

Introducing TI's Integrated Development Environment – CCS (Code Composer Studio) to Expert Engineers

1. Introduction

1.1. Intended Audience – Expert DSP engineer that is new to TI's Code Composer Studio (CCS)

CCS is Texas Instruments' Integrated Development Environment (IDE) based on the open source Eclipse architecture. It is used to build, debug, and run DSP applications as well as other processor applications.

TI provides a great amount of CCS training, documentation, and other help that covers all aspect of CCS. The following section provides links for the training. So who needs this document?

The intended readers of this document are DSP experts who have not yet worked with TI tools, yet are very knowledgeable and have worked with other vendors' tools. So they know what to expect from tools, understand the logic behind tools, and only need to know the mechanics of the tools. They may not have patience or time to go through training. Their goal is to jump in and try to run a test application.

In addition to the CCS tool, TI provides a great deal of software blocks to facilitate easy development of applications on TI's devices, including a set of optimized libraries for standard Mathematics (MATHLIB), Signal Processing (DSPLIB) and Image Processing (IMGLIB). A DSP expert would like to use these optimized functions in applications. This document shows an expert DSP engineer how to develop applications that call optimized library functions.

Steps to take when Start Porting DSP Algorithm into TI Environments

When porting an existing DSP algorithm that was developed under a different environment into TI's Integrated Development Environment CCS, the expert engineer will go through the following steps:

1. TI's Processor SDK is a comprehensive set of software and firmware tools, utilities and example modules that supports many TI processors. Each module has a unit test project that demonstrates how to use the module. The easy way to understand how to use a library function is to import the unit test of the said function and run it on some hardware such as Evaluation Module (EVM). Chapter 2 shows how to import a project from the release, build it, and run it on standard hardware.
2. The next step is building a new application that utilizes the library function that was used in the previous step. Chapter 3 shows how to build a new non-trivial (e.g., fairly complex) project, build it, and run it on standard hardware such as TI Evaluation Module (EVM).
3. Processor SDK is a uniform release of software blocks that guarantees working together. Three standard libraries are included in the Processor SDK release: DSPLIB, MATHLIB, and IMGLIB. In addition, TI developed a set of optimized libraries that are not part of the Processor SDK release. These libraries include IQMATH, FASTRTS, VICP, VLIB, FAXLIB and VOLIB (see http://processors.wiki.ti.com/index.php/Software_libraries for more details). In addition, there are devices that are not supported by the standard Processor SDK, but rather by their own Software Development Kit (SDK). Chapter 4 shows how to build an example code (unit test) C674X that is not supported by Processor SDK using a library function from a dedicated FFTLIB library.

1.2. CCS On-Line Training Resources

The following is a partial list of

- [CCS Training Page](#) contains lots of training materials includes Videos and documents
- [TMS320C6000 Optimization Workshop](#): Chapter 2 discusses CCS (and provides an introduction to C6000 architecture)
- [The Code Composer Studio \(CCS\) Integrated Development Environment \(IDE\)](#) is the location to download CCS. It has links to other CCS information.
- [Processor SDK RTOS Setup CCS](#) has a good introduction to using CCS with Processor SDK. Some of the materials referenced in this document are covered.
- TI's [Code Composer e2e Forum](#) is a public forum dedicated to questions and answers about everything CCS. Almost any issue that you may encounter has probably been discussed previously in this forum.

If the above list is not what you are looking for, continue through this document.

Getting Started with CCS

The instructions and the screen shots in this document are taken from CCSv6 (6.1.3). Different versions of CCS might have slightly different screen shots. This document assumes that the user has installed CCS already.

CCS puts all the metadata that is associated with its operation in the workspace. There is a default workspace (usually in `c:/users/user_name/workspace_v6` or similar, where `user_name` is the user login name) where multiple projects can reside. In addition, the user can define other locations as workspace for a specific project.

The first time CCS is opened in a new workspace, the display window (see below) provides links to collateral that provide training and other support documents.

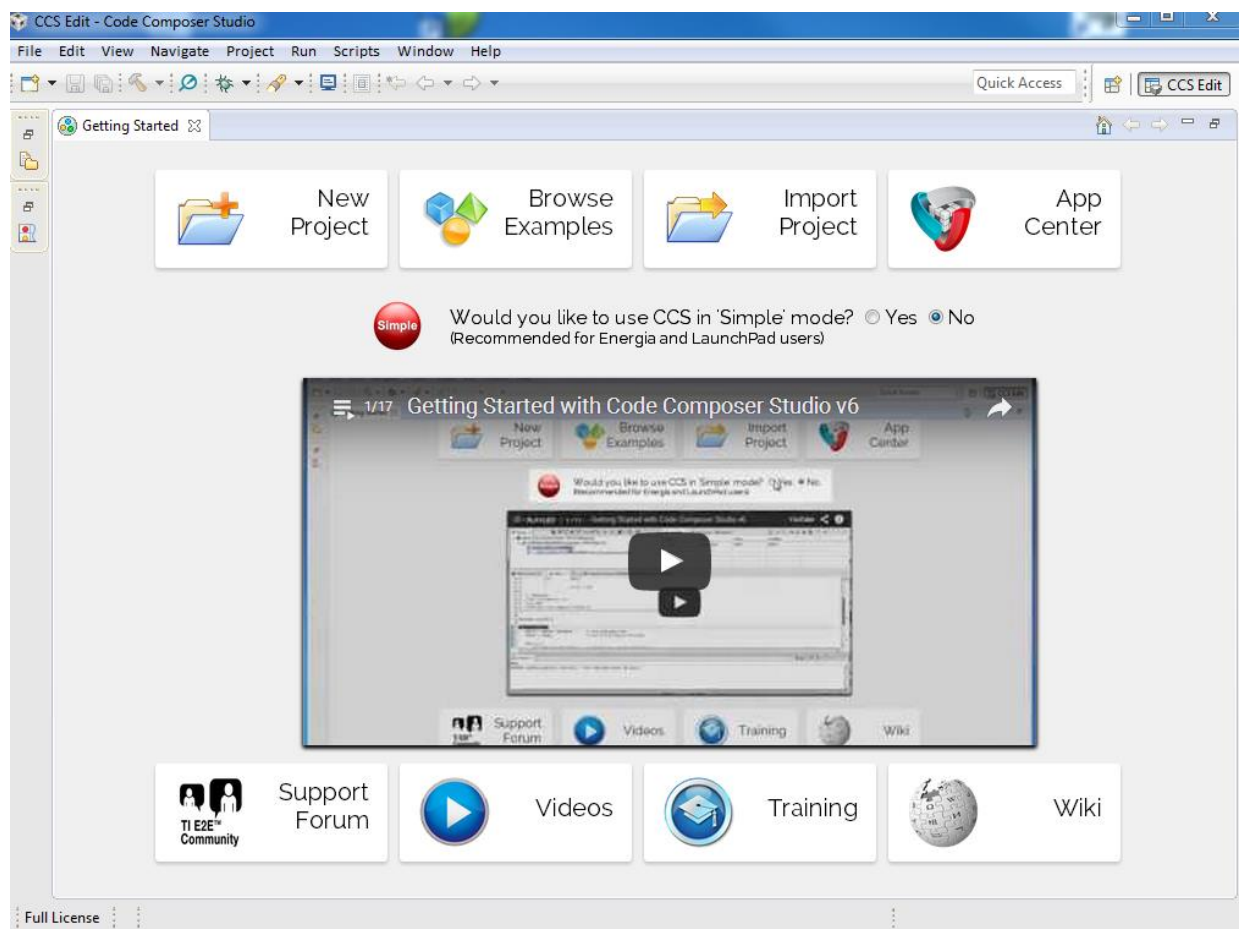


Figure 1.2.1: CCS Getting Started Display

1.3. CCS Edit and Debug Perspectives

CCS has two perspectives:

- **CCS Edit** perspective is used for creating projects and building code. To switch to the CCS Edit perspective, click on *Window* → *Perspective* → *Open Perspective* → *CCS Edit*.
- **CCS Debug** perspective is used for execution and debugging of code on the customer EVM. To switch to the CCS Debug perspective, click on *Window* → *Perspective* → *Open Perspective* → *CCS Debug* (See Figure 2).

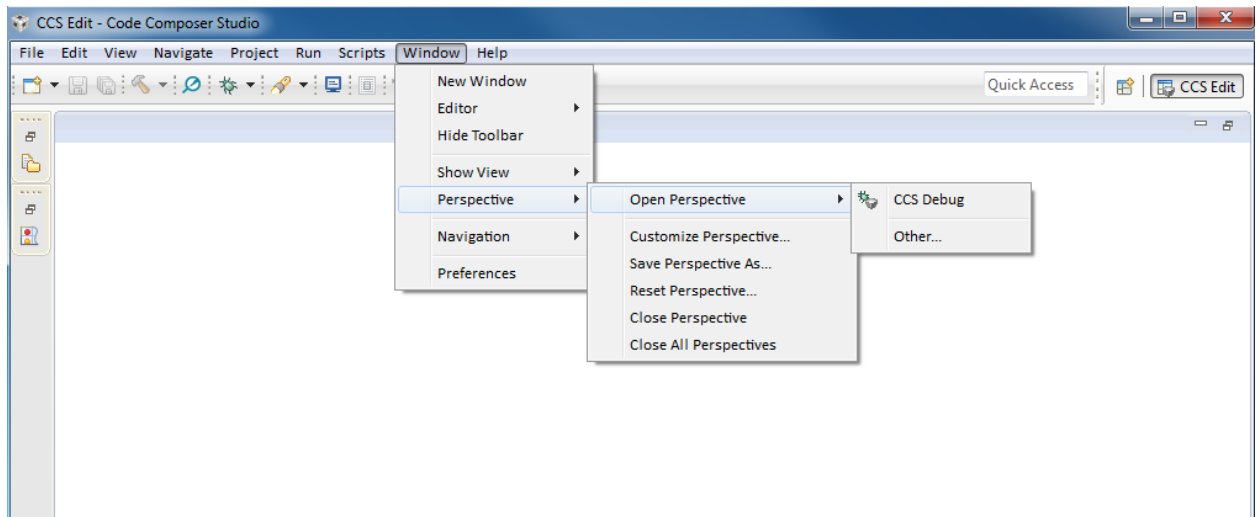


Figure 1.3.1: Changing the CCS Perspective

The current perspective can be seen in the upper right corner of the CCS window, as shown in Figure 2. Upon starting CCS, the default perspective is the CCS Edit perspective.

2. Import CCS Project from Release Examples

2.1. Before Importing a Project

Processor SDK has many examples and unit tests within a release that can be imported into a project. Instructions on how to import a project from a release are provided in this chapter.

Most of the examples in the release are based on the Real Time Software Component (RTSC) scheme. RTSC enables the system to rebuild drivers and utilities for a user-defined platform from a configuration file. To achieve that the CCS environment must be aware of the location of the various building modules in the Processor SDK release, In other words, the user must verify that CCS “sees” all the modules in the release.

Assume a new release was installed in directory C:\ti\Releases\Release_3_0_0_4\C667X. The following steps are needed to add or verify that CCS sees the new release.

1. Click the *Window* tab and select *Preferences*

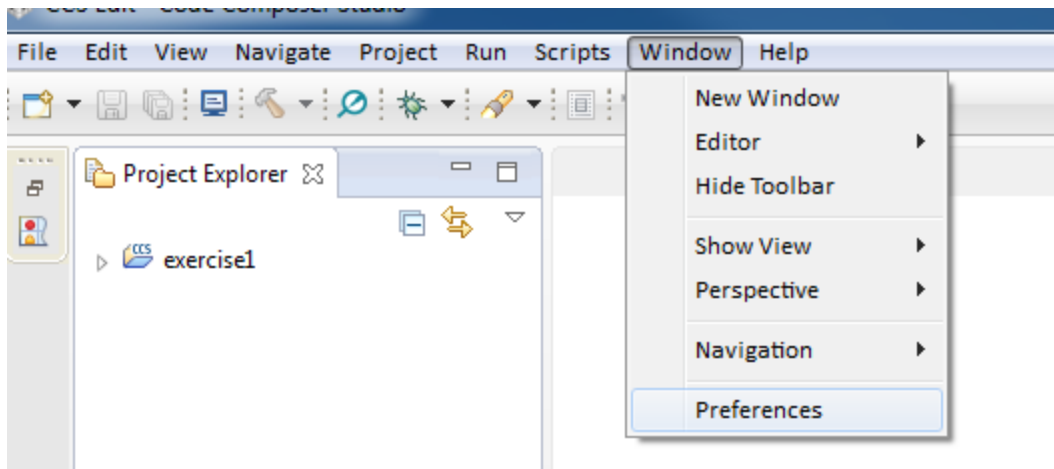


Figure 2.1.1: CCS Edit Perspective: Window Drop-down Menu

2. The Preferences dialog box opens. Navigate to *Code Composer Studio* → *RTSC* → *Products*.

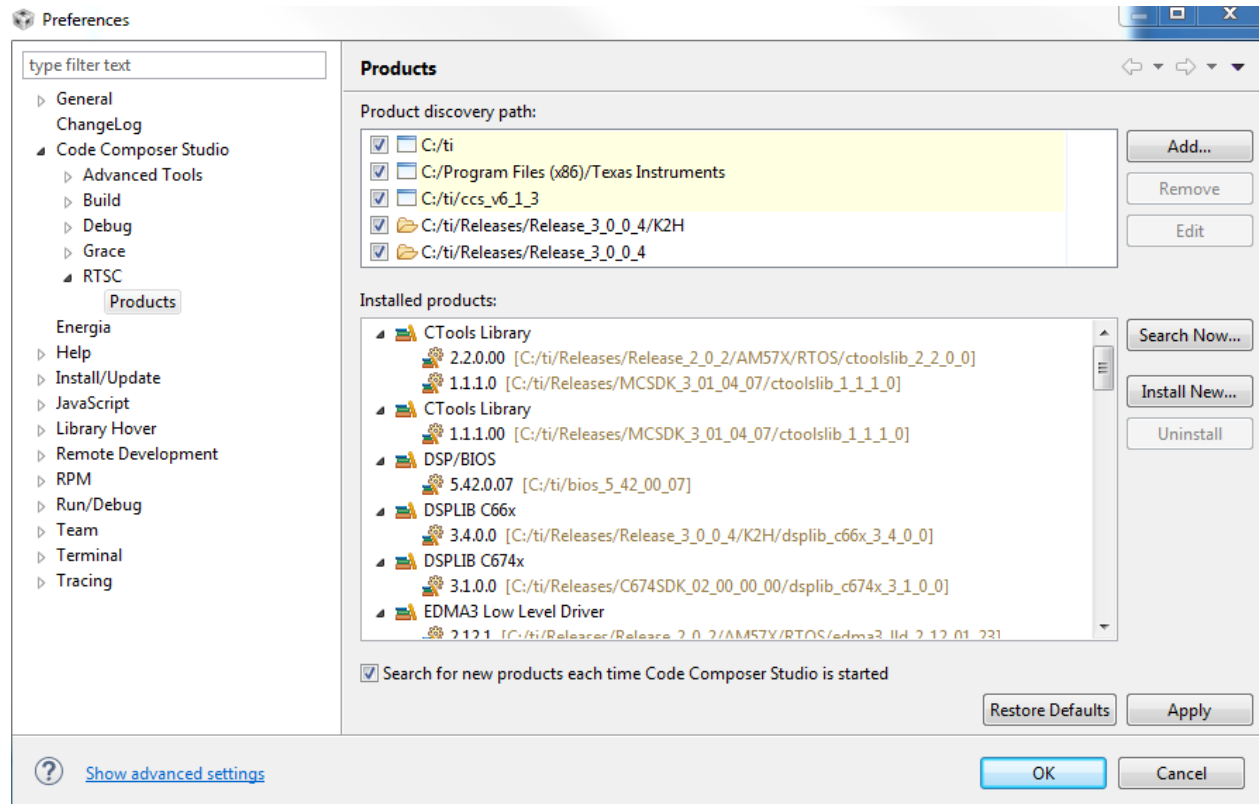
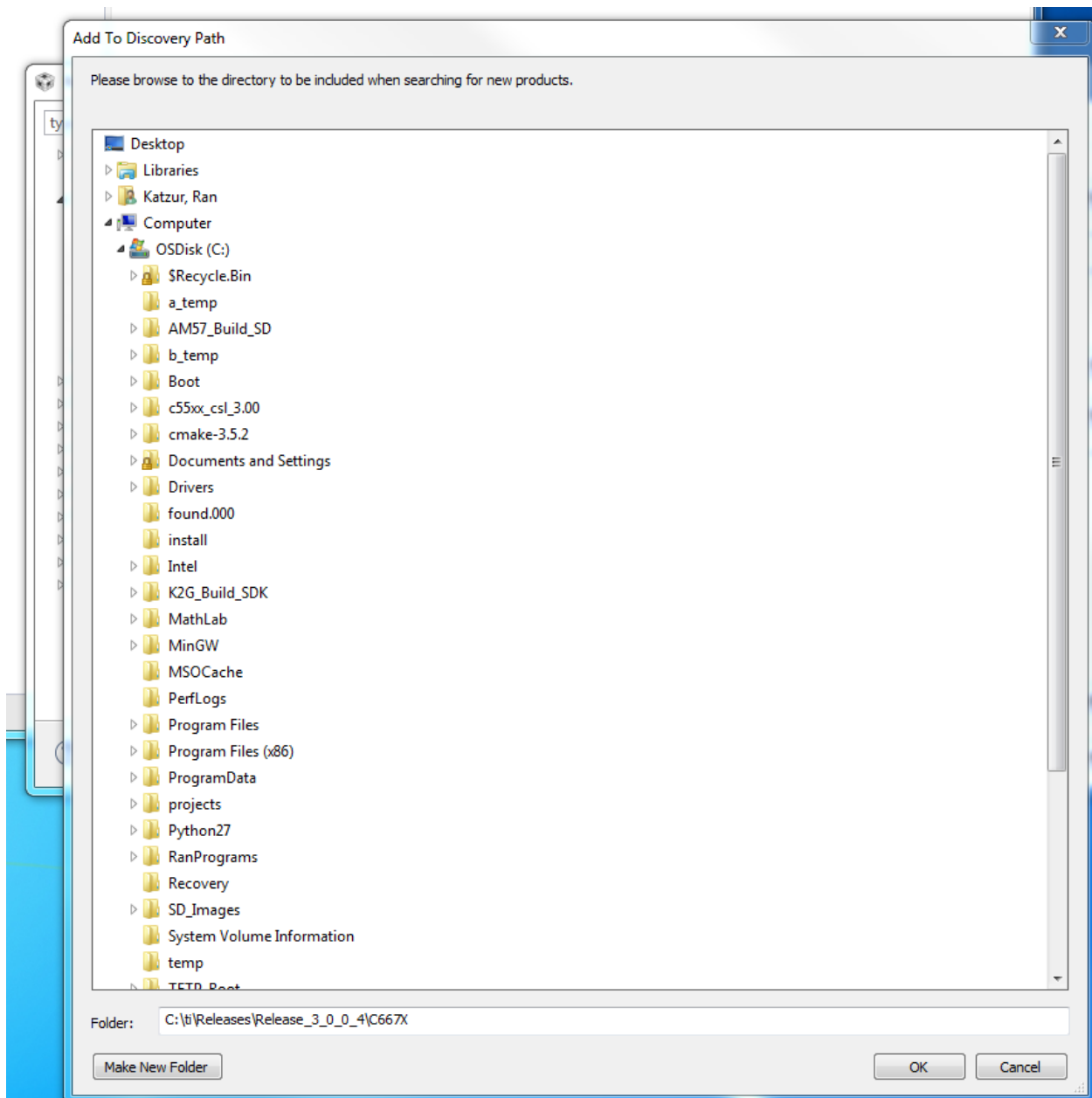


Figure 2.1.2:

3. In the Product Discovery Path, make sure that the location of the new release is specified. If the path is not there, click on the *Add* tab, add the directory name or browse to the directory and click *OK*.



CCS will scan the new location and report back what new modules it found. Click *Finish*. CCS will add the new module. A dialogue box may ask if the user trust the software and the answer is *Yes*, and then restart CCS now.

Note – Some releases have issue with multiple NDK releases. If CCS reports error when it loads NDK the user can un-checks NDK before clicking on *Finish*.

2.2. Import the FFT Project

In this next section, we will use and FFT project as an example. To get started - left click on the *Project* Tab in the CCS EDIT perspective, select import CCS Projects and left click.

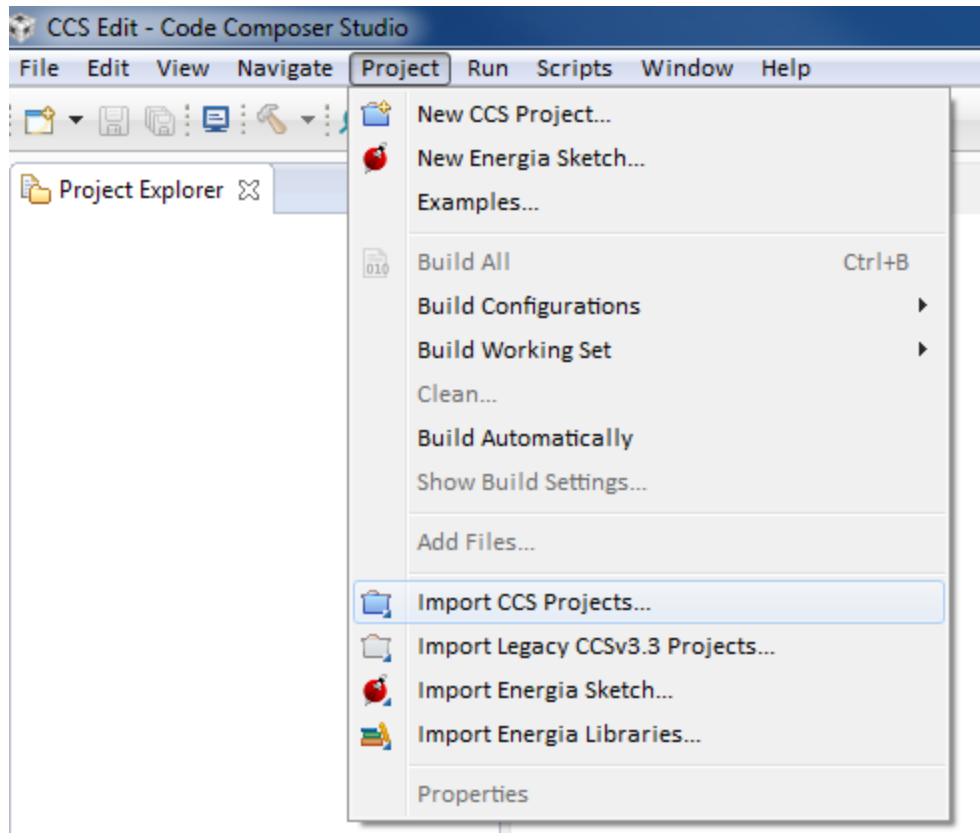
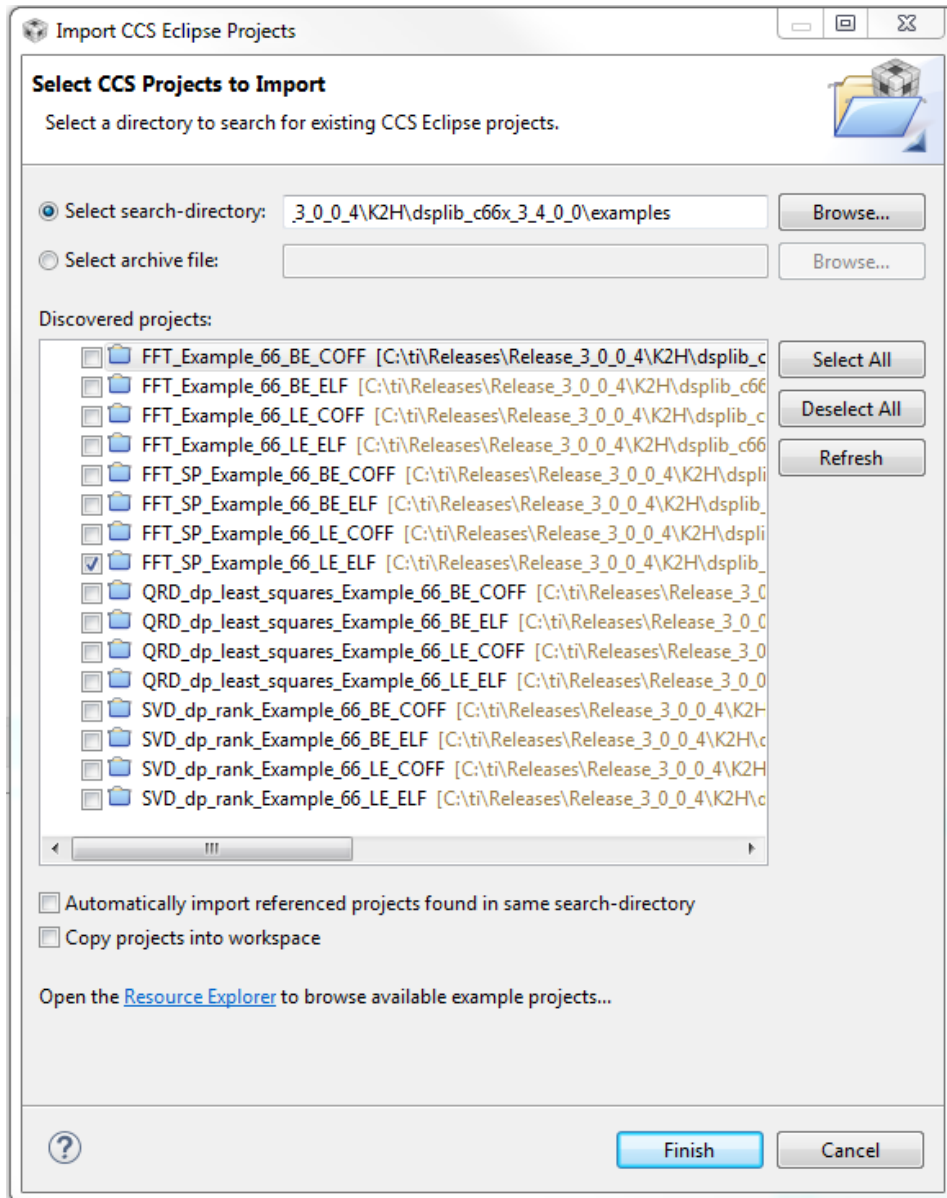


Figure 2.2.1:

A dialog box is opened. For the Select search-directory click on *Browse* and navigate to the location where the dsplib directory was installed on your system->examples and select *OK*. CCS will look for all the examples in this directory. We will work with FFT_sp_Example_66_LE_ELF. LE stands for little endian format and ELF stands for the standard executable format. The window will look like the following:



Click on *Finish*. CCS imports the project but may give some warnings in the problems window. The problem may refer to Invalid Project Path. This may be the result of different directory structure between the developer of the project and the user. The next step is to fix these issues.

Clicking in the small arrow next to the project name opens the project explorer. There are three files, the test source code – `fft_example_sp.c`, the linker command file `lnk.cmd`, and an initialization file `macros.ini_initial`. Double click on the *macros.ini_initial* opens the file in the editor windows. This file defines three locations.

```
MATHLIB_INSTALL_DIR=c:/ti/mathlib_c66x_3_1_0_0
DSPLIB_INSTALL_DIR=c:/nightlybuilds/dsplib
EXT_ROOT__FFT_SP_EXAMPLE_66_LE_ELF_FFT_SP_EX=../..
```

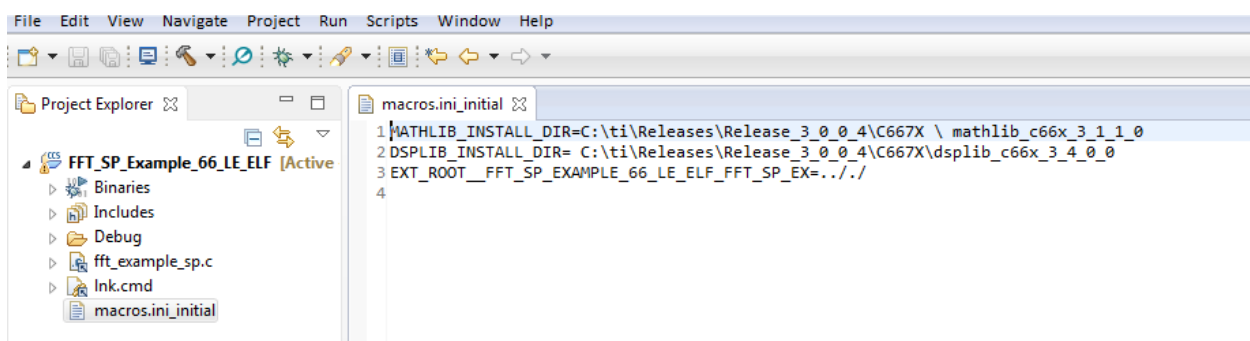
The last location is relative to the example directory and is correct. But the other two point to locations in the developer system. The user has to change these paths.

MATHLIB is an optimized library for mathematical functions. It is part of the release. So its location depends where the user installed the Processor SDK. The screen shots were taken from a system where the Processor SDK release location is `C:\ti\Releases\Release_3_0_0_4\C667X` and the mathlib version is `mathlib_c66x_3_1_1_0`, thus the first location will be defined as

```
MATHLIB_INSTALL_DIR=C:\ti\Releases\Release_3_0_0_4\C667X\mathlib_c66x_3_1_1_0
```

Similarly, the second location is the location of the DSPLIB. For the same system the location will be defined as `DSPLIB_INSTALL_DIR=`

`C:\ti\Releases\Release_3_0_0_4\C667X\dsplib_c66x_3_4_0_0`. The screen shot shows the updated locations. As was mentioned before, the user paths depend on the user install directory of the Processor SDK.



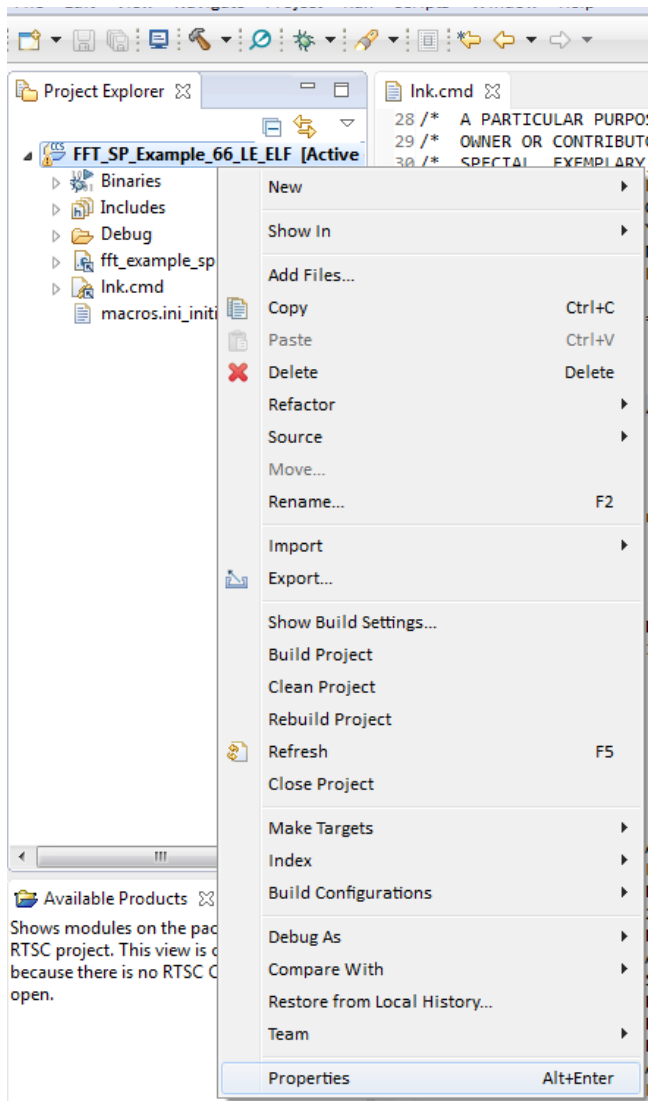
Save the updated file by either select file->Save or by clicking on the disk icon just below the edit tab. The user can close the file by clicking on the x next to the file name in the edit window.

Before building the project let's look at the linker command file `lnk.cmd`. To open it the user selects the file and right clicks to open with text editor. In addition to stack size and heap size,

linking a generic library. During the building process the correct library will be linked, depends on the properties of the project. For little endian ELF format case dsplib.ae66 will be linked. For little endian COEF format case dsplib.a66 will be linked. The COEF format is an old TI proprietary format that is used only in backward compatibility projects. For big endian ELF format case dsplib.ae66e will be linked. For big endian COEF format case dsplib.a66e will be linked.

Two memory segments are defined for this project, the internal L2 memory and the shared MSMCRAM memory. The internal L1P and L1D memories are configured as cache. Each section of memory should be allocated in one of the memory segments; otherwise the linker will allocate it in a default segment and give warning message.

Last we look at the project properties. Right click on the project and select the last item – *Properties*:



In the properties dialog box, the optimization should be set to off, and the debug option to full symbolic debug. Note that library routines that will be called are optimized routines that were built with full optimization and with no symbolic debug. The user is encouraged to explore the project properties, and then close the properties window.

Rebuild the project by right clicking the project name and select *Rebuild Project*. The following screen shot shows the result of the build:



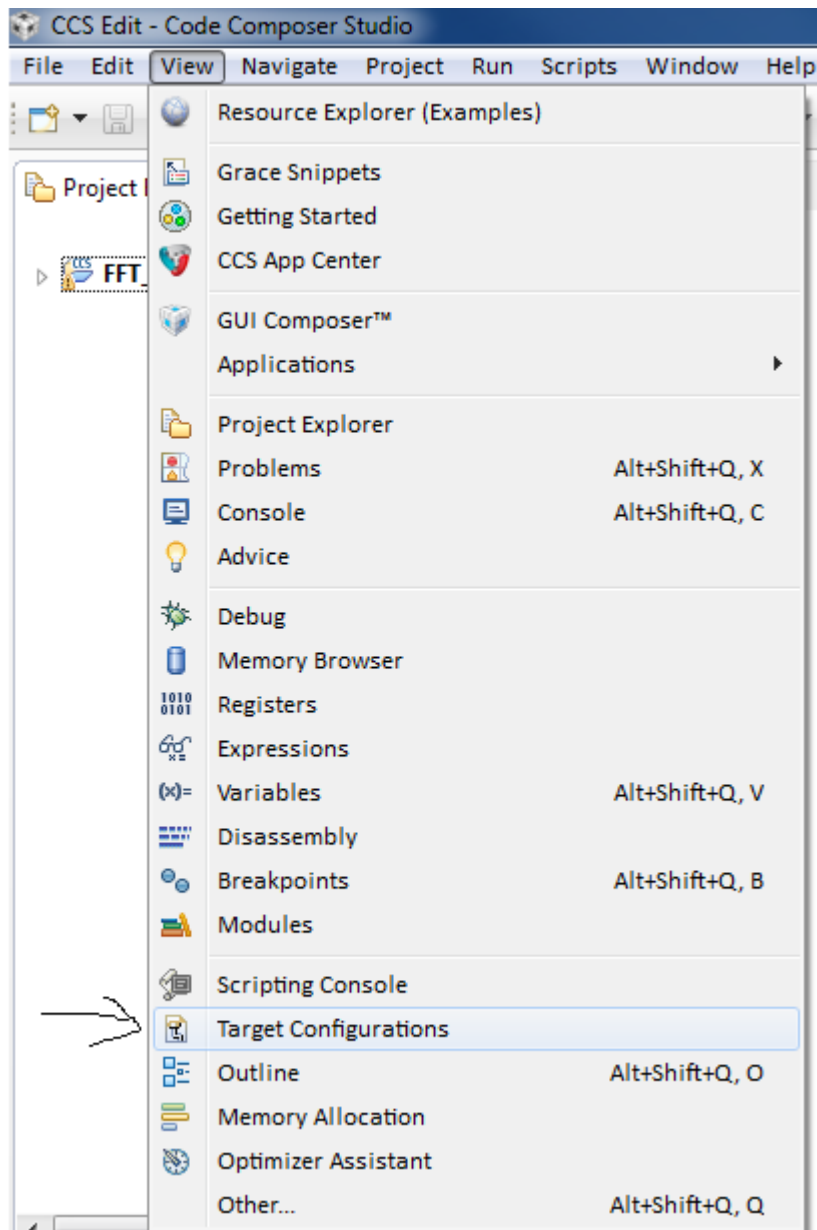
```
Console
:DT Build Console [FFT_SP_Example_66_LE_ELF]
'Finished building: C:/ti/Releases/Release_3_0_0_4/K2H/dsplib_c66x_3_4_0_0/examples/tft_sp_ex/tft_example_sp.c'
'Building target: FFT_SP_Example_66_LE_ELF.out'
'Invoking: C6000 Linker'
'C:/ti/ccsv6_1_3/ccsv6/tools/compiler/c6000_7.4.16/bin/cl6x" -mv6600 --abi=eabi -g --define=ti_targets_elf_C66 --diag_wrap=off
--diag_warning=225 --display_error_number --mem_model:data=far --debug_software_pipeline -k --strip_coff_underscore -z
-m"FFT_SP_Example_66_LE_ELF.map" -i"C:/ti/ccsv6_1_3/ccsv6/tools/compiler/c6000_7.4.16/lib"
-i"C:/ti/ccsv6_1_3/ccsv6/tools/compiler/c6000_7.4.16/include" -i"../../../../" -i"C:/ti/mathlib_c66x_3_1_0_0/packages"
-i"C:/nightlybuilds/dsplib" -i"../../../../" --reread_libs --diag_wrap=off --display_error_number --warn_sections
--xml_link_info="FFT_SP_Example_66_LE_ELF_linkInfo.xml" --rom_model -o "FFT_SP_Example_66_LE_ELF.out" "./fft_example_sp.obj"
'C:/ti/Releases/Release_3_0_0_4/K2H/dsplib_c66x_3_4_0_0/examples/fft_sp_ex/link.cmd" -llibc.a
<Linking>
warning #10349-D: creating output section ".init_array" without a SECTIONS specification. For additional information on this
section, please see the 'C6000 EABI Migration' guide at
http://processors.wiki.ti.com/index.php/C6000_EABI:C6000_EABI_Migration#C6x_EABI_Sections
'Finished building target: FFT_SP_Example_66_LE_ELF.out'

**** Build Finished ****
```

2.3. Define Target- Emulator

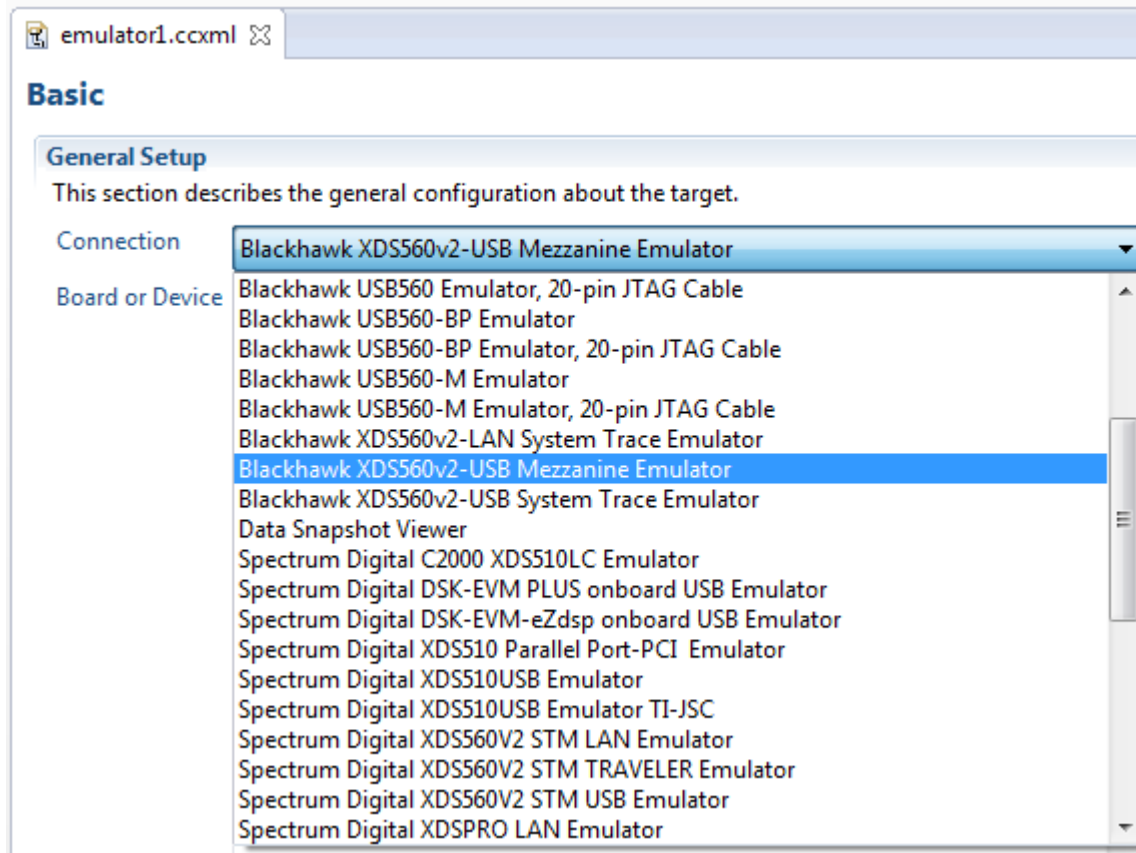
CCS communicates with the board via an emulator. In this example the EVM that is used is TMS320C6678 Evaluation Modules with a daughter card that is Blackhawk XDS560v2-USB Mezzanine Emulator so the following instructions will be for this emulator. If a different emulator or/and different EVM is used, the instructions will be changed accordingly.

From the CCS Edit perspective click on *View->Target Configurations* (see below) . A target Configuration window will be opened.

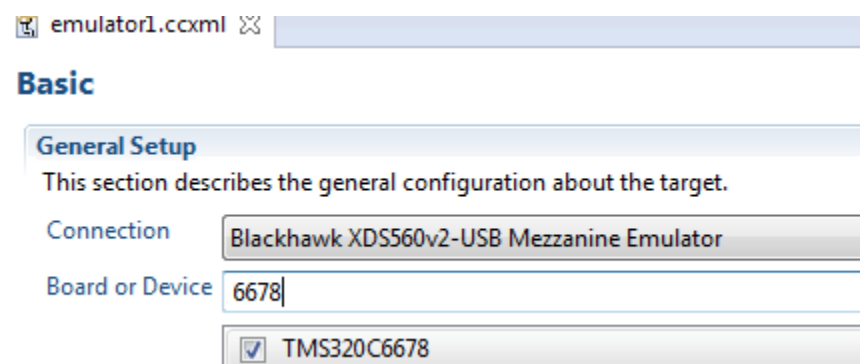


In the User-Defined section, the user right-clicks and selects *New Target Configuration*. In the opened window give a name. For the purpose of this document the target name is emulator1. After clicking on *Finish*, the emulator definition is opened in the editor window.

The first step is to choose the *Connection*. From the Pull down menu the user selects the emulator that is used, see the screen shot below.



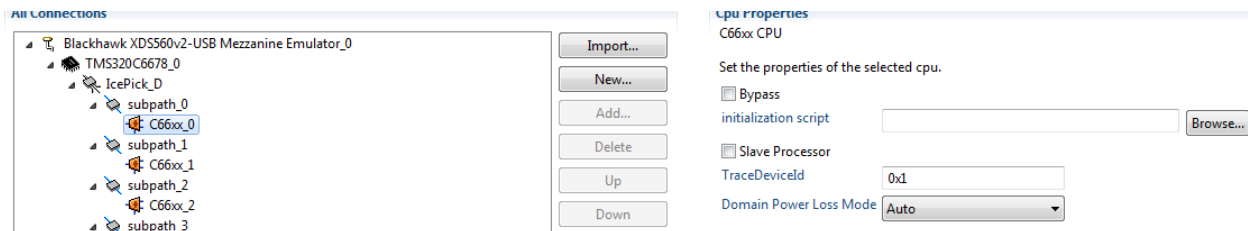
Next a set of supported boards and devices are in the Board or Device window. A filter can apply to help find the desired board. For this document TMS320C6678 was chosen. After a board is chosen the user can save the configuration. If the board or the EVM is powered and the emulator is linked, the user can test the connection using the *Test Connection* tab in the middle of the window.



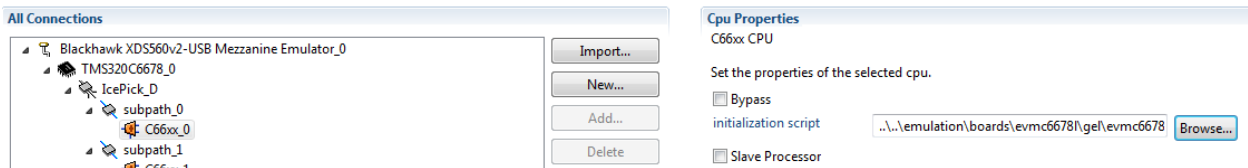
To initialize the hardware CCS uses a script written in “General Extension Language” or gel. <http://processors.wiki.ti.com/index.php/GEL> gives more information about gel files. When a

target is defined, the user should attach the correct gel file to cores in the target. (Usually it is enough to connect the gel to core 0, since core 0 does all the system initialization)

At the bottom of the emulator1.ccxml (or whatever name the user gave to the target) window there is an “Advanced” tab, clicking on this tab will open a display of all the CPUs in the system. The user selects core 0 and browse for the correct gel file



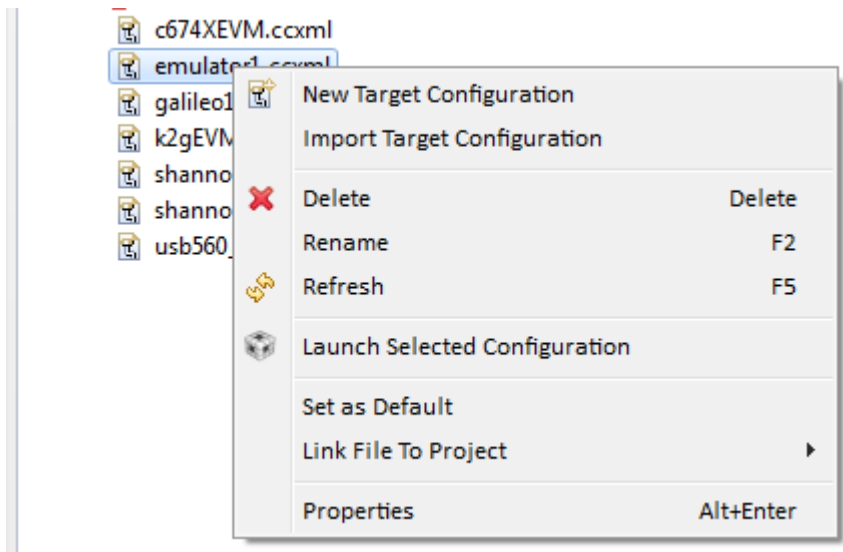
Gel file are located in the directory where CCS was install in the sub-directory `\ccsv6\ccs_base\emulation\boards\BOARDNAME\gel` where BOARDNAME is the board that is used. For this example evmc6678l is used. After selecting the gel file and clicking the *Open* tab at the bottom of the dialog box, the gel location is in the target configuration as seen in the next screen shot.



The last thing is to save the configuration (clicking on the *Save* tab). The user can close the emulator1.ccxml window.

2.4. Connect to the Target and Run the Project

Selecting the target in the target configuration window and right click opens a menu. The user can set the target as a default target and the user can launch Selected Configuration.



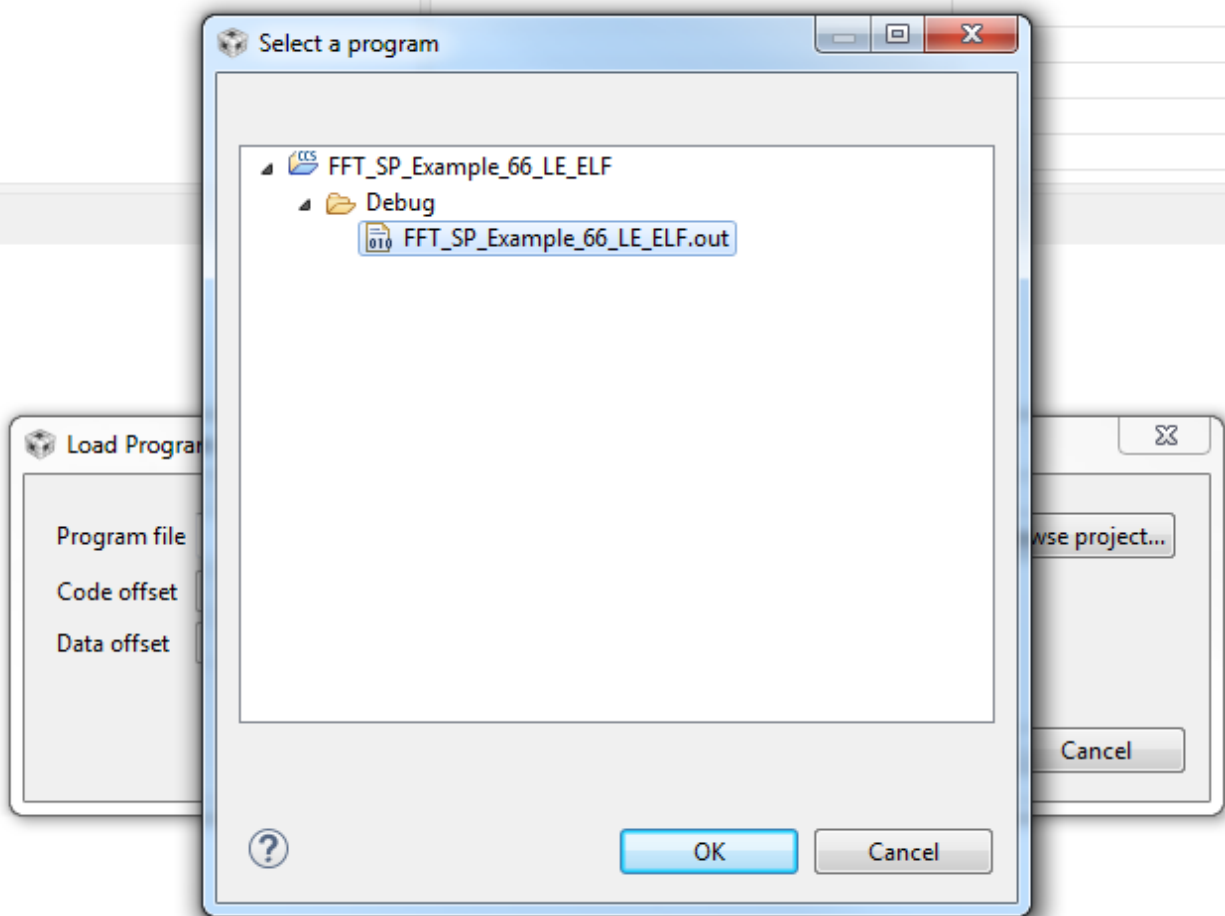
The CCS changes perspective to debug and displays all the CPUs in the system. Next the cores that are involved in the execution need to be connected. In this case the code runs only on a single core, so core zero is selected and is connected. This is done by selecting core 0, right click and select *Connect Core*. Core 0 will go through all the initialization steps that are defined in the gel file, and prints the progress in the Console window. See the screen shot below for the last printing in the Console.

```

Console
emulator1.ccxml
C66xx_0: GEL Output: PA PLL Setup... Done.
C66xx_0: GEL Output: DDR3 PLL (PLL2) Setup ...
C66xx_0: GEL Output: DDR3 PLL Setup... Done.
C66xx_0: GEL Output: DDR begin (1333 auto)
C66xx_0: GEL Output: XMC Setup ... Done
C66xx_0: GEL Output:
DDR3 initialization is complete.
C66xx_0: GEL Output: DDR done
C66xx_0: GEL Output: DDR3 memory test... Started
C66xx_0: GEL Output: DDR3 memory test... Passed
C66xx_0: GEL Output: PLL and DDR Initialization completed(0) ...
C66xx_0: GEL Output: configSGMIISerdes Setup... Begin
C66xx_0: GEL Output:
SGMII SERDES has been configured.
C66xx_0: GEL Output: Enabling EDC ...
C66xx_0: GEL Output: L1P error detection logic is enabled.
C66xx_0: GEL Output: L2 error detection/correction logic is enabled.
C66xx_0: GEL Output: MSMC error detection/correction logic is enabled.
C66xx_0: GEL Output: Enabling EDC ...Done
C66xx_0: GEL Output: Configuring CPSW ...
C66xx_0: GEL Output: Configuring CPSW ...Done
C66xx_0: GEL Output: Global Default Setup... Done.

```

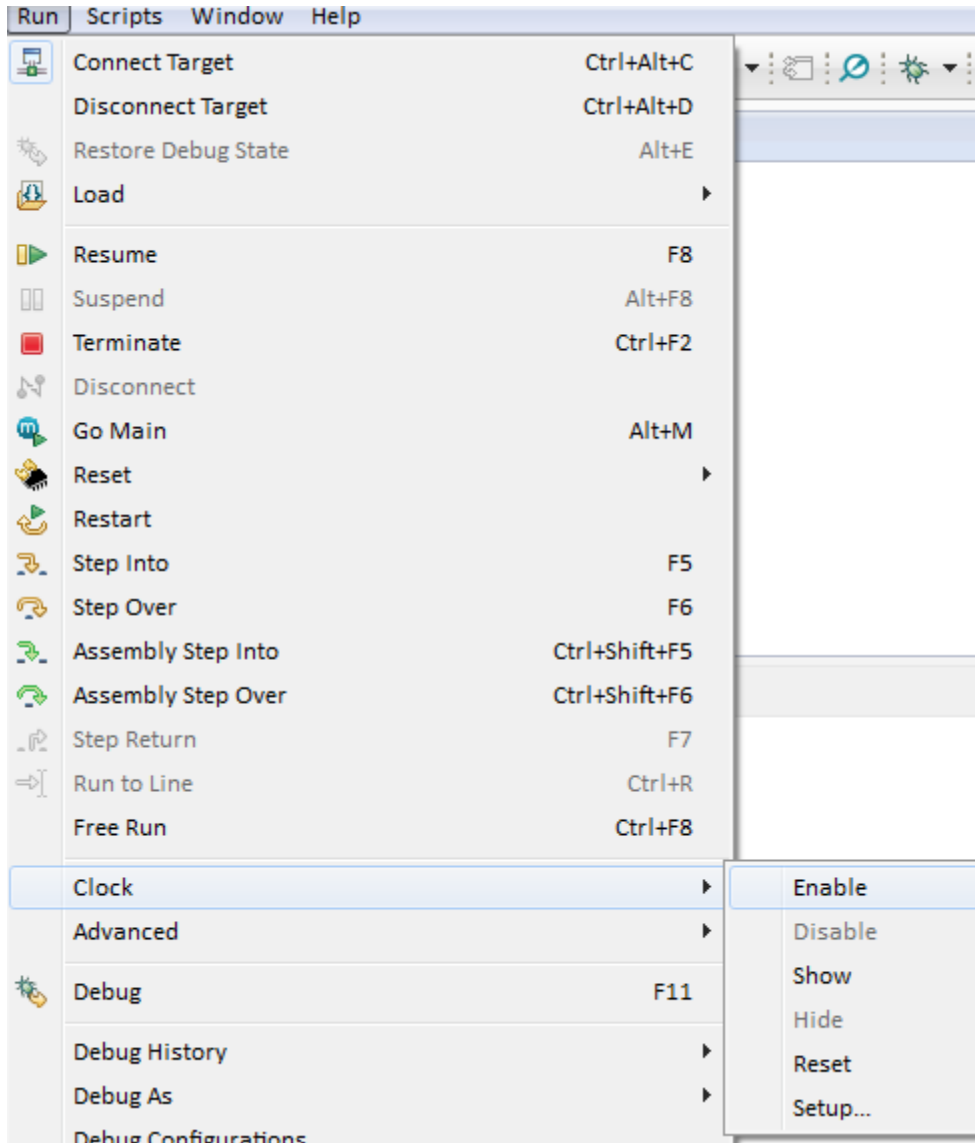

Next the executable is loaded into the core. There are multiple ways to load code (as well as to Run and other operations) but in this document only one way is described. Core 0 is still selected, from the RUN menu right click on *Load* and Load Program. The window that is opened enables the user to Browse, or Browse only Project. The easiest way is to Browse a project and go to the Debug directory and select the out file:



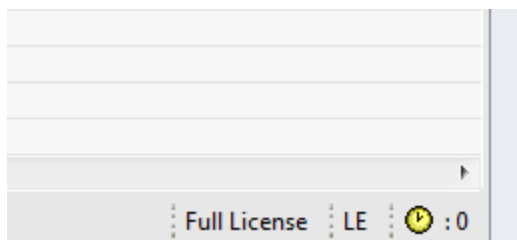
Click *OK* twice, the code is loaded and the main function appears in the edit window.

2.5. Code Execution and measure cycles

Enabling the CCS clock is done from the Run menu. Clicking on *Clock Enable* (see below) opens a small clock window with value of 0. Double click on the *Cycle count* always set the clock to zero.



And the clock window looks like the following (right bottom corner of the CCS window)



Next we step through the code using *F6* key (or from the Run menu click *Step Over*). After three steps the execution is about to execute the DSPF_sp_fftSPxSP routine. At this point the clock is set (*double click*) to zero.

Before executing the DSPF_sp_fftSPxSP routine let's look at the parameters for the function. The document that describes the function and the parameters that are used is [TMS320C67x DSP Library Programmer's Reference Guide](#) –page 49. The first parameter is the number of elements must be power of 2 and up to 8K. Note that the twiddle factors that are generated by the function gen_twiddle_fft should be called with the same value. Next are the pointers to the input data, the twiddle factors and the output vector. Each of these vectors are of $2*N$ floating point size. The bit reversal vector brev is next. According to the documents the brev size is 64 regardless of the FFT size. The next 3 parameters are used to optimize the execution. n_min is 4 if N is power of 4, and 2 otherwise. This value tells the program if it can use all Radix 4 butterflies (4) or must use Radix 2 butterflies, at least once (2). The last two parameters enable the program to break the FFT into multiple executions so that the data fits into L1D cache. is [TMS320C67x DSP Library Programmer's Reference Guide](#) explains the concept in great details.

Click *F6* one more time, the code progress after the DSPF_sp_fftSPxSP routine and the cycle counts (the clock) shows ~1513 cycles.

3. Chapter 3 – Build a New CCS Project

The previous chapter shows how to import a project. Each module of Processor SDK including all functions in the optimized libraries have unit test that shows the user how to use the function. The next step is to build a new project using the same library function that was used in the previous chapter.

While different devices may (or may not) have different implementation of library functions, the interface and the parameter list of the function are the same across different platforms. Thus in this example we use the TI's EVMK2H evaluation module with the 66AK2H12 processor.

In this example we build a 66AK2H12 project a single C66XX core that generates random numbers as input, calculates the energy in the sequence, execute FFT function from a library, calculate the energy in the frequency domain, and printout the difference between the two energies. (Parseval's theorem implies that the two energies must be equal)

There are multiple ways to use library functions and other software modules that are part of Processor SDK. The first method is direct usage of libraries and other utilities. The other method is using RTSC – Real-Time Software Component. While many TI examples are using RTSC to facilitate fast and accurate building of projects, the project in this chapter will be created without RTSC support.

3.1. Create a New Project

Start from the file menu (at the upper left corner);

File → New → CCS Project

A dialogue box is opened. First the user must configure the Target. There is a pull down menu at the upper right corner of the dialogue window. The target can be a generic processor, a device or a TI EVM.

Each target has a set of processors. To illustrate this, the following two screen shots show the dialogue window when a board called IDK_AM427X is selected and when the device 66AK2H12 is selected respectively;

New CCS Project

CCS Project

Project name must be specified

Target:

<select or type filter text>

IDK_AM437X

Connection:

Verify...

Cortex A [ARM]

Cortex M [ARM]

PRU

Project name:

☒ Use default location

Location:

C:\Users\A0270985.ENT\workspace_v6_try

Browse...

Compiler version:

GNU v4.9.3 (Linaro)

More...

Advanced settings

Project templates and examples

type filter text

Empty Projects

Empty Project

Empty Project (with main.c)

SYS/BIOS

GNU Target Examples

System Analyzer (UIA)

Creates an empty project fully initialized to run in "non-hosted" mode on the selected device. The project will contain an empty 'main.c' source-file.

?

< Back

Next >

Finish

Cancel

New CCS Project

CCS Project

Project name must be specified

Target: <select or type filter text> 66AK2H12

Connection: Verify...

Cortex A [ARM] C66XX [C6000]

Project name:

☒ Use default location

Location: Browse...

Compiler version: More...

▶ Advanced settings

▼ Project templates and examples

- Empty Projects
 - Empty Project
 - Empty Project (with main.c)
 - Empty Assembly-only Project
 - Empty RTSC Project
- Basic Examples
 - Hello World

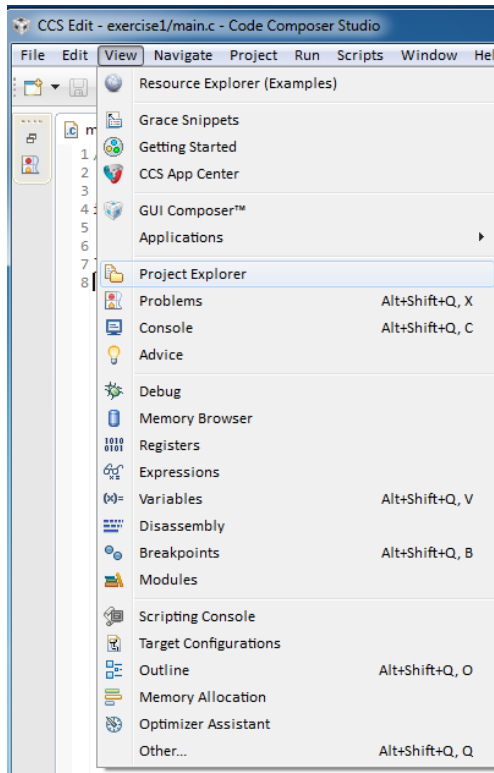
Creates an empty project fully initialized for the selected device. The project will contain an empty 'main.c' source-file.

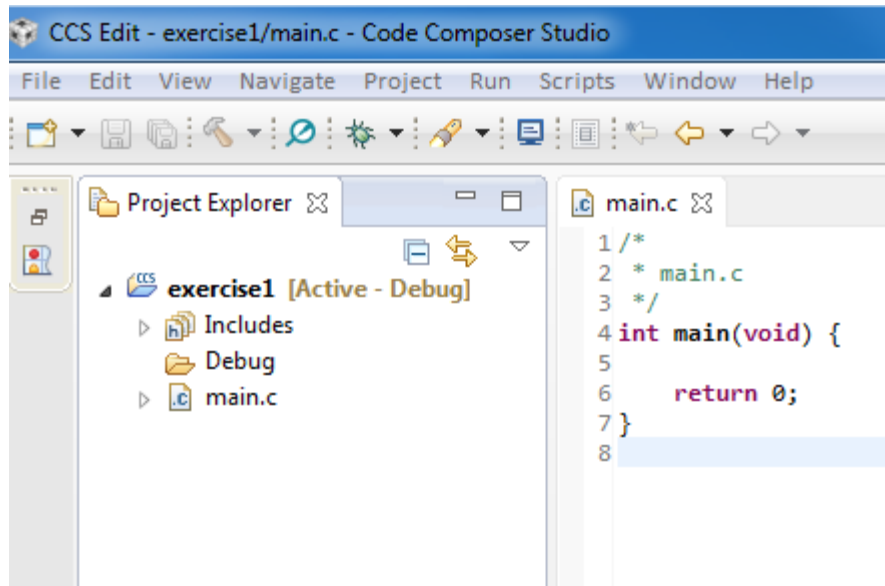
< Back Next > Finish Cancel

The user who selects IDK_ AM437X can choose one of three programmable processors, either ARM cortex A (AM437X has Cortex A9), ARM cortex M4 or PRU. 66AK2H12 has two processors to choose from, either ARM Cortex A (A15) or C66XX DSP. Each processor has its own list of default project's templates. All processors have several Empty Projects templates as well as Basic Example (Hello World) template.

To start the project we choose *Empty Project* with main.c. Next the project name should be chosen. After a Project name (for example “exercise1”) is written in the Project Name Tab, the Finish Tab at the bottom of the dialogue window is highlighted.

Left Click on the *Finish* TAB and the new project with main.c file is created. To open Project Explorer (if it was not opened in the past) the user left-clicks on the *View* tab, selects *Project Explorer* and then left-click. The following two windows show how to enable Project Explorer and the Project Explorer display. Clicking on the small arrow next to the Project Name opens the project structure:





Next we add to files to the project and modified the main.c file. The first file that is added is an include file called for example exercise1.h. The include file will have all the constants that are used in the code, as well as all the routines' prototypes and all standard include files. Since the project will use random number generation the C standard include file <stdlib.h> must be included. Since the project will use I/O functions like printf, the standard C I/O include file <stdio.h> must be included. The following is a Pseudo C code for the example1.h file:

```

/*
 * exercise1.h
 *
 * Created on: Aug 26, 2016
 * Author: a0270985
 */

#ifndef EXAMPLE1_H_
#define EXAMPLE1_H_

#include <stdlib.h>
#include <stdio.h>

#define DATA_SIZE 256
#define MAXIMUM_VALUE 1000

extern void generateFloatingPointInputData (float *p_out, int
numberOfElements);
extern double calculateEnergy (float *p_in, int numberOfElements) ;

```

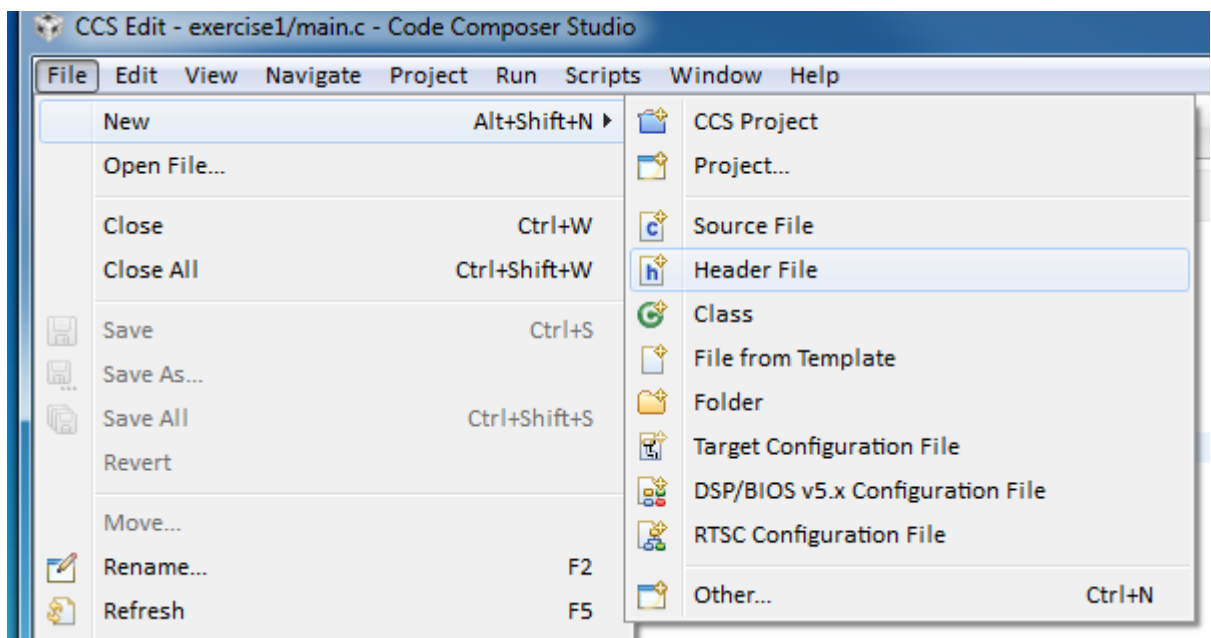


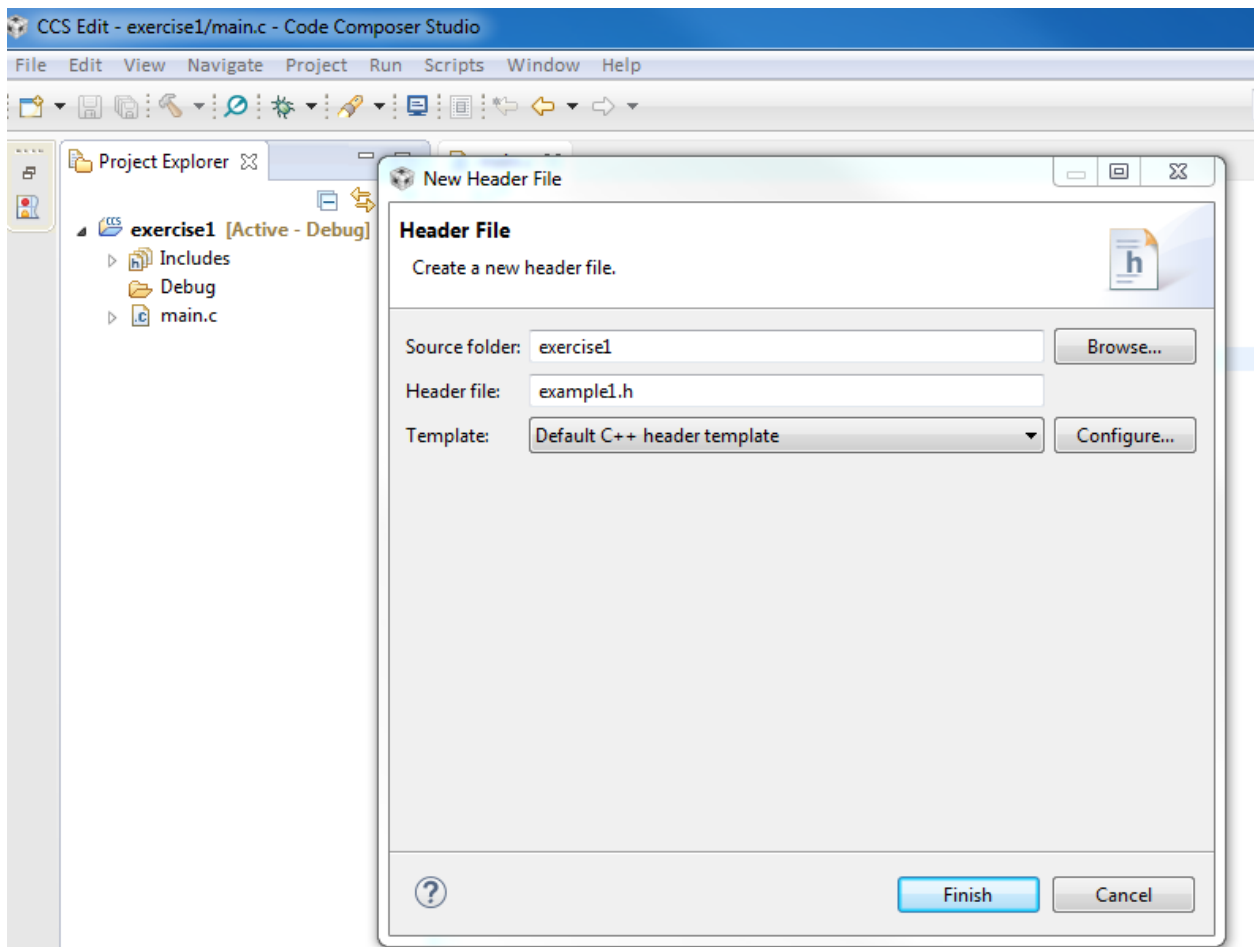
```
#endif /* EXAMPLE1_H_ */
```

Notice that the include file does not declare the FFT prototype. The FFT function is part of the DSPLIB library that is part of the release and the prototype is defined in a different include file that will be added later.

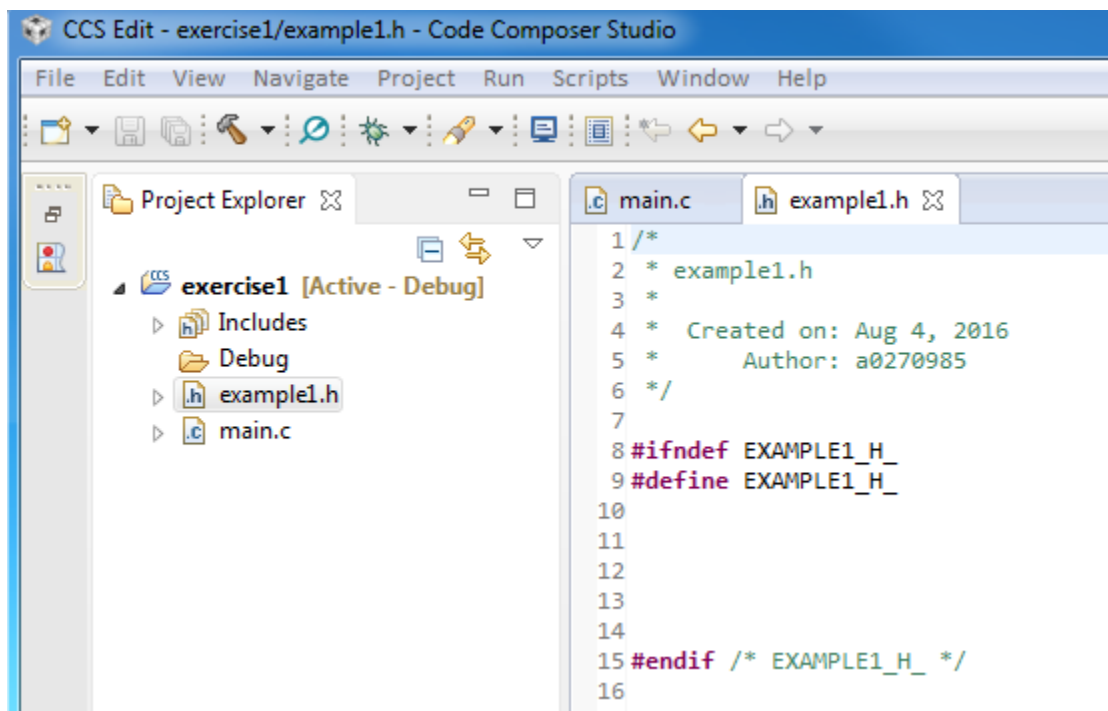
The include file as any other source file can be written using any text editor and then copied into the project, or it can be written within CCS. To use the later, one should left click on the *File* tab;

File->New->Header file and a dialogue window is opened. In the dialogue box one writes the include file name and click *Finish*. See the next two screen shots:

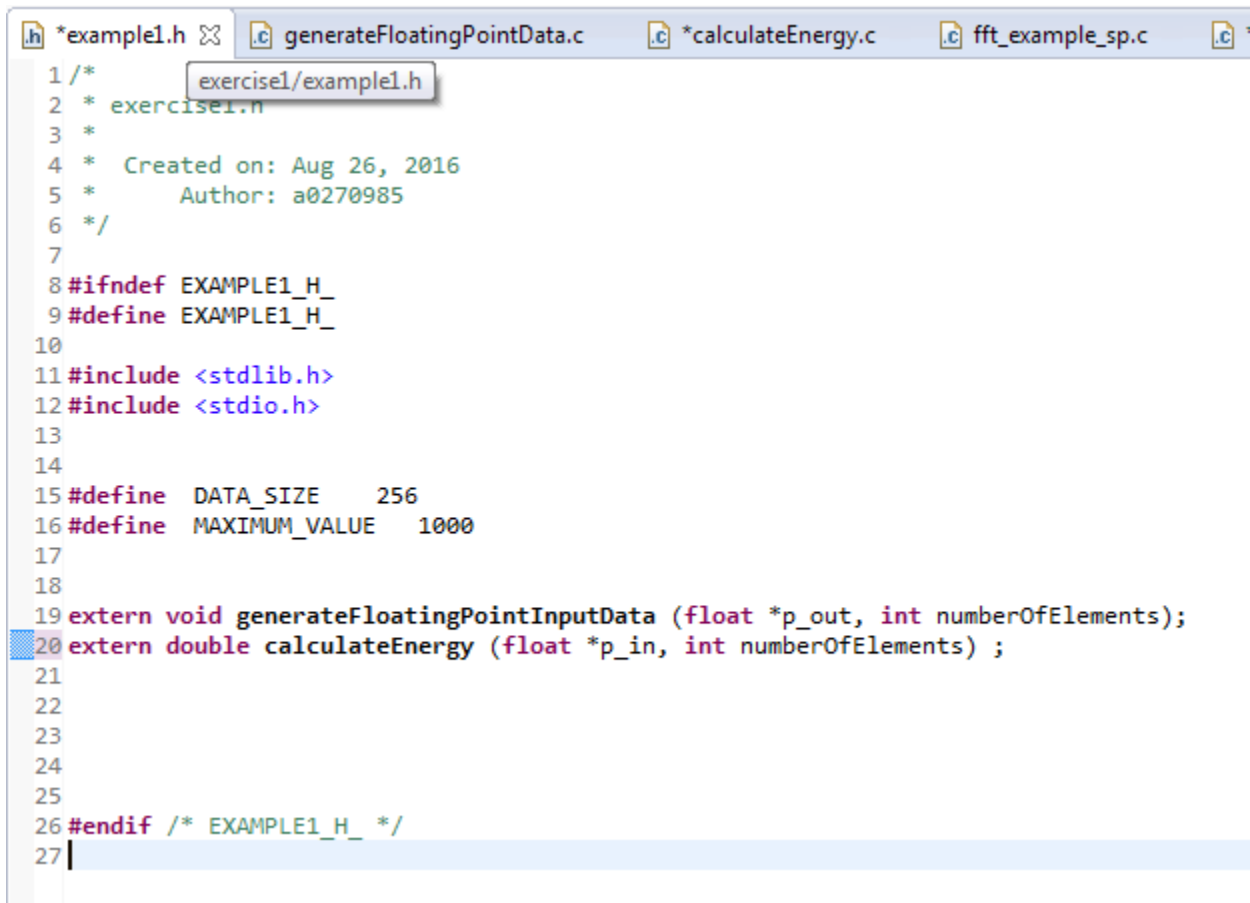




The include file is opened with the first two lines and the last line:



Adding the body of the example1.h from above, one can copy and paste from the pseudo code above to get the following include file:



```
1 /*  
2  * exercise1.h  
3  *  
4  * Created on: Aug 26, 2016  
5  * Author: a0270985  
6  */  
7  
8 #ifndef EXAMPLE1_H_  
9 #define EXAMPLE1_H_  
10  
11 #include <stdlib.h>  
12 #include <stdio.h>  
13  
14  
15 #define DATA_SIZE 256  
16 #define MAXIMUM_VALUE 1000  
17  
18  
19 extern void generateFloatingPointInputData (float *p_out, int numberOfElements);  
20 extern double calculateEnergy (float *p_in, int numberOfElements) ;  
21  
22  
23  
24  
25  
26 #endif /* EXAMPLE1_H_ */  
27
```

One advantage of CCSv6 is that the user can assign compilation parameters such as level of optimization and level of debug-ability for each project and for each source file in the project. This feature enables the user to compile the main program (say) without optimization and with full symbolic debugger feature, and compile processing code with high optimization and no symbolic debug, to make it easier to profile performances of each function and to optimize each function. In this project two source files will be added, one for generating floating point random number and one for calculating the energy. The prototype of these function is defined in the include file.

Adding a C source file is similar to adding an include file, namely, from the File Tab, New, Source File. After developing each file the user can compile each file separately by selecting the file name (left click on the source file name), right click and *Build Selected File(s)*. Multiple files can be selected using the Ctrl key. The following is two screen shots from the two files; generateFloatingPointData and calculateEnergy.c after compilation of each file. The compilation message is at the Console window usually in the bottom of the CCS window. Pseudo code for the two files are given below, so the user can copy and paste:

```

#include "example1.h"

#define HALF_MAXIMUM_VALUE (MAXIMUM_VALUE /2)

void generateFloatingPointInputData (float *p_out, int numberOfElements)
{
    int r1;
    float x1;
    int i;
    for (i=0; i<numberOfElements;i++)
    {
        r1 = rand() % MAXIMUM_VALUE ;
        x1 = (float) (r1 - HALF_MAXIMUM_VALUE) ;
        *p_out++ = x1 ;
    }
}

/*
 * calculateEnergy.c
 *
 * Created on: Aug 26, 2016
 * Author: a0270985
 */

#include "example1.h"

double calculateEnergy (float *p_in, int numberOfElements)
{
    double sum ;
    int i ;
    float x,y ;
    sum = 0.0 ;
    for (i=0; i<numberOfElements;i++)
    {
        x = *p_in++ ;

        sum = sum + (double) (x*x) ;
    }
    return (sum) ;
}

```

Project Explorer

- exercisel [Active - Debug]
 - Includes
 - Debug
 - calculateEnergy.c
 - example1.h
 - generateFloatingPointData.c
 - main.c

main.c

```
1 /*
2  * generateFloatingPointData.c
3  *
4  * Created on: Aug 24, 2016
5  * Author: a0270985
6  *
7  * Note - The floating point number that are generated are between
8  * -HALF_MAXIMUM_VALUE to +HALF_MAXIMUM_VALUE
9  */
10
11
12 #include "example1.h"
13
14 #define HALF_MAXIMUM_VALUE (MAXIMUM_VALUE / 2)
15
16 void generateFloatingPointInputData (float *p_out, int numberOfElements)
17 {
18     int r1 ;
19     float x1 ;
20     int i ;
21     for (i=0; i<numberOfElements;i++)
22     {
23         r1 = rand() % MAXIMUM_VALUE ;
24         x1 = (float) (r1 - HALF_MAXIMUM_VALUE) ;
25         *p_out++ = x1 ;
26     }
27 }
28
29
30
```

Console

CDT Build Console [exercisel]

```
**** Build of configuration Debug for project exercisel ****

"C:\ti\ccsv6_1_3\ccsv6\utils\bin\gmake" -k generateFloatingPointData.obj
'Building file: ../generateFloatingPointData.c'
'Invoking: C6000 Compiler'
"C:\ti\ccsv6_1_3\ccsv6\tools\compiler\ti-cgt-c6000_8.1.0/bin/cl6x"
--include_path="C:\ti\ccsv6_1_3\ccsv6\tools\compiler\ti-cgt-c6000_8.1.0/include" -g --diag_warning=225 --diag_wrap=off
--display_error_number --preproc_with_compile --preproc_dependency="generateFloatingPointData.d"
"../generateFloatingPointData.c"
'Finished building: ../generateFloatingPointData.c'
'
'

**** Build Finished ****
```

```
calculateEnergy.c
1 /*
2  * calculateEnergy.c
3  *
4  * Created on: Aug 26, 2016
5  * Author: a0270985
6  */
7
8
9 #include "example1.h"
10
11 double calculateEnergy (float *p_in, int numberOfElements)
12 {
13     double sum ;
14     int i ;
15     float x,y;
16     sum = 0.0 ;
17     for (i=0; i<numberOfElements;i++)
18     {
19         x = *p_in++ ;
20
21         sum = sum + (double) (x*x) ;
22     }
23     return (sum) ;
24 }
25
26
27
```

Console

CDT Build Console [exercise1]

```
**** Build of configuration Debug for project exercise1 ****
"C:\ti\ccsv6_1_3\ccsv6\utils\bin\gmake" -k calculateEnergy.obj
'Building file: ../calculateEnergy.c'
'Invoking: C6000 Compiler'
"C:/ti/ccsv6_1_3/ccsv6/tools/compiler/ti-cgt-c6000_8.1.0/bin/cl6x"
--include_path="C:/ti/ccsv6_1_3/ccsv6/tools/compiler/ti-cgt-c6000_8.1.0/include" -g --diag_warning=225 --diag_wrap=off
--display_error_number --preproc_with_compile --preproc_dependency="calculateEnergy.d" "../calculateEnergy.c"
"../calculateEnergy.c", line 15: warning #179-D: variable "y" was declared but never referenced
'Finished building: ../calculateEnergy.c'
**** Build Finished ****
```

The main function should create the input data (using the generateFloatingPointData function), then it calculates the energy in the input data, performs FFT and calculates the energy of the transformed frequency domain data. The FFT function is part of the DSPLIB optimized TI library that is part of the release. In this document we use Processor SDK RTOS release 3.0.0.4 with dsplib_c66x_3_4_0_0. The library contains multiple FFT functions. For this project the single precision floating point DSPF_sp_fftSPxSP is chosen.

The sub-directory /Release_3_0_0_4\C667X\dsplib_c66x_3_4_0_0\packages\ti\dsplib\lib have four versions of the dsplib optimized library and four versions of not-optimized version. Library dsplib.a66 is little endian COFF format, dsplib.a66e is big endian COFF format. COFF format is a TI proprietary format that was used for backward compatibility with older projects. The library dsplib.ae66 is the ELF version of little endian format while dsplinae66e is the big endian version.

The ELF format is a standard format that is currently used. For the purpose of this project the little endian ELF format is used, that is dsplib.ae66 library

The include file dsplib.h in directory

/Release_3_0_0_4\C667X\dsplib_c66x_3_4_0_0\packages\ti\dsplib includes all dsplib include functions. This file will be included in the project.

The documentations how to use the library function is in directory

\Release_3_0_0_4\C667X\dsplib_c66x_3_4_0_0\packages\ti\dsplib\docs\doxygen in a chm format, as well as the [TMS320C67x DSP Library Programmer's Reference Guide](#) –page 49. The following is a screen shots that shows how to use the function DSPF_sp_fftSPxSP

The screenshot shows the DSPLIB API Reference web page. The left sidebar contains a tree view of the API reference, with 'Fast Fourier Transform' expanded. The main content area displays the 'DSPF_sp_fftSPxSP' function documentation. The page header includes the Texas Instruments logo and navigation tabs for 'Main Page', 'Modules', and 'Files'. The function signature is shown as: `void DSPF_sp_fftSPxSP (int N, float *ptr_x, float *ptr_w, float *ptr_y, unsigned char *brev, int n_min, int offset, int n_max)`. The 'Parameters' section lists the arguments: `N` (length of FFT in complex samples), `ptr_x` (pointer to complex data input), `ptr_w` (pointer to complex twiddle factor), `ptr_y` (pointer to complex output data), `brev` (pointer to bit reverse table containing 64 entries), `n_min` (should be 4 if N can be represented as Power of 4 else, n_min should be 2), `offset` (index in complex samples of sub-fft from start of main fft), and `n_max` (size of main fft in complex samples).

Even with the documentations it may not be clear how to use the function. To understand better how to use the function one can look at the unit test. The unit test main function is called

DSPF_sp_fftSPxSP_d.c and is located in directory
\\Release_3_0_0_4\C667X\dsplib_c66x_3_4_0_0\packages\ti\dsplib\src\DSPF_sp_fftSPxSP\c66.
Even though the previous chapter test program was built for a different device, the way to use
the library routines is the same.

From the imported project of the previous chapter we know that the FFT routine needs two
other vectors in addition to the input, the 64-elements bit reversal vector (brev) and the twiddle
factor. The DSPLIB has several twiddle factor generation functions, but they all for fixed point
arithmetic and not for floating point. Thus this project will use the same Twiddle Factor
generation that the developer in the imported project used.

In addition just like the imported test project from the previous Chapter, we add two include
files DSPF_sp_fftSPxSP.h for the function that the code uses , and math.h. The main source code
looks like the following:

```
#include "example1.h"
#include <math.h>
#include "DSPF_sp_fftSPxSP.h"

#pragma DATA_ALIGN(inputVector, 8);
float inputVector[ 2*DATA_SIZE] ; //    complex vector
#pragma DATA_ALIGN(outputVector, 8);
float outputVector[2* DATA_SIZE] ;
#pragma DATA_ALIGN(twiddleFactors, 8);
float twiddleFactors[2* DATA_SIZE] ;

void gen_twiddle_fft_sp (float *w, int n)
{
    int i, j, k;
    double x_t, y_t, theta1, theta2, theta3;
    const double PI = 3.141592654;

    for (j = 1, k = 0; j <= n >> 2; j = j << 2)
    {
        for (i = 0; i < n >> 2; i += j)
        {
            theta1 = 2 * PI * i / n;
            x_t = cos (theta1);
            y_t = sin (theta1);
            w[k] = (float) x_t;
            w[k + 1] = (float) y_t;

            theta2 = 4 * PI * i / n;
            x_t = cos (theta2);
            y_t = sin (theta2);
            w[k + 2] = (float) x_t;
            w[k + 3] = (float) y_t;

            theta3 = 6 * PI * i / n;
            x_t = cos (theta3);
```



```

        y_t = sin (theta3);
        w[k + 4] = (float) x_t;
        w[k + 5] = (float) y_t;
        k += 6;
    }
}

unsigned char brev[64] = {
    0x0, 0x20, 0x10, 0x30, 0x8, 0x28, 0x18, 0x38,
    0x4, 0x24, 0x14, 0x34, 0xc, 0x2c, 0x1c, 0x3c,
    0x2, 0x22, 0x12, 0x32, 0xa, 0x2a, 0x1a, 0x3a,
    0x6, 0x26, 0x16, 0x36, 0xe, 0x2e, 0x1e, 0x3e,
    0x1, 0x21, 0x11, 0x31, 0x9, 0x29, 0x19, 0x39,
    0x5, 0x25, 0x15, 0x35, 0xd, 0x2d, 0x1d, 0x3d,
    0x3, 0x23, 0x13, 0x33, 0xb, 0x2b, 0x1b, 0x3b,
    0x7, 0x27, 0x17, 0x37, 0xf, 0x2f, 0x1f, 0x3f
};

int main(void)
{
    double sumInput, sumOutput ;
    int i,j ;

    generateFloatingPointInputData (inputVector, 2*DATA_SIZE);
    sumInput = calculateEnergy (inputVector, 2* DATA_SIZE) ;

    gen_twiddle_fft_sp (twiddleFactors, DATA_SIZE) ;

    DSPF_sp_fftSPxSP(DATA_SIZE, inputVector, twiddleFactors, outputVector,
brev, 4, 0, DATA_SIZE);
    sumOutput = calculateEnergy (outputVector, 2* DATA_SIZE) ;

    printf(" input energy %e  output energy  %e  difference  %e  \n",
sumInput, sumOutput, sumInput-sumOutput) ;

    return 0;
}

```

Note: The include file in the imported project was ti\dsplib\dsplib.h. This is a generic include file that includes ALL the include files in the DSPLIB release. This include file is generic, so for functions that were optimized for a certain architecture, the user must provide the device name. For the C66 architecture, the device name is _TMS320C6600. Adding device name to a project is done from properties-> Advanced Options ->Predefined Symbols lower window (Pre-define NAME)

3.2. Building the New Project

Right click on the project name and select *Rebuild Project*. After the build the following error message is displayed in the Console window:

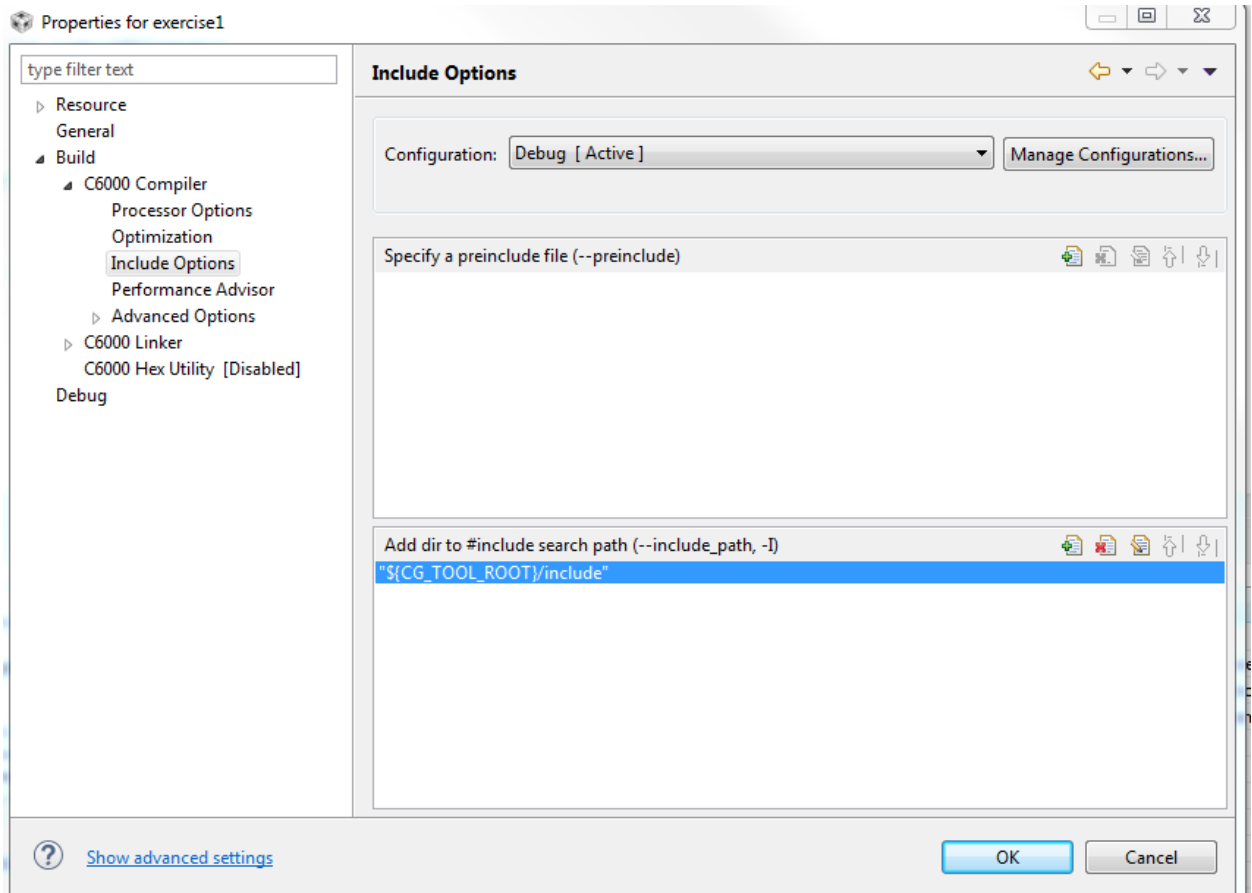
```
"C:\\ti\\ccs_v6_1_3\\ccsv6\\utils\\bin\\gmake" -k all
'Building file: ../calculateEnergy.c'
'Invoking: C6000 Compiler'
"C:/ti/ccs_v6_1_3/ccsv6/tools/compiler/ti-cgt-c6000_8.1.0/bin/cl6x"
--include_path="C:/ti/ccs_v6_1_3/ccsv6/tools/compiler/ti-cgt-c6000_8.1.0/include"
--include_path="C:/ti/Releases/Release_3_0_0_4/K2H/dsplib_c66x_3_4_0_0/packages" -g --define=_TMS320C6600 --diag_warning=225
--diag_wrap=off --display_error_number --preproc_with_compile --preproc_dependency="calculateEnergy.d"  "../calculateEnergy.c"
"../calculateEnergy.c", line 15: warning #179-D: variable "y" was declared but never referenced
'Finished building: ../calculateEnergy.c'
'
'
'Building file: ../generateFloatingPointData.c'
'Invoking: C6000 Compiler'
"C:/ti/ccs_v6_1_3/ccsv6/tools/compiler/ti-cgt-c6000_8.1.0/bin/cl6x"
--include_path="C:/ti/ccs_v6_1_3/ccsv6/tools/compiler/ti-cgt-c6000_8.1.0/include"
--include_path="C:/ti/Releases/Release_3_0_0_4/K2H/dsplib_c66x_3_4_0_0/packages" -g --define=_TMS320C6600 --diag_warning=225
--diag_wrap=off --display_error_number --preproc_with_compile --preproc_dependency="generateFloatingPointData.d"
"../generateFloatingPointData.c"
'Finished building: ../generateFloatingPointData.c'
'
'
'Building file: ../main.c'
'Invoking: C6000 Compiler'
"C:/ti/ccs_v6_1_3/ccsv6/tools/compiler/ti-cgt-c6000_8.1.0/bin/cl6x"
--include_path="C:/ti/ccs_v6_1_3/ccsv6/tools/compiler/ti-cgt-c6000_8.1.0/include"
--include_path="C:/ti/Releases/Release_3_0_0_4/K2H/dsplib_c66x_3_4_0_0/packages" -g --define=_TMS320C6600 --diag_warning=225
--diag_wrap=off --display_error_number --preproc_with_compile --preproc_dependency="main.d"  "../main.c"

>> Compilation failure
subdir_rules.mk:21: recipe for target 'main.obj' failed
"../main.c", line 7: fatal error #1965: cannot open source file "DSPF_sp_fftSPxSP.h"
1 catastrophic error detected in the compilation of " ../main.c".
Compilation terminated.
gmake: *** [main.obj] Error 1
gmake: Target 'all' not remade because of errors.

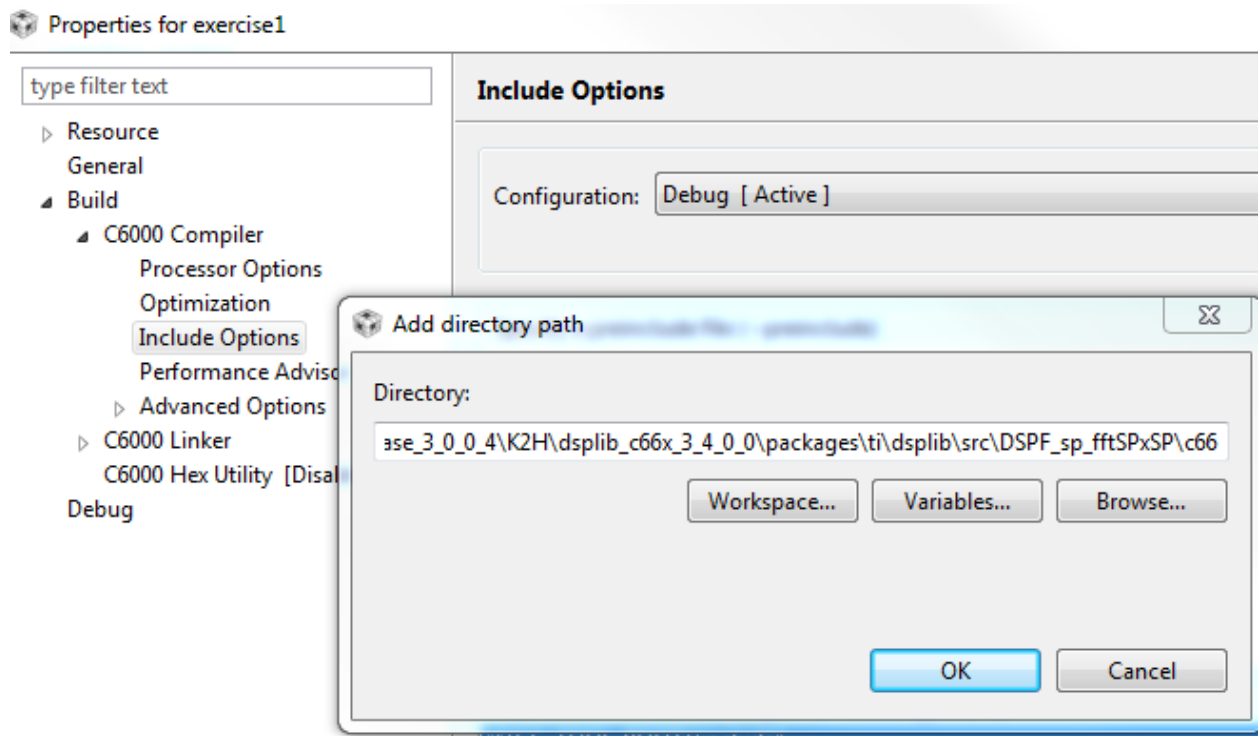
**** Build Finished ****
```

The project does not find the include file DSPF_sp_fftSPxSP.h. We need to add the path to it. Searching in the release the file DSPF_sp_fftSPxSP.h is in directory

INSTALL_DIR\dsplib_c66x_3_4_0_0\packages\ti\dsplib\src\DSPF_sp_fftSPxSP\c66 where INSTALL_DIR is the directory name where the user installed Processor SDK. Adding the path to ti\dsplib is done from the properties windows. Right click on the Project name and select Properties (this is the last item in the list). In the properties window select Include Options .



The upper window enables the user to add a pre-include file. The lower window is used to add a path. Click on the green plus sign (+) and add the path to the ti\dsplib\dsplib.h. In the system that is used here it looks like the following:



Click **OK** twice and try to rebuild the project again. This time compilation went through, but there are few issues with linking the program, see the screen shot below:

```

"./generateFloatingPointData.obj" "./main.obj" -llibc.a
<Linking>
warning #10247-D: creating output section ".text" without a SECTIONS specification
warning #10247-D: creating output section ".const" without a SECTIONS specification
warning #10247-D: creating output section ".fardata" without a SECTIONS specification
warning #10247-D: creating output section ".cinit" without a SECTIONS specification
warning #10247-D: creating output section ".stack" without a SECTIONS specification

warning #10247-D: creating output section ".sysmem" without a SECTIONS specification
>> Compilation failure
makefile:141: recipe for target 'exercise1.out' failed
warning #10247-D: creating output section ".far" without a SECTIONS specification
warning #10247-D: creating output section ".switch" without a SECTIONS specification
warning #10247-D: creating output section ".cio" without a SECTIONS specification
warning #10210-D: creating ".stack" section with default size of 0x400; use the -stack option to change the default size
warning #10210-D: creating ".sysmem" section with default size of 0x400; use the -heap option to change the default size

undefined      first referenced
symbol          in file
-----
DSPF_sp_fftSPxSP ./main.obj

error #10234-D: unresolved symbols remain
error #10010: errors encountered during linking; "exercise1.out" not built
gmake: *** [exercise1.out] Error 1
gmake: Target 'all' not remade because of errors.

**** Build Finished ****

```

The error tells us that the library function DSPF_sp_fftSPxSP that is called by main was not found, but in addition it gives us warning that there are no section specifications. Indeed the project does not have a linker command file that defines what memories are used and what sections are used. As a starting point we will copy the linker command file from the imported project into the new project. Later on the user can modified the linker command file for the “real” application. For example, the linker command file that was used in the imported project does not include the external memory DDR which usually is used in real applications. The linker command file of the imported projects lnk.cmd is the following:

```
-heap 0x8000
-stack 0xC000
-l../../../../../packages/ti/dsplib/lib/dsplib.lib

MEMORY
{
    L2SRAM (RWX) : org = 0x800000, len = 0x100000
    MSMCSRAM (RWX) : org = 0xc000000, len = 0x200000
}

SECTIONS
{
    .text: load >> L2SRAM
    .text:touch: load >> L2SRAM

    GROUP (NEAR_DP)
    {
        .neardata
        .rodata
        .bss
    } load > L2SRAM

    .far: load >> L2SRAM
    .fardata: load >> L2SRAM
    .data: load >> L2SRAM
    .switch: load >> L2SRAM
    .stack: load > L2SRAM
    .args: load > L2SRAM align = 0x4, fill = 0 {_argsize = 0x200; }
    .systemem: load > L2SRAM
    .cinit: load > L2SRAM
    .const: load > L2SRAM START(const_start) SIZE(const_size)
    .pinit: load > L2SRAM
    .cio: load >> L2SRAM
    xdc.meta: load >> L2SRAM, type = COPY
}
```

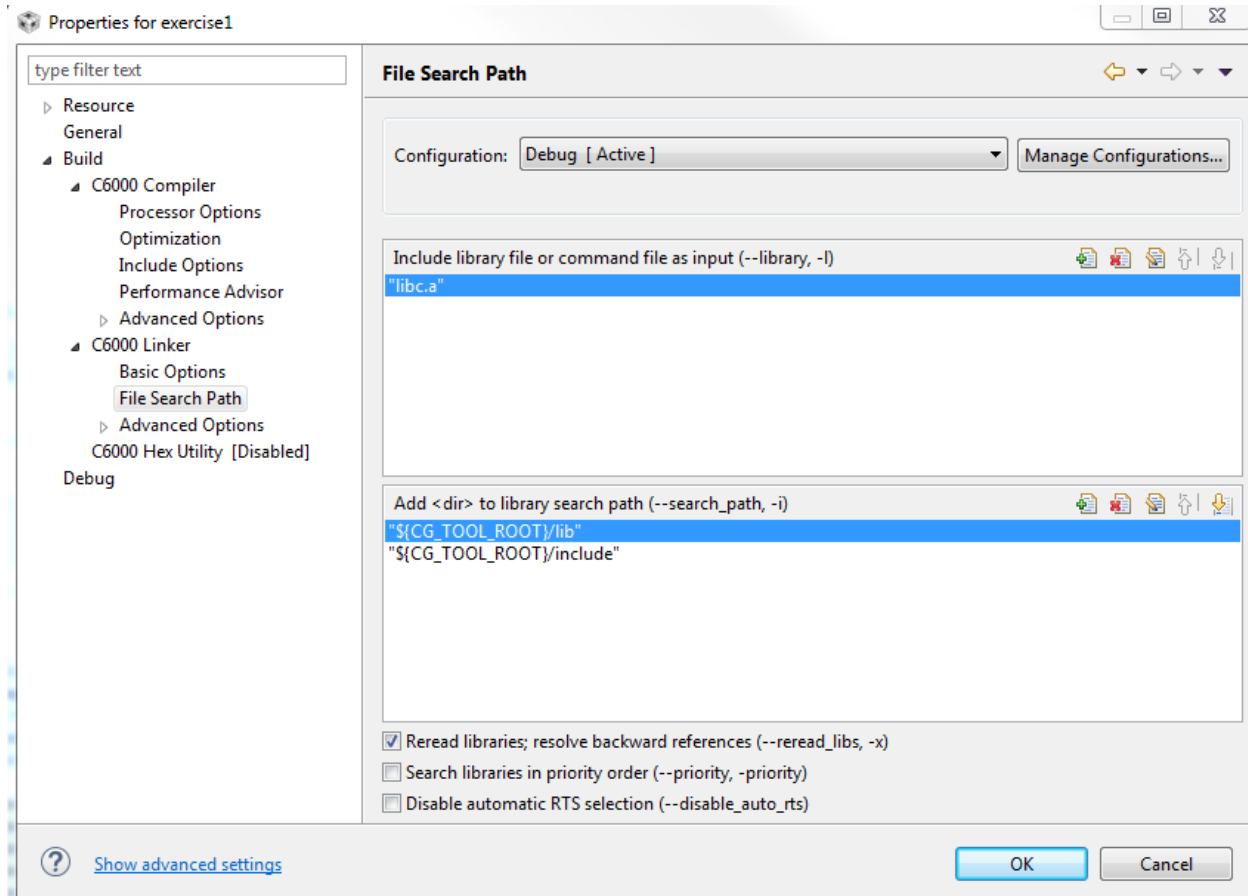
Next we add the DSPF_sp_fftSPxSP function library. A complete set of the entire DSPLIB libraries are in directory INSTALL_DIRECTORY\ dsplib_c66x_3_4_0_0\packages\ti\dsplib\lib. In addition, each DSPLIB function has its own small library. This is the library that is going to be used in this project.

From the comment of type of libraries in chapter 2.2, and building this project as little endian and ELF format, the library that is used is dsplib.ae66 in directory

INSTALL_DIRECTORY\ \dsplib_c66x_3_4_0_0\packages\ti\dsplib\lib

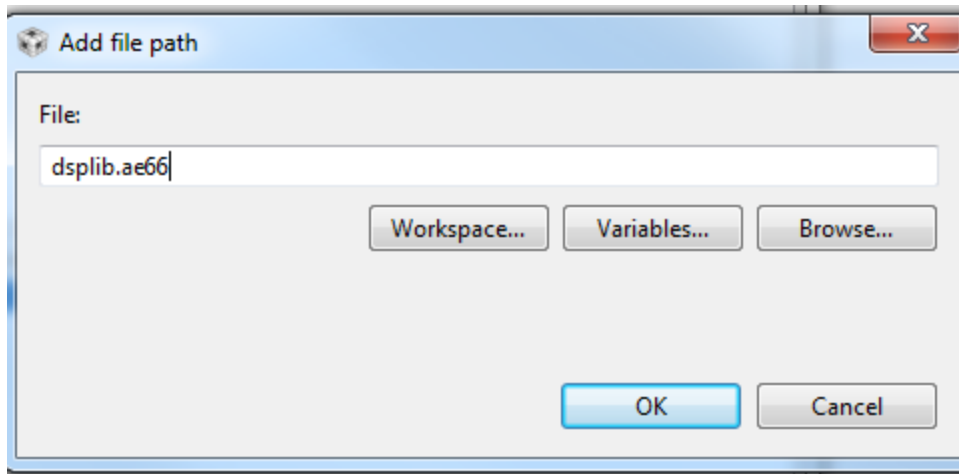
To add the library and a path to the library to the project the user must go to

Properties->C6000 Linker ->File Search Path, see the screen shot:

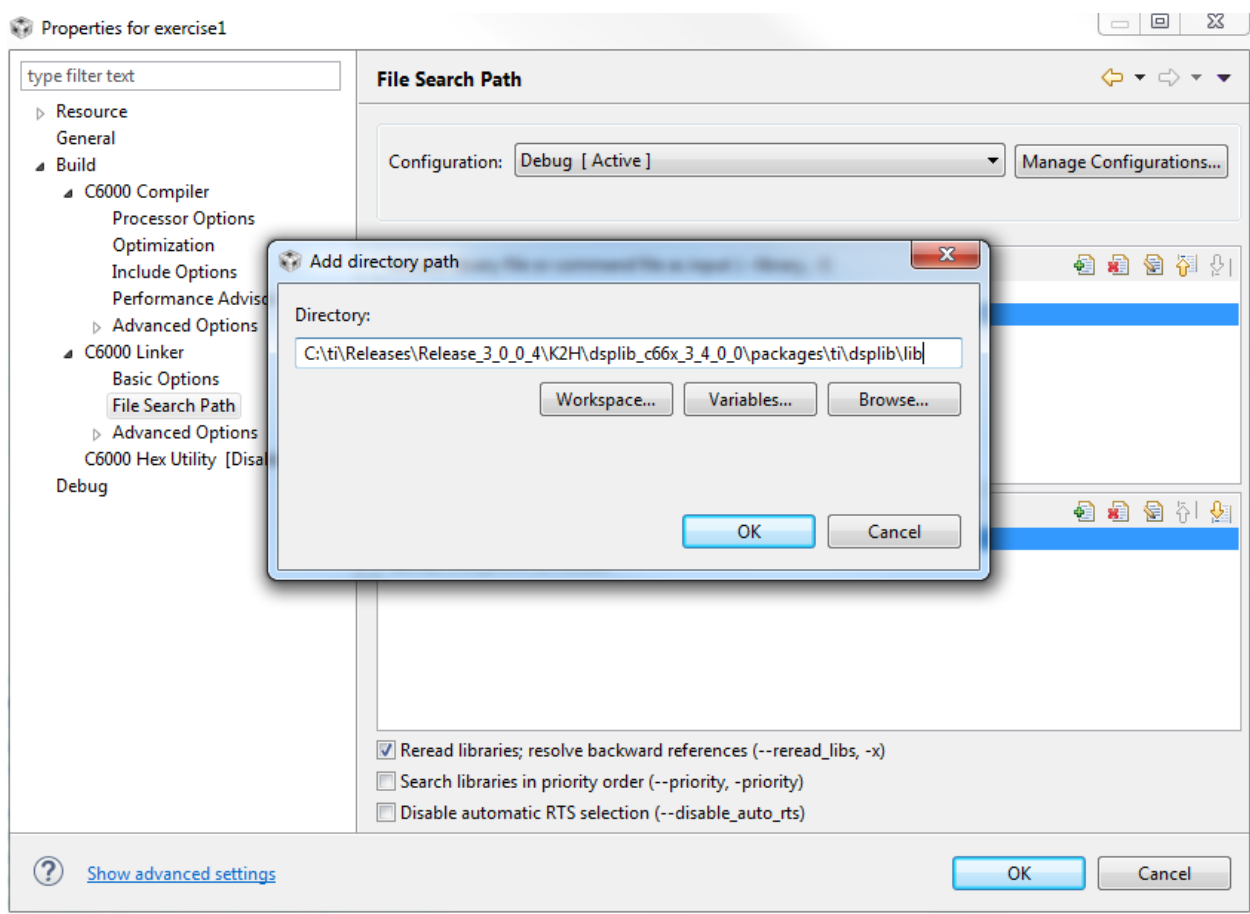


The upper window should have the library name, while the lower window has the path to the library.

Select the green + sign at the top window opens a dialogue box where we enter the library name:



And in the lower window we add the path to the library:



Click *OK* three times, one for the library, one for the Path and one for the Properties and rebuild the project. This time the project was built and the Console shows the following:

```

Console
CDT Build Console [exercise1]
--include_path="C:/ti/Releases/Release_3_0_0_4/K2H/dsplib_c66x_3_4_0_0/packages"
--include_path="C:/ti/Releases/Release_3_0_0_4/K2H/dsplib_c66x_3_4_0_0/packages/ti/dsplib/src/DSPF_sp_fftSPxSP/c66" -g
--define=_TMS320C6600 --diag_warning=225 --diag_wrap=off --display_error_number --preproc_with_compile
--preproc_dependency="generateFloatingPointData.d" "../generateFloatingPointData.c"
'Finished building: ../generateFloatingPointData.c'
'
'Building file: ../main.c'
'Invoking: C6000 Compiler'
"C:/ti/ccsv6_1_3/ccsv6/tools/compiler/ti-cgt-c6000_8.1.0/bin/cl6x"
--include_path="C:/ti/ccsv6_1_3/ccsv6/tools/compiler/ti-cgt-c6000_8.1.0/include"
--include_path="C:/ti/Releases/Release_3_0_0_4/K2H/dsplib_c66x_3_4_0_0/packages"
--include_path="C:/ti/Releases/Release_3_0_0_4/K2H/dsplib_c66x_3_4_0_0/packages/ti/dsplib/src/DSPF_sp_fftSPxSP/c66" -g
--define=_TMS320C6600 --diag_warning=225 --diag_wrap=off --display_error_number --preproc_with_compile
--preproc_dependency="main.d" "../main.c"
"../main.c", line 66: warning #179-D: variable "i" was declared but never referenced
"../main.c", line 66: warning #179-D: variable "j" was declared but never referenced
'Finished building: ../main.c'
'
'Building target: exercise1.out'
'Invoking: C6000 Linker'
"C:/ti/ccsv6_1_3/ccsv6/tools/compiler/ti-cgt-c6000_8.1.0/bin/cl6x" -g --define=_TMS320C6600 --diag_warning=225 --diag_wrap=off
--display_error_number -z -m"exercise1.map" -i"C:/ti/ccsv6_1_3/ccsv6/tools/compiler/ti-cgt-c6000_8.1.0/lib"
-i"C:/ti/Releases/Release_3_0_0_4/K2H/dsplib_c66x_3_4_0_0/packages/ti/dsplib/lib"
-i"C:/ti/ccsv6_1_3/ccsv6/tools/compiler/ti-cgt-c6000_8.1.0/include" --reread_libs --diag_wrap=off --display_error_number
--warn_sections --xml_link_info="exercise1_linkInfo.xml" --rom_model -o "exercise1.out" "../calculateEnergy.obj"
"../generateFloatingPointData.obj" "../main.obj"
"C:/ti/Releases/Release_3_0_0_4/K2H/dsplib_c66x_3_4_0_0/examples/fft_sp_ex/lnk.cmd" -llibc.a -ldsplib.ae66
<Linking>
warning #10349-D: creating output section ".init_array" without a SECTIONS specification. For additional information on this
section, please see the 'C6000 EABI Migration' guide at
http://processors.wiki.ti.com/index.php/C6000_EABI:C6000_EABI_Migration#C6x_EABI_Sections
'Finished building target: exercise1.out'
'
**** Build Finished ****

```

There are some warnings that the user can easily eliminate, but the executable exercise1.out was built and is in the Debug directory.

3.3. Code Execution understanding the results

Repeat the steps in Chapter 2.5 to launch the target, connect core 0 and load the code of exercise1 (Run->load_> Load Program and from the dialogue box choose *Browse Project* and then choose *exercise1->Debug->Exercise1.out* and then OK and OK)

Step through the code. The last instruction prints the following on the Console:

```
input energy 4.216463e+07 output energy 1.079414e+10 difference -1.075198e+10
```

So the input energy is not equal to the output energy. I leave it to the user to understand what my mistake is, I will just give a hint. If the following two lines are added to the code:

```
sumInput = sumInput * (float) DATA_SIZE ;
printf(" input energy %e output energy %e difference %e \n", sumInput,
sumOutput, sumInput-sumOutput) ;
```

Then the second printf gives the following results, (error of about e-8)

```
input energy 1.079415e+10 output energy 1.079414e+10 difference 4.998233e+02
```


4. Import Function from Library that is not Part of Processor SDK

4.1. Import an Example from FFTLIB (C674X version)

In chapter 2 there are instructions how to import a project. In Chapter 3 there are instructions how to build a new project. In both cases the build process was relatively easy and simple.

Processor SDK supports many TI digital devices and covers many building blocks. However, there are some devices that are not supported (currently) by Processor SDK. There are TI's Libraries that are not part of Processor SDK.

Importing and building examples in a library that is not part of Processor SDK requires more configurations since the example project is un-aware of the software environment.

In addition, in the previous examples the projects were not RTSC projects. RTSC requires some additional considerations. Chapter 2.1 – Before Importing a Project describes how to verify that RTSC system sees all the software modules that it may require.

Chapter 4 imports a project that has some build issues, and shows how to debug and fix these issues. The techniques that are demonstrated here can be used for other projects with similar issues.

The software tools that are used are CCS V6.1.3, and the library that is used is a FFTLIB for floating point devices. The library can be loaded from





























http://software-dl.ti.com/libs/fftlib/2.0.0/2_0_0_2/index_FDS.html

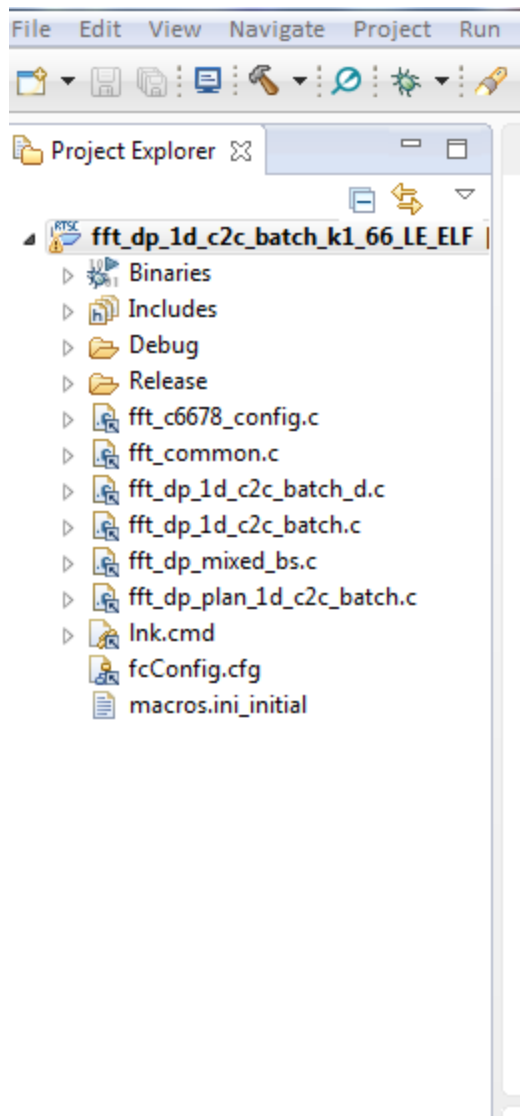
This library supports C674X devices which are not supported by Processor SDK but by set of tools for C674X. Download page for the C674X set of software tools is in http://software-dl.ti.com/dsps/dsps_public_sw/c6000/web/bios_c6sdk/latest/index_FDS.html

Following the process from 2.2 in directory `INSTALL_DIRECTORY\fftlib_2_0_0_2\packages\ti\fftlib\src` where `INSTALL_DIRECTORY` is the directory where FFTLIB was installed. We choose the second function in the following list, `fft_dp_1d_c2c_batch`. This function calculates FFT of double precision values (and the calculation is double precision) and one dimension complex FFT on multiple vectors (thus the batch). See the location of the example in the following screen shut:

Browse For Folder

Select root directory of the projects to import

- ▲  fftlib_2_0_0_2
 - ▷  components
 - ▷  docs
 -  eclipse
 -  package
 - ▲  packages
 - ▲  ti
 - ▲  fftlib
 - ▷  docs
 - ▷  package
 - ▲  src
 - ▷  common
 - ▷  fft_dp_1d_c2c
 - ▷  fft_dp_1d_c2c_batch
 - ▷  fft_dp_1d_r2c
 - ▷  fft_dp_1d_r2c_batch
 - ▷  fft_dp_2d_c2c
 - ▷  fft_dp_2d_r2c
 - ▷  fft_dp_3d_c2c
 - ▷  fft_dp_3d_r2c
 - ▷  fft_omp_dp_1d_c2c
 - ▷  fft_omp_dp_1d_r2c
 - ▷  fft_omp_dp_2d_c2c
 - ▷  fft_omp_dp_2d_r2c
 - ▷  fft_omp_dp_3d_c2c
 - ▷  fft_omp_dp_3d_r2c
 - ▷  fft_omp_sp_1d_c2c
 - ▷  ...

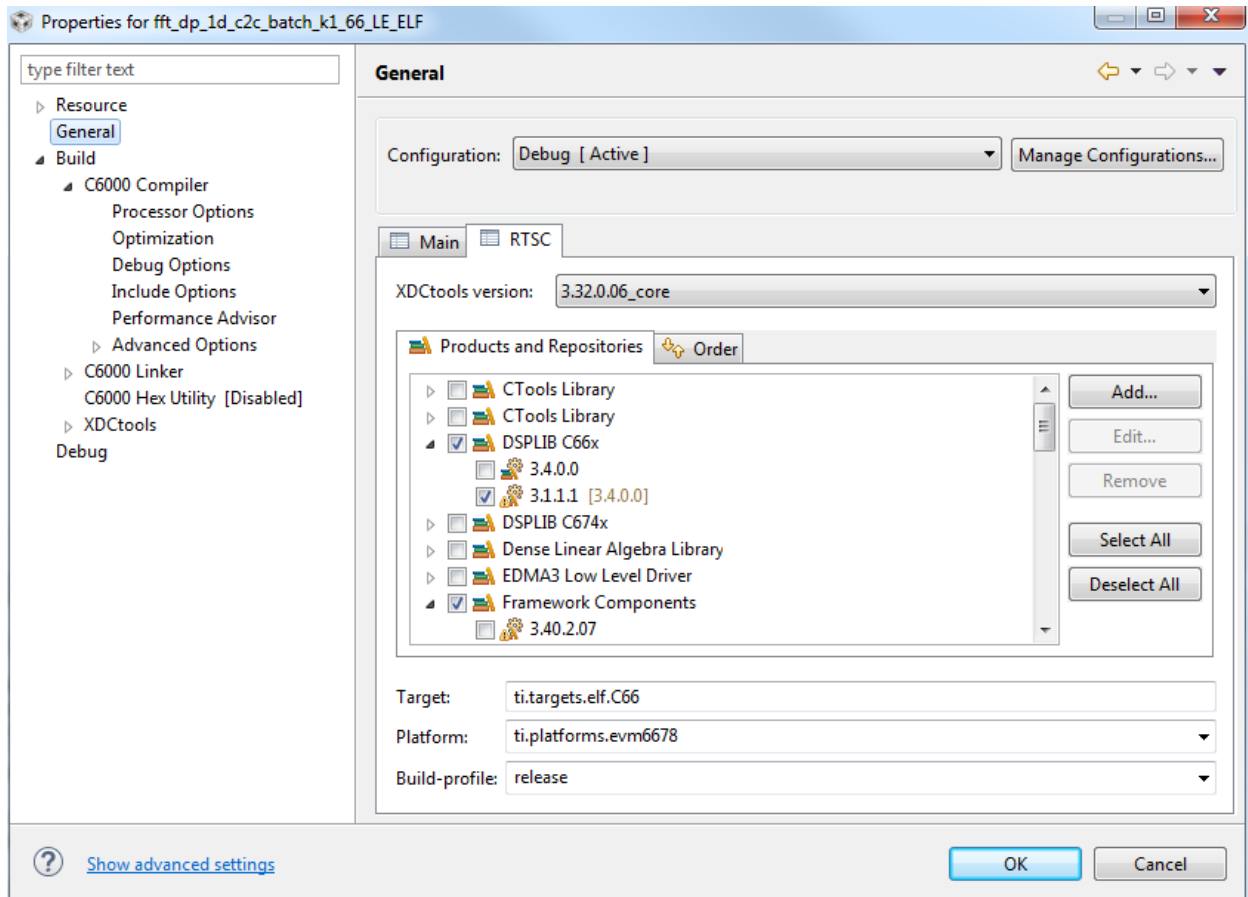


First notice the small exclamation mark next to the project name. This means that the project needs adjust its properties before it can be built. And indeed, if I click on Rebuild Project, this is the result:

```
'Building file: C:/ti/libraries/fftlb_2_0_2/packages/ti/fftlb/src/fft_dp_1d_c2c_batch/fft_dp_plan_1d_c2c_batch.c'
'Invoking: C6000 Compiler'
"C:/ti/ccsv6_1_3/ccsv6/tools/compiler/c6000_7.4.16/bin/cl6x" -mv6600 --abi=eabi -g
--include_path="C:/ti/ccsv6_1_3/ccsv6/tools/compiler/c6000_7.4.16/include" --include_path="../../../../../../../../"
--include_path="../../../../common" --include_path="../../../../" --include_path="../../../../common/fft"
--include_path="../../../../common/nonmp" --define=ti_targets_elf_c66 --define=SOC_C6678 --diag_wrap=off --diag_warning=225
--display_error_number --mem_model:data=far --debug_software_pipeline -k --preproc_with_compile
--preproc_dependency="fft_dp_plan_1d_c2c_batch.d" --cmd_file="configPkg/compiler.opt"
"C:/ti/libraries/fftlb_2_0_2/packages/ti/fftlb/src/fft_dp_1d_c2c_batch/fft_dp_plan_1d_c2c_batch.c"

>> Compilation failure
subdir_rules.mk:52: recipe for target 'fft_dp_plan_1d_c2c_batch.obj' failed
"C:/ti/libraries/fftlb_2_0_2/packages/ti/fftlb/src/fft_dp_1d_c2c_batch/fft_dp_plan_1d_c2c_batch.c", line 44: fatal error #5:
could not open source file "ti/csl/csl_cacheAux.h"
1 fatal error detected in the compilation of
"C:/ti/libraries/fftlb_2_0_2/packages/ti/fftlb/src/fft_dp_1d_c2c_batch/fft_dp_plan_1d_c2c_batch.c".
Compilation terminated.
gmake: *** [fft_dp_plan_1d_c2c_batch.obj] Error 1
gmake: Target 'all' not remade because of errors.
```

When an include file is not recognized by the system, the project properties must be checked. For RTSC projects, verifying that all the RTSC projects are well defined is essential. To look at the RTSC definition right click on the project name and select *Properties* (the last item in the pull down menu). RTSC definitions are in the *General* ->*RTSC* tab:



If one of the needed element is not available, or has the wrong address, the system will flag it out. Going through all RTSFC elements (see the next two screen shots) one of the additional depositories has a small exclamation mark next to it. (second screen shot)

General

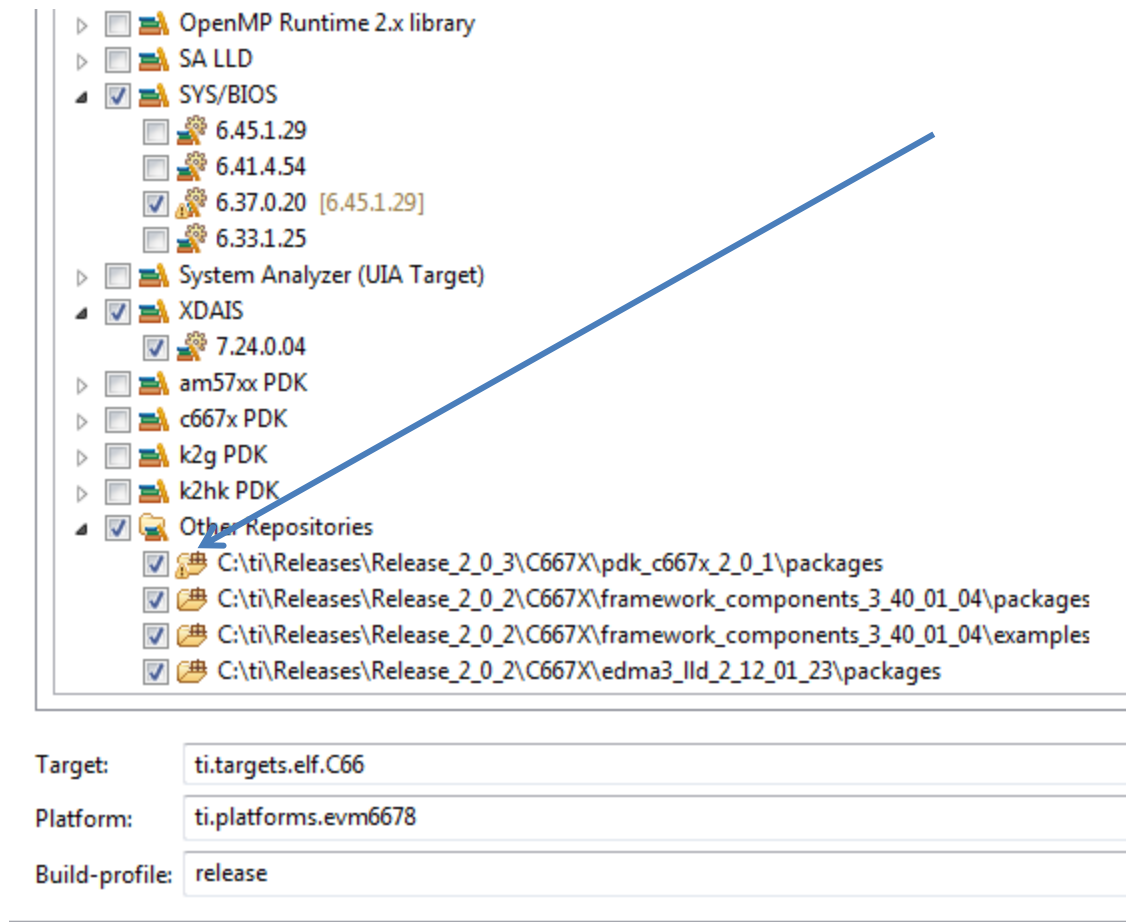
Configuration: **Debug [Active]**

Main **RTSC**

XDCtools version: **3.32.0.06_core**

Products and Repositories **Order**

- ▶ ☐ CTools Library
- ▶ ☐ CTools Library
- ▶ ☒ DSPLIB C66x
 - ☐ 3.4.0.0
 - ☒ 3.1.1.1 [3.4.0.0]
- ▶ ☐ DSPLIB C674x
- ▶ ☐ Dense Linear Algebra Library
- ▶ ☐ EDMA3 Low Level Driver
- ▶ ☒ Framework Components
 - ☐ 3.40.2.07
 - ☐ 3.40.1.04
 - ☒ 3.30.0.06 [3.40.2.07]
- ▶ ☐ IMGLIB C64x+
- ▶ ☐ IMGLIB C66x
- ▶ ☐ IPC
- ▶ ☐ Keystone2 PDK
- ▶ ☐ Library Architecture and Framework
- ▶ ☐ MATHLIB C66x
- ▶ ☐ MATHLIB C674x
- ▶ ☐ MCSDK
- ▶ ☐ MSP430ware
- ▶ ☐ NDK
- ▶ ☐ OpenMP Runtime 2.x library
- ▶ ☐ SA LLD



The screenshot shows a configuration window with a tree view on the left and a form on the right. The tree view contains the following items:

- OpenMP Runtime 2.x library
- SA LLD
- SYS/BIOS
 - 6.45.1.29
 - 6.41.4.54
 - 6.37.0.20 [6.45.1.29]
 - 6.33.1.25
- System Analyzer (UIA Target)
- XDAIS
 - 7.24.0.04
- am57xx PDK
- c667x PDK
- k2g PDK
- k2hk PDK
- Other Repositories
 - C:\ti\Releases\Release_2_0_3\C667X\pdk_c667x_2_0_1\packages
 - C:\ti\Releases\Release_2_0_2\C667X\framework_components_3_40_01_04\packages
 - C:\ti\Releases\Release_2_0_2\C667X\framework_components_3_40_01_04\examples
 - C:\ti\Releases\Release_2_0_2\C667X\edma3_1ld_2_12_01_23\packages

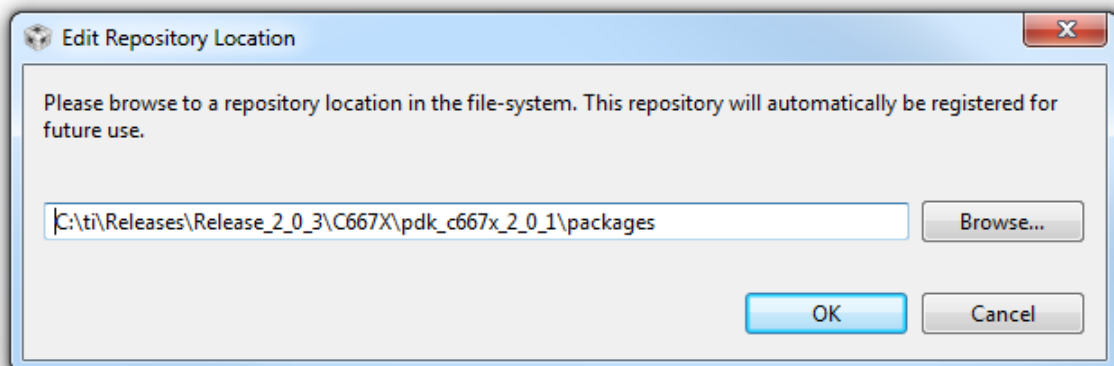
A blue arrow points from the 'Other Repositories' folder to the first entry in its list. Below the tree view, there is a form with the following fields:

Target: ti.targets.elf.C66

Platform: ti.platforms.evm6678

Build-profile: release

And indeed, if one search for `pdk_c667x_2_0_1\packages` , it is found at `c:\ti\Releases\release_2_0_2\C667X\`. To fix the error one should select the repository (left click) and click on the *Edit* tab at the right side of the window. A dialogue box is opened:



The screenshot shows a dialog box titled "Edit Repository Location". The dialog box contains the following text:

Please browse to a repository location in the file-system. This repository will automatically be registered for future use.

Below the text is a text input field containing the path `C:\ti\Releases\Release_2_0_3\C667X\pdk_c667x_2_0_1\packages`. To the right of the input field is a "Browse..." button. At the bottom of the dialog box are "OK" and "Cancel" buttons.

Fixing the path to the correct one and then click **OK** and **OK** remedies the problem. Then if the user rebuilds the project again the build process finished and generates executable;

```
DT Build Console [fft_dp_1d_c2c_batch_k1_66_LE_ELF]
--include_path=../../../../../common --include_path=../../../../../common/ti
--include_path=../../../../../common/nonmp --define=ti_targets_elf_c66 --define=SOC_C6678 --diag_wrap=off --diag_warning=225
--display_error_number --mem_model:data=far --debug_software_pipeline -k --preproc_with_compile
--preproc_dependency="fft_dp_mixed_bs.d" --cmd_file="configPkg/compiler.opt"
"C:/ti/libraries/fftlb_2_0_0_2/packages/ti/fftlb/src/common/fft/fft_dp_mixed_bs.c"
'Finished building: C:/ti/libraries/fftlb_2_0_0_2/packages/ti/fftlb/src/common/fft/fft_dp_mixed_bs.c'
,

'Building file: C:/ti/libraries/fftlb_2_0_0_2/packages/ti/fftlb/src/fft_dp_1d_c2c_batch/fft_dp_plan_1d_c2c_batch.c'
'Invoking: C6000 Compiler'
"C:/ti/ccsv6_1_3/ccsv6/tools/compiler/c6000_7.4.16/bin/cl6x" -mv6600 --abi=eabi -g
--include_path="C:/ti/ccsv6_1_3/ccsv6/tools/compiler/c6000_7.4.16/include" --include_path=../../../../../
--include_path=../../../../../common --include_path=../../../../../ --include_path=../../../../../common/fft
--include_path=../../../../../common/nonmp --define=ti_targets_elf_c66 --define=SOC_C6678 --diag_wrap=off --diag_warning=225
--display_error_number --mem_model:data=far --debug_software_pipeline -k --preproc_with_compile
--preproc_dependency="fft_dp_plan_1d_c2c_batch.d" --cmd_file="configPkg/compiler.opt"
"C:/ti/libraries/fftlb_2_0_0_2/packages/ti/fftlb/src/fft_dp_1d_c2c_batch/fft_dp_plan_1d_c2c_batch.c"
'Finished building: C:/ti/libraries/fftlb_2_0_0_2/packages/ti/fftlb/src/fft_dp_1d_c2c_batch/fft_dp_plan_1d_c2c_batch.c'
,

'Building target: fft_dp_1d_c2c_batch_k1_66_LE_ELF.out'
'Invoking: C6000 Linker'
"C:/ti/ccsv6_1_3/ccsv6/tools/compiler/c6000_7.4.16/bin/cl6x" -mv6600 --abi=eabi -g --define=ti_targets_elf_c66
--define=SOC_C6678 --diag_wrap=off --diag_warning=225 --display_error_number --mem_model:data=far --debug_software_pipeline -k -z
-m"fft_dp_1d_c2c_batch_k1_66_LE_ELF.map" -i"C:/ti/Releases/Release_2_0_2/C667X/edma3_lld_2_12_01_23/packages/ti/sdo/edma3/rm"
-i"C:/ti/ccsv6_1_3/ccsv6/tools/compiler/c6000_7.4.16/include" -i"C:/ti/ccsv6_1_3/ccsv6/tools/compiler/c6000_7.4.16/lib"
--reread_libs --diag_wrap=off --display_error_number --warn_sections
--xml_link_info="fft_dp_1d_c2c_batch_k1_66_LE_ELF_linkInfo.xml" --rom_model -o "fft_dp_1d_c2c_batch_k1_66_LE_ELF.out"
"/fft_c6678_config.obj" "/fft_common.obj" "/fft_dp_1d_c2c_batch.obj" "/fft_dp_1d_c2c_batch_d.obj" "/fft_dp_mixed_bs.obj"
"/fft_dp_plan_1d_c2c_batch.obj" "C:/ti/libraries/fftlb_2_0_0_2/packages/ti/fftlb/src/common/nonmp/lnk.cmd"
-l"configPkg/linker.cmd" -llibc.a
<Linking>
'Finished building target: fft_dp_1d_c2c_batch_k1_66_LE_ELF.out'
,

**** Build Finished ****
```

Note: RTSC projects encapsulate the used modules in the RTSC window. Most of the build error is due to the wrong definition of RTSC module or the path to a repository.