Working With Large Projects in Code Composer Studio 2.0

Ki-Soo Lee *Code Composer Studio, Applications Engineering*

ABSTRACT

This application report examines the issues surrounding large, multi-site project development in Code Composer Studio™, and offers techniques on how to use the extensive features of Code Composer Studio to resolve these issues. Some of the features discussed are source control and external makefile use, as well as symbolic debugging of executables built, using a large number of project files.

Contents

1 Introduction

When dealing with large projects that involve a large number of source files, dependence on other projects, and numerous developers across different sites, it naturally becomes more difficult to manage these projects. However, Code Composer Studio v2.0 offers many features that, if used effectively, can be of great assistance when working with large projects. These features, and how to use them for large projects, are discussed in the following pages.

This application report focuses more on techniques that can be applied using the various features of Code Composer Studio v2.0, and less on detailed steps of how to use these features – as many of these steps are covered in other documentation. For more information and a basic description of these features, please refer to the online help provided with the installation of Code Composer Studio v2.0. Other useful documents to refer to when reading this application note are *Managing Code Development Using the CCS Project Manager* (SPRA762), which presents the new features available in Code Composer Studio v2.0 that facilitate efficient project management, and *Example of GEL Usage With File I/O for Code Composer v2.0* (SPRA774), which covers such topics as using GEL to automate tasks in more detail.

2 Large Project Management Techniques

2.1 Using Multiple Configurations

When dealing with large projects with many source files, it is often advantageous to have multiple configurations within the project. In addition to having the standard Debug and Release configurations, additional custom configurations can be created to have a variety of build options or settings. This is useful when the developer wishes to change a build option or add/remove a common symbol definition that would force a rebuild of many files and consume a great deal of time. For example, code built to run on a target while running the debugger might need to behave differently than code that will run without the debugger (bootloaded from flash). The reason is that perhaps the actual target application needs to await some kind of user input (pressing a power ‘on’ button), which would cause a type of reset and cause the host debugger to lose connection with the target if a debugger is used. In that case, the step where the application waits for a particular user input may be bypassed when the application is built with a certain symbol definition. Instead of having to rebuild every time the developer wishes to switch between running with the debugger or not; simply have different configurations for each set of build options or common symbol definition can save countless hours of rebuilding.

Having multiple configurations can also allow the developer to have different files be included in the build for different configurations. This is helpful when there are different versions of a particular file. Instead of having to manually switch between the two (or more) files when a particular result is desired, different configurations can be created that each use a version of the file.

2.2 Using File-Specific Options

File-specific options can be of great assistance when dealing with large projects. Being able to define symbols, change some build optimization settings, or add pre/post-step build options on a single file saves the hassle of having to rebuild entire projects, if project level build options are changed. Another useful feature is the ability to exclude a file from a particular build configuration.

2.3 Using Libraries

As the number of source files increase, it can be a rather daunting task maintaining all those files in one project. A library, created with some of those source files, can be used to reduce the number of source files in a project. Often libraries are created from a group of related files. For example, a library can be created that contains all the code in the application for JPEG encoding and decoding, and it can be called jpeg.lib to help easily identify what source files are contained in the library. Another advantage of using libraries is being able to do a ‘rebuild all’, without having to rebuild all the files in the library.

2.4 Creating Project Dependencies

When dealing with a project that uses many libraries, it may be beneficial to set up some project dependencies, so that the main project is dependent upon the successful build of the libraries used by the main project. This is helpful when a complete rebuild of the entire project (libraries and all) is desired. Such project dependencies can be set up by using the timake.exe utility as a post-step build option to automatically begin a build of a project, once a dependent project has been successfully built. The timake.exe utility allows you to run the CCStudio project build process external to CCStudio. It allows your project contained in the *.pjt file to be built directly from the command line. For example, say you have divided your large project into a 'main' project (main.pjt) and two library projects – lib1.pjt and lib2.pjt. If you wished to set up the project so that a complete rebuild of the entire project and associated libraries can be done by beginning the build of one project, it can be done like this:

1. Create a new configuration (for example – 'dependency') for lib1.pjt, lib2.pjt and main.pjt.
2. For the lib1.pjt, in the 'Final build Steps' in the 'General' tab under the project build options for the new configuration 'dependency', use the 'timake.exe' utility to begin the build of lib2.pjt once the build of lib1.pjt is complete:

   timake.exe lib2.pjt dependency –a

   This will build the 'dependency' configuration of lib2.pjt, once lib1.pjt has been successfully built.
3. For the lib2.pjt, in the 'Final build Steps' in the 'General' tab under the project build options for the new configuration 'dependency', use the 'timake.exe' utility to begin the build of main.pjt 'dependency' configuration once the build of lib2.pjt is complete:

   timake.exe main.pjt dependency –a

   This will build the 'dependency' configuration of main.pjt, once lib2.pjt has been successfully built.

In the example shown above, a new configuration 'dependency' was created for all the projects and then 'linked' together by using the 'Final build steps' in the 'General' tab of the project build options. Lib1.pjt was set as the project file to be 'active' when the initial build is started, and the rebuild all option can be selected here to kick off the build process. Once the lib1 library has been built, the build of lib2 will automatically begin by using the timake.exe utility as a 'Final build step'. Once the lib2 library has been built, the build of the main project will automatically begin. Because the build of the main project will not begin until after lib2.pjt is done building, and because the build of lib2.pjt does not begin until lib1.pjt is done building; when the main project links in lib1.lib and lib2.lib, both libraries have been completely rebuilt. This way we have created project dependencies, where main.pjt is dependent upon the successful rebuild of lib2.pjt (which is dependent up the successful rebuild of lib1.pjt).

2.5 Using Initial/Final Build Steps

Initial/Final build steps (found under the ‘General’ tab of the project build options) can be a very effective means to automate tasks. As seen in section 2.4, timake.exe was run as a Final build step to automatically kick off another build of a project. In addition to the timake.exe utility, other utilities like the hex conversion utility (a program that accepts COFF files and converts them into one of several standard ASCII hexadecimal formats suitable for loading into an EPROM programmer) can be run as a Final build step, or batch files can be executed as an Initial or Final build step. As projects grow in size, it becomes advantageous to be able to automate as much of the build procedure as possible. Creating batch files to do some specific tasks before/after a build (such as appending some binary resource to the end of a binary executable to be burned to ROM), and being able to have Code Composer Studio execute the batch file automatically, can be a helpful and time saving step.

Initial/Final build steps can also be applied to just a single file by using the file-specific options discussed in section 2.3.

2.6 Source Code Control

Source code control is essential for managing complex software development projects. Large projects require teams of software engineers working on many source files. Source code control tools make it possible to keep track of the changes made to individual source files, and prevent files from being accessed by more than one person at a time. Other benefits include: being able to retrieve an older version of code checked into the source control database, and to help facilitate sharing of consistent versions across teams. The Code Composer Studio IDE supports any source control provider that correctly implements the Microsoft SCC Interface. Once Code Composer Studio is interfaced with the source code control application, the project displays visually which files are checked in or out. Also, many of the basic features/actions of the source control application can be accessed within the Code Composer Studio IDE.

2.7 Using GEL

GEL (General Extension Language) can be a very effective tool when dealing with large projects. GEL is very useful in automating tasks in Code Composer Studio, such as using the built–in GEL functions to automatically open a project (GEL_ProjectLoad()), begin a build (GEL_Build()), and load an executable (GEL_Load()) once Code Composer Studio has started up. GEL can also be used for batch building of multiple projects.

2.8 Working With Makefiles

A Code Composer Studio project can be exported to a standard makefile, saving the user the time of manually recreating the detailed build steps that are followed by Code Composer Studio. The standard makefile is compatible with any make utility that is derived from the standard make utility, such as gmake. Code development projects can be created using Code Composer Studio v2.0, and final builds done in some other environment, such as UNIX.

2.9 CodeMaestro

When working on large projects with many developers, it is common that a developer will be using many APIs written by other developers. When this is the case, the new CodeMaestro technology can help save valuable coding time, by adding those API's to its auto-complete database. This way, the APIs will appear as options in the listbox that pops up when you press the tab to use CodeMaestro's Complete Word option (assuming the first letter or two of the API was typed). This is extremely helpful when there are hundreds of APIs available, and the developer cannot quite remember what the exact API call is. For example, when an application is divided into libraries worked on by different teams or individuals, CodeMaestro's auto-complete option can save the developer the time and effort of having to look up the API in header files or documentation. To make sure that the desired APIs are in the auto-complete database, please make sure that the header file that contains the prototypes of the API is included (#include <headerfile.h>) in the source file that will use them, and that the source file is saved. Also, the project being used needs to be closed and reopened for the auto-complete feature to take effect.

2.10 Enhancements from CCStudio v1.x

2.10.1 Scanning Dependencies

Unlike CCStudio v1.x, when a CCStudio 2.0 project is opened, it will not automatically scan all dependencies. It will only do so when the 'Project–>Scan All Dependencies' menu option is selected, or when a build is executed. This was done so a developer does not have to sit through a dependency scan for a large project every time when all the developer wants to do is just open the project.

3 Debugging Large Projects

3.1 Using the Symbol Browser

As projects get larger, it becomes more difficult to find files, functions, and symbols on your own when debugging an application. Also in large projects, the teams that do the code development and system integration/debug may be separate. Therefore, it is not uncommon for the development team to just throw the executable and source files over the wall to the debug team. Hence, the Symbol Browser can be a very useful tool, since the Symbol Browser displays all of the associated files, functions, global variables, types, and labels of a loaded COFF output file (*.out). From the Symbol Browser, you can open the source code for a file or function in the text editor, thus taking you right to the file or function.

3.2 Using GEL

GEL can also be a very useful tool when debugging large projects. GEL functions can be used to automate some debugging tasks, such as adding an expression to the Watch Window (GEL_WatchAdd()), setting Breakpoints (GEL_BreakPtAdd()) and Probe Points (GEL_ProbePtAdd()), or starting execution of target applications (GEL_Run()). You can call GEL functions from anywhere that you can enter an expression. You can also add GEL functions to the Watch Window, so they execute at every breakpoint. Again, the benefit of automating tasks by using GEL is saved time.

4 Conclusion

As we have seen, Code Composer Studio provides many features to help deal with issues that may come up when working with large projects. As applications become larger in size and complexity, it is important for the developers to be aware of features that are available to them that can aid in working with these larger and more complex projects.

5 References

It is strongly encouraged that you refer to the following documentation when using this application report, for more information on the various features of Code Composer Studio.

1. Code Composer Studio 2.0 online help.
2. *Managing Code Development Using the CCS Project Manager* (SPRA762).
3. *Example of GEL Usage With File I/O for Code Composer v2.0* (SPRA774).

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, license, warranty or endorsement thereof.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations and notices. Representation or reproduction of this information with alteration voids all warranties provided for an associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Resale of TI's products or services with *statements different from or beyond the parameters* stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Also see: Standard Terms and Conditions of Sale for Semiconductor Products. www.ti.com/sc/docs/stdterms.htm

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265