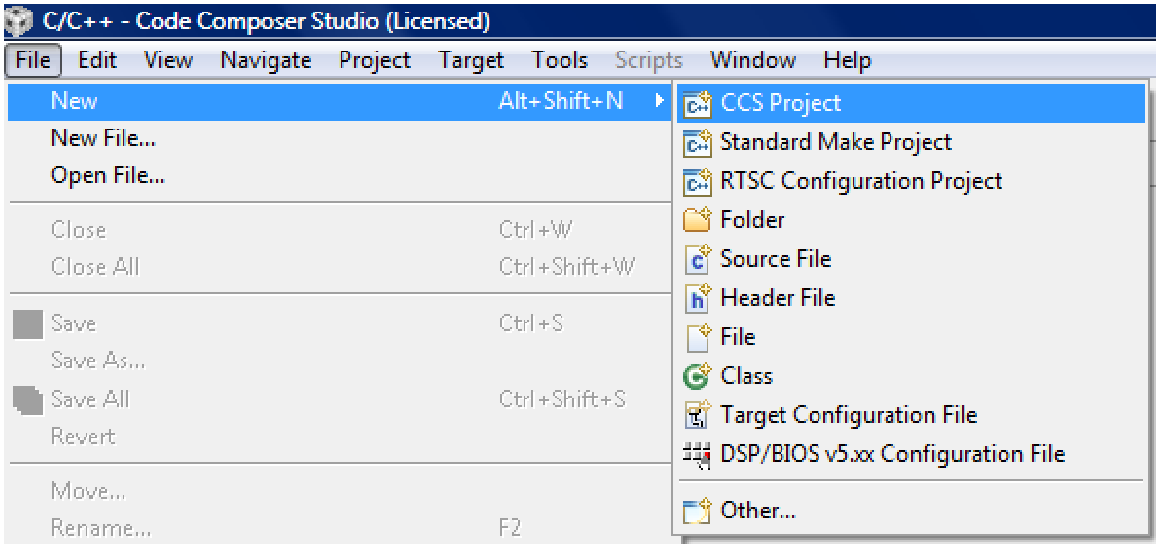
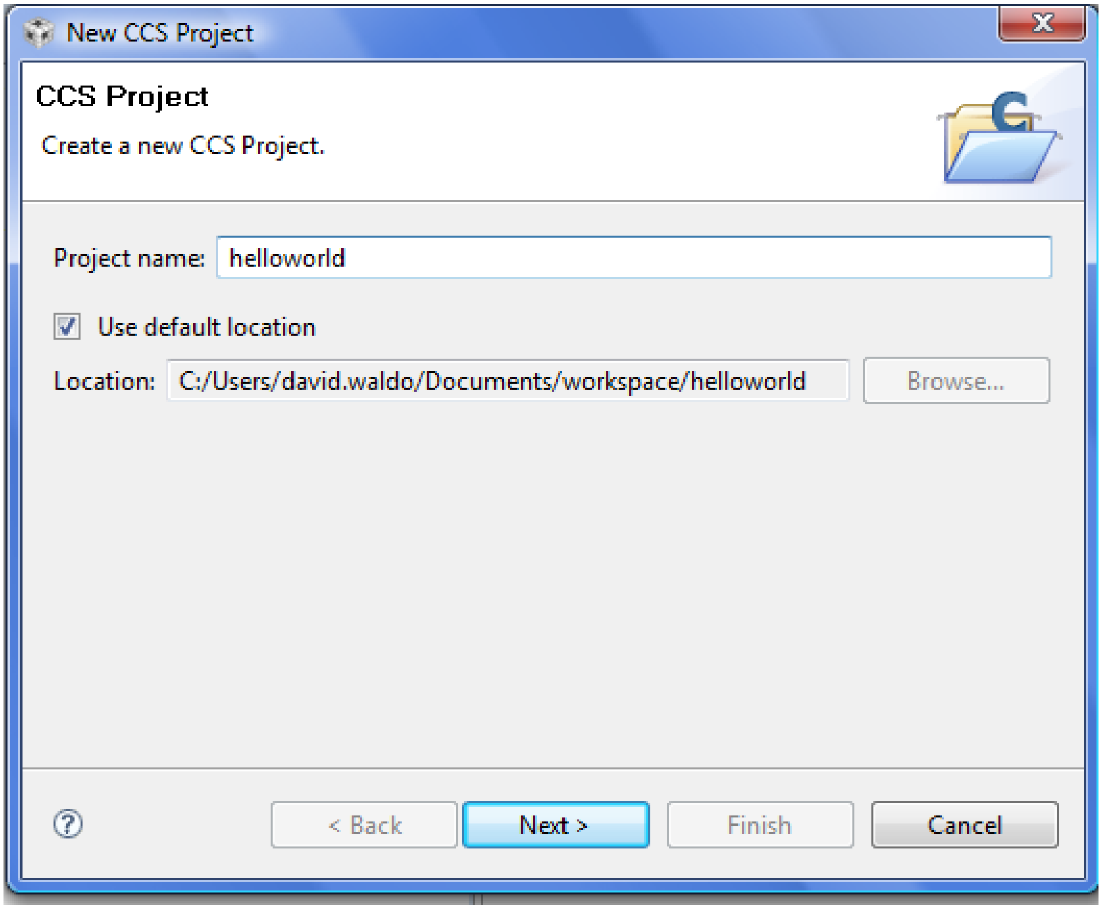


Figure 2.1: Screenshot of CCS Project menu

This will bring up a window where you can enter the name of the project. The location selected is the default location for project files. Press **Next**.



New CCS Project

CCS Project

Create a new CCS Project.

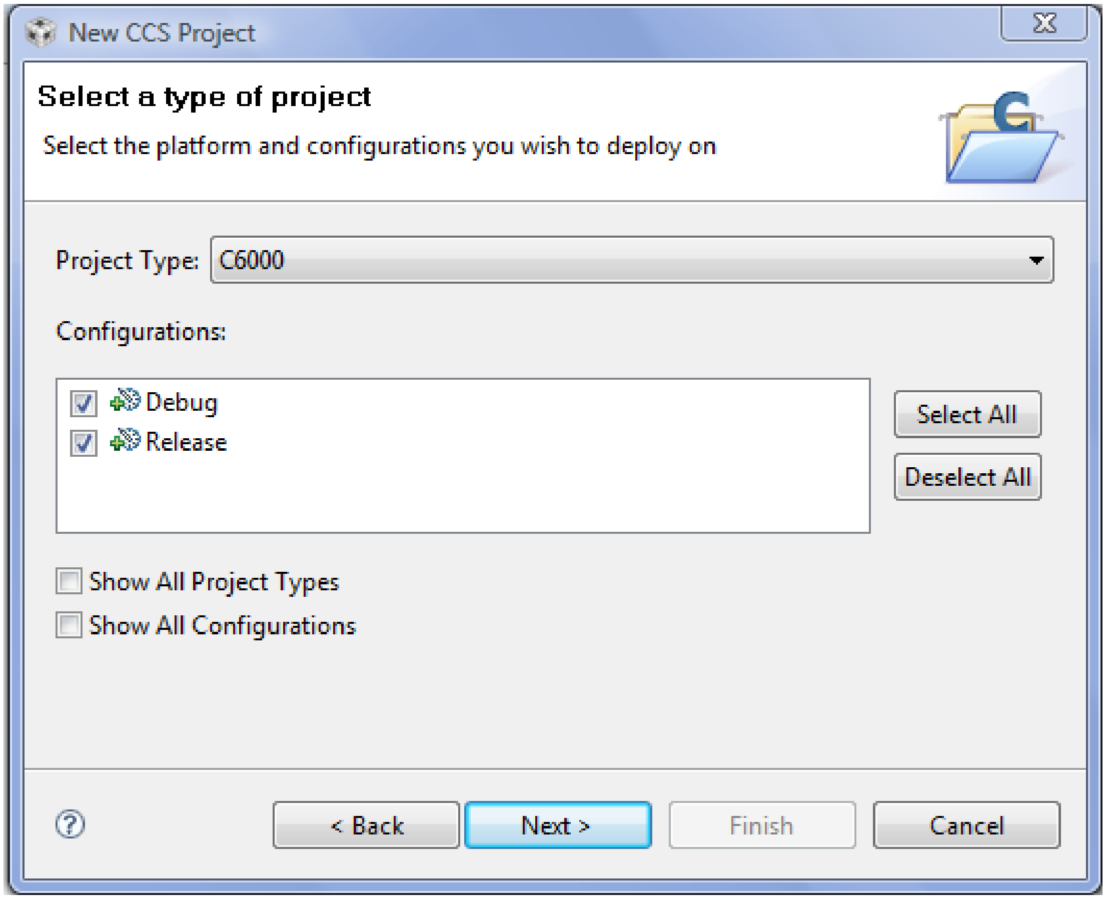
Project name: helloworld

☑ Use default location

Location: C:/Users/david.waldo/Documents/workspace/helloworld

Figure 2.2: CCS Project name and location

Since the example uses the TMS320C67xx processor the project type selected is **C6000**. The project configurations are **Debug** and **Release**. Select the **Next** button.



New CCS Project

Select a type of project

Select the platform and configurations you wish to deploy on

Project Type: C6000

Configurations:

☑ Debug

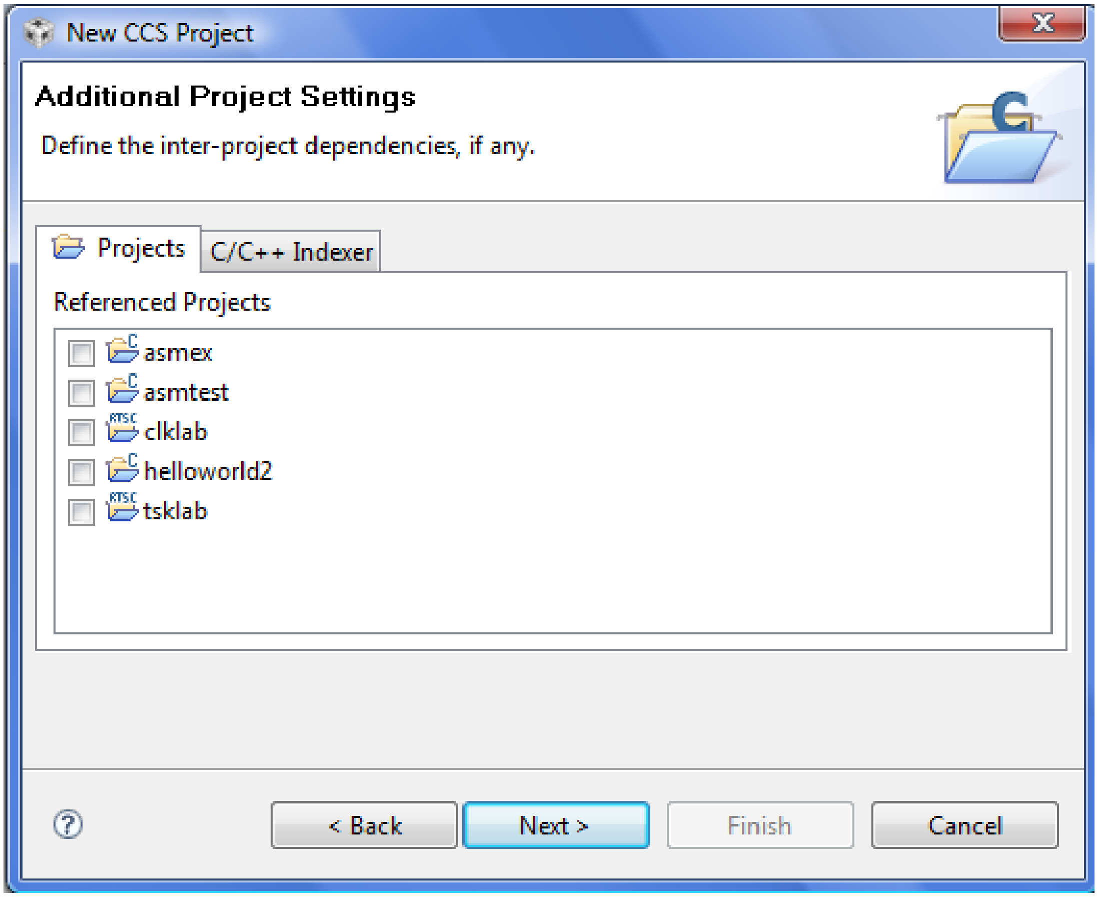
☑ Release

□ Show All Project Types

□ Show All Configurations

Figure 2.3: Type of CCS project

If there are any project dependencies they are selected on the next screen. Select the **Next** button.



New CCS Project

Additional Project Settings

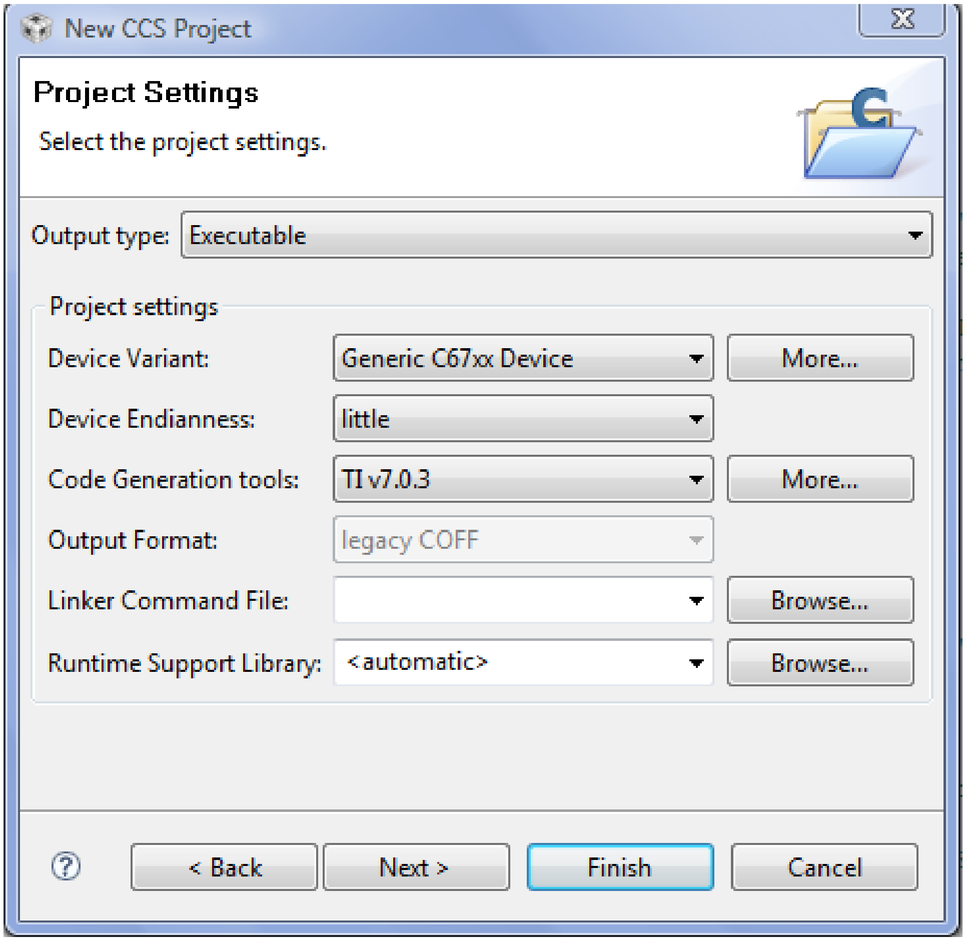
Define the inter-project dependencies, if any.

Referenced Projects

- □ asmex
- □ asmtest
- □ clklab
- □ helloworld2
- □ tsklab

Figure 2.4: CCS Project dependencies

On the **Project Settings Screen**, select the items that pertain to the type of project to be created. Since the project will be executed select **Output Type: Executable**. The processor type is TMS320C67xx so the **Device Variant** is **Generic C67xx Device**. This project will use **Little Endian**. The code generation tools are the latest version (in this case TI v7.0.3). The runtime support library is selected to match the device variant and the endianness selected. The library can be selected automatically. Press **Next**.



New CCS Project

Project Settings

Select the project settings.

Output type: Executable

Project settings

Device Variant: Generic C67xx Device

Device Endianness: little

Code Generation tools: TI v7.0.3

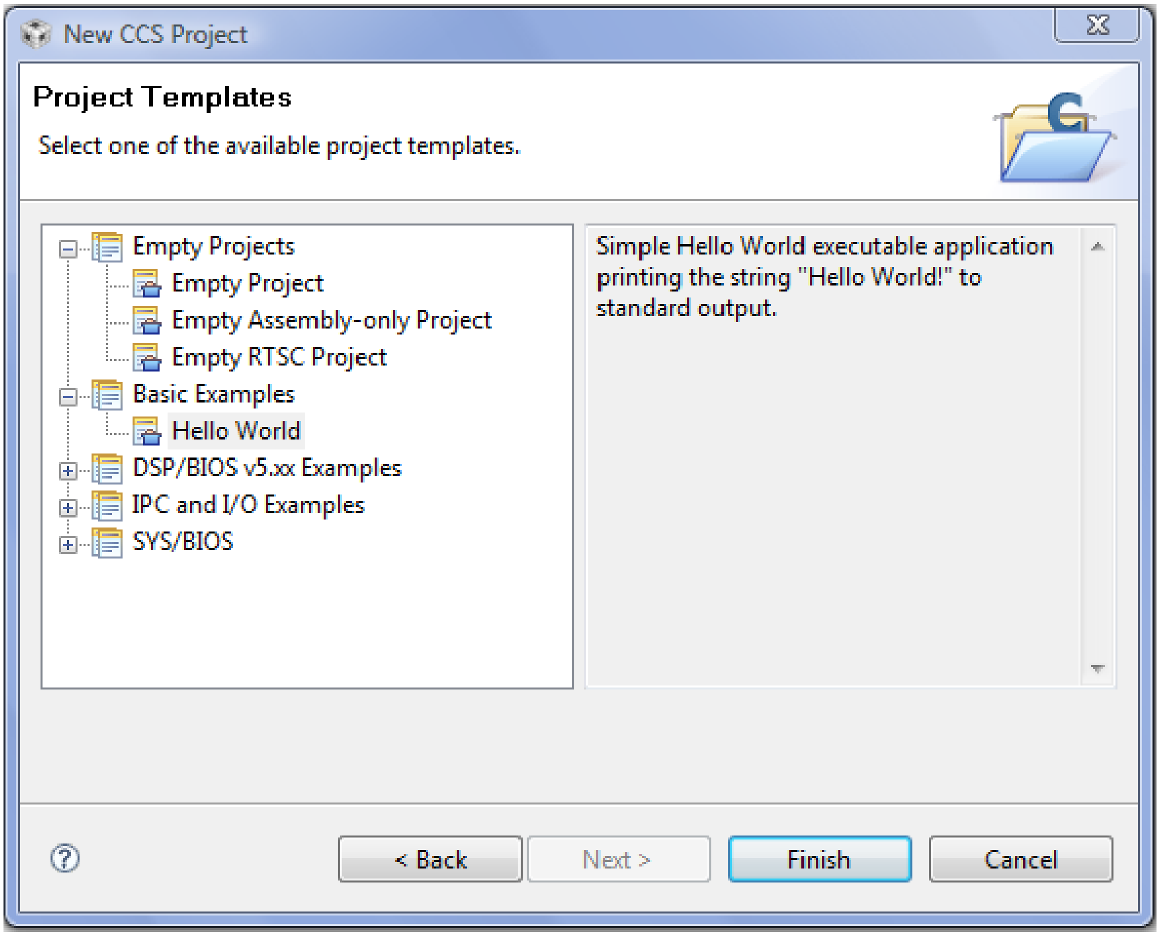
Output Format: legacy COFF

Linker Command File:

Runtime Support Library: <automatic>

Figure 2.5: CCS project settings

Since this project will be a simple C program project, select either the **Empty Project** or the **Hello World** example.



Select one of the available project templates.

Simple Hello World executable application printing the string "Hello World!" to standard output.

Figure 2.6

After the projects settings have been set, select **Finish** and the project will show up in the **C/C++ Projects** view.

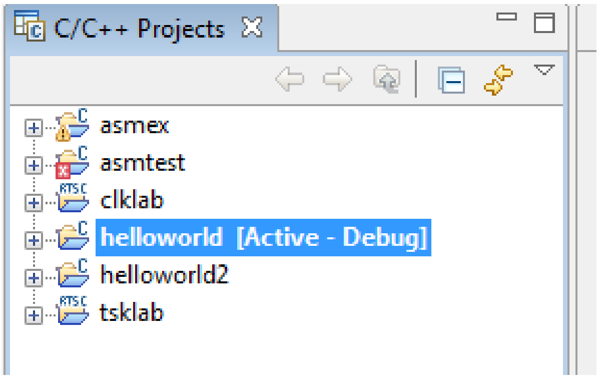


Figure 2.7: C/C++ projects view

2.1.3 Add code files

Projects in CCS must have a `main` function. This function gets executed first by default. So create a program file that contains a `main` function. Select **File->New->Source File**.

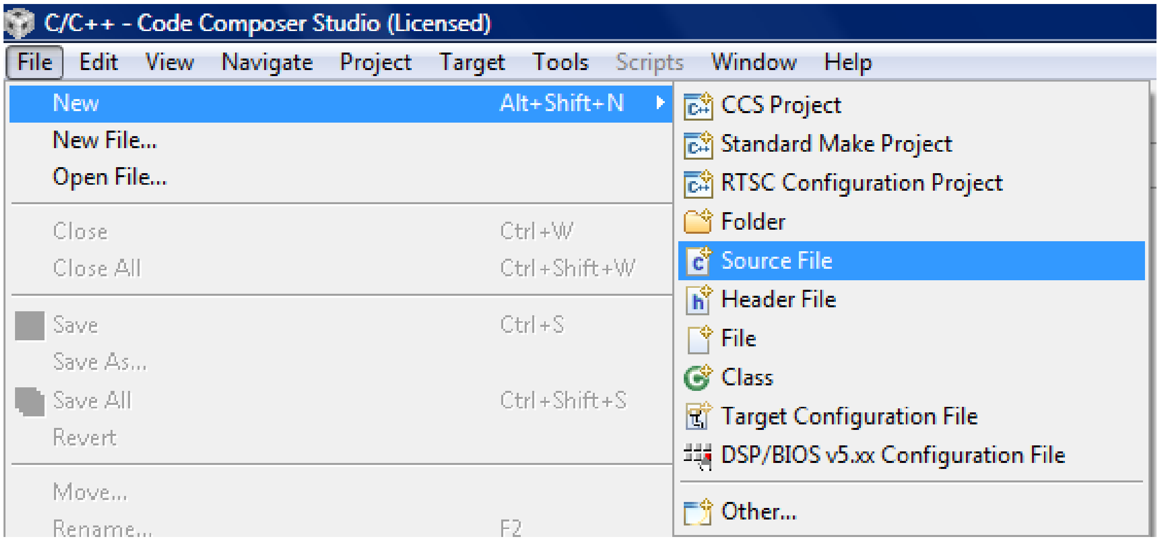
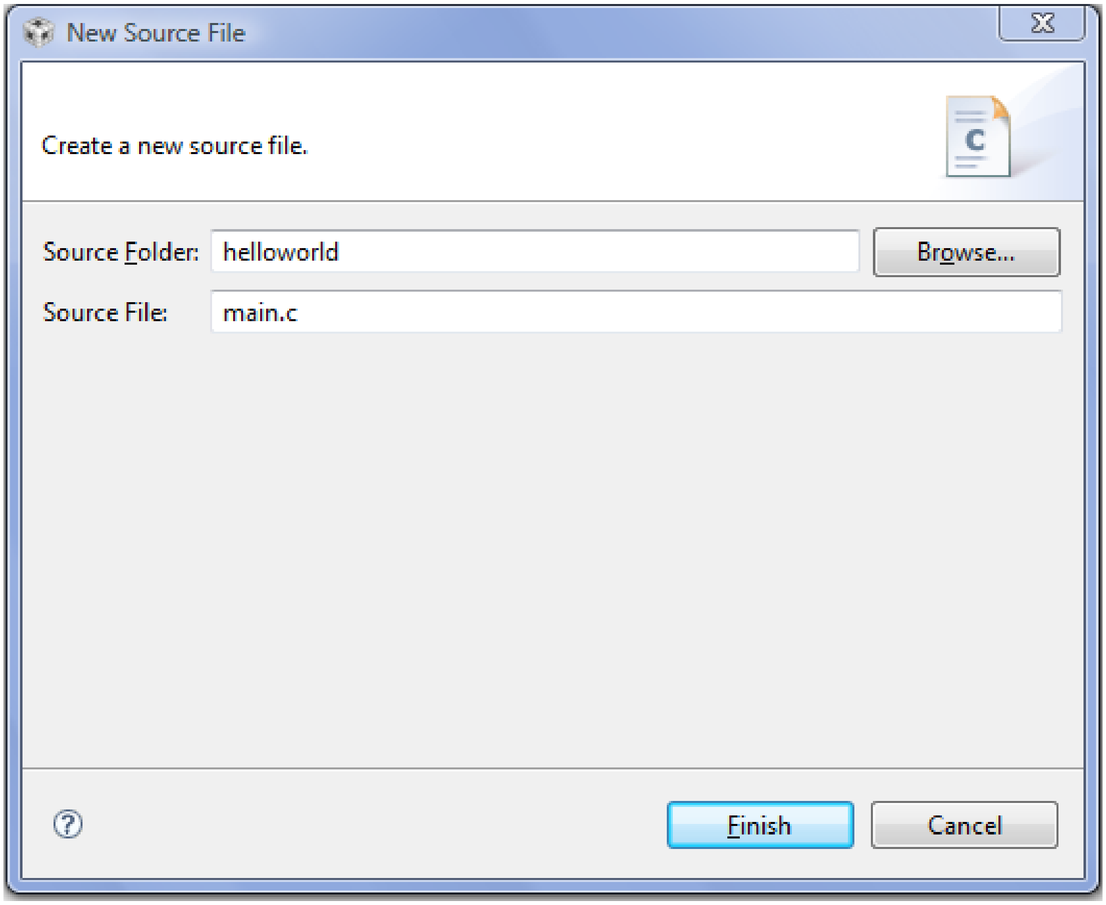


Figure 2.8: New source file

The **New Source File** dialog opens and you can enter the source file name. Add a **.c** extension for a C program file. The source folder should be the folder of your project. Select **Finish** and a new file is opened.



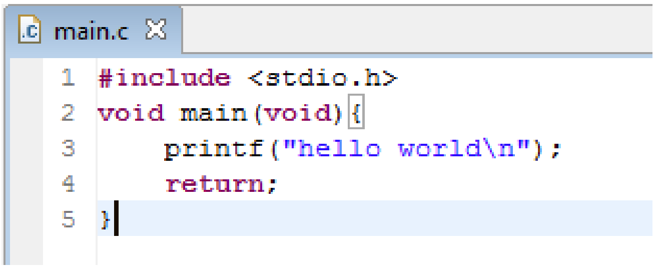
Create a new source file.

Source Folder: helloworld

Source File: main.c

Figure 2.9: New source file dialog

Enter the C code. The file must contain a **main** function. After entering the code, save the file.



```
#include <stdio.h>
void main(void){
    printf("hello world\n");
    return;
}
```

Figure 2.10: Code in main.c

2.1.4 Create a target configuration

In order to build and run your project you need to have a target configuration. This file will contain information about the emulator that will be used and the target processor. In this module the target configuration will be the TI simulator for the C6713 processor.

First select **File->New->Target Configuration File**.

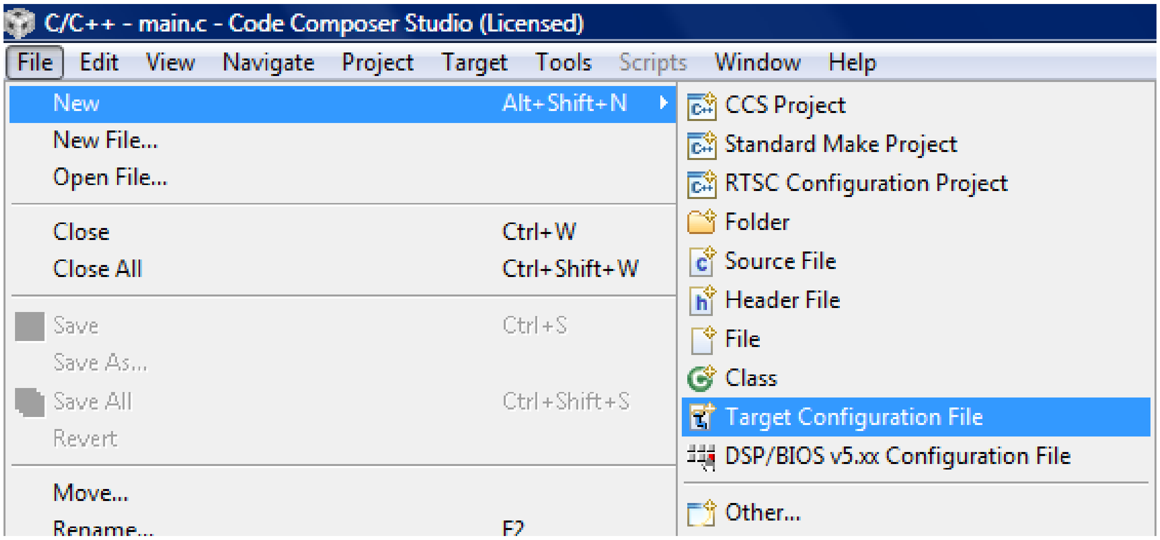
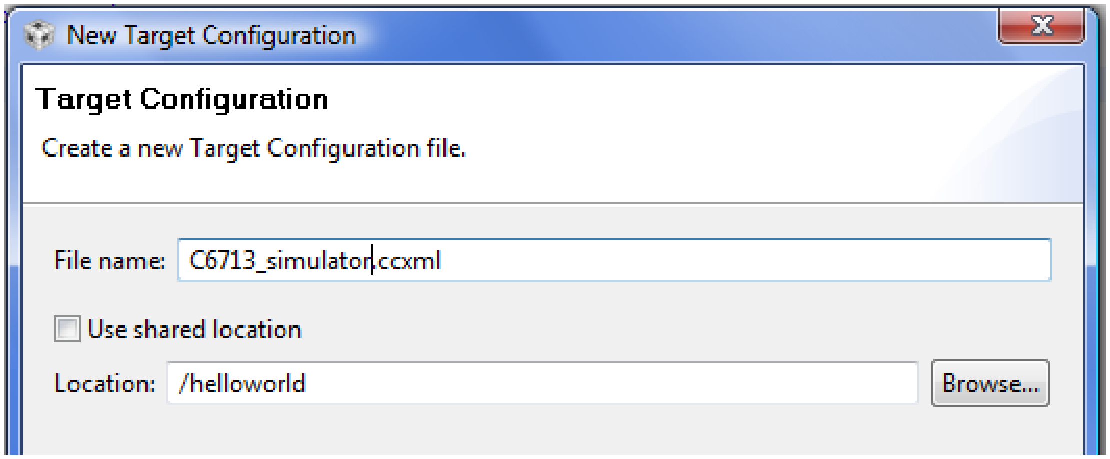


Figure 2.11: Open new target configuration file

In the New Target Configuration dialog enter the configuration file name ending in the **.ccxml** file extension. Put the file in the project directory. Select **Finish** and a new target configuration file opens.



New Target Configuration

Target Configuration

Create a new Target Configuration file.

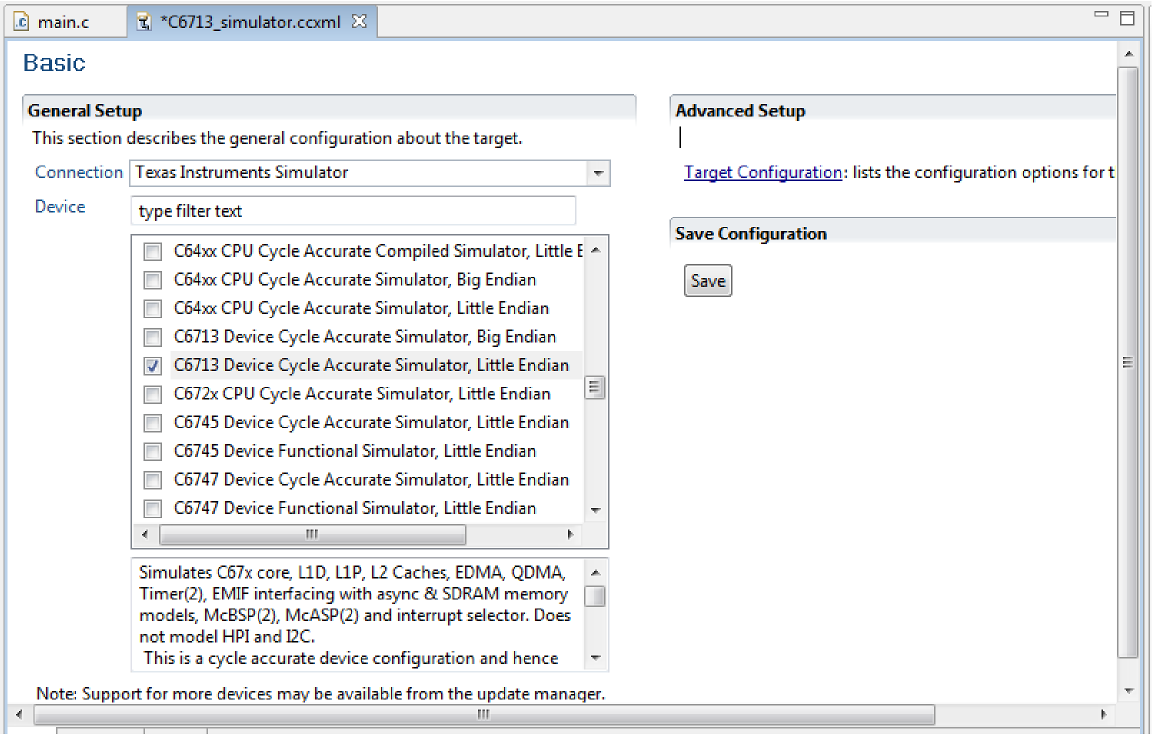
File name: C6713_simulator.ccxml

□ Use shared location

Location: /helloworld

Figure 2.12: Target configuration dialog

In the configuration, select the connection needed (simulator or some type of emulator, etc.) and the type of device. In this example the TI simulator is used with the C6713 little endian simulator device. Select **Save** when done.



Basic

General Setup

This section describes the general configuration about the target.

Connection: Texas Instruments Simulator

Device: type filter text

- ☐ C64xx CPU Cycle Accurate Compiled Simulator, Little E
- ☐ C64xx CPU Cycle Accurate Simulator, Big Endian
- ☐ C64xx CPU Cycle Accurate Simulator, Little Endian
- ☐ C6713 Device Cycle Accurate Simulator, Big Endian
- ☑ C6713 Device Cycle Accurate Simulator, Little Endian
- ☐ C672x CPU Cycle Accurate Simulator, Little Endian
- ☐ C6745 Device Cycle Accurate Simulator, Little Endian
- ☐ C6745 Device Functional Simulator, Little Endian
- ☐ C6747 Device Cycle Accurate Simulator, Little Endian
- ☐ C6747 Device Functional Simulator, Little Endian

Simulates C67x core, L1D, L1P, L2 Caches, EDMA, QDMA, Timer(2), EMIF interfacing with async & SDRAM memory models, McBSP(2), McASP(2) and interrupt selector. Does not model HPI and I2C.
This is a cycle accurate device configuration and hence

Note: Support for more devices may be available from the update manager.

Advanced Setup

Target Configuration: lists the configuration options for t

Save Configuration

Save

Figure 2.13: Target configuration setup

Now the target configuration is in the project.

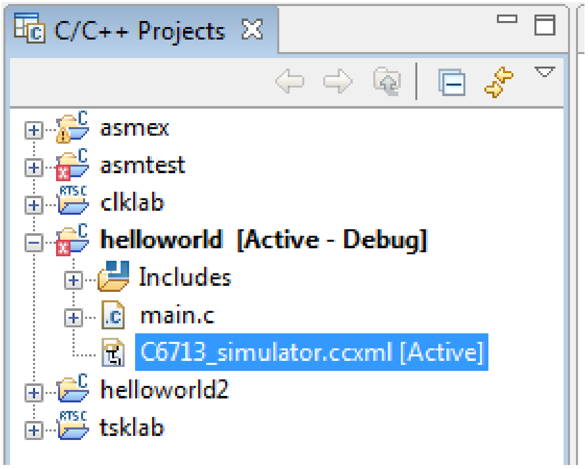
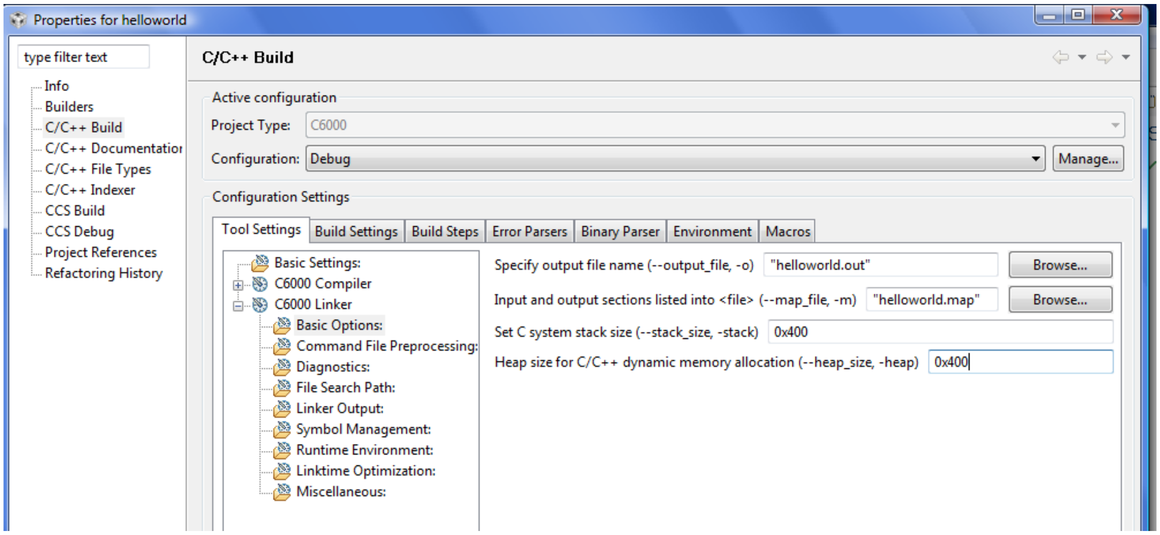


Figure 2.14: Project files

2.1.5 Debug the project

The default project settings do not have the stack and heap sizes defined. This will produce warnings when building the project. To change the project properties, select **Project->Properties** and then click on **C/C++ Build** on the left and the **C6000 Linker/Basic Options** under **Tool Settings**. For the stack size and heap size enter 0x400 and click **OK**.



Project Type: C6000

Configuration: Debug

Specify output file name (--output_file, -o) "helloworld.out"

Input and output sections listed into <file> (--map_file, -m) "helloworld.map"

Set C system stack size (--stack_size, -stack) 0x400

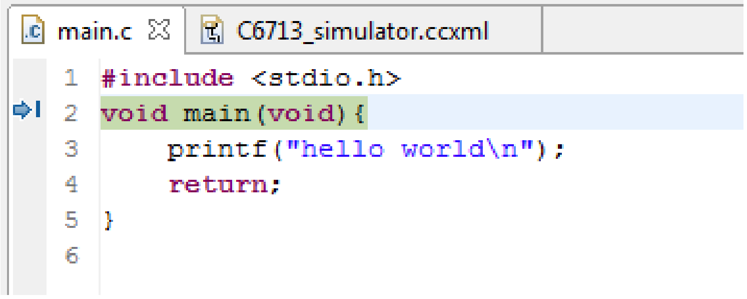
Heap size for C/C++ dynamic memory allocation (--heap_size, -heap) 0x400

Figure 2.15: Setting the stack and heap sizes

To start a debug session either select **Target->Debug Active Project** or press the debug icon in the menu bar.

Figure 2.16: Debug icon

When the debug session starts the code files are compiled and linked into an executable file and the file is loaded onto the target. The initialization code is executed and the program is halted at the beginning of the `main` function.



```
#include <stdio.h>
void main(void){
    printf("hello world\n");
    return;
}

```

Figure 2.17: Debugging session

Icons on the debug window can be used to run the program and other debugging tools.

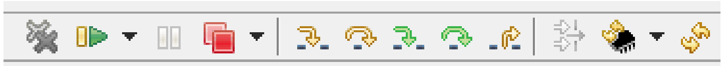
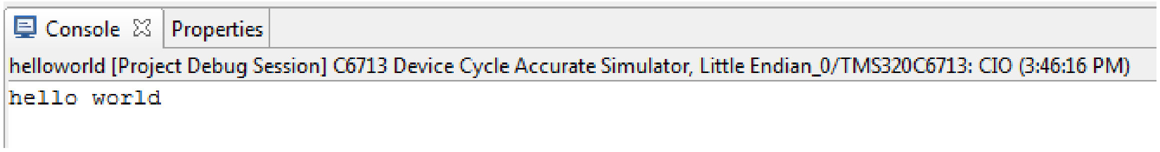


Figure 2.18: Debugging tools

Press the green arrow to run the program. The `main` function will be executed and the program will leave the `main` function. In the program the printf function prints to the console which if not shown by default can be viewed by selecting **View->Console**. In the console is shown the "hello world" that was printed.



```
helloworld [Project Debug Session] C6713 Device Cycle Accurate Simulator, Little Endian_0/TMS320C6713: CIO (3:46:16 PM)
hello world
```

Figure 2.19: Console output of the program

2.2 C Programming Lab[2]

2.2.1 Skills

This laboratory deals with compiling and running a C program on the TMS320C6713 DSP using Code Composer Studio. By the end of the laboratory you will be able to:

- Compile and build a C language program
- Run a C language program on the TMS320C6713 DSP
- Link multiple files with functions into one project

2.2.2 Reading

- SPRU 187: Optimizing C Compiler
- C programming reference card[3]

2.2.3 Description

If you have forgotten how to program in C/C++ you should review C programming on the many tutorials on the internet or look through your favorite textbook or reference book. This document will not give you a tutorial on how to program in C/C++.

You must remember that when programming the TI DSP you are loading your program into a system that has its own DSP and memory and is apart from your computer. In order for you to print anything, the data must be sent from the DSP board to CCS and then displayed in the stdio display. This can be very slow so you will want to use the debugging tools to view variables.

In this lab you will be writing very simple C programs. If you want to print something you will obviously need to include the stdio.h file.

Your main program should look like:

```
#include <stdio.h>
main()
{
variables
code
}
```

Be sure to use good programming style.

2.2.4 Pre-Laboratory

Write a C program that performs a dot product of the following vectors. The data for `a` and `b` should be stored in integer arrays. Use a *for* loop.

a = [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30]

b = [30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1]

$$y = \sum_k a_k \cdot b_k \tag{2.1}$$

[2] This content is available online at <http://cnx.org/content/m36345/1.3/>.

[3] http://www.cise.ufl.edu/class/cop4600/docs/C_Ref_Card.pdf

2.2.5 Laboratory

2.2.5.1 Part 1

- In this part you will create a project that performs the array multiply and sum within the main function of the program. In order to be able to accurately count clock cycles, use the C6713 simulator.
- Create a new project, call it clab.
- Create your C file. Call it `main.c`. Add this file to your project.
- You should use the following options for the project (Project->Build Options):
- After the project has been loaded, record the values in the variables at each step as you single step through the program. Count the number of clock cycles and compare to the assembly program completed in a previous lab. Be sure to zero the counter before stepping through your code.

2.2.5.2 Part 2

- In this part you are going to write a function that performs an array multiply and sum algorithm and returns the sum. This function will be located in another file and you will also make a header file.
- Create a C file called `multsum.c`. In this file make a function called `multsum`. The function should have as inputs:

- pointer to integer array `a`
- pointer to integer array `b`
- number of elements in the arrays
- The output of the function should be the integer sum. The function should perform the multiply and sum algorithm.

- Create a header file for the multsum function and call it `multsum.h`. In the header include the following code:

```
   #ifndef _MULTSUM_H_
#define _MULTSUM_H_
extern int multsum(int *a, int *b, int length);
#endif /*MULTSUM_H_*/
```

This uses some compiler directives to determine if the current header file has already been included by another include file. The macro `_MULTSUM_H_` is not defined the first time it is encountered so the macro is defined and the definition is included. The `extern` keyword tells the compiler that the function is defined in another file.

- In `main.c`:

- include the `multsum.h` file

```
#include "multsum.h"
```

- declare the `a` and `b` arrays
- call the `multsum` function with the sum returned to a variable

- Include the `multsum.c` file in your project.
- Run the program and verify that the sum that is returned is correct.

Chapter 3

Echo Lab

3.1 Circular Buffers[1]

3.1.1 Introduction

When performing DSP algorithms data needs to be stored as it is being input. Only a certain amount of data is kept. New data is saved and the old data is removed. This is a first-in-first-out (FIFO) buffer. There are two ways to implenent the FIFO buffer. One is to shift the data and the other is to implement a circular buffer.

3.1.2 Shifted buffer

There are two ways of implementing a FIFO buffer. Both methods start with defining an array that is the length of data needed to be stored. The first method of implementing a FIFO buffer is to add the new data to the beginning of the buffer and shifting the old data in the buffer. The last value that is shifted out is discarded or used before it is removed.

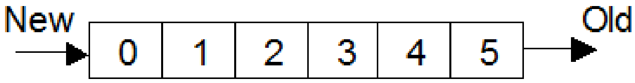


Figure 3.1

To implement the FIFO buffer by shifting the following pseudo-code can be used.
N - number of coefficients
x(n) - stored input data, n = 0...N-1
input_sample - variable that contains the newest input sample

```
// Shift in the new sample. Note the data is shifted starting at the end.
for (i=N-2; i>=0; i--){
x[i+1] = x[i];
}
x[0] = input_sample
```

[1] This content is available online at <http://cnx.org/content/m36654/1.2/>.

3.1.3 Circular buffer

The second method is the more preferable method since it takes less processing. In this method a circular buffer is implemented. An index keeps track of the current point in the buffer. The data in the buffer does not get shifted, the index gets incremented and wrapped around when it gets to the end of the buffer. The element after the current index is the last value in the circular buffer. The following figure shows the circular buffer after the index has already wrapped around.

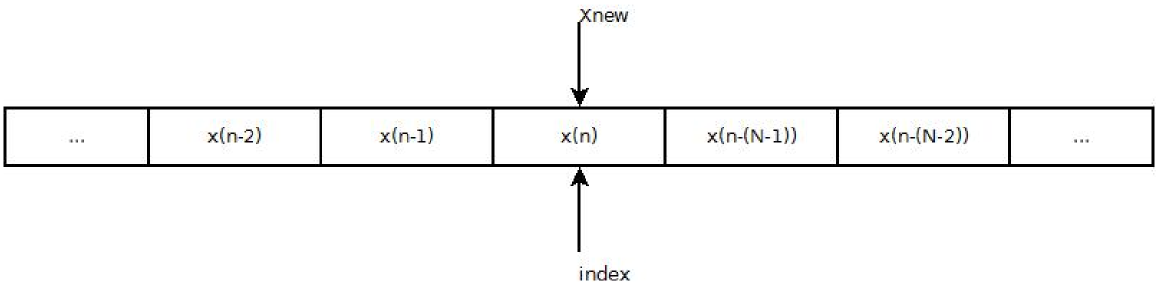


Figure 3.2

In the following figure is an example of a FIFO circular buffer with the number elements equal to 7. In the figure the current index is 3 and that is where the new sample is stored. The last element in the buffer is in element 4.

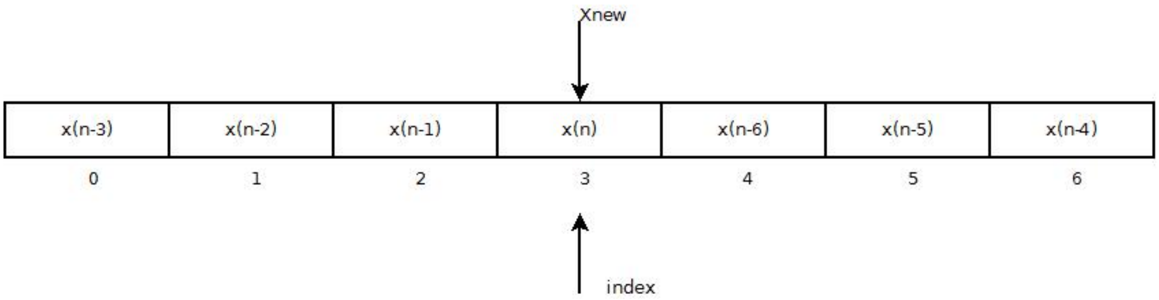


Figure 3.3: FIFO buffer implemented using a circular buffer

When a new value is received, the index is incremented (and possibly wrapped around to zero), the old value where the index points can be saved in a temporary variable to be used later and the new value is written where the index points thus deleting the old value. After another increment the new representation of the data will look like the following figure.

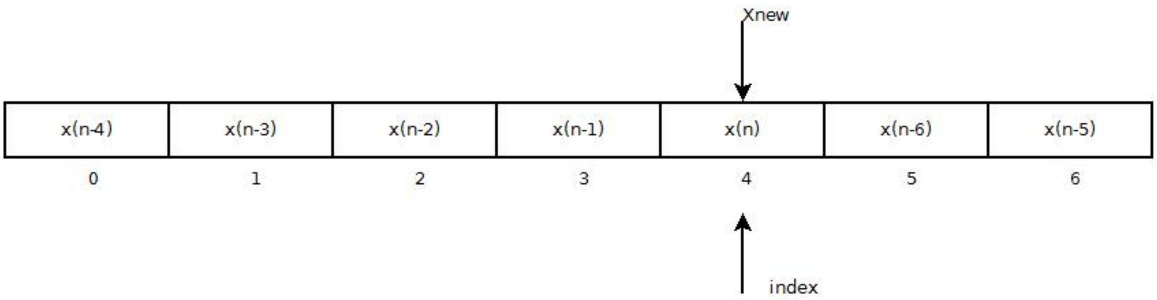


Figure 3.4: FIFO buffer implemented using a circular buffer after increment

In terms of signal processing notation, the value at the index 4 is x(n). Since the value at the index 3 came in one time sample before the index 4, it corresponds to x(n-1). Similarly, the value at index 2 is x(n-2).

The following code shows pseudo-code that can be used to implement a circular buffer.

`N` - number of coefficients

`x(n)` - stored input data, n = 0...N-1

`input_sample` - variable that contains the newest input sample

`index` - location in the buffer where the current sample is stored

```
index = (index+1)%N; // use the remainder to wrap around the index
x(index) = input_sample;
```

3.2 Single Sample and Block I/O[2]

3.2.1 Introduction

Signal processing applications that run in "real-time" input data, process the data, and then output the data. This can be done either on a single sample basis or a block of data at a time.

3.2.2 Single Sample I/O

A simple way to perform signal processing algorithms is to get one input sample from the A/D converter, process the sample in the DSP algorithm, then output that sample to the D/A converter. When I/O is done this way there is a lot of overhead processing to input and output that one sample. As an example, suppose we will process one sample at a time with the following equation

$$y(n) = 2x(n) \tag{3.1}$$

where $x(n)$ is the input and $y(n)$ is the output. Here is how the processing takes place. When an interrupt occurs, systems usually input a sample and output a sample at the same time. Suppose we have a value for the output, y(0), in memory before the interrupt occurs.

[2] This content is available online at <http://cnx.org/content/m36655/1.1/>.

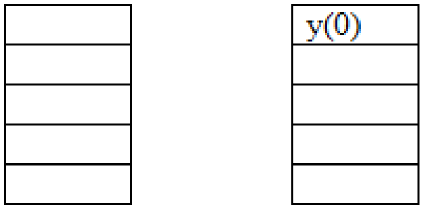


Figure 3.5

Then when the interrupt occurs, the y(0) is output and x(0) is input.

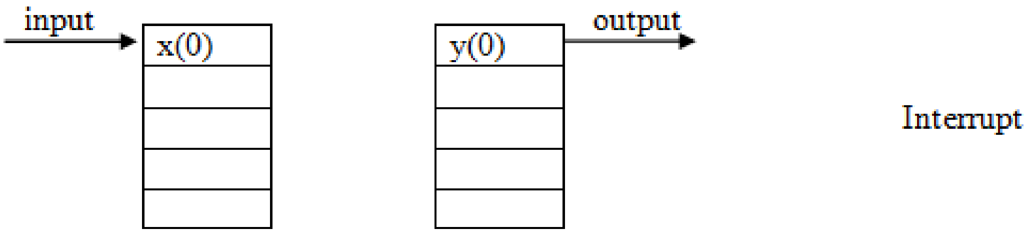


Figure 3.6

After this we can use x(0) to compute the next output. So y(1) = 2x(0).

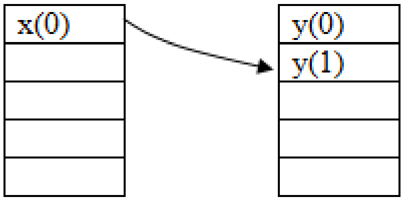


Figure 3.7

Then another interrupt occurs and the next values are input and output.

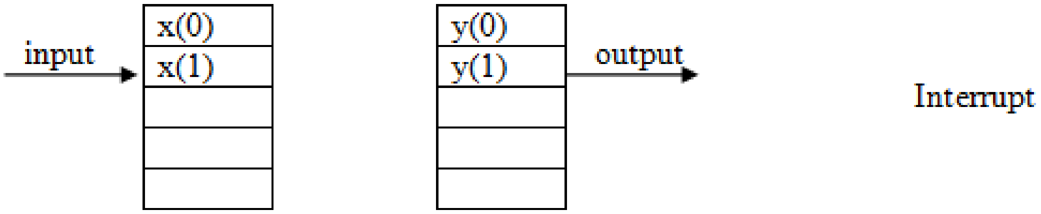


Figure 3.8

And the process repeats.

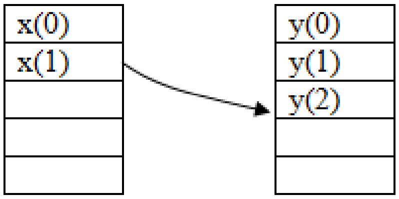


Figure 3.9

Notice that the processing of the input sample must be done before the next interrupt occurs or else we won't have any data to output. Also notice that the output is always one sample delayed from the input. This means that the current input cannot be instantly output but must wait for the next interrupt to occur to be output.

3.2.3 Block I/O

In order to minimize the amount of overhead caused by processing one sample at a time in-between each interrupt, the data can be buffered and then processed in a block of data. This is called "block processing." This is the method that will be used in this module and the Code Composer Studio (CCS) project. The benefit of block processing is it reduces overhead in the I/O processes. The drawback to block processing is that it introduces a larger delay in the output since a block of data must be input before any data is processed and then output. The length of the block determines the minimum delay between input and output.

Suppose we are processing the data with the same algorithm as above:

$$y\left(n\right) = 2x\left(n\right) \tag{3.2}$$

Also, suppose we are going to process the data with a block size of 3. This means we need to start with 3 values in our output buffer before we get more data.

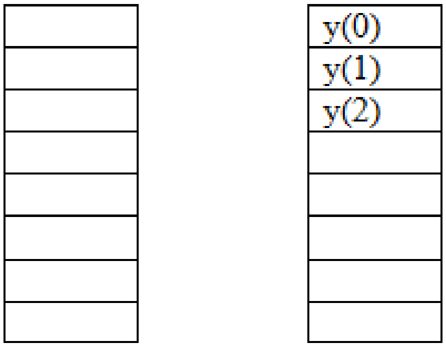


Figure 3.10

Three interrupts occur and we get 3 new values and output the three values that are in the output buffer.

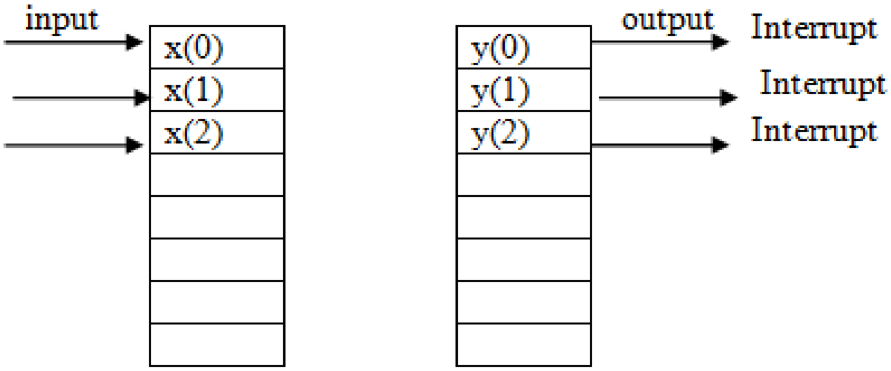


Figure 3.11

At this point we process the values that were received.

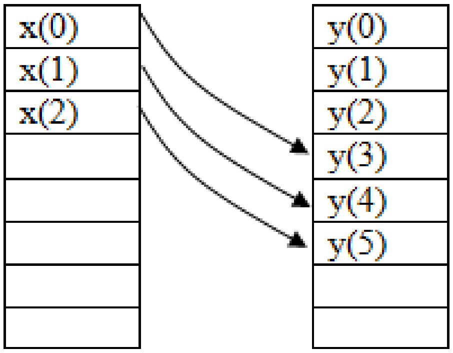


Figure 3.12

Then it starts all over again. Notice how far down x(0) is moved. This is the delay amount from the input to the output.

3.3 Code Composer Studio v3.3 DSP/BIOS and C6713 DSK[3]

3.3.1 Introduction

This module describes the TMS320C6713 DSK development board and how to use it in a Code Composer Studio (CCS) v3.3 project that uses DSP/BIOS. An example project is included.

3.3.2 Reading

- TMS320C6713 DSK Technical Reference
- SLWS106D: TLV320AIC23 Technical Manual
- SPRA677: A DSP/BIOS AIC23 Codec Device Driver for the TMS320C6713 DSK
- SPRU616: DSP/BIOS Driver Developer's Guide
- SPRA846: A DSP/BIOS EDMA McBSP Device Driver for TMS320C6x1x DSPs

3.3.3 Project Files

The files referred to in this module can be found in this ZIP file: DSK6713_audio.zip[4]

3.3.4 DSK Hardware

The following figure shows the block diagram of the TMS320C6713 DSK hardware. The heart of the DSK is the TMS320C6713 DSP chip which runs at 225 MHz. The DSP is in the center of the block diagram and connects to external memory through the EMIF interface. There are several devices connected to this interface. One device is a 16 Mbyte SDRAM chip. This memory, along with the internal DSP memory, will be where code and data are stored.

[3] This content is available online at <http://cnx.org/content/m36721/1.2/>.

[4] http://cnx.org/content/m36656/latest/DSK6713_audio.zip

On the DSK board there is a TLV320AIC23 (AIC23) 16-bit stereo audio CODEC (coder/decoder). The chip has a mono microphone input, stereo line input, stereo line output and stereo headphone output. These outputs are accessible on the DSK board. The AIC23 figure shows a simplified block diagram of the AIC23 and its interfaces. The CODEC interfaces to the DSP through its McBSP serial interface. The CODEC is a 16-bit device and will be set up to deliver 16-bit signed 2's complement samples packed into a 32-bit word. Each 32-bit word will contain a sample from the left and right channel in that order. The data range is from - $2^{(16-1)}$ to ($2^{(16-1)}$-1) or -32768 to 32767.

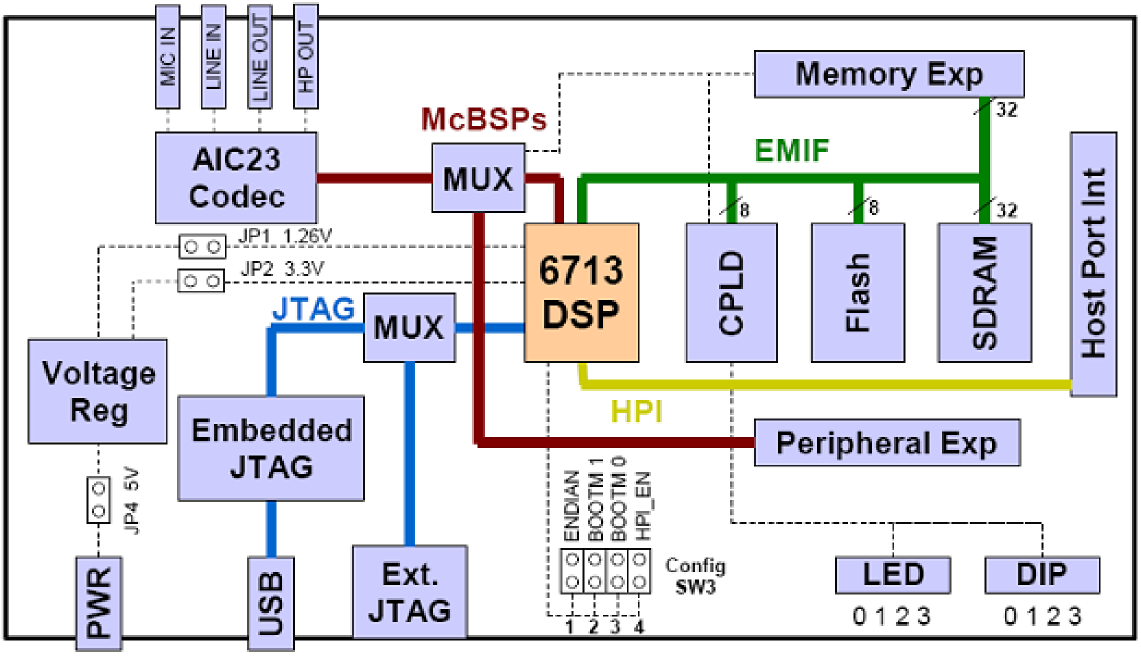


Figure 3.13: TMS320C613 DSK Block Diagram taken from TMS320C6713 DSK Technical Reference

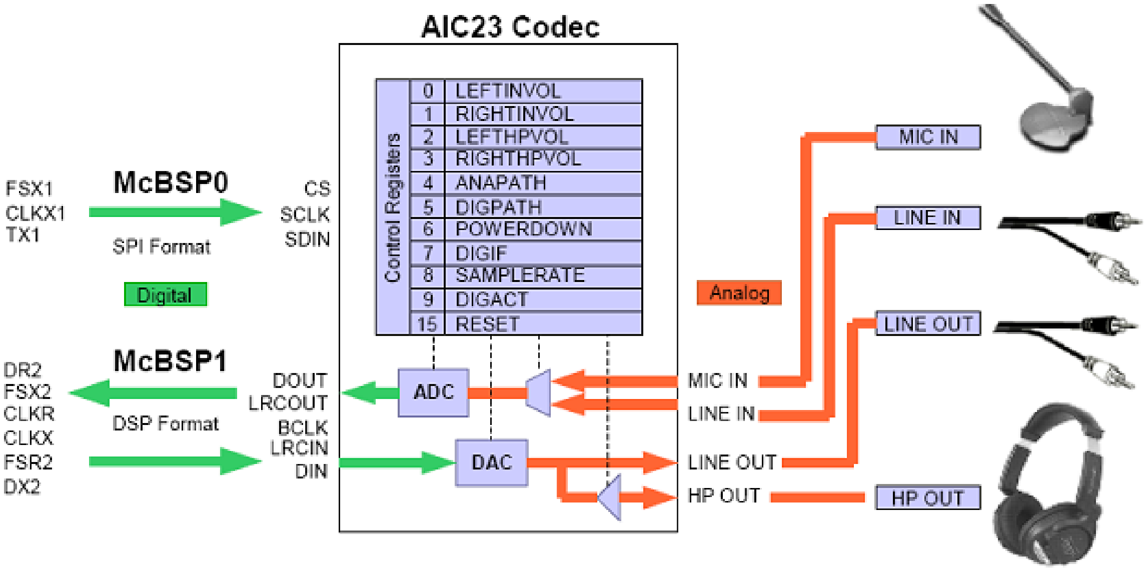


Figure 3.14: Simplified AIC23 CODEC Interface taken from TMS320C6713 DSK Technical Reference

3.3.5 DSK6713 Audio Project Framework

The following figure shows a diagram of the software that will be used in this module. Texas Instruments has written some drivers for the McBSP that get data from the AIC23 and write data to the AIC23. The input data is put into an input stream (input buffer) and the output data is read from an output stream (output buffer). The signal processing software simply needs to get a buffer from the input stream, process the data, then put the resulting buffer in the output stream.

The project is set up using DSP/BIOS, a real time operating systems developed by TI. This module does not explain how to use DSP/BIOS but it will explain what objects are used in this project. The main objects are an input stream, `inStream`, an output stream, `outStream`, and a task, `TSK_processing`, which uses the function `processing()`.

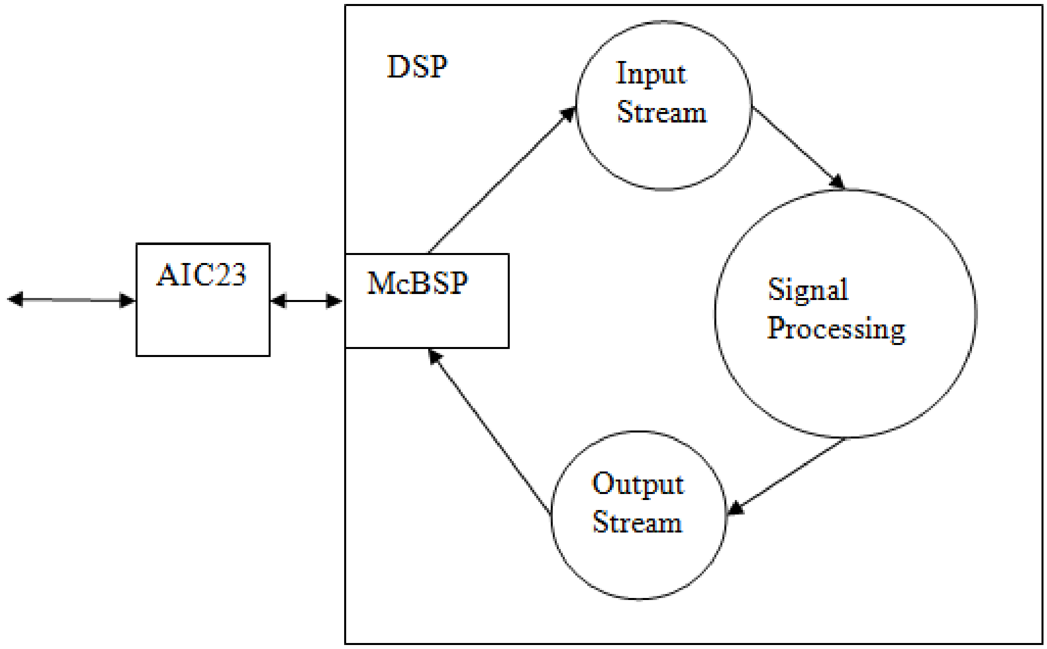


Figure 3.15: Block diagram of the signal processing framework.

There are a few things that can be easily modified by the user in the main program file `DSK6713_audio.c`.

3.3.5.1 CODEC Setup

The CODEC can be set up for different inputs and different sample rates. In the program there is a variable that sets up the registers in the CODEC:

```
DSK6713_EDMA_AIC23_DevParams AIC23CodecConfig ={
0x0000AB01, //VersionId
0x00000001, //cacheCalls
0x00000008, //irqId
AIC23_REG0_DEFAULT,
AIC23_REG1_DEFAULT,
AIC23_REG2_DEFAULT,
AIC23_REG3_DEFAULT,
AIC23_REG4_LINE,// Use the macro for Mic or Line here
AIC23_REG5_DEFAULT,
AIC23_REG6_DEFAULT,
AIC23_REG7_DEFAULT,
AIC23_REG8_48KHZ,// Set the sample rate here
AIC23_REG9_DEFAULT,
0x00000001, //intrMask
0x00000001 //edmaPriority
};
```

This is set up to use the Line In as the input with the macro AIC23_REG4_LINE. If the microphone input is needed then change this macro to AIC23_REG4_MIC. The sample rate is set to 48kHz with the macro AIC23_REG8_48KHZ. This sample rate can be changed by changing the line to one of the following macros:

```
AIC23_REG8_8KHZ
AIC23_REG8_32KHZ
AIC23_REG8_44_1KHZ
AIC23_REG8_48KHZ
AIC23_REG8_96KHZ
```

3.3.5.2 Signal Processing Function

The main part of the software is an infinite loop inside the function processing(). The loop within the function is shown here.

```
while(1) {
/* Reclaim full buffer from the input stream */
if ((nmadus = SIO_reclaim(&inStream, (Ptr *)&inbuf, NULL)) < 0) {
SYS_abort("Error reclaiming full buffer from the input stream");
}

/* Reclaim empty buffer from the output stream to be reused */
if (SIO_reclaim(&outStream, (Ptr *)&outbuf, NULL) < 0) {
SYS_abort("Error reclaiming empty buffer from the output stream");
}

// Even numbered elements are the left channel (Silver on splitter)
// and odd elements are the right channel (Gold on splitter).
/* This simply moves each channel from the input to the output. */
/* To perform signal processing, put code here */
for (i = 0; i < CHANLEN; i++) {
outbuf[2*i] = inbuf[2*i];// Left channel (Silver on splitter)
outbuf[2*i+1] = inbuf[2*i+1];// Right channel (Gold on splitter)
}

/* Issue full buffer to the output stream */
if (SIO_issue(&outStream, outbuf, nmadus, NULL) != SYS_OK) {
SYS_abort("Error issuing full buffer to the output stream");
}

/* Issue an empty buffer to the input stream */
if (SIO_issue(&inStream, inbuf, SIO_bufsize(&inStream), NULL) != SYS_OK) {
SYS_abort("Error issuing empty buffer to the input stream");
}
}
```

This loop simply gets buffers to be processed with the SIO_reclaim commands, processes them, and puts them back with the SIO_issue commands. In this example the data is simply copied from the input buffer to the output buffer. Since the data comes in as packed left and right channels, the even numbered samples are the left channel and the odd numbered samples are the right channel. So the command that copies the left channel is:

```
outbuf[2*i] = inbuf[2*i];// Left channel (Silver on splitter)
```

Suppose you wanted to change the function so that the algorithm just amplifies the left input by a factor of 2. Then the program would simply be changed to:

```
outbuf[2*i] = 2*inbuf[2*i];// Left channel (Silver on splitter)
```

Notice that there is a `for` loop within the infinite loop. The loop is executed `CHANLEN` times. In the example code, `CHANLEN` is 256. So the block of data is 256 samples from the left channel and 256 samples from the right channel. When writing your programs use the variable `CHANLEN` to determine how much data to process.

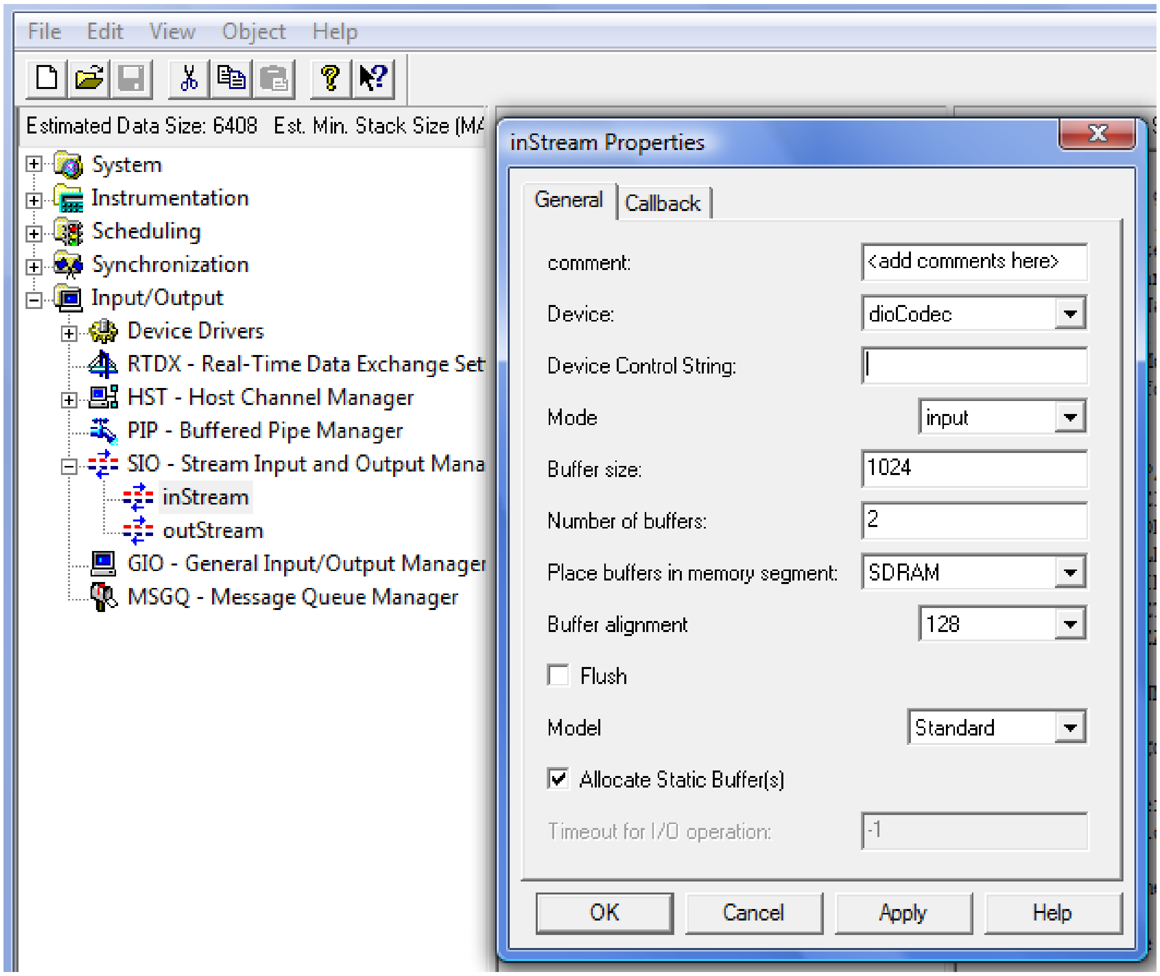
3.3.5.3 Buffer Sizes

In the program file is a macro that is used to define the size of the buffers that are in the `inStream` and `outStream` buffers. The macro is

```
#define CHANLEN 256 // Number of samples per channel
```

The CCS project uses TI's device drivers described in SPRA846. In the TI document SPRA846 it states that "If buffers are placed in external memory for use with this device driver they should be aligned to a 128 bytes boundary. In addition the buffers should be of a size multiple of 128 bytes as well for the cache to work optimally." If the SIO streams `inStream` and `outStream` are in external memory then the buffer sizes need to adhere to this requirement. If the streams are placed in internal memory then they can be any size desired as long as there is enough memory.

Suppose you want to keep the buffers in external memory but change the size. As an example, you want to change the length of each channel to 512 (multiple of 128). Then change `CHANLEN` to 512 and then open the configuration file `DSK6713_audio.tcf` and examine the properties for `inStream`.



File Edit View Object Help
Estimated Data Size: 6408 Est. Min. Stack Size (MA
System
Instrumentation
Scheduling
Synchronization
Input/Output
Device Drivers
RTDX - Real-Time Data Exchange Set
HST - Host Channel Manager
PIP - Buffered Pipe Manager
SIO - Stream Input and Output Mana
inStream
outStream
GIO - General Input/Output Manager
MSGQ - Message Queue Manager
inStream Properties
General
Callback
OK
Cancel
Apply
Help

comment: <add comments here>

Device: dioCodec

Device Control String:

Mode: input

Buffer size: 1024

Number of buffers: 2

Place buffers in memory segment: SDRAM

Buffer alignment: 128

☐ Flush

Model: Standard

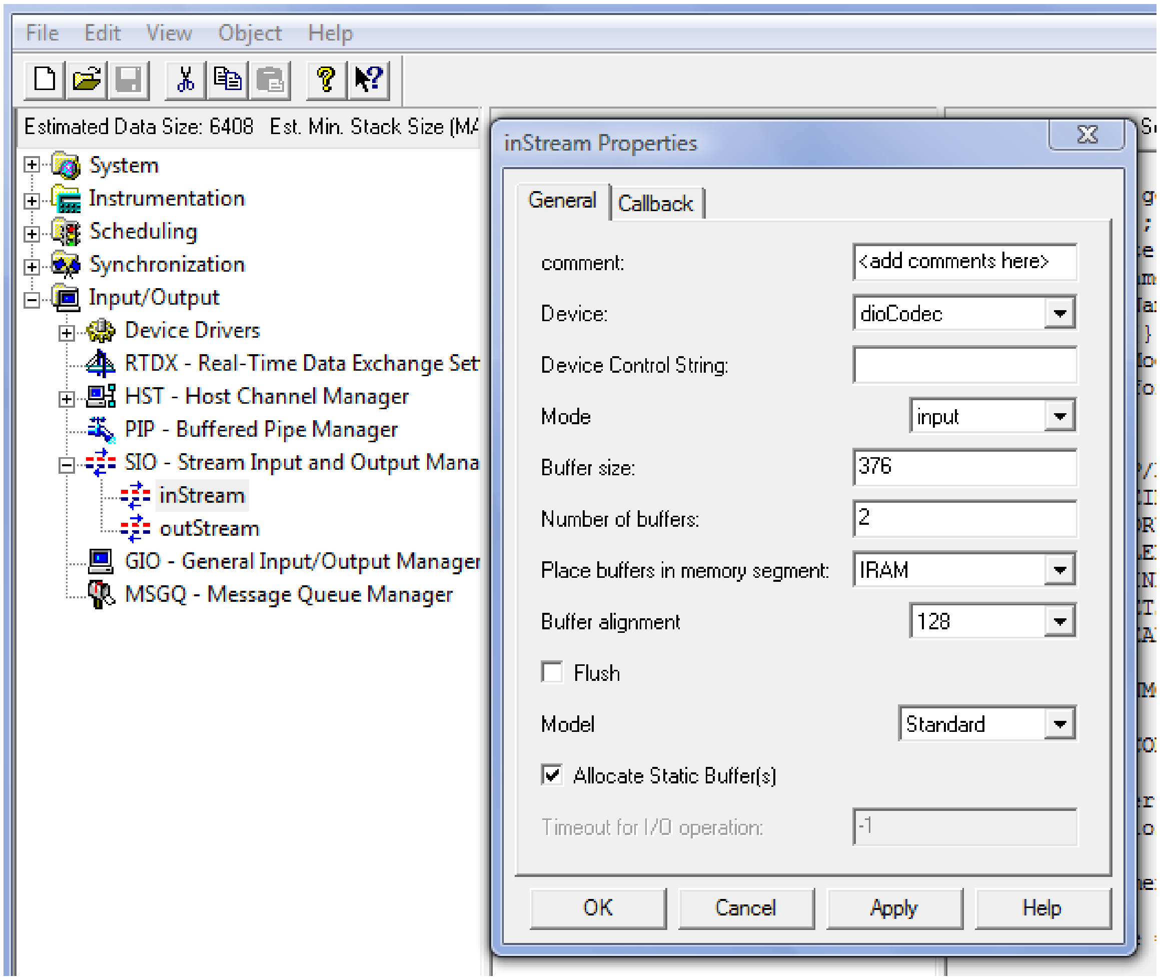
☑ Allocate Static Buffer(s)

Timeout for I/O operation: -1

Figure 3.16: DSP/BIOS SIO object inStream properties

Notice that the buffer size is 1024. This was for a channel length of 256. In the C6713 a Minimum Addressable Data Unit (MADU) is an 8-bit byte. The buffer size is in MADUs. Since each sample from the codec is 16-bits (2 MADUs) and there is a right channel and a left channel, the channel length is multiplied by 4 to get the total buffer size needed in MADUs. 256*4 = 1024. If you want to change the channel length to 512 then the buffer size needs to be 512*4 = 2048. Simply change the buffer size entry to 2048. Leave the alignment on 128. The same change must be made to the `outStream` object so that the buffers have the same size so change the buffer size in the properties for the `outStream` object also.

Suppose you have an algorithm that processes a block of data at a time and the buffer size is not a multiple of 128. To make the processing easier, you could make the stream buffer sizes the size you need for your algorithm. In this case, the stream objects must be placed in internal memory. As an example suppose you want a channel length of 94. Set the `CHANLEN` macro to 94 and then open the configuration file `DSK6713_audio.tcf` and examine and modify the properties for `inStream` and `outStream`. The buffer size will now be 94*4 = 376 and the buffer memory segment must be change from external RAM (SDRAM) to the internal RAM (IRAM).


File
Edit
View
Object
Help
Estimated Data Size: 6408 Est. Min. Stack Size (MA
System
Instrumentation
Scheduling
Synchronization
Input/Output
Device Drivers
RTDX - Real-Time Data Exchange Set
HST - Host Channel Manager
PIP - Buffered Pipe Manager
SIO - Stream Input and Output Mana
inStream
outStream
GIO - General Input/Output Manager
MSGQ - Message Queue Manager
inStream Properties
General
Callback
OK
Cancel
Apply
Help

comment: <add comments here>

Device: dioCodec

Device Control String:

Mode input

Buffer size: 376

Number of buffers: 2

Place buffers in memory segment: IRAM

Buffer alignment 128

□ Flush

Model Standard

☑ Allocate Static Buffer(s)

Timeout for I/O operation: -1

Figure 3.17: DSP/BIOS SIO object inStream properties showing IRAM

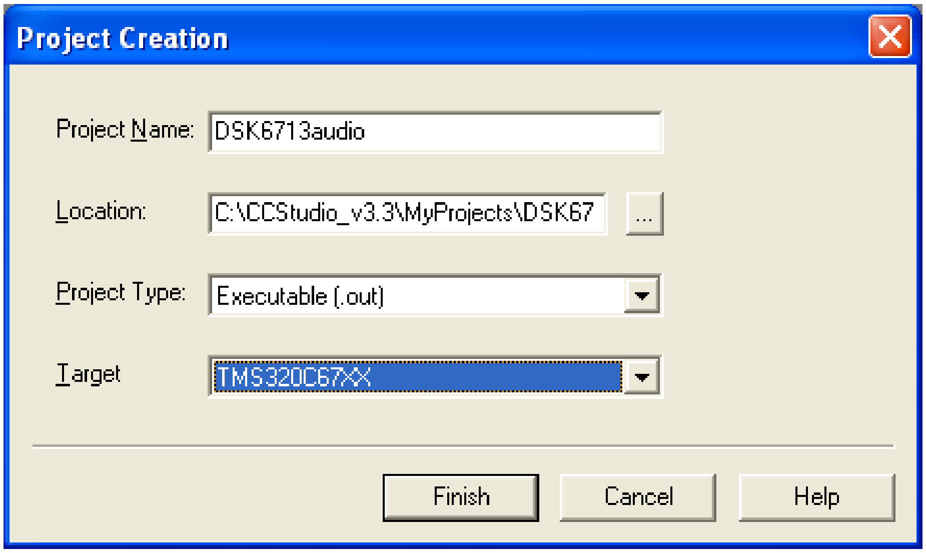
3.3.6 Chip Support Library

CCS v3.3 comes with the chip support library (CSL). If for some reason you don't have it or it needs to be updated, download it from the TI website and install it.

3.3.7 Project Setup

3.3.7.1 Make an empty DSP/BIOS project

To create a CCS project select **Project->New...**. This will bring up a window where you can enter the name of the project. The location selected is the default location for project files. Note that it is in the install directory for CCS v3.3. Select the Target type as **TMS320C67xx**. Press **Finish**.



Project Name: DSK6713audio

Location: C:\CCStudio_v3.3\MyProjects\DSK67

Project Type: Executable (.out)

Target TMS320C67XX

Figure 3.18: CCS Project name and location

Copy files from the zip file to project directory. When they are copied to the project directory the files need to be added to the project. Right click on the project name and select **Add Files To Project...**. There should be 4 files added to the project as well as a header file and a couple of automatically generated files.

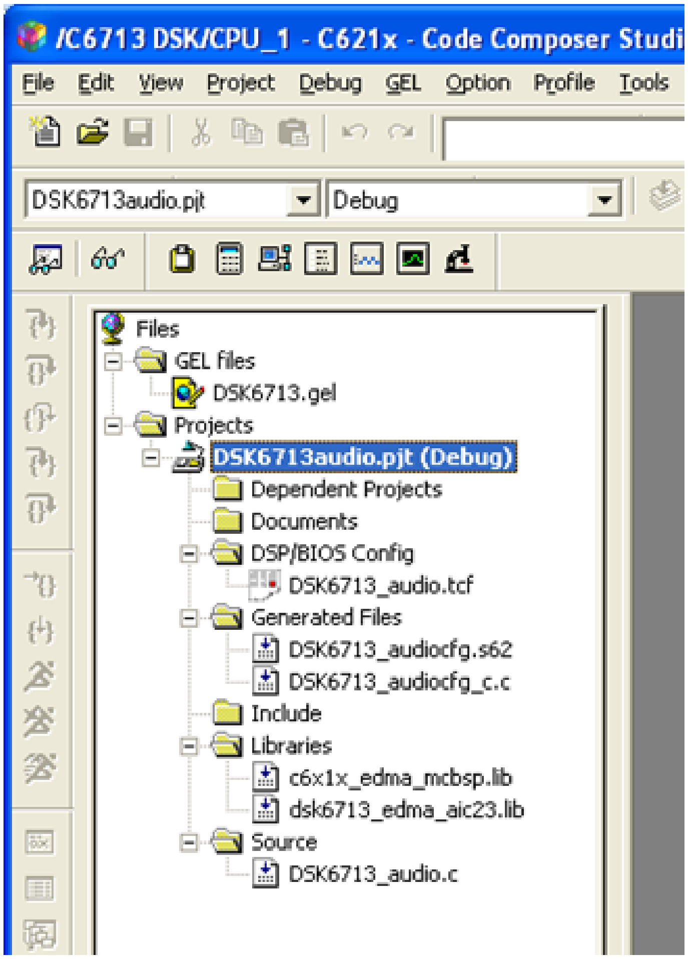
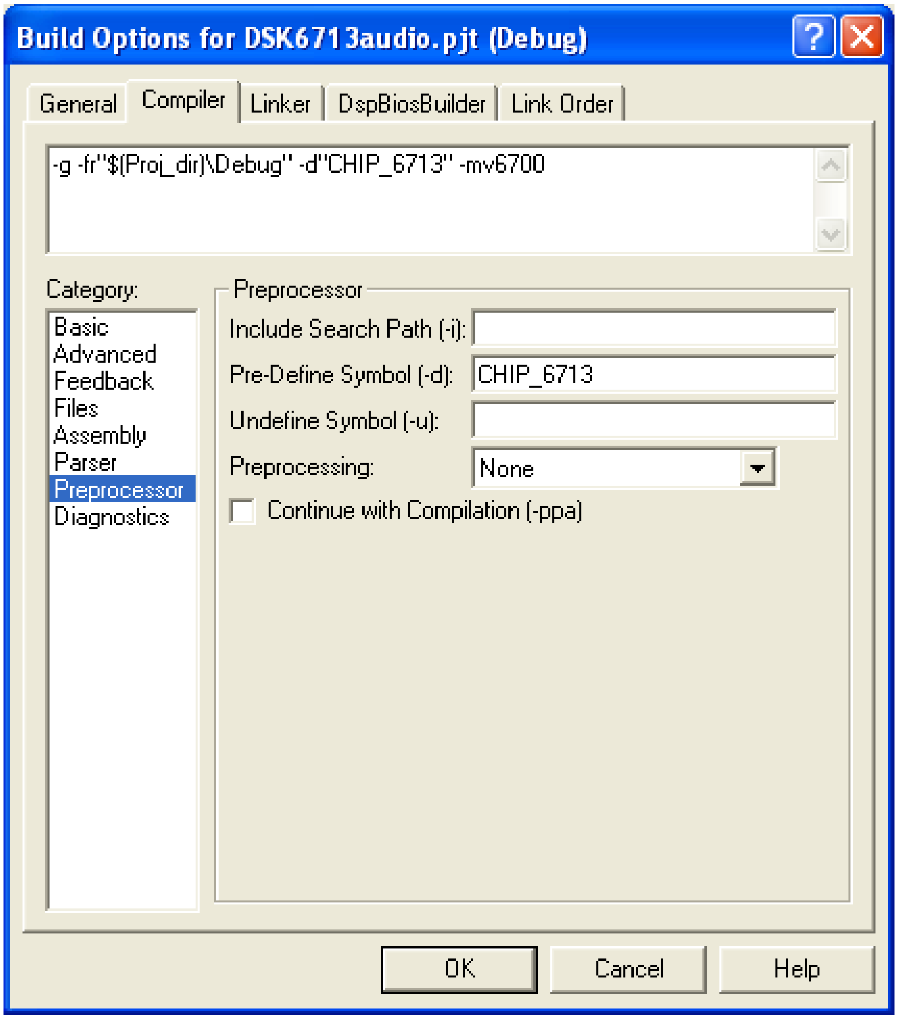


Figure 3.19: Project view after adding files

3.3.7.2 Set up the CSL

Some options in the project need to be changed in order to use the CSL. To get to the project properties select **Project->Build Options**. Under Preprocessor enter the chip that we are using, in this case **CHIP_6713**.



Build Options for DSK6713audio.pjt (Debug)

```
-g -fr"$(Proj_dir)\Debug" -d"CHIP_6713" -mv6700
```

Preprocessor

Include Search Path (-i):

Pre-Define Symbol (-d): CHIP_6713

Undefine Symbol (-u):

Preprocessing: None

□ Continue with Compilation (-ppa)

Figure 3.20: Preprocessor setting for CSL

The library file for the CSL also needs to be added to the project. Right click on the project or where it says Libraries and search for the CSL library file named `csl6713.lib`. It should be located in the directory `C:\CCStudio_v3.3\C6000\csl\lib`.

3.3.7.3 Add Command File

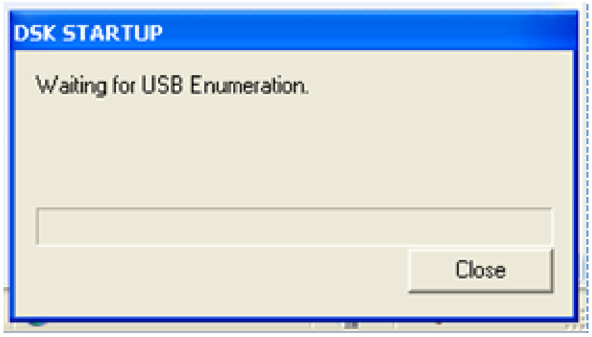
The configuration file, `DSK6713_audio.tcf`, generates several files, one of them being a command file titled `DSK6713_audio.cmd`. Add this file to the project. If it is not there, you may need to attempt to build the project so that the file gets generated.

3.3.8 Using CCS with the DSK

3.3.8.1 Startup CCS with DSK connected to PC

After installing CCS v3.3 the C6713 DSK drivers need to be installed also. If they are not already installed then download them from Spectrum Digital, manufacturer of the DSK, or install them from the install disk that comes with the DSK.

Add power to the DSK board and then plug the USB connector into the PC you are using. Start up CCS. As the splash screen comes, under Windows XP, there should be a little window that pops up in the bottom right corner of the screen.

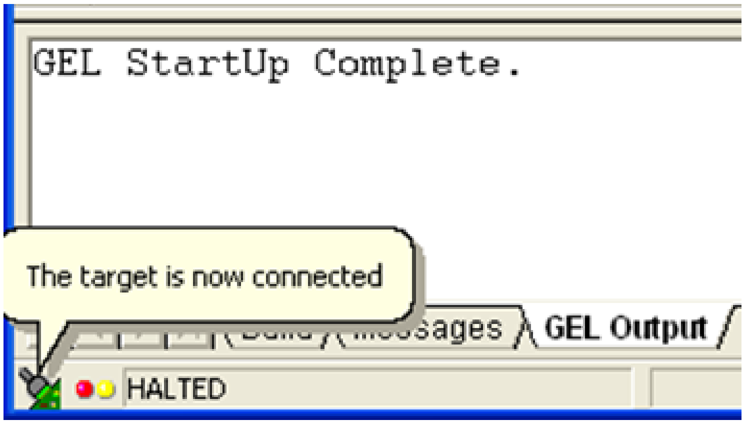


Waiting for USB Enumeration.

Figure 3.21: Popup window when CCS starts and DSK is connected

3.3.9 Connect to target

CCS v3.3 must first be connected to the target before the executable can be loaded onto the board. Select **Debug->Connect**. This will cause CCS to connect to the board and the board status should be in the lower left corner of the CCS window.



GEL StartUp Complete.

The target is now connected

Figure 3.22: Target status

3.3.10 Debug/test the project

In order to debug and run the program you need to build it and then load it onto the DSK board. To build the project, click the Incremental Build button.

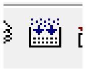
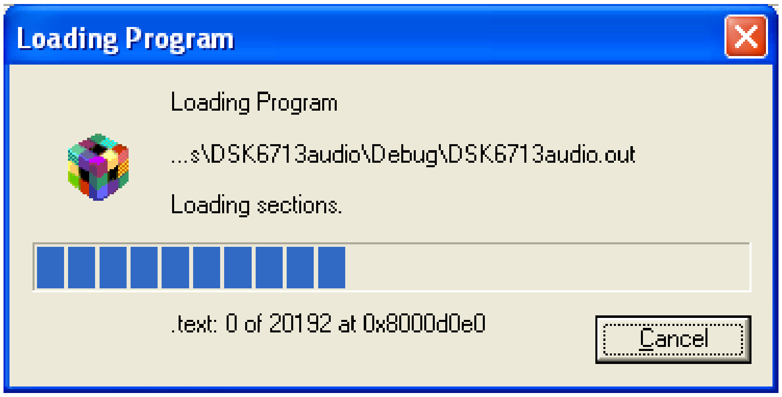


Figure 3.23: Incremental Build button

After the project builds successfully, load the `.out` file to the DSK board but selecting **File->Load Program...** and selecting the file. Or, if you want it loaded automatically, select **Options->Customize** and then on the **Project/Program/CIO** tab select the checkmark next to **Load Program After Build**. When you load the program you will see a status window like the following figure.



Loading Program

Loading Program

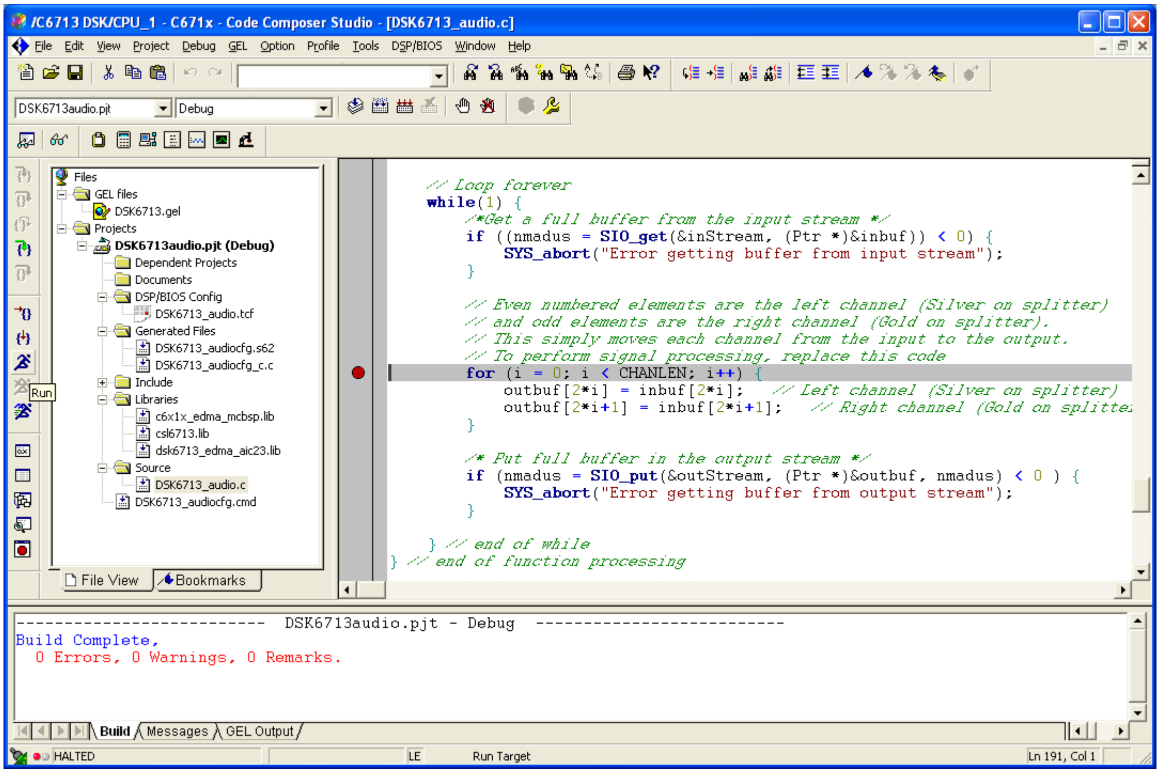
...s\DSK6713audio\Debug\DSK6713audio.out

Loading sections.

.text: 0 of 20192 at 0x8000d0e0

Figure 3.24: Load Program status window

Once the program is loaded it is ready to be used. The following figure shows the final project with the main program file.



```
    // Loop forever
    while(1) {
        /*Get a full buffer from the input stream */
        if ((nmadus = SIO_get(&inStream, (Ptr *)&inbuf)) < 0) {
            SYS_abort("Error getting buffer from input stream");
        }

        // Even numbered elements are the left channel (Silver on splitter)
        // and odd elements are the right channel (Gold on splitter).
        // This simply moves each channel from the input to the output.
        // To perform signal processing, replace this code
        for (i = 0; i < CHANLEN; i++) {
            outbuf[2*i] = inbuf[2*i];    // Left channel (Silver on splitter)
            outbuf[2*i+1] = inbuf[2*i+1];    // Right channel (Gold on splitte
        }

        /* Put full buffer in the output stream */
        if (nmadus = SIO_put(&outStream, (Ptr *)&outbuf, nmadus) < 0 ) {
            SYS_abort("Error getting buffer from output stream");
        }

    } // end of while
} // end of function processing
```

```
-------------------------  DSK6713audio.pjt - Debug  -------------------------
Build Complete,
  0 Errors, 0 Warnings, 0 Remarks.
```

Figure 3.25: Final project view

3.4 Echo Lab[5]

3.4.1 Introduction

In this laboratory you will use a program that gathers data from an A/D converter, processes the data, and outputs to a D/A converter. The signal processing portion will be added by you to perform an echo function. After this laboratory you should:

- Understand how data is received from an A/D converter and sent to a D/A converter
- Understand how signal processing is done in "real-time"
- Be able to implement a simple signal processing algorithm using "block processing" techniques

3.4.2 Laboratory Assignment

In the laboratory you will be making a delay function or an echo. To make an echo you will need to delay the data that is coming into the DSP. This is usually accomplished using a buffer that is used to store previous input values. Use a circular buffer to store the input data. The length of the buffer is determined by

$$N = \text{Delay} \cdot \text{SampleRate} \tag{3.3}$$

In order to implement the delay an array of sufficient size must be defined. When a function is called, the local variables will go onto the stack. The stack is usually small compared to other memory sections. Therefore, the number of local variables for a function is limited. Since the array for storing the previous inputs will be large, the array should be stored somewhere else in memory besides on the stack. If the Memory Model in the build options is set to `Far Aggregate` then the global variables and the local static variables will be put into the `.far` section. This section can be set to the large external memory. Therefore, the array that stores the delay values should be defined as either a **global or static local variable**. A static local variable is a variable that is not initialized each time a function is called. The static variable will retain the value it had the last time the function was called. To make a static variable just use the static keyword in front of the variable definition.

```
static int static_var, static_array[10];
```

You need to make sure the `.far` section is set to a large memory section. For our project, set it to the SDRAM memory. To change the location of the `.far` section in the project you should:

- Open the project `DSK6713_audio.pjt`
- Open the configuration file `DSK6713_audio.tcf`
- Right click on System->MEM and open Properties
- Click on the Compiler Sections tab
- On the pull down menu next to the `.far` section select `SDRAM`

3.4.3 Pre-Laboratory

- If your sample rate is 8000 Hz and you want a delay of 1/2 second, how long should your delay buffer be?
- The main loop in the default I/O program simply inputs a block of data and then outputs a block of data. You need to modify the main loop so that it will take an input sample, store it in the circular buffer and output the sample that is delayed by 1/2 second. Your code should implement a circular buffer. Also, make sure you initialize the buffer with zeros so that you won't output whatever happens to be in the memory.

[5] This content is available online at <http://cnx.org/content/m36657/1.1/>.

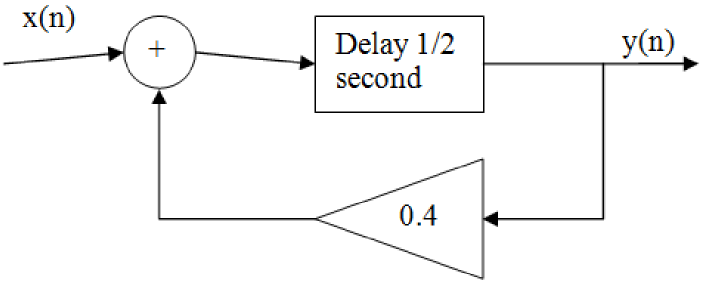
- What is the transfer function, H(z), of the system in the following figure? What is the difference equation? What would happen if you set the feedback gain to a value greater than 1?
- Write another program that will implement a fading echo shown in following figure. The output of the delay is fed back to the input of the delay and added to the input.
- 

Figure 3.26

3.4.4 Laboratory

3.4.4.1 Part 1

- In this part you will just get the template project working that simply outputs whatever was input.
- Open the project `DSK6713_audio.pjt`.
- Change the parameters in the `DSK6713_audio.c` file so that the sampling rate is 8000 Hz and the input is from the microphone.
- Connect a microphone to the DSK board microphone input and the speakers to the line out.
- Build, load and run the project.
- Verify that as you speak into the microphone you hear yourself on the speakers.

3.4.4.2 Part 2

- Modify the program `DSK6713_audio.c` so that one channel of output is delayed by 1/2 second. Make sure to define your stored samples array as a global or static array in external memory.
- Build, load and run the project.
- Demonstrate that you get a 1/2 second delay in the output.

3.4.4.3 Part 3

- Modify the program `DSK6713_audio.c` to implement the fading echo.
- Build, load and run the project.
- Demonstrate that you get a 1/2 second delay in the output.
- What would happen if you set the feedback gain to a value greater that 1?

Chapter 4

Cosine Generator Lab

4.1 Second Order System Impulse Response Generation[1]

4.1.1 Introduction

This module examines the recursive generation of the impulse response of a second order system. This analysis can be used to determine how to generate a cosine or sine function recursively.

4.1.2 Second Order Systems

The transfer function of a general second order system is given as:

$$H(z) = \frac{b_0 z^2 + b_1 z + b_2}{z^2 + a_1 z + a_2} \tag{4.1}$$

The Direct Form II structure for the implementation of the second order system is shown in the following figure.

[1] This content is available online at <http://cnx.org/content/m36658/1.3/>.

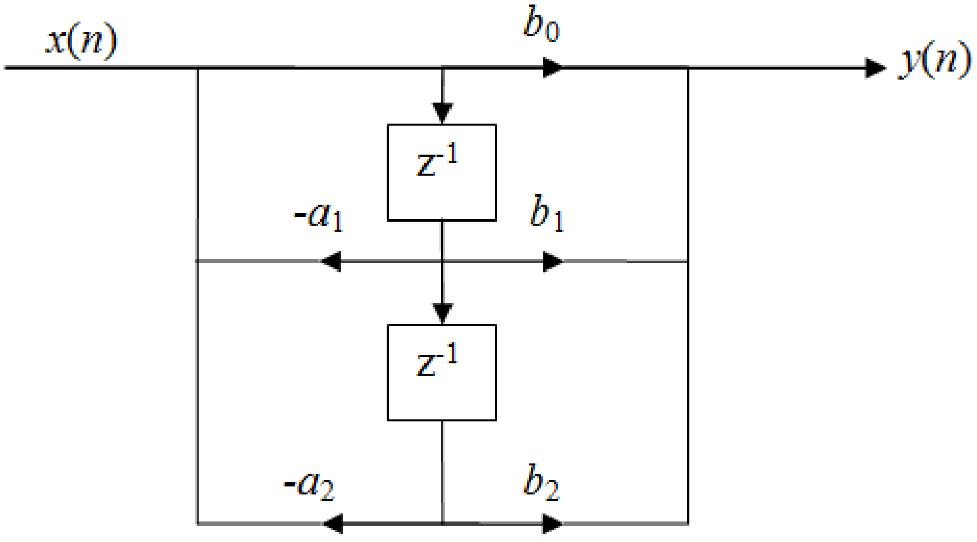


Figure 4.1

The difference equation for the second order system is

$$y(n) + a_1 y(n-1) + a_2 y(n-2) = b_0 x(n) + b_1 x(n-1) + b_2 x(n-2) \quad (4.2)$$

$$y(n) = -a_1 y(n-1) - a_2 y(n-2) = b_0 x(n) + b_1 x(n-1) + b_2 x(n-2) \quad (4.3)$$

To recursively determine the impulse response of the system start with the following:

$$h(-1) = h(-2) = 0, x(n) = \delta(n) \quad (4.4)$$

Then using Equation 3 the impulse response can be found recursively by:

$$h(0) = -a_1 h(-1) - a_2 h(-2) + b_0 \delta(0) + b_1 \delta(-1) + b_2 \delta(-2) = b_0 \quad (4.5)$$

$$h(1) = -a_1 h(0) - a_2 h(-1) + b_0 \delta(1) + b_1 \delta(0) + b_2 \delta(-1) \quad (4.6)$$

$$h(1) = -a_1 b_0 + b_1 \quad (4.7)$$

$$h(2) = -a_1 h(1) - a_2 h(0) + b_0 \delta(2) + b_1 \delta(1) + b_2 \delta(0) \quad (4.8)$$

$$h(2) = -a_1(-a_1 + b_1) - a_2 b_0 + b_2 \quad (4.9)$$

For the values of n>2 the recursive equation reduces to

$$h(n) = -a_1 h(n-1) - a_2 h(n-2) \quad (4.10)$$

because the values of $x(n), x(n-1)$ and $x(n-2)$ will all be zero for n>2.

So, if you know the values of $h(0)$ and $h(1)$, then the Equation 10 can be used to find future values of $h(n)$.

4.2 Cosine Generator Lab[2]

4.2.1 Skills

In this laboratory you write a program that generates a cosine on the output of the D/A converter. After this laboratory you should:

- Be able to work with transfer functions and difference equations.
- Be able to get a Visual Basic program to communicate with your DSP program on the DSK board.

4.2.2 Reading

- Chapter 4: Emmanuel Ifeachor and Barrie W. Jervis, *Digital Signal Processing: A Practical Approach*, Pearson Education Limited, England, 2002.
- Chapter 13 & 14: Mrinal Mandal, Amir Asif, *Continuous and Discrete Time Signals and Systems*, Cambridge University Press, 2007.

4.2.3 Description

In this lab you will be making a cosine function generator. This will require you to generate the values that go to the D/A converter in order to produce a cosine function. Since we are only concerned with the output, the input values will be ignored. You will need to determine the values of the cosine function necessary to produce it on the output.

4.2.4 Pre-Laboratory

- In order to generate a cosine function we will generate the impulse response of a system that has the impulse response

$$h\left(n\right)=\cos\left(\alpha \mathrm{n}\right)u\left(n\right) \tag{4.11}$$

- For the given impulse response find the following:

1. Transfer function of the system. Look it up in a ZT table.
2. Direct Form II structure of the system.
3. Pole/zero diagram of the system. Assume $\alpha=\pi/4$ for the example diagram.
4. Difference equation of the system.

- Calculate $h(0)$, $h(1)$, and $h(2)$ assuming the input is an impulse $x(n)=\delta(n)$. Simplify the $h(2)$ term.
- Show how you will recursively determine the impulse response of this system.
- Suppose the sample rate of the system is 8000 Hz and we want to generate a cosine with a frequency of 500 Hz, what should α be to implement this cosine function?
- In MATLAB, write a program that will recursively determine the impulse response of this system with sample rate 8000Hz and cosine of 400 Hz. Put the data into an array and plot the impulse response. Generate 100 points. Compare the recursively generated values to the direct calculation of the cosine function.

[2] This content is available online at <http://cnx.org/content/m36659/1.1/>.

- Modify the while loop Write a program in C that will recursively determine the impulse response of this system.

1. The recursive portion should be in an infinite `while` loop and the data should not go into an array but in a single variable.
2. All of the variables should be of type `double`.
3. In your program you will need to include the header file `math.h`.
4. The data we are generating is for the D/A converter. The D/A converter accepts `Int16` type data which has the limits of [-32768, 32767]. Since the impulse response has the limits [-1, 1] you will need to multiply the result by a number to scale the value up to a level that will work with the D/A converter without overflowing. Use the value `32000.0`. The statement will be something like `(Int16)(32000.0*y)`.
5. To calculate π, use the following statement:

```
pi=16.0*atan(1.0/5.0)-4*atan(1.0/239.0);
```

4.2.5 Laboratory

4.2.5.1 Part 1

- Use the sample I/O program for the DSK and generate a 400 Hz cosine sampled at 8000 Hz using the recursive program you developed.
- Change the frequency of the cosine to at least 2 other values from about 50 Hz to 4000 Hz and record the results.
- What happens when you change the frequency of the cosine to 5000 Hz? Explain what you see.

4.2.5.2 Part 2

- In this part you will be creating a Visual Basic (VB) application that communicates with a target application through the Real Time Data Exchange (RTDX). The VB application will have two buttons that when pressed will send different numbers to the target (DSP board). The target application will print the value received to a LOG object and turn the output cosine either on or off.
- Create a new Visual Basic project (These may not be exactly the steps depending on your version of Visual Studio):

1. Start Visual Studio.
2. Select Create Project.
3. In the Project types: select Visual Basic:Windows.
4. Under Templates: select Windows Forms Application.
5. The Solution Name will be `CosineOnOff`.
6. Click OK.

- Add two buttons to your form with the following characteristics:

1. (Name): Button1
2. Text: On
3. (Name): Button2

4. Text: Off

- In the Form code in the Form1:(Declarations) section add code for the status constants and other variables. This will be right under the statement `Public Class Form1`. The code should be:

```
’---------------------------------------------------------------------------
’ RTDX Return Status
’---------------------------------------------------------------------------
Const Success = &H0 ’ Method call is valid
Const Failure = &H80004005 ’ Method call failed
Const ENoDataAvailable = &H8003001E ’ No data currently available.
Const EEndOfLogFile = &H80030002 ’ End of transmission

’---------------------------------------------------------------------------
’ Variable Declarations
’---------------------------------------------------------------------------
Dim rtdx As Object ’ Holds the rtdx object
Dim bufferstate As Integer ’ Holds the number of bytes xmitted
’ or pending xmission
Dim status As Integer ’ RTDX Function call return status
```

- The first function to be executed when a project is started is `Form:Load`. In this function put the code for initializing the RTDX channel. Name the channel `HtoTchan` for Host to Target channel.

```
Try
~~~ rtdx_out = CreateObject("RTDX")
~~~~status = rtdx.Open("HtoTchan", "W") ’ Open channel for writing
Catch ex As Exception
~~~ ’Exception occured when opening the RTDX object
~~~ System.Diagnostics.Debug.WriteLine("Opening of channel HtoTchan failed")
~ ~~rtdx_out = Nothing
~~~ End ’ Force program termination
End Try
```

- One of the last functions to execute is the `Form:FormClosing` function. In this function put the code for cleaning things up.

```
status = rtdx.Close() ’ Close rtdx
Set rtdx = Nothing ’ Free memory reserved for rtdx obj
```

- In the `Button1:Click` function put the code to be executed when the button is clicked. When the button is clicked it should send a 4 byte integer, the number 1, to the target indicating that button 1 was clicked. The code for doing this is:

```
Dim Data As Integer

Data = 1
status = rtdx.WriteI4(Data, bufferstate)

If status = Success Then
Debug.Print "Value " & Data & " was sent to the target"
Else
Debug.Print "WriteI4 failed"
End If
```

- In the `Button2:Click` function put the code to be executed when the button is clicked. When the button is clicked it should send a 4 byte integer, the number 0, to the target indicating that button 2 was clicked. The code is the same as above, the data is just 0.
- Now the target code needs to be updated.
- Save the `DSK6713_audio.c` file as `DSK6713_audio2.c` and use this as the main file in your project.
- Include a header file so we can use the RTDX.

```
#include <rtdx.h>
```

- Create a global input channel called `HtoTchan`. This code will go in the global variables section.

```
RTDX_CreateInputChannel( HtoTchan );
```

- Add code to the top of function `processing`:

```
int data;
int status;
int busystatus=0;

/* enable the Host to Target channel */
RTDX_enableInput( &HtoTchan );
```

- In the `while` loop add the code:

```
/* check to see if the channel is busy before the read */
if (busystatus == 0)
{
/* Print the data if something was actually read */
if (RTDX_sizeofInput(&HtoTchan) == sizeof(data))
{
// data received here
LOG_printf(&trace,"Value sent = %d",data);
}
status = RTDX_readNB( &HtoTchan, &data, sizeof(data) );
}

/* get the status of the channel */
busystatus = RTDX_channelBusy(&HtoTchan);
```

- The above code will read 4 byte values from the `HtoTchan` input channel. When a value is received the program will get to the statement `// data received here` and will have the value in the variable `data`.
- Since the variable data will contain either a 1 or a 0 this can be used to multiply the cosine value to turn it on and off. `(Int16)(32000.0*y*data);`
- Set up RTDX by opening the Configuration window with Tools->RTDX->Configuration Control, right click to select properties and then click on Continuous mode. After closing the properties check the Enable RTDX box.
- Run the target application. It is important that the target application be started before the VB application. If the VB application is started first the channel may not get initialized.
- Run the VB application.
- Verify that the tone that is output can be turned on and off with your VB program.

Chapter 5

FIR Lab

5.1 FIR Filter Structures[1]

Consider causal FIR filters: $y(n) = \sum_{k=0}^{M-1} h(k)\, x(n-k)$; this can be realized using the following structure

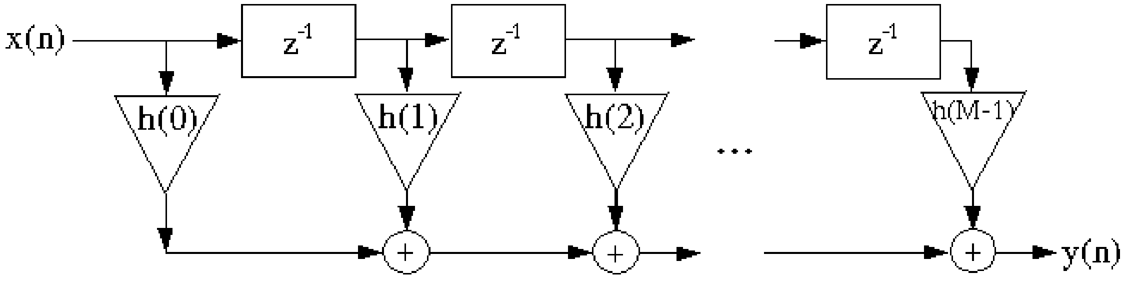


Figure 5.1

or in a different notation

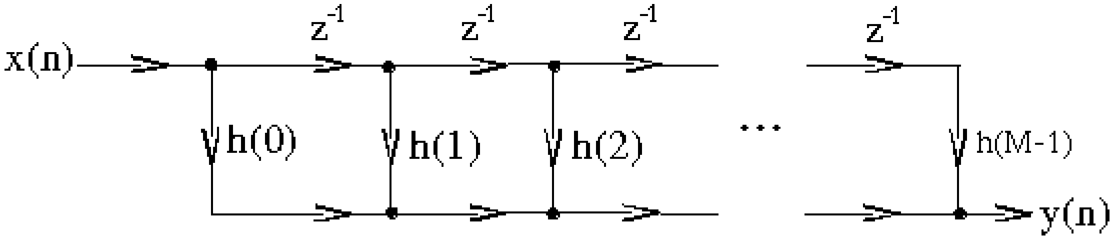
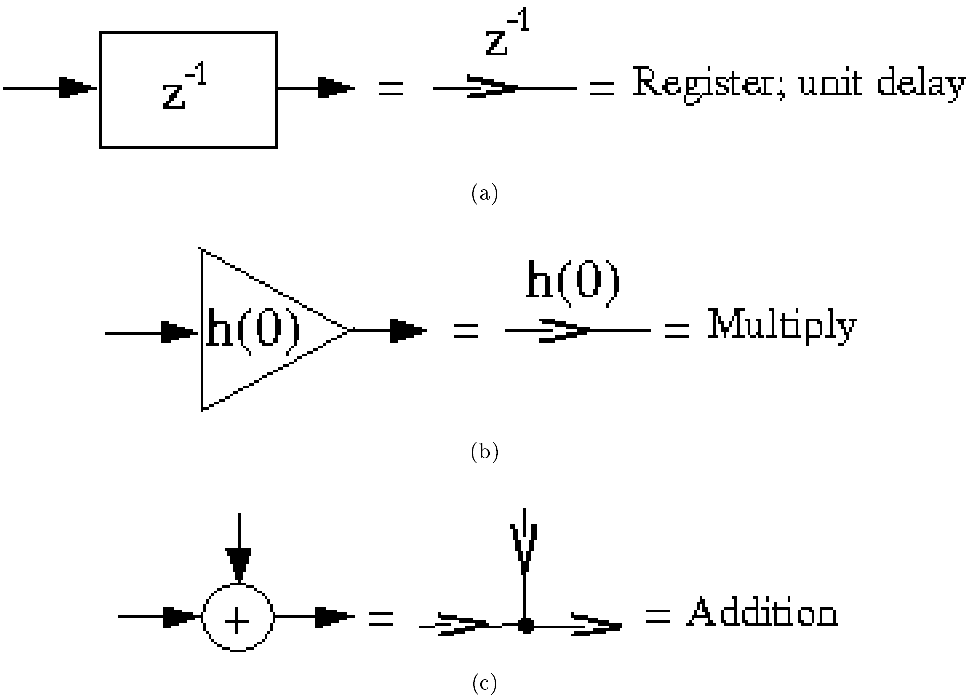


Figure 5.2

[1] This content is available online at <http://cnx.org/content/m11918/1.2/>.



(a)

(b)

(c)

Figure 5.3

This is called the **direct-form FIR filter structure**.

There are no closed loops (no feedback) in this structure, so it is called a **non-recursive structure**. Since any FIR filter can be implemented using the direct-form, non-recursive structure, it is always possible to implement an FIR filter non-recursively. However, it is also possible to implement an FIR filter **recursively**, and for some special sets of FIR filter coefficients this is much more efficient.

Example 5.1

$$y(n) = \sum_{k=0}^{M-1} x(n-k)$$

where

$$h(k) = \left\{ 0, 0, \underset{\overset{\wedge}{k=0}}{1}, 1, \dots, 1, \underset{\overset{\wedge}{k=M-1}}{1}, 0, 0, 0, \dots \right\}$$

But note that

$$y(n) = y(n-1) + x(n) - x(n-M)$$

This can be implemented as

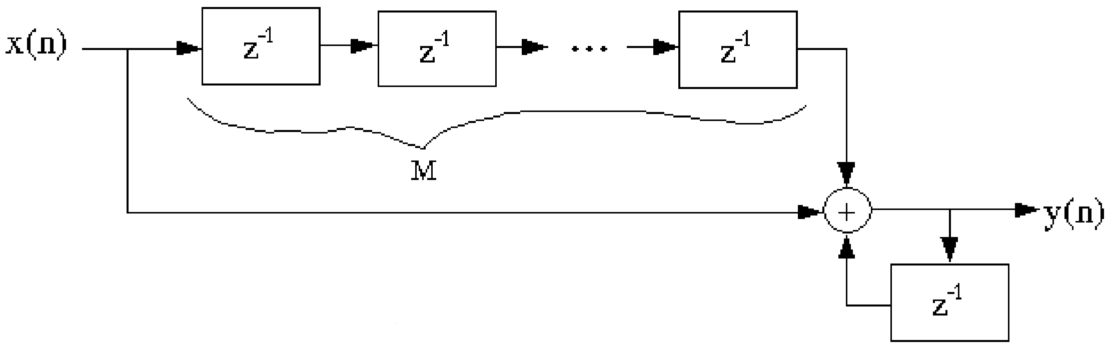


Figure 5.4

Instead of costing $M - 1$ adds/output point, this comb filter costs only two adds/output.

Exercise 5.1.1

Is this stable, and if not, how can it be made so?

IIR filters must be implemented with a **recursive** structure, since that's the only way a finite number of elements can generate an infinite-length impulse response in a linear, time-invariant (LTI) system. Recursive structures have the advantages of being able to implement IIR systems, and sometimes greater computational efficiency, but the disadvantages of possible instability, limit cycles, and other deletorious effects that we will study shortly.

5.1.1 Transpose-form FIR filter structures

The **flow-graph-reversal theorem** says that if one changes the directions of all the arrows, and inputs at the output and takes the output from the input of a reversed flow-graph, the new system has an identical input-output relationship to the original flow-graph.

Direct-form FIR structure

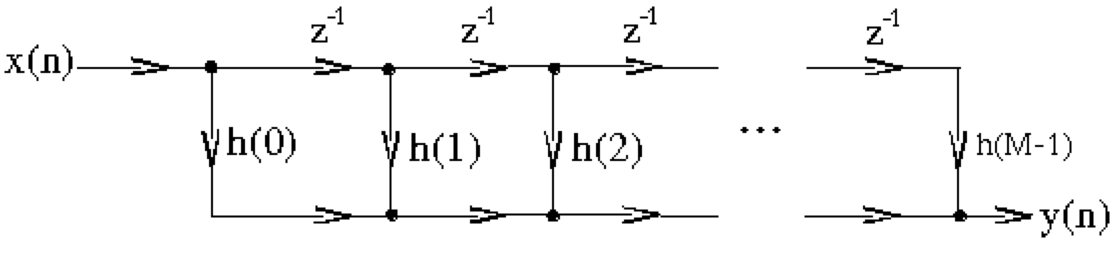


Figure 5.5

reverse = transpose-form FIR filter structure

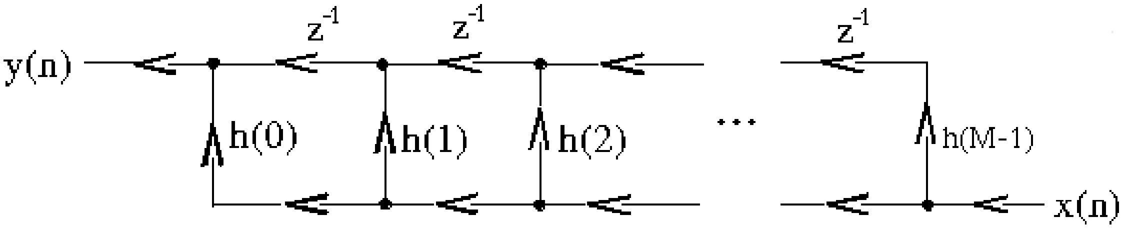


Figure 5.6

or redrawn

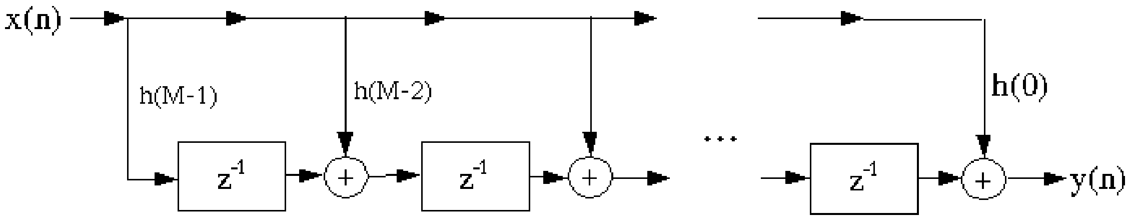


Figure 5.7

5.1.1.1 Cascade structures

The z-transform of an FIR filter can be factored into a cascade of short-length filters

$$b_0 + b_1 z^{-1} + b_2 z^{-3} + \dots + b_m z^{-m} = b_0 \left(1 - z_1 z^{-1}\right)\left(1 - z_2 z^{-1}\right) \dots \left(1 - z_m z^{-1}\right)$$

where the z_i are the zeros of this polynomial. Since the coefficients of the polynomial are usually real, the roots are usually complex-conjugate pairs, so we generally combine $\left(1 - z_i z^{-1}\right)\left(1 - \overline{z_i} z^{-1}\right)$ into one quadratic (length-2) section with **real** coefficients

$$\left(1 - z_i z^{-1}\right)\left(1 - \overline{z_i} z^{-1}\right) = 1 - 2\Re\left(z_i\right) z^{-1} + \left(\left|z_i\right|\right)^2 z^{-2} = H_i\left(z\right)$$

The overall filter can then be implemented in a **cascade** structure.

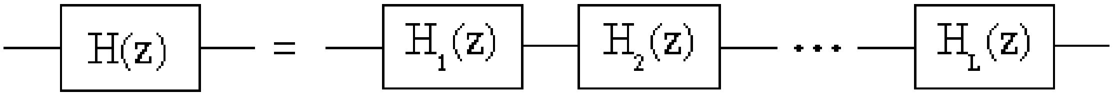


Figure 5.8

This is occasionally done in FIR filter implementation when one or more of the short-length filters can be implemented efficiently.

5.1.1.2 Lattice Structure

It is also possible to implement FIR filters in a lattice structure: this is sometimes used in adaptive filtering

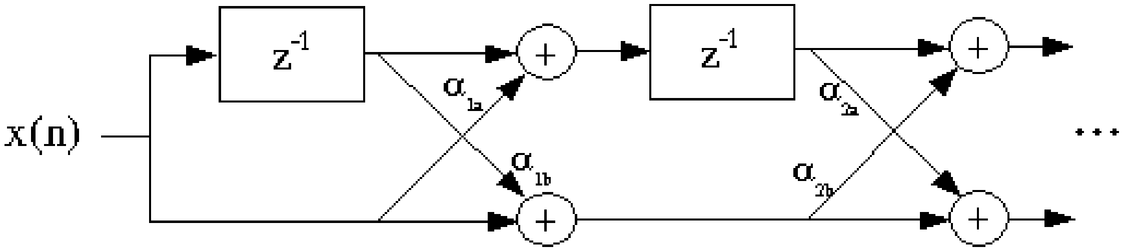


Figure 5.9

5.2 FIR Filter Implementation[2]

5.2.1 Introduction

This module explains how to implement an FIR filter in the Direct-Form structure. It explains how data is stored as samples are received and how to use a circular buffer in the algorithm.

5.2.2 Reading

- For a description of FIR Direct-Form structures see the module FIR Filter Structures (Section 5.1).
- For a description of circular buffers see Circular Buffers (Section 3.1).

5.2.3 FIR Filtering

The difference equation for an N-order FIR filter is given by:

$$y(n) = \sum_{k=0}^{N-1} b_k x(n-k) = \sum_{k=0}^{N-1} h(k)\, x(n-k) \tag{5.1}$$

[2] This content is available online at <http://cnx.org/content/m37136/1.1/>.

For each new input sample, $x(n)$, the filter produces an output $y(n)$. To calculate the new output sample, the input must be convolved with the FIR filter impulse response, $h(n)$. Notice that the summation requires the input samples starting with the most recent sample, $x(n)$ through the N-1 most recent samples, through $x(n$ - N -1). In a DSP system, these previous samples must be saved in an array in memory. Usually there will be one new sample received, $x(n)$, and any previous samples that were received must be saved in memory.

5.2.4 FIFO Buffer

Since the algorithm only needs the previous N-1 samples to be saved there must be a way for the algorithm to update the array when a new sample is input. This can be accomplished in two ways. The first way is to shift all the samples in the array by copying them to the shifted location and then storing the new value at the beginning of the array. This is a first-in-first-out (FIFO) buffer. If this method is used then the FIR algorithm is given by

$$y_{\text{out}} = \sum_{k=0}^{N-1} h\left(k\right) x_{\text{saved}}\left(k\right) \tag{5.2}$$

Figure 1 shows a diagram of the implementation of the FIR filter using the array shifting method. In the diagram the "*" indicates multiplication. The new value is shifted into the $x(0)$ spot and all the others are shifted to the right.

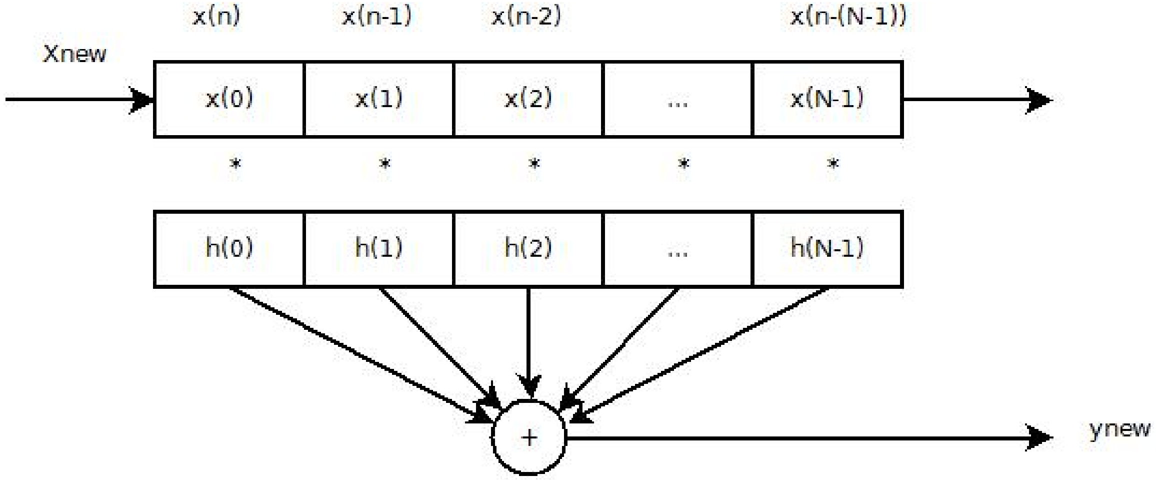


Figure 5.10: Diagram of the FIR filter implementation using the shifting method

The following pseudo-code shows how to implement the FIR filter using the FIFO type buffer. In the code the following definitions are used:

`N` - number of coefficients
`h(n)` - filter coefficients, n = 0...N-1
`x(n)` - stored input data, n = 0...N-1
`input_sample` - variable that contains the newest input sample

```
// Shift in the new sample. Note the data is shifted starting at the end.
for (i=N-2; i>=0; i--){
```

```
x[i+1] = x[i];
}
x[0] = input_sample
// Filter the data
ynew = 0;
for(i=0;i<N;i++){
ynew = ynew + h(i)*x(i);
}
```

5.2.5 Double Size FIFO Buffer

In the example above, the data in the FIFO buffer is shifted every time a new sample is received. This could lead to inefficient code. It is better to do processing on blocks of data. For instance, you could use a DMA to copy a block of data from one place to another and free up the processor from doing the work.

One solution could be to make a FIFO buffer that is twice as large as the data that is needed to store. When this is done, shifting of the data can be done after a block of data is received.

The following figures show the data buffer as data is being received and stored. Figure 2 shows the buffer when the first new value is received. Of course the buffer would be zeroed before processing begins. The shaded area shows the part of the buffer that is used in the algorithm.

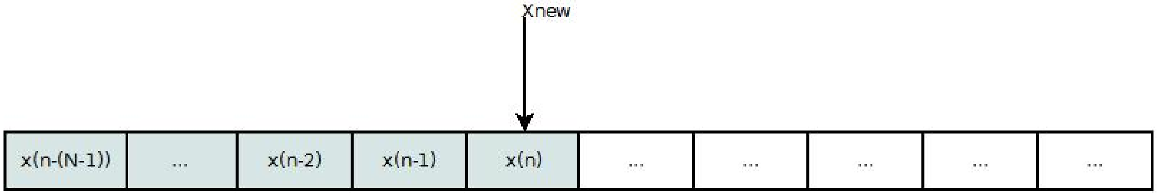


Figure 5.11: Data buffer at the beginning

Figure 3 shows the buffer after M samples have been received. Note that there is no need to shift data in the buffer. The shaded area is what is being used for the fitler.

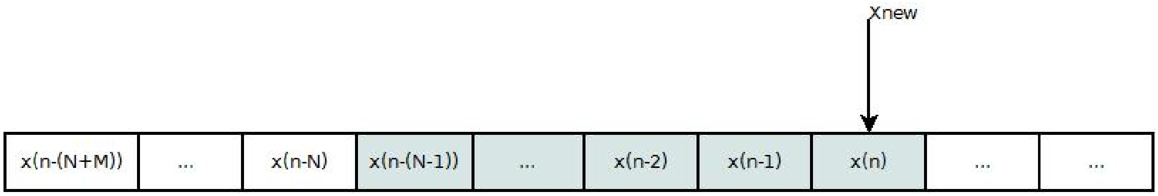


Figure 5.12: Data buffer after M values have been stored

Figure 4 shows the buffer when the new sample is stored in the last element of the buffer. At this point the shifting of the data is needed to remove old data. The old data is at the beginning of the buffer. This data is just overwritten with the data at the end of the buffer.

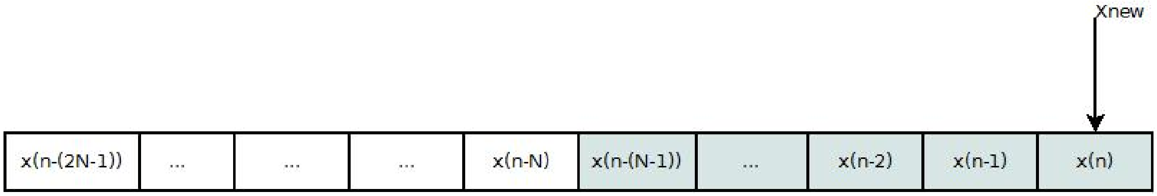


Figure 5.13: Data buffer when the index reaches the end of the buffer

Figure 5 shows the buffer after the data has been copied. The algorithm will now use the data at the beginning of the buffer just as it did in figure 2. Notice that a large block of data is copied at one time.

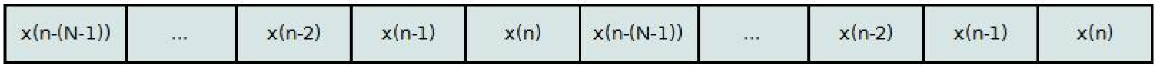


Figure 5.14: Data buffer showing the data that is copied from the end to the beginning

The following pseudo-code shows how to implement the FIR filter using the double size FIFO type buffer. In the code the following definitions are used:

`N` - number of coefficients
`h(n)` - filter coefficients, n = 0...N-1
`x(n)` - stored input data, n = 0...2N-1
`input_sample` - variable that contains the newest input sample
`index` - variable that contains the current place where the new sample is to be stored

```
x[index] = input_sample
// Filter the data
ynew = 0;
for(i=0;i<N;i++){
ynew = ynew + h(i)*x(index-i);
}
// update the index and copy the data if necessary
index = index+1;
if(index>=2*N){
for (i=N-2; i>=0; i--){
x[i+N+1] = x[i];
}
index = N - 1;
}
```

5.2.6 Circular Buffer

The second method for storing input data is use a circular buffer. To do this you will need to keep track of an index that tells where the last input was stored. This index is incremented and used to store the new value. When the index reaches the end of the array it is wrapped around to the beginning of the array. If this method is used then the FIR algorithm is

$$y_{\text{out}} = \sum_{k=0}^{N-1} h\left(k\right) x\left(\left(\text{index} - k\right)\right)_N \tag{5.3}$$

where $(*)_N$ is modulo-N. This assumes that the index is incremented as new data is stored in the buffer. The following figure shows how the data for $x(n)$ is stored in a circular buffer.

When implementing the FIR filter using a circular buffer, notice that the coefficient buffer indexes going forward and the data buffer indexes going backward.

`N` - number of coefficients

`h(n)` - filter coefficients, n = 0...N-1

`x(n)` - stored input data, n = 0...2N-1

`input_sample` - variable that contains the newest input sample

`index` - variable that contains the current place where the new sample is to be stored

```
x[index] = input_sample
// Filter the data
ynew = 0;
for(i=0;i<N;i++){
if((index-i)<0){
ynew = ynew + h[i]*x[index-i+N];
}
else {
ynew = ynew + h[i]*x[index-i];
}
}
index = (index+1)%N;
```

5.2.7 Double Size Circular Buffer

In the previous implementation of the circular buffer notice that there is an if statement in the filter loop. This is very time consuming. So to remove this from the loop a buffer that is twice the length can be used. The new value that is received is put in two places within the buffer.

In the following figure shows a buffer for a circular buffer of size 4. Therefore the total length of the buffer is 8. The two indexes point to the places where the input sample is stored. In the figure the shaded boxes are the intems that make up the filter elements. Since the samples do not wrap around the end of the buffer there is no need to check the offset in the loop.

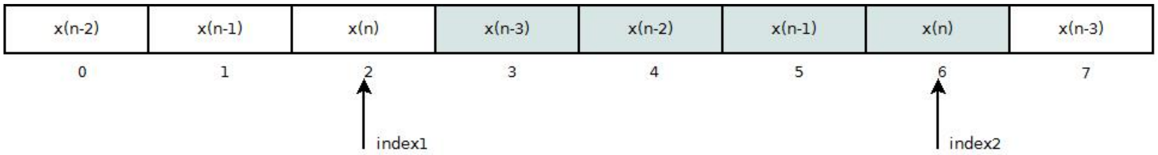


Figure 5.15: Double sized circular buffer with two indexes

After the filter output is determine the indexes are incremented and checked to see if they go past the end of the buffer. The following figure shows the buffer and indexes after one increment.

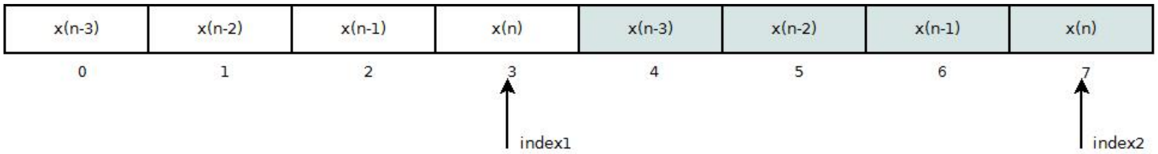


Figure 5.16: Buffer after increment of indexes

The following figure shows the indexes after another increment. Notice that the indexes wrap around and the element that are used still don't wrap around the end of the buffer.

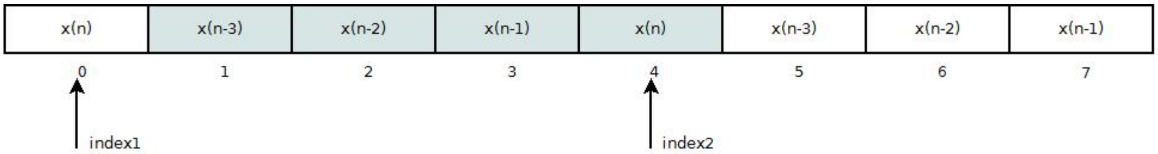


Figure 5.17: Buffer after indexes wrap around the end

To implement the double size circular buffer the following pseudo-code can be used.
`N` - number of coefficients
`h(n)` - filter coefficients, n = 0...N-1
`x(n)` - stored input data, n = 0...2N-1
`input_sample` - variable that contains the newest input sample
`index1` - the first place in x(n) where the new sample is to be stored
`index2` - the second place in x(n) where the new sample is to be stored

```
// variable are initialized to the following values
int index1=0;
int index2=N;

// As a new sample is received put it in two places
x[index1] = input_sample;
x[index2] = input_sample;
ynew = 0;
for(i=0;i<N;i++){
ynew = ynew + h[i]*x[index2-i];
}
index1 = (index1+1)%N;
index2 = index1+N;
```

5.3 FIR Filter Lab[3]

5.3.1 Introduction

This module is a lab assignment for implementing an FIR filter using a circular buffer.

5.3.2 Reading

- For a description of FIR Direct-Form structures see the module FIR Filter Structures (Section 5.1).
- For a description of circular buffers see Circular Buffers (Section 3.1).
- FIR Filter Implementation (Section 5.2).

5.3.3 Filter Implementation

In this lab you will be writing a function `fir_filt` that will take as its input one new sample and produce one output sample. The `fir_filt` function will have the following basic structure:

```
Store the input value in a circular buffer
Compute the new output value
Return the new output value
Store the coefficients in in a header file
```

Here are some tips for writing the function.

- Store the coefficients as `float` values.
- Store the input as `Int16` values (use a global variable or a local static variable). This is not very good programming practice but we will do it to simplify the processing.
- The intermediate sum in the filter calculation should be stored in a float variable initialized to 0.0.
- Be sure to cast the output `float` variable to an `Int16` before output. e.g. `return (Int16) var;`
- When calling the function use a statement like `outbuf[2*i] = fir_filt(inbuf[2*i]);`

5.3.4 Pre-Laboratory

- Design a Bandpass FIR filter with the following requirements:

1. Bandpass
2. Stopband attenuations: 40 dB
3. Passband: 1300Hz to 2000 Hz
4. Transition widths: 600 Hz
5. Sample rate: 8000 samples/sec.
6. DO NOT USE BUILT IN FUNCTIONS IN MATLAB for filter designs. You can use functions like `window` and `sinc`.

- For the filter design, your prelab report should include a plot of the filter coefficients and the magnitude and phase plots of the filter spectrum with non-normalized Hz as the x-axis.
- Save the coefficients to an ASCII data file to be used in your C function below. It would be advantageous to have this completely automated so that you could simply change the bandpass region and output all of the above for a new frequency range (in MATLAB see `fopen`, `fprintf`, `fclose`). Your header file should have the coefficients in an array, be saved in a format like this: `3.007445391758509e-003` and be data type `float`.

[3] This content is available online at <http://cnx.org/content/m37137/1.3/>.

- Write a C function called `fir_filt` that takes as its input one input time point and outputs one filtered time point. Use the coefficients designed above. Use a circular buffer to store the saved values of the input.
- Write a program that filters one channel of the CODEC input and zeros the other channel. The data should be filtered with the function `fir_filt`.
- Your function `fir_filt` should be located in a separate file from you main program. Also, create a header file to contain the declaration of your function `fir_filt`.
- Your files should be organized something like this:

1. `DSK6713_audio.c` - contains your main code for sending and receiving data and filtering the signal.
2. `bpf.h` - contains your filter coefficients and a macro definition for the length of your filter (e.g. `#define N 70`). This file will be included in `fir_filt.c`.
3. `fir_filt.c` - contains your FIR filter function called `fir_filt`.
4. `fir_filt.h` - contains the declaration of your FIR filter `fir_filt`. This will be included in the `DSK6713_audio.c` file.

5.3.5 Laboratory

- Run the program on the DSK and demonstrate the filtered channel. When you build the project you may need to have the -o3 option selected. This will help optimize the code. If you are debugging the code you will need to set this back to None since the optimized code does not work well in debugging.
- Use an input sinusoid and measure the amplitude of the output at different frequencies to test the frequency response. You could use a spectrum analyzer to get a power spectral density of your filter or you could use your laptop to analyze the system.
- You should verify your spectrum that you determined in your pre-lab work. Use a good experimental write-up for this lab.

Chapter 6

Noise Cancellation Lab

Index of Keywords and Terms

Keywords are listed by the section with that keyword (page numbers are in parentheses). Keywords do not necessarily appear in the text of the page. They are merely associated with that section. *Ex.* apples, § 1.1 (1) **Terms** are referenced by the page they appear on. *Ex.* apples, 1

Attributions

Collection: *DSP Lab with TI C6x DSP and C6713 DSK*
Edited by: David Waldo
URL: http://cnx.org/content/col11264/1.6/
License: http://creativecommons.org/licenses/by/3.0/

Module: "C6x Assembly Programming"
By: David Waldo
URL: http://cnx.org/content/m36319/1.1/
Pages: 1-19
Copyright: David Waldo
License: http://creativecommons.org/licenses/by/3.0/
Based on: C62x Assembly Primer II
By: Hyeokho Choi
URL: http://cnx.org/content/m11051/2.3/

Module: "Code Composer Studio v4 Assembly Project"
By: David Waldo
URL: http://cnx.org/content/m36321/1.3/
Pages: 20-33
Copyright: David Waldo
License: http://creativecommons.org/licenses/by/3.0/

Module: "Creating a TI Code Composer Studio Simulator Project"
By: David Waldo
URL: http://cnx.org/content/m33263/1.5/
Pages: 33-39
Copyright: David Waldo
License: http://creativecommons.org/licenses/by/3.0/

Module: "Assembly Programming Lab"
By: David Waldo
URL: http://cnx.org/content/m33373/1.4/
Page: 40
Copyright: David Waldo
License: http://creativecommons.org/licenses/by/3.0/

Module: "Code Composer Studio v4 C Project"
By: David Waldo
URL: http://cnx.org/content/m36322/1.4/
Pages: 41-55
Copyright: David Waldo
License: http://creativecommons.org/licenses/by/3.0/

Module: "C Programming Lab"
By: David Waldo
URL: http://cnx.org/content/m36345/1.3/
Pages: 56-57
Copyright: David Waldo
License: http://creativecommons.org/licenses/by/3.0/

Module: "Circular Buffers"
By: David Waldo
URL: http://cnx.org/content/m36654/1.2/
Pages: 59-61
Copyright: David Waldo
License: http://creativecommons.org/licenses/by/3.0/

Module: "Single Sample and Block I/O"
By: David Waldo
URL: http://cnx.org/content/m36655/1.1/
Pages: 61-65
Copyright: David Waldo
License: http://creativecommons.org/licenses/by/3.0/

Module: "Code Composer Studio v3.3 DSP/BIOS and C6713 DSK"
By: David Waldo
URL: http://cnx.org/content/m36721/1.2/
Pages: 65-79
Copyright: David Waldo
License: http://creativecommons.org/licenses/by/3.0/

Module: "Echo Lab"
By: David Waldo
URL: http://cnx.org/content/m36657/1.1/
Pages: 80-81
Copyright: David Waldo
License: http://creativecommons.org/licenses/by/3.0/

Module: "Second Order System Impulse Response Generation"
By: David Waldo
URL: http://cnx.org/content/m36658/1.3/
Pages: 83-84
Copyright: David Waldo
License: http://creativecommons.org/licenses/by/3.0/

Module: "Cosine Generator Lab"
By: David Waldo
URL: http://cnx.org/content/m36659/1.1/
Pages: 85-89
Copyright: David Waldo
License: http://creativecommons.org/licenses/by/3.0/

Module: "FIR Filter Structures"
By: Douglas L. Jones
URL: http://cnx.org/content/m11918/1.2/
Pages: 91-95
Copyright: Douglas L. Jones
License: http://creativecommons.org/licenses/by/1.0

Module: "FIR Filter Implementation"
By: David Waldo
URL: http://cnx.org/content/m37136/1.1/
Pages: 95-101
Copyright: David Waldo
License: http://creativecommons.org/licenses/by/3.0/

Module: "FIR Filter Lab"
By: David Waldo
URL: http://cnx.org/content/m37137/1.3/
Pages: 102-103
Copyright: David Waldo
License: http://creativecommons.org/licenses/by/3.0/

DSP Lab with TI C6x DSP and C6713 DSK
This laboratory offers a basic introduction to digital signal processing using Texas Instruments C6x DSPs. Labs include topics such as assembly language programming, C programming, and basic filter implementation. The laboratory assignments use the C67x simulator or the C6713 DSK.

About Connexions
Since 1999, Connexions has been pioneering a global system where anyone can create course materials and make them fully accessible and easily reusable free of charge. We are a Web-based authoring, teaching and learning environment open to anyone interested in education, including students, teachers, professors and lifelong learners. We connect ideas and facilitate educational communities.

Connexions's modular, interactive courses are in use worldwide by universities, community colleges, K-12 schools, distance learners, and lifelong learners. Connexions materials are in many languages, including English, Spanish, Chinese, Japanese, Italian, Vietnamese, French, Portuguese, and Thai. Connexions is part of an exciting new information distribution system that allows for **Print on Demand Books**. Connexions has partnered with innovative on-demand publisher QOOP to accelerate the delivery of printed course materials and textbooks into classrooms worldwide at lower prices than traditional academic publishers.