Report 2 for Real-Time Signal Processing

# LAB3: Using C

Through this lab, I learn how to build a project using C source files and two compiler options: Debug build configuration and Release (Optimised) build configuration. I also try to examine fixed and floating-point solutions.

## Create the Lab3 project

Firstly, I create a project named LAB3.PJT and choose TMS320C62XX as my target.

## Create a CDB file

Then I create a new CDB file with the dis6711.cdb template and save it in the same directory as my project LAB3.PJT.

## Adding files to the project

After that, I add following three files into this project:

LAB3.C

LAB3.CDB

LAB3cfg.CMD

## Building the program (.OUT)

Finally, I build the program with the *Debug* configuration*.*

Because I do not enable the *Program Load after Build* option, I should load the program into the microprocessor manually.

## Run and Debug Code

Run to the main function with:

Debug: Go Main

The debugger should run the system initialization code until **main() is reached.**

## Watch Variables

In order to watch variables, I add y in the LAB3.C window in to Watch Window. Because **main()** does not have any local variables, the *Watch Locals* tab is empty.

Then I single-step the debugger into the **dotp()** function, which contains local variables. After several single-steps, I find the *Watch Locals* has been updated.

## Using the Command Window

I also learn how to and variables to the watch window using wa or take commands in the *Command window.*

## Run and Halt

I run the processor by pressing F5 and **CPU RUNNING** is displayed. Then I halt it by pressing shift-F5.

## What’s the Result?

Y=2829056 decimal

## Benchmark / Profile Code

This procedure is used to analyze how long it takes the dot-product function to execute.

Restart the program and make the processor is **halted.**

Open a new profiling session called **Debug**, choose Profile All Functions and expand the lab3.c entry.

After that, I set a breakpoint at the end of **main()**, to stop the processor automatically.

Then I run the program and make one single-step. I examine the Debug Profile window and get that:

The minimum number of cycles for the dotp function is **7992**.

How fast was your dotp? Min Incl **7992** clock cycles.

How fast was your dotp? Max Incl **8598** clock cycles.

Because there is a cache when program initializes, it will take more time to read data into cache. After initialization, the program keeps running, but can read data from cache directly, that will save some time.

If I run it a third time, I will get the **Min**.

## Benchmark with Optimization

In this section, I try to use the *Release* configuration to optimize the code.

Open the build configuration dialog and select *Release* and click the **Add** button.

I call my new configuration “**Optimize**” and it will copy the settings from *Release* and set this new configuration **active**. Then I modify the *Optimize* configuration with adding the **Function Profile Debug (-gp)** option.

Open a new profile session called *Optimize*.

Run the program and single-step again.

How fast was dotp this time?

**Min Incl** 168 clock cycles

**Max Incl** 277 clock cycles

So the code runs faster using the complier’s optimizer.

## Lab 3 Summary

|  |  |  |  |
| --- | --- | --- | --- |
| **Lab Step** | **Lab** | **Build Configuration** | **Cycles** |
| Lab 3 step 42 | Lab 3 integer | Debug | 7992 |
| Lab3 step 60 | Lab 3 Integer | Optimize  (variation of Release) | 168 |

# Lab 3a – Floating-Point Dot-Product

## Create float-point project

Firstly, I create a project called lab3a, open LAB3.CDB and save it as LAB3a.CDB then add the following files into lab3a.

LAB3a.c

LAB3a.CDB

LAB3CFG.CMD

I build and load the program with the *Debug* build configuration.

Open a new profile session and set a breakpoint at the end of **main()**.

Run it to the breakpoint and single-step once.

Y = 2829056 decimal

This is same as the integer result.

How fast was the floating-point **dotp**?

**Min Incl** 9272 clock cycles

**Max Incl** 9525 clock cycles

Obviously, it takes more time than the integer solution. Because the floating-point calculation needs more time than integer calculation, and data in the floating-point format will need more memories to store.

## Benchmark optimized floating-point code

Following the same procedures in the previous section, I get:

Y = 2829056 decimal

How fast was the optimized floating-point **dotp**?

**Min Incl** 308 clock cycles

**Max Incl** 530 clock cycles

It also takes more time than the integer solution.

# Lab 3b – Faster floating-point dotp() using DATA\_ALIGN

Create a project named lab3b.

Open the files LAB3a.c, LAB3a.CDB and data.h and save them as LAB3b.C, LAB3b.CDB and data.h.

Add the following files to the lab3b

LAB3b.C

LAB3b.CDB

LAB3bCFG.CMD

Then, uncomment the following two lines from LAB3b.C

#pragma DATA\_ALIGN(a, 8);

#pragma DATA\_ALIGN(x, 8);

Following the previous procedures, I got:

Y = 2829056 decimal

How fast was the optimized floating-point **dotp** using the pragma DATA\_ALIGN?

**Min Incl** 206 clock cycles

**Max Incl** 425 clock cycles

It takes more time than the integer solution.

|  |  |  |  |
| --- | --- | --- | --- |
| **Take Home Exercises** | **Floating-point Lab** | **Build Configuration** | **Cycles** |
| Lab 3a Step 10 | Lab 3a Floating-Point | Debug | 9272 |
| Lab 3a Step 19 | Lab 3a Floating-Point | Optimize(variation of Release) | 308 |
| Lab 3b Step 12 | Lab 3b Float | #pragma DATA\_ALIGN | 206 |

# Lab 4

This lab exercise will produce a sine wave output via the speaker, we can also so the the sine wave in the oscillograph.

## Open Project

Open the project LAB4.PJT and add the compiler options as following:

-iC:\ti\c6000\dsk\include

-iC:\c60001day\ti\preliminary\_bsl\include

-dCHIP\_6711

-dBOARD\_6711DSK

## Configure Hardware Interrupt (HWI)

### Interrupt Service Routine

Open LAB4.C and get the codec\_out() function:

void codec\_out(void)

{

AD535\_HWI\_write(hAD535, sineGen());

}

This function is to output sine sample.

How can you make the *XINT0\_HWI* function into an interrupt service routine?

There are three steps:

1. Open the DSP/BIOS Configuration Tool, expand the HWI and choose one of the available interrupts(from HWI\_INT4 to HWI\_INT15)
2. Set ***HWI\_INT9***as *McBSP0 transmit interrupt*(XINT0)
3. Enter \_XINT\_HWI as the function name into the *Function* textbox

Open LAB4.CDB and open the **HWI** module.

**INT9** is associated with *MCSP\_0\_transmit* by default.

The function name of the HWI service routine is **XINT0\_HWI**.

Also we should use **Dispatcher**.

Finally, save and close LAB4.CDB.

### Initializing Hardware Interrupts

Open LAB4.C and change fill in the three missing code segments as following:

void init\_HWI(void)

{

IRQ\_**globalEnable**(); // Enable ints globally

IRQ\_enable(**IRQ\_EVT\_XINT0**); // Enable McBSP0 interrupt

}

## Code Functions

### Code Initialization

Open LAB4.C and Replace the ??? entries with the following codes:

void codec\_init()

{

// Use the BSL routines to: Open, reset, & config the AD535

hAD535 = AD535\_**open**(AD535\_localId); // Open AD535 and set hAD535 handle

AD535\_reset(**hAD535**); // Reset AD535

AD535\_**config**(**hAD535**, &**my\_AD535\_Config**); // Configure AD535

### Code Output

The following function is used to output a sine sample from the function sineGen().

void codec\_out(void)

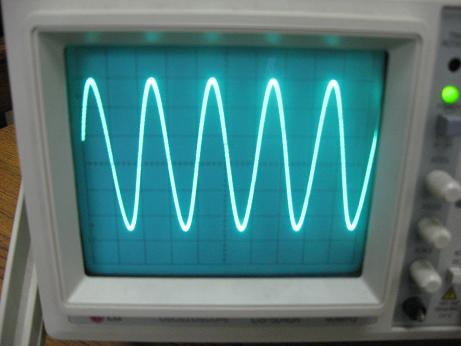
{

AD535\_HWI\_write(hAD535, sineGen());

}

## Build and run…

I can hear the sine wave tone and display it on the oscillograph.



## Using IDL

In the **main()** function, I comment out the **while(1);**loop and replace it with:

Return:

I get the same result as above.

# LAB4a – Try graphing the Sine Wave

Open the project lab4a and setup a breakpoint at the end of the for-loop.

void main()

{

block\_sine(buffer, BUFF\_SIZE);

}

After build the program, I change some graph properties as following:

Graph Title: Graphical

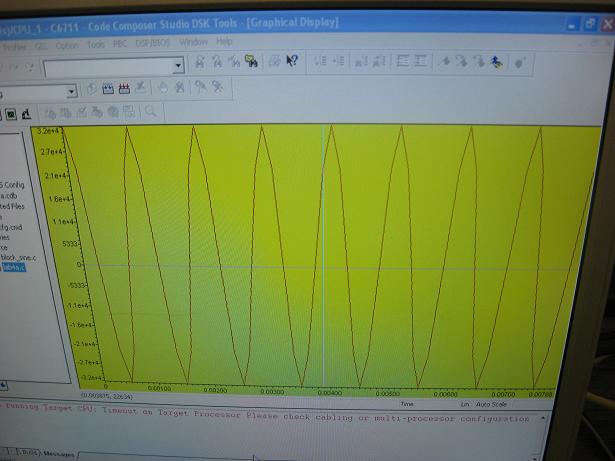
Start Address: buffer

Acquisition Buffer Size: 64

Display Data Size: 64

Sampling Rate (Hz): 8000

Then I get the graph below:



This is the graph of sine wave.

# Lab4b – Compare floating vs. fixed-point sine algorithm

## Float

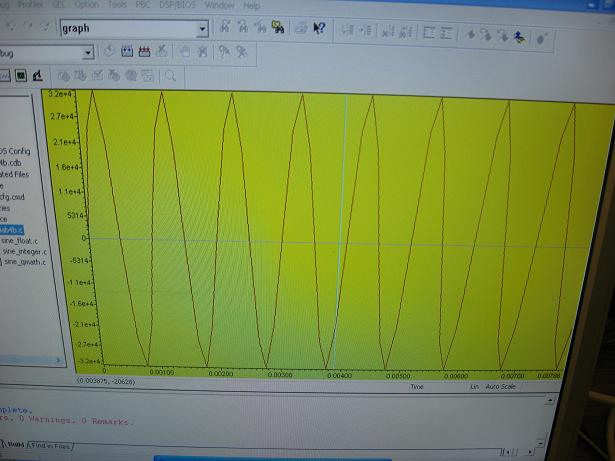
Graph Title: Float

Start Address: graph

Acquisition Buffer Size: 64

Display Data Size: 64

Sampling Rate (Hz): 8000



## Fractional fixed-point

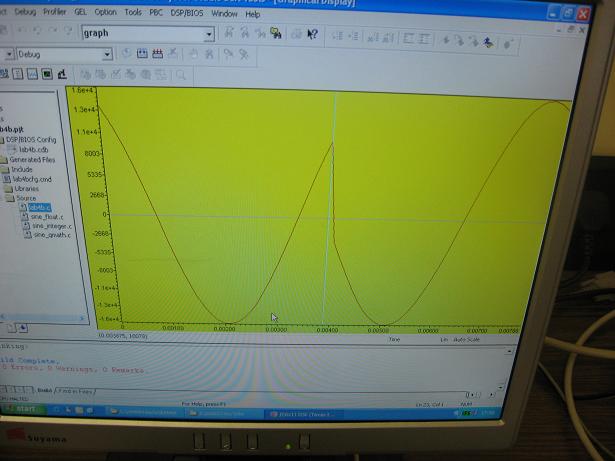
Graph Title: Q

Start Address: graphQ

Acquisition Buffer Size: 64

Display Data Size: 64

Sampling Rate (Hz): 8000



## Standard fixed-point

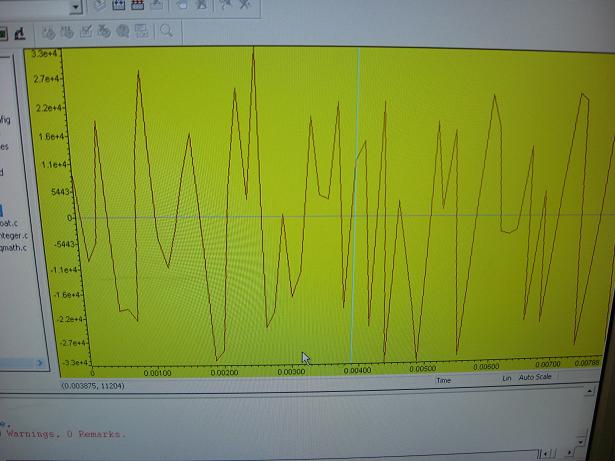
Graph Title: Integer

Start Address: graphInt

Acquisition Buffer Size: 64

Display Data Size: 64

Sampling Rate(Hz): 8000



We can see that the float and fractional solution produce the near-sine wave, but standard solution can not.

# LAB 5

## Open Project

Open the project LAB5.PJT and set the compiler option as following:

-iC:\ti\c6000\dsk\include

-iC:\c60001day\ti\preliminary\_bsl\include

-dCHIP\_6711

-dBOARD\_6711DSK

## Interrupt Service Routine

Fill-in the correct function name in Lab5.c as below:

void myEdma\_HWI(void)

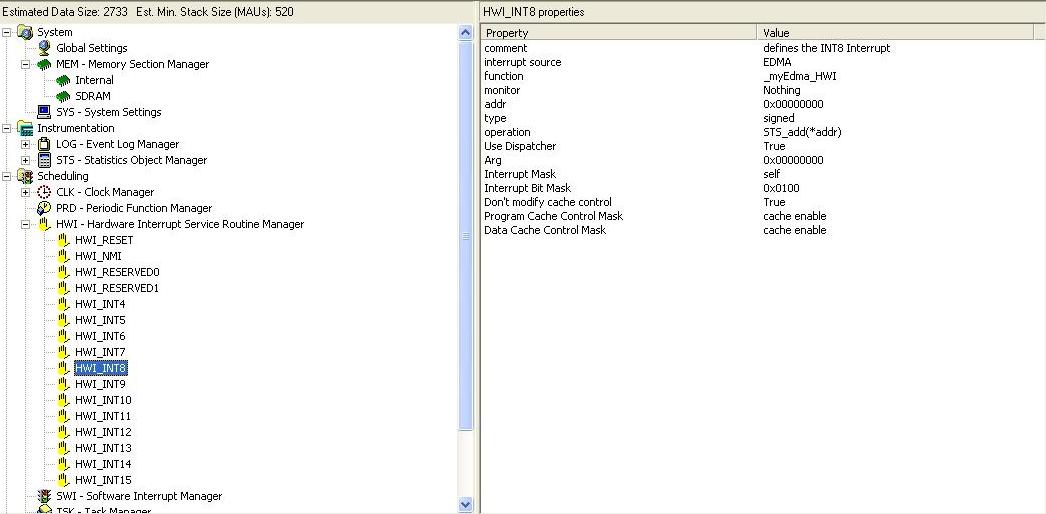
{

/\* Clear the CPU interrupt pending bit in the EDMA \*/

EDMA\_**intClear**(5);

// Call function to that fills the buffer

block\_sine(buffer, BUFF\_SIZE);

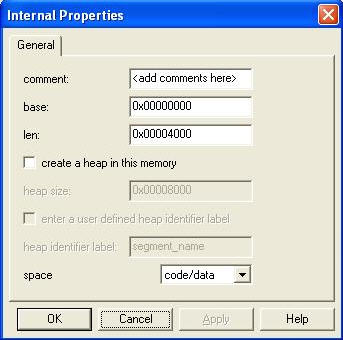


Open the LAB5.CDB, it is **INT8** associated with EDMA and the name of the **HWI** serviceroutine is ***\_my\_Edma\_HWI***

Also I use the interrupt dispatcher routine to handle all the hardware interrupt context save/restore operations.

## Configuring Cache and memory buffer

Insert a new MEM named ***Internal*** and configure it as below:



Then change **C variables Section (.far)** to ***Internal*** in the compiler sections of MEM projerties.

### Initializing Hardware Interrupts

Open LAB5.C to the **init\_HWI** function and fill in the missing code as below:

void init\_HWI(void)

{

IRQ\_globalEnable(); // Enable ints globally

IRQ\_enable(IRQ\_EVT\_**EDMAINT**); // Enable EDMA interrupt (to CPU)

}

## EDMA Functions

### EDMA Initialization

Fill-in the missing code as below:

EDMA\_Config my\_edma\_Config = {

EDMA\_OPT\_RMK(

EDMA\_OPT\_PRI\_LOW,

EDMA\_OPT\_ESIZE\_**16**BIT, // Hint, how many bits wide is the codec?

EDMA\_OPT\_2DS\_NO,

EDMA\_OPT\_SUM\_INC,

EDMA\_OPT\_2DD\_NO,

EDMA\_OPT\_DUM\_NONE,

EDMA\_OPT\_TCINT\_YES,

EDMA\_OPT\_TCC\_OF(5),

EDMA\_OPT\_LINK\_YES,

EDMA\_OPT\_FS\_NO

),

EDMA\_SRC\_OF(buffer), // Where does the EDMA move the data from?

EDMA\_CNT\_OF(BUFF\_SIZE),

EDMA\_DST\_OF(0), // 0x018C0004

EDMA\_IDX\_OF(0),

EDMA\_RLD\_OF(0)

};

void edma\_init()

{

// Which EDMA event are we interested in?

// Hint 1: look up EDMA\_open in CSL Reference Guide

// Hint 2: we want the Xmit EVenT 0 (XEVT0)

hEdma = EDMA\_open( **EDMA\_CHA\_XEVT0**, EDMA\_OPEN\_RESET );

// What's the name of our EDMA\_Config data structure

// used to configure the EDMA channel and reload parameters?

EDMA\_config(hEdma, &**my\_edma\_Config**);

EDMA\_config(hEdmaLINK, &**my\_edma\_Config**);

## Build and Run

Rebuild