Application Report
SPRA762 - June 2001

Managing Code Development Using CCS Project Manager

Chuck Farrow *Northeast Applications*
Andy Thé

ABSTRACT

This application report presents the new features available in Code Composer Studio™ (CCS) v2 that facilitate efficient project management. The CCS Project Management system has been broadened and improved to include: multi-project/multi-configuration support, source code control, as well as the ability to interface to external build processes.

The material presented in this application note focuses on this and other new features specific to project management, and how to apply them in your code development process. Some examples help guide the reader through the various topics and techniques, pointing out the features that help provide efficient code development.

Contents

1 Project Development Overview . . . 2
2 Use External Makefile . . . 3
2.1 Limitations and Restrictions . . . 6
3 Batch Builds Using the timake Utility . . . 7
4 Building an Archive . . . 7
5 Multiple Configurations . . . 7
5.1 To Change the Active Project Configuration . . . 8
5.2 To Add a New Project Configuration . . . 8
5.3 To Remove a Project Configuration . . . 9
6 Source Code Control . . . 9
7 File and Project Build Options . . . 9
7.1 File Build Options . . . 10
7.2 Custom Build Options . . . 11
7.2.1 Specify Custom Build Steps for a Project . . . 11
7.2.2 Specify Custom, Pre- and Post-Build Steps for a File . . . 13
8 Export to Makefile . . . 14
9 References . . . 16
Appendix A timake.exe Utility . . . 17
Appendix B dosrun.bat Batch File . . . 18
Appendix C Syntax for Parsing Imported Makefiles . . . 19

Code Composer Studio is a trademark of Texas Instruments.

All trademarks are the property of their respective owners.

List of Figures

Figure 1. Build Environment Flow Chart . . . 3
Figure 2. Incorporating an External Makefile With a CCS Project . . . 4
Figure 3. Scanning for Source Files . . . 5
Figure 4. Sample Project for External Makefile . . . 6
Figure 5. Creating an Archive in CCS . . . 7
Figure 6. Project Toolbar . . . 8
Figure 7. Project Configuration Selection . . . 8
Figure 8. Build Sequence Diagram . . . 10
Figure 9. Pre- and Post-Project Build Options . . . 12
Figure 10. File-Specific Build Options . . . 13
Figure 11. Export to a Makefile . . . 16

List of Tables

Table C–1. Character Set Descriptions for Parsing Imported Makefiles . . . 20

1 Project Development Overview

The Project Development capabilities in CCS have been significantly improved. CCS 2.0 offers a highly sophisticated build environment that can be tailored to the individual project needs. This application report details these improvements and gives examples of how to use them.

The improvements include the capability to track source code changes from within CCS. Once a particular Source Code Management application is associated with CCS v2, its capabilities are accessible from within CCS via pulldown menus. This allows all code management to happen within the CCS integrated development environment (IDE). The benefit is that there is now no need to jump between multiple applications when using source control.

Another new feature within CCS is support for multiple build configurations. Multiple Build configurations allow, for example, a Debug Configuration and a Release Configuration of build options. Separate configurations for Debug and Release are helpful because they allow for two independent builds without making any changes to the project options. Not only are the compile/link options separate, but independent folders are created to hold all the output files generated during the build process. The names of the folders match the configuration name so it is easy to track where the output files for a particular configuration are. Any build configuration that you want to capture and save can be stored and later retrieved for bit exact rebuilds.

Using external makefiles and exporting makefiles is another new feature that facilitates code management from within CCS. By allowing CCS to use an external makefile a developer can integrate more complex builds into CCS. Complex builds could include source files from a UNIX environment. The following flow diagram highlights the steps that could be taken when developing a CCS project using source control and makefiles (see Figure 1).

The rest of this application report focuses on the specific portions of the project development cycle that are different from the prior releases (1.x) of CCS.

New CCS project ?
Yes
No
Create new project
Open existing project
Retrieve files from source control
External makefile ?
Yes
Import makefile
No
Set configuration
Edit or create
Store files in source control
Set file options
Set project options
Export makefile ?
Yes
No
Export makefile
Internal build
External build
End

Figure 1. Build Environment Flow Chart

2 Use External Makefile

This feature of the Project Manager allows the developer to use a makefile (*.mak) created for legacy product development. If, for instance, you had an existing build process outside of Code Composer Studio and now you wanted to build within CCS, you would use the External Makefile build option. The existing makefile will be called from within CCS, giving the effect of a fully integrated build environment. Refer to Figure 2 for an example of the available options related to the use of an external makefile. Clicking the Next button brings up the dialog box allowing the user to specify how source files are defined in the makefile, so that the makefile can be parsed and the source files displayed in the source folder of the Project View. Refer to Figure 3 for a sample view of this dialog box.

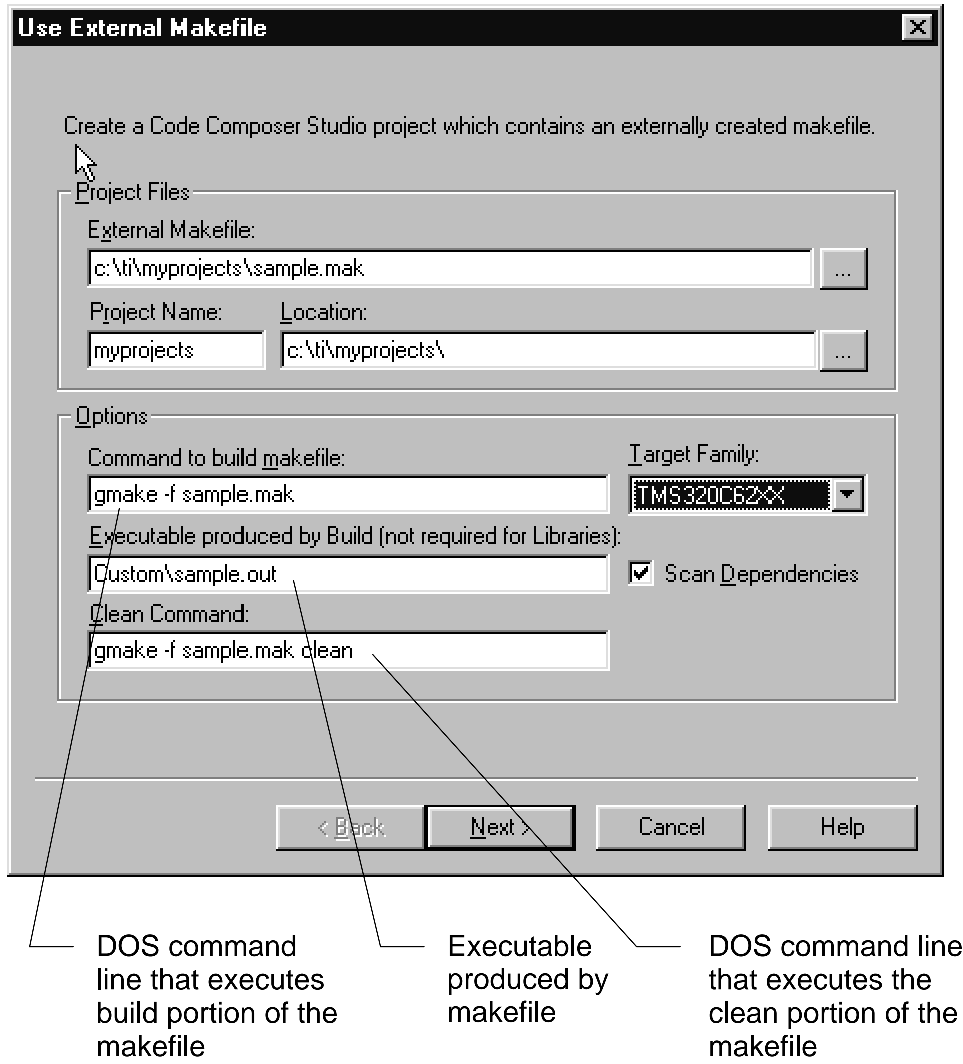

Figure 2. Incorporating an External Makefile With a CCS Project

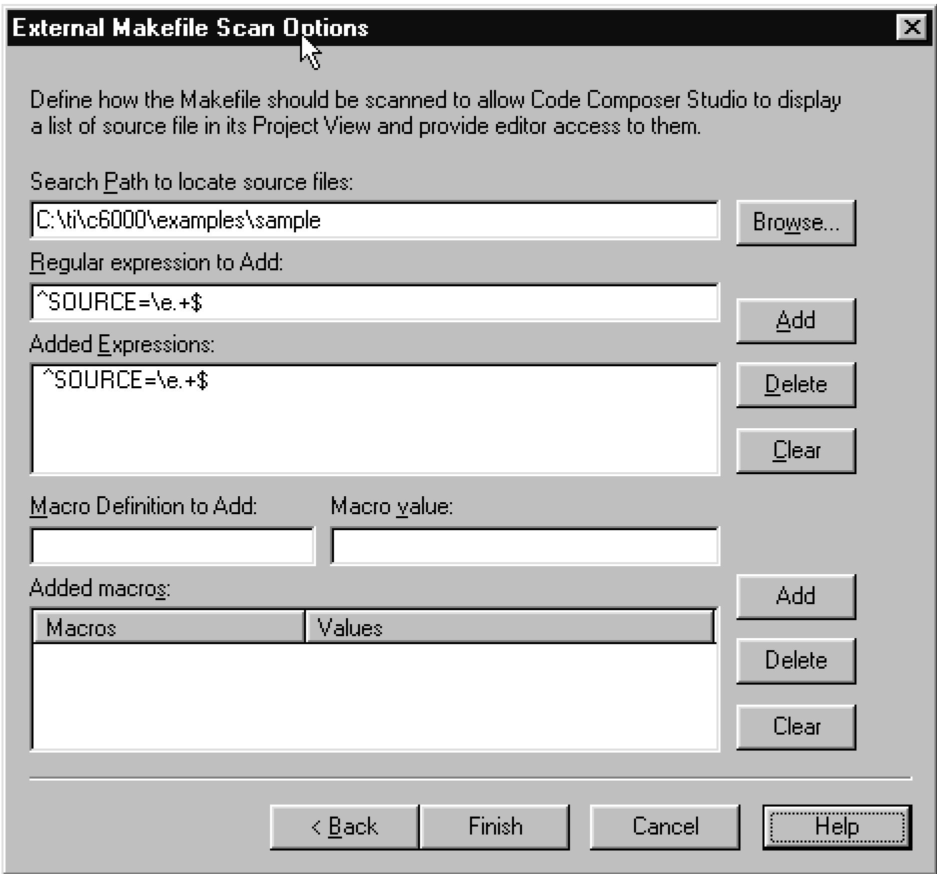

Figure 3. Scanning for Source Files

The search path indicates the location of the source files that are associated with the external makefile. The remaining textboxes are used to define how the makefile will be parsed to select source files, and any remaining macro definitions that are needed. Specifically, the Regular expression to Add textbox is where the user enters the expression that defines how the makefile will be searched to identify the source files. Once entered, the Add button adds the expression to the Added Expressions text box. The Macro Definition to Add textbox is where the user enters any required macro names. The value for this macro is then entered in the Macro Value textbox. Once complete, selecting the Add button places the newly defined macro into the Added macro display box. These values are now available for use by the project build environment.

Here are some syntax examples of to extract source files from the makefile:

```
"^SOURCE=\e.+$"       Extracts all source files after the SOURCE=macro definition.
"^.+\.obj\s*:\e.+$"   Extracts all the source dependencies in a makefile.
"^\w+s*=\e.+$"        Extracts all macro definitions.
```

For more information on how to construct these expressions, refer to Appendix C.

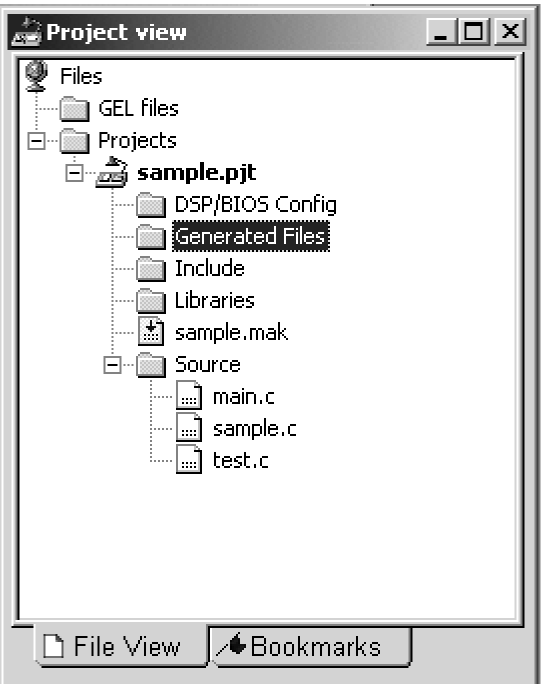

Figure 4. Sample Project for External Makefile

After clicking Finish, the CCS project is created that contains the external makefile. Refer to Figure 4 for an example of what this newly created project will look like. Now that a CCS project is associated with the makefile, the project and its contents can be displayed in the Project view window, and the Project–>Build and Project–>Rebuild All commands can be used to build the project. If you are creating multiple configurations in your build process, refer to section 5 for details on how to do this. Code Composer Studio allows you to edit the project file (*.pjt) on a text-edit basis. To open the project file for editing, right-click on it in the Project view window and select Open for Editing. This capability gives you greater flexibility over your project build process by allowing you to manually edit any component of the project build process.

2.1 Limitations and Restrictions

It is recommended that source files not be added to or removed from the project view manually. The preferred method is to make modifications to the makefile and rescan it by right-clicking on it and selecting rescan makefile. It is Important to understand that changes made in the Project view do not change the contents of the makefile. Also, if files are added manually to the Project view, build errors will be generated when building. Files in the project view–>source folder that were scanned from the makefile will not have a down arrow, while files that are manually added will have the down arrow in the file icon. File-specific options for source files that are added in the Project View are disabled. The Project–>Compile File command is also disabled. However, when the project is saved, the current state of the Project view is preserved.

Important: Before using Code Composer Studio commands to build your program using an external makefile, it is necessary to set the environment variables by executing the batch file *DosRun.bat.*

Refer to Appendix B for details on the contents of this batch file.

3 Batch Builds Using the timake Utility

CCS provides an external build utility that allows projects (*.pjt) to be built outside of CCS. This allows CCS projects to be integrated into larger system builds. These system builds, for example, could include a host processor. Now the DSP and Host processor code bases can be built from one master makefile or batch file. The master makefile or batch file calls the timake utility to build the DSP portion of the system. The timake.exe utility takes a *.pjt file and other options and generates an output file specified by the options set in the *.pjt file. Refer to Appendix A for specific options and use of the utility.

4 Building an Archive

CCS supports the creation of archives using the Project–>New menu selection. Refer to Figure 5 for the various options. When a project name is entered, the appropriate folder name is added to the location item. Select Library(.lib) as the Project Type. Select the appropriate Target for the application. When Finish is selected, an archive project is created. All the build options are selectable as with any other project, except the linker. Since there is no executable output, the linker is not needed.

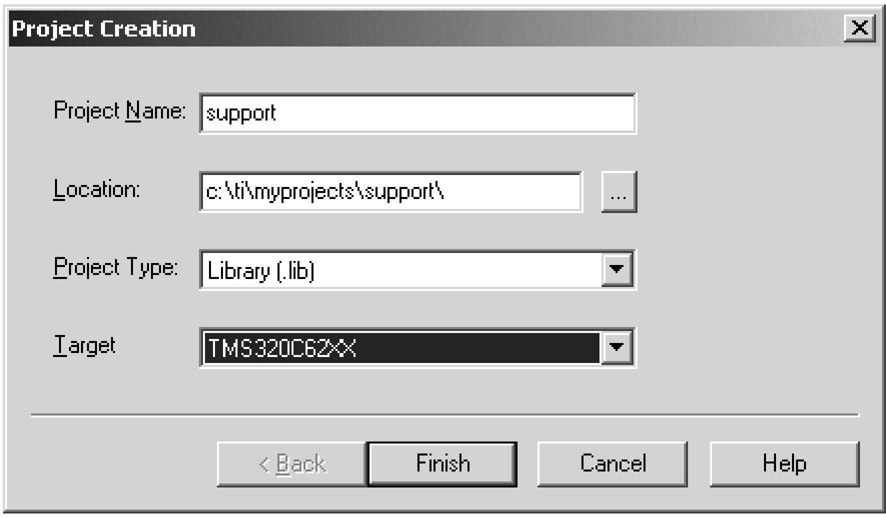

Figure 5. Creating an Archive in CCS

5 Multiple Configurations

A configuration defines a set of project level build options as well as all file specific options. Configurations independently contain all the options associated with a particular build. As such each configuration is uniquely named.

Build configurations enable you to define complete build options for the each phase of program development. For example, you can define a Debug configuration to use while debugging your program, and a Release configuration for building the finished product.

Each project is created with two default configurations: Debug and Release. Additional configurations can be defined. Whenever a project is created or an existing project is initially opened, the first configuration (in alphabetical order) is set active.

The active configuration setting is preserved in the Code Composer Studio workspace. To change the active configuration, refer to section 5.1.

When you build your program, the output files generated by the software tools are placed in a configuration-specific subdirectory. For example, if you have created a project in the directory TestProject, the output files for the Debug configuration are placed in TestProject\Debug. Similarly, the output files for the Release configuration are placed in TestProject\Release.

5.1 To Change the Active Project Configuration

There are two methods for selecting the active Configuration. Refer to Figure 6, then click on the Select Active Configuration field in the Project toolbar, and select a configuration from the drop-down list.

Figure 6. Project Toolbar

The active configuration can also be changed using the Project Configurations dialog box as shown in Figure 7.

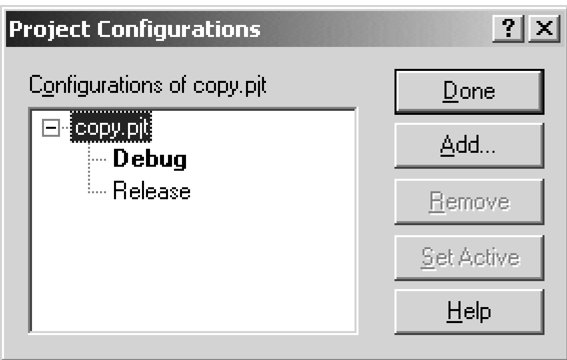

Figure 7. Project Configuration Selection

1. Select Project–>Configurations, or right-click on the project's filename in the Project View window and select Configurations.
2. In the Project Configurations dialog box, select the configuration you want to make active.
3. Click Set Active. The selected configuration is highlighted to indicate that it is now the active configuration.
4. Click Close.

5.2 To Add a New Project Configuration

1. Select Project–>Configurations, or right-click on the project's filename in the Project View window and select Configurations.
2. In the Project Configurations dialog box, click Add.

3. In the Add Project Configuration dialog box, specify the name of the new configuration in the Create Configuration field, and choose Use Default Settings (build options) or Copy Settings from an existing configuration.
4. Click OK to accept your selections, and exit the Add Project Configuration dialog.
5. Click Close to exit the Project Configurations dialog.

5.3 To Remove a Project Configuration

1. Select Project–>Configurations, or right-click on the project's filename in the Project View window and select Configurations.
2. In the Project Configurations dialog box, select the configuration to delete.
3. Click Remove.
4. Click Close.

6 Source Code Control

CCS allows a developer the capability to interface directly to their preferred source code control software application through CCS pulldown menus. The Code Composer Studio IDE supports any source control provider that correctly implements the Microsoft SCC Interface. Some of the more popular source control systems that are compatible with CCS are Rational's ClearCase, Microsoft's SourceSafe, or MKS. Once CCS is interfaced with the source code control application, the project displays visually which files are checked in or out. Source code control is essential for software development projects of all sizes. It helps with revision control, branching and merging of changes, and backups.

The project view window lists all the source files that are in a particular project in the source folder. Files that are checked out to the source control system have green checks to the left of the name. If a file is checked out, it can be opened in an Editor window and modified accordingly. Files that are checked into the source code control system are indicated by red checks to the left of the name. These checked-in files can be opened for viewing in an Editor window, but they are read-only, so changes CANNOT be saved.

Once a source code control provider is selected, that provider's tools are used whenever any of the source code control commands are invoked within CCS. The source code control features are limited by the capabilities of the provider.

7 File and Project Build Options

Use the following procedures to set the compiler and linker options that are used when building your program. Instructions are also provided on how to set custom build options that may run programs or utilities outside of the CCS environment.

Define a set of project level options that apply to all files in the project. Then optimize your program development by defining file specific options for individual source code files. If a particular file required very specific optimization settings you would use file level options to specify the optimization level.

To better understand the overall sequence of steps taken during the build process refer to Figure 8. This diagram highlights the steps taken after the Build or Build All menu item is selected.

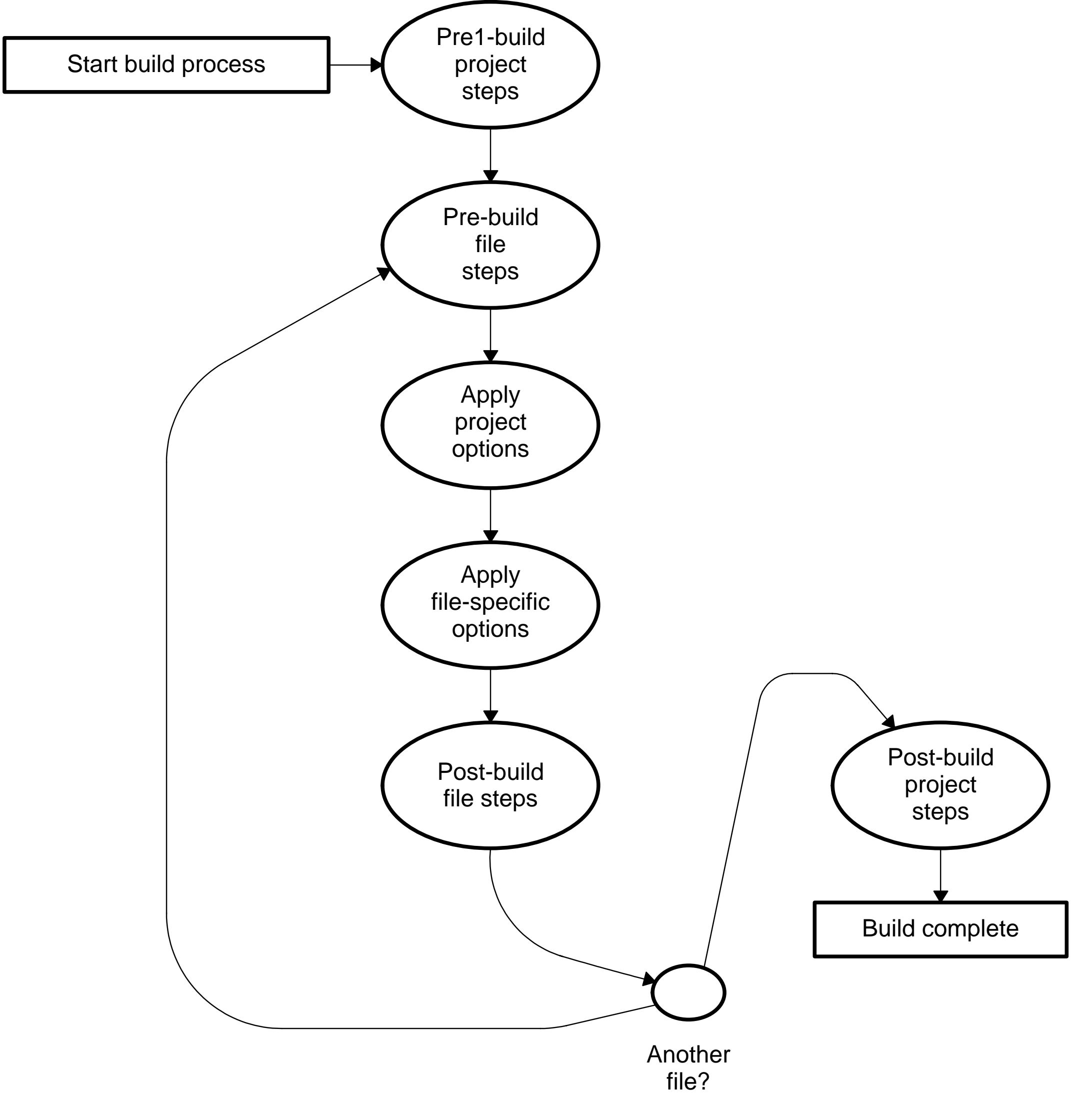

Figure 8. Build Sequence Diagram

7.1 File Build Options

Options set for a specific file override the project level settings.

1. Right-click on the name of the source file in the Project View window and select File Specific Options from the context menu.
2. Select the options to be used when compiling this file.
3. Click OK to accept your selections.

File-specific options are stored in the project file by recording only the differences between the project options and those set for the file. Project build options set at the project level affect all project files.

1. Select Project–>Build Options, or right-click on the project's filename in the Project View window and select Options.
2. In the Build Options dialog box, select the appropriate tab: General, Compiler, Linker, Archiver, or Link Order.
3. Select the options to be used when building your program.
4. Click OK to accept your selections.

7.2 Custom Build Options

The General dialog enables you to specify custom build steps.

Important: If any custom build step involves launching Texas Instruments' code generation tools, you must execute the batch file DosRun.bat to set up the necessary PATH and environment variables. It is recommended that you execute DosRun.bat as the first step of the initial build steps. If Code Composer Studio is installed in the default location, the batch file is located in the the main installation directory, which is usually c:\ti. Refer to Appendix B for more information.

7.2.1 Specify Custom Build Steps for a Project

Custom build steps are extremely helpful at automating things that are done before and/or after the actual compiling/linking phase. An example of some post build steps would be creating an archive or converting the coff output file to hex format file. If you wanted to create a library or convert the coff output file to hex format, the command would be placed in the Final Build Step textbox. Copying files from one location to another is another example for either a pre- or post-build step. Refer to Figure 8 for examples.

1. Select Project–>Build Options, or right-click on the project name in the Project View and select Options.
2. In the Build Options dialog box, select the General tab.
3. Click OK to exit the Build Options dialog box.

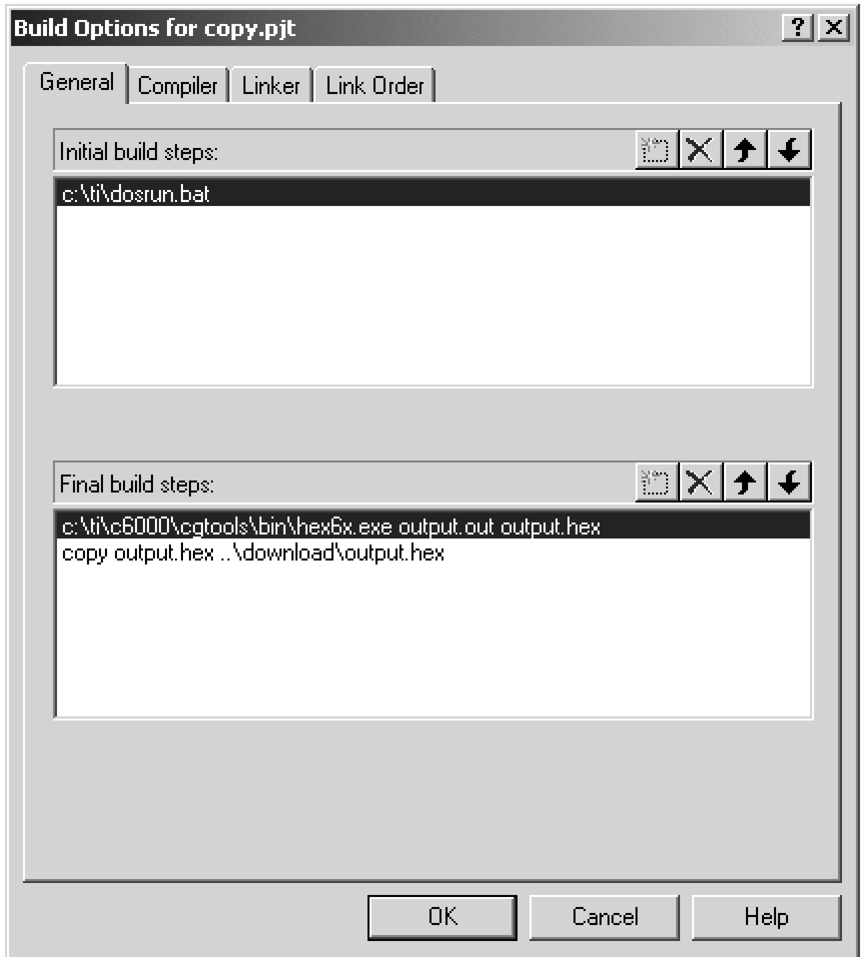

Figure 9. Pre- and Post-Project Build Options

The General dialog offers the following options. Refer to Figure 9 for a description of the options available for setting pre- and post-build options.

Initial build steps and final build steps. Displays the list of initial and final build steps to be executed. The build steps are executed in the order specified.

New. To add a new initial or final build step, click New, then type the desired command. By default, the new command is added to the bottom of the list.

Delete. Select a command, then click Delete to remove that command from the list.

Up. Select a command, then click Up to move that command higher in the list.

Down. Select a command in the list, then click Down to move that command lower in the list.

7.2.2 Specify Custom, Pre- and Post-Build Steps for a File

If file-specific custom build options are required, they can be specified here in the file-specific Build Options dialog box. You may want to exclude a particular file from a build while still keeping it in the project. These options are used to override the standard build tools and replace them with user specific ones. Typically, this is used in conjunction with external makefiles. This is because the makefile appears as a file in the project. To better understand the order in which project and file-specific build options are executed, refer to Figure 8.

1. Right-click on the source file name in the Project View, and select File Specific Options.

   Alternatively, with the source file open in the integrated editor, you can select Project–>File Specific Options.

2. In the Build Options dialog box, select the General tab.
3. Click OK to exit the Build Options dialog box.

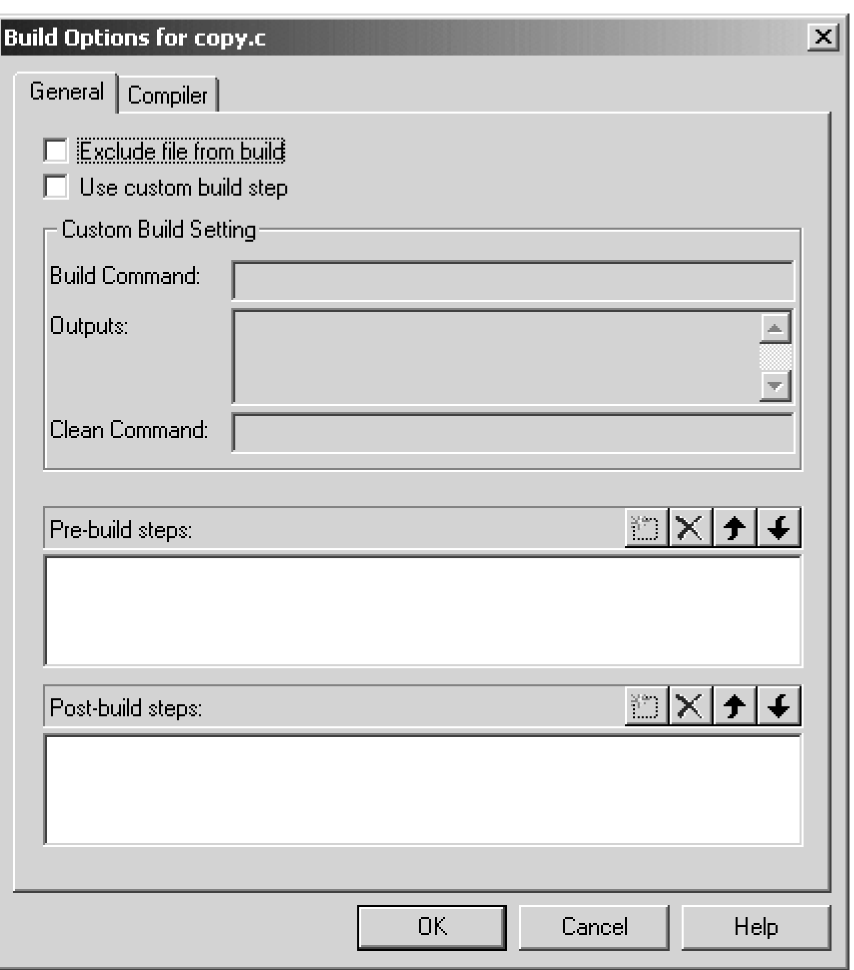

Figure 10. File-Specific Build Options

The General dialog for file-specific options offers the following options. Refer to Figure 10 for a description of the options available for selecting file-specific build options.

Exclude file from build. The selected file is excluded from the build process. By default, this option is turned off.

Use custom build step. Activate the Custom Build Setting fields. Choosing this option enables you to specify custom build commands. By default, this option is turned off.

Custom Build Setting. After selecting Use custom build step, you can specify custom build commands for the selected file.

Build Command. Enter the command line to be executed to complete a build.

Outputs. Enter a path and filename for the executable file and any intermediate files generated by the build command.

Clean Command. Enter the command line to be executed to remove any files generated by the build command.

Pre-build steps and Post-build steps. Displays the list of pre-build and post-build steps to be executed. The build steps are executed in the order specified. Whereas custom build steps are executed instead of the normal build steps specified in the compiler/linker settings, pre- and post-build steps are executed in addition to them. Pre-build steps are executed before the standard steps, and post-build steps are executed afterwards. These steps can be used to run such things as preprocessors and the moving or copying of files. To better understand the order that project and file-specific build options are executed, refer to Figure 8.

New. To add a new pre-build or post-build step, click New, then type the desired command. By default, the new command is added to the bottom of the list.

Delete. Select a command, then click Delete to remove that command from the list.

Up. Select a command, then click Up to move that command higher in the list.

Down. Select a command in the list, then click Down to move that command lower in the list.

8 Export to Makefile

A Code Composer Studio project can be exported to a standard makefile saving the user the time of manually recreating the extensive build steps that are followed by CCS. The standard makefile is compatible with any make utility that is derived from the standard make utility. Code development projects can be developed using CCS v2 and final builds done in some other environment. This environment could even be on another Operating system like UNIX. By using the Export to a makefile utility, a makefile is created for use in this environment. This may be necessary if your build environment is on a UNIX platform.

Important: Before running a makefile, it is necessary to set the appropriate environment variables. To set environment variables on a PC, run the batch file

DosRun.bat

Refer to Appendix B for details on the contents of this batch file.

To Export a CCS Project to a Standard Makefile:

1. Make the desired Code Composer Studio project active by selecting the project name from the Select Active Project drop-down list on the Project toolbar.
2. Select Project–>Export to Makefile.
3. In the Exporting <filename>.pjt dialog box, specify the configurations to export, the default configuration, the host operating system for your make utility, and the filename for the standard makefile.
4. Click OK to accept your selections and generate a standard makefile.

Build Configurations. Lists all the configurations that are available for this project. Each configuration defines a set of project level build options that apply to all files in the project.

Builds to Export. Lists the configurations that are to be exported to the standard makefile. The active configuration is added to this list by default. Use the buttons >>, <<, Add All, and Clear All to modify the list of Builds to Export. The makefile is created and the specified configuration(s) become target(s) when the makefile is invoked. Refer to Figure 11.

>>. To add a configuration to the list of Builds to Export, select a configuration in the list of Build Configurations, and then click this button.

<<. To remove a configuration, select a configuration in the list of Builds to Export, and then click this button.

Add All. To add all configurations, click Add All.

Clear All. To remove all configurations, click Clear All.

Default Build Configuration. The drop-down list includes all configurations listed under Builds to Export. From the drop-down list, select the configuration to be se tup as the default configuration in the standard makefile.

Operating System. From the drop-down list, select the host operating system that you use to run your make utility.

Makefile Name. Accept the default filename, or enter a filename and the location where the standard makefile is to be created. You can use the Browse button to navigate to a directory and specify a filename.

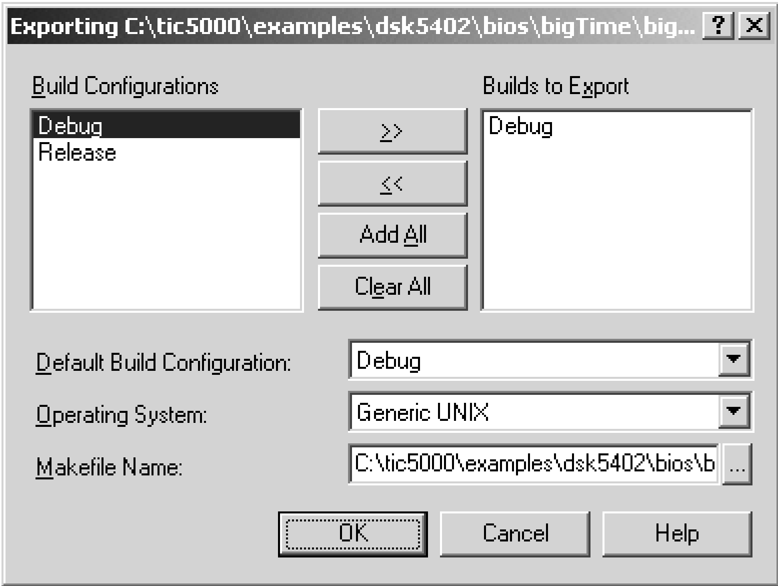

Figure 11. Export to a Makefile

9 References

1. Code Composer Studio 2.0 online help and tutorials.
2. *Code Composer Studio Getting Started Guide* (SPRU509).
3. *Migrating CCS 1.20/CCS 1.0 Projects to CCS 2.0* (SPRA745).
4. *Using Example Projects, Code and Scripts to Jump-Start Customers With Code Composer Studio 2.0* (SPRA766).
5. *Building DSP/BIOS Programs in UNIX* (SPRA660).

Appendix A timake.exe Utility

The timake.exe utility allows you to run the CCS project build process external to CCS. It allows your project contained in the *.pjt file to be built directly on the DOS command line.

```
Usage: timake pjt –lc | cfg [–a | –clean]
The first parameter <pjt> is required and must be a valid project file (.pjt)

The second parameter must be one of the following:
  –lc     List configuration names
  <cfg>   A configuration name
If the second parameter is a configuration name there can be a third option.

The third option can be one of the following:
  –a      Rebuild all
  -clean  Remove all intermediate files
```

Appendix B dosrun.bat Batch File

The following is a partial listing of the contents of the dosrun.bat batch file. This batch file is located in the directory c:\ti. If you installed Code Composer Studio IDE in a directory other than c:\ti, the batch file will be located in the directory you specified during installation. This batch file must be executed before running any of the external build features of CCS. Usually the dosrun.bat batch file is included, as the first pre-build step, since it is likely that CCS build tools will be called within the prebuild and/or post-build stages. This covers the situation when custom build features are required as pre- or post-build steps. In the case where external makefiles are being used, the dosrun.bat batch file must be executed within the context of the DOS command shell. The environment variables set by dosrun.bat are only valid within the scope which they were set. In other words, if dosrun.bat was executed as a pre-build step, then the environment variables are only defined with the CCS build scope. If dosrun.bat is executed within a DOS command shell, then the variables are only in scope within that DOS command window.

```
SET PATH=c:\ti\bin;c:\ti\c6000\cgtools\bin;c:\ti\c6000\evm6x\bin;c:\ti\bin
\utilities\vislink;%PATH%;
SET TI_DIR=c:\ti
SET UNIFY_PODDRVR_A=dsk6211.dll
SET UNIFY_PODPORT_A=0x378
SET UNIFY_PODMODE_A=0
SET C6X_C_DIR=c:\ti\c6000\cgtools\include;c:\ti\c6000\cgtools\lib;c:\ti\c6000
\bios\include;c:\ti\c6000\bios\lib;c:\ti\c6000\rtdx\lib;c:\ti\c6000\rtdx
\include;
SET C6X_A_DIR=c:\ti\c6000\cgtools\include;c:\ti\c6000\cgtools\lib;c:\ti\c6000
\bios\include;c:\ti\c6000\bios\lib;c:\ti\c6000\rtdx\lib;c:\ti\c6000\rtdx
\include;
SET D_SRC=c:\ti\c6000\evm6x\lib
```

Appendix C Syntax for Parsing Imported Makefiles

The following character set defines the syntax used by CCS for parsing imported makefiles. The expressions are created in the Using External Makefiles dialog, and they specify how the source files are selected and displayed with the CCS project view.

Table C–1. Character Set Descriptions for Parsing Imported Makefiles

Character	Description
\	Marks the next character as special. To define a special character in a regular expression, precede the special character with the backslash character. Example: The regular expression /n/ matches the character n. The regular expression /\n/ matches a linefeed or newline character.
^	Matches the beginning of input or line. In this implementation, this character cannot be defined in character set.
$	Matches the end of input or line. In this implementation, this character cannot be defined in a character set.
*	Matches the preceding character zero or more times. In this implementation, this character cannot be defined if only one character is specified in the regular expression. This means that /zo*/ matches z and zoo, but /z*/ matches nothing because only one character has been specified.
+	Matches the preceding character one or more times.
?	Matches the preceding character zero or one time. In this implementation, this character cannot be defined if only one character is specified in the regular expression.
.	Matches any single character except “\n”.
(pattern)	Matches the pattern and remembers the match. The matched substring can be retrieved by using “\0”–“\9” in the regular expression, where “0”–“9” are the numbers of the pattern. Example: Regular expression “(re).*\0s+ion” will match “regular expression”. First, the pattern “(re)” matches the first two letters of “regular expression”, and the pattern is remembered with index 0. The pattern “*” matches “gular exp” in “regular expression”. The pattern “\0” retrieves the pattern that has been remembered with index 0, and this “re” matches the second occurrence of “re” in “regular expression”. Finally, the pattern “s+ion” matches “ssion”.
x|y	Matches either the character “x” or “y”. You can combine more than two characters. Example: “x|y|z”.
{n}	Means the preceding character will match exactly n times (nonnegative).
{n,}	Means the preceding character will match at least n times (nonnegative).
{n,m}	Means the preceding character will match at least n times, and at most m times (both nonnegative).
[xyz]	A character set. Matches any one of the enclosed characters.
[^xyz]	A non-matching character set. Matches any character that is not in the set.
\b	Matches a word boundary, that is, the boundary between any character excluding space characters (“\f\n\r\t\v”) and non-space characters.
\B	Matches a non-word boundary. Matches any boundary between space characters or between non-space characters.
\d	Matches any digit /0–9/.

Table C–1. Character Set Descriptions for Parsing Imported Makefiles

Character	Description
\D	Matches any non-digit.
\f	Matches a formfeed.
\n	Matches a newline character.
\r	Matches a carridge return character.
\s	Matches any white space character.
\S	Matches any non-white space character.
\t	Matches a tab character.
\v	Matches any vertical tab character.
\w	Matches any word character, including underscore. [A–Za–z0–9_]
\W	Matches any non-word character (any character that does not match \w).
\num	Where num is a value between 0 and 9. Matches remembered pattern. (See description of pattern.)
/n/	Where n is a value between 1 and 255. Matches supplied in n ASCII code.
\e	Marks the start position of text to extract; used in conjunction with \E. Example: “FIND\e.+” would match “FIND.+” but only return the characters after “FIND”.
\E	Marks the end position of text to extract; used in conjunction with \e. If the \E is not supplied, the end of line ($) will be used as the end position. Example: “Find\e.+\ETheRest” would match “Find.+TheRest”, but only return the characters between “Find” and “TheRest”.
	Examples
“^SOURCE=\e.+$”	Extracts all source files after the SOURCE=macro definition.
“^.+\.obj\s*:\e.+$”	Extracts all the source dependencies in a makefile.
“^\w+s*=\e.+$”	Extracts all macro definitions.

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

Copyright © 2001, Texas Instruments Incorporated