DSP/BIOS 5.30 Textual Configuration (Tconf) User’s Guide

Literature Number: SPRU007H
May 2006

Preface

Read This First

About This Manual

DSP/BIOS allows you to develop embedded real-time software applications for Texas Instruments TMS320 DSP devices. DSP/BIOS provides a small firmware real-time library and easy-to-use tools for real-time tracing and analysis.

This book described the Tconf configuration scripts used with DSP/BIOS. It is intended as an addendum to the *TMS320 DSP/BIOS User's Guide*. In addition, the TMS320 DSP/BIOS API Reference Guide for your platform provides details about DSP/BIOS modules.

Important: This manual is for use with DSP/BIOS 5.30. See Appendix A for conversion information.

Notational Conventions

This document uses the following conventions:

❏ Program listings, program examples, and interactive displays are shown in a `special typeface`. Examples use a **`bold version`** of the special typeface for emphasis; interactive displays use a **`bold version`** of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).

❏ Square brackets ([and]) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets. Unless the square brackets are in a **bold** typeface, do not enter the brackets themselves.

❏ BIOS_INSTALL_DIR is the top-level folder of the DSP/BIOS installation. It is best to define this environment variable to point to your DSP/BIOS installation. However, only the example applications actually require this environment variable.

Trademarks

Licences

Contents

Figures

Tables

DSP/BIOS Tconf Overview

This chapter introduces Tconf, which is used to configure DSP/BIOS applications.

Topic **Page**

1.1 DSP/BIOS Configuration Roadmap

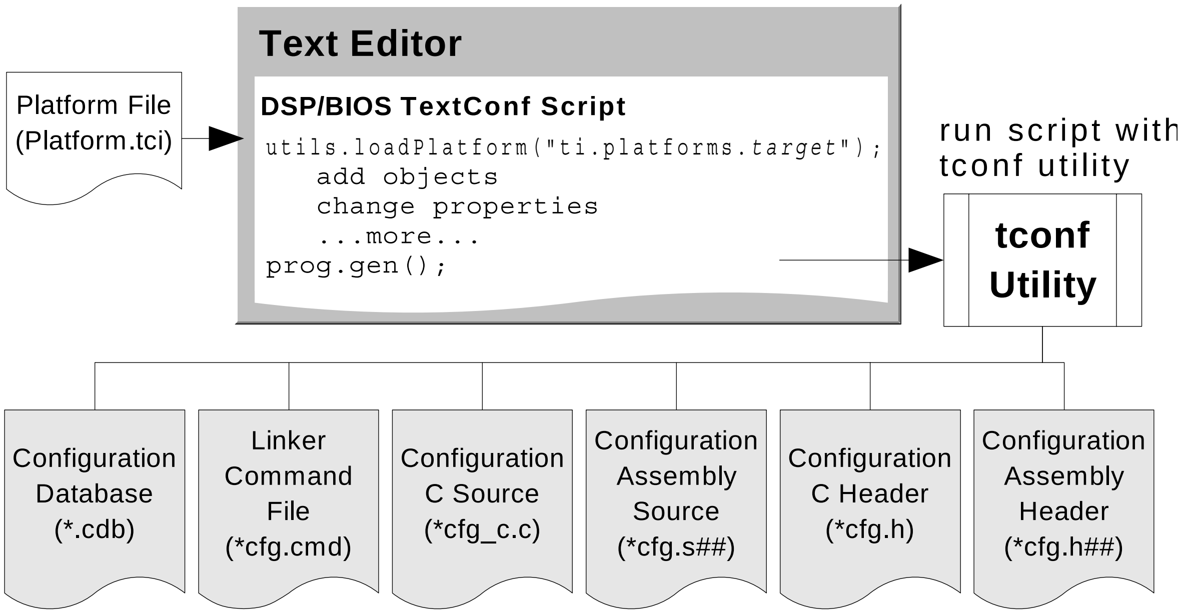
DSP/BIOS configuration allows you to create and configure static objects and properties for use by the DSP/BIOS API as part of your application design. For information about DSP/BIOS objects and properties, see the *TMS320 DSP/BIOS API Reference Guide* for your platform.

Typically, you use the graphical DSP/BIOS Configuration Tool (Gconf) to create your initial configuration. This tool acts as a macro recorder for Tconf scripts (TCF files). You see the script change in the right pane of the tool as you change the configuration. Later, you can edit the Tconf script generated by the DSP/BIOS Configuration Tool with a text editor.

Tconf scripts are now the source files for DSP/BIOS configurations. The CDB files previously used as configuration source files can now only be opened in read-only mode or converted to TCF files.

When you save a configuration with the DSP/BIOS Configuration Tool, the files shown in gray in Figure 1-1 are generated. See Section 2.1.1, *Generated Files*, page 2-2 for more details.

Figure 1-1 DSP/BIOS Configuration


Text Editor
Platform File (Platform.tci)
DSP/BIOS TextConf Script
utils.loadPlatform("ti.platforms.target");
add objects
change properties
...more...
prog.gen();
run script with tconf utility
tconf Utility
Configuration Database (*.cdb)
Linker Command File (*cfg.cmd)
Configuration C Source (*cfg_c.c)
Configuration Assembly Source (*cfg.s##)
Configuration C Header (*cfg.h)
Configuration Assembly Header (*cfg.h##)

The roadmap for future configuration is to provide the same capabilities for all target content written for TI DSPs as those that are available for DSP/BIOS modules. This is part of a broad component re-use strategy. Tconf is just one utility in a set of component tools to be provided in the eXpress DSP Component (XDC) Tools to enable component re-use. For more information about the XDC Tools and future content see https://www-a.ti.com/downloads/sds_support/targetcontent/rtsc/index.html

The XDC Tools (including the Tconf utility) are supported for Microsoft Windows and UNIX (Solaris and x86 Linux).

1.2 DSP/BIOS Configuration Benefits

The following sections describe the benefits of the static (design-time) configuration and reasons to use the DSP/BIOS Configuration Tool vs. a text editor to modify a Tconf script.

1.2.1 Benefits of Static Configuration

The DSP/BIOS API also supports dynamic creation of objects at run-time. Creating objects at run-time is easier, but extra code is required to support the object creation and deletion.

Design-time configuration provides the following benefits over run-time configuration:

❏ Improves run-time performance by reducing the time your program spends performing system setup.

❏ Reduces program size by eliminating run-time code required to dynamically create and configure objects. For a typical module, the functions to create and delete objects make up 50% of the code in the module.

❏ Optimizes internal data structures.

❏ Detects errors earlier by validating object properties before program compilation.

❏ Automatically sets a variety of properties that are dependent on other properties. This helps ensure that your configuration is valid.

❏ Provides object names the DSP/BIOS Analysis Tools can show at run-time. Objects created at run-time are either not shown or have generated names.

1.2.2 The DSP/BIOS Configuration Tool vs. a Text Editor

Both the DSP/BIOS Configuration Tool and direct text editing of scripts have advantages in certain situations. You can use either configuration method alone, or you can switch between these methods to perform tasks in the environment best suited to each task.

The DSP/BIOS Configuration Tool provides the following advantages over editing Tconf scripts with a text editor:

❏ The Windows Explorer-like interface makes it easy to see a list of the available properties for each module and its objects.

❏ You are prevented from making a number of errors through drop-down lists of valid values and through disabled commands and fields.

❏ Syntax errors cannot occur when generating configuration files.

❏ You do not need to learn the Tconf script syntax.

Using a text editor to manually edit a Tconf script has the following benefits:

❏ Supported on UNIX, Linux, and Windows. (The DSP/BIOS Configuration Tool is not supported on UNIX and Linux.)

❏ Allows you to import sub-scripts (TCI files) so that you can modularize platform-specific, application-specific, or other categories of settings. This makes it easier to port and maintain applications. For example, if a set of applications all run on a target with minimal memory, all applications can import a TCI file that minimizes the DSP/BIOS memory footprint.

❏ Enables use of standard code editing tools. For example, text-based configuration makes it easier to merge changes from multiple developers, compare configurations used by multiple applications, cut and paste between program configurations, and perform repetitive tasks such as creating several similar objects.

❏ Supports branching, looping, and other programming constructs within a configuration procedure.

❏ Allows you to ensure that symbol definitions in the configuration and program sources always match. You can do this by defining variables for use in scripts and generating a C header file from the script to be included by the program source code.

1.3 Creating a Tconf Script

To configure an application in DSP/BIOS, you need a Tconf script.

Typically, you use the graphical DSP/BIOS Configuration Tool (Gconf) to create your initial configuration. This tool acts as a macro recorder for Tconf scripts (TCF files). You see the script change in the right pane of the tool as you change the configuration. Later, you can edit the Tconf script generated by the DSP/BIOS Configuration Tool with a text editor.

Tconf scripts contain statements in the JavaScript language (see Section 3.1, *JavaScript Language Highlights*). These statements are executed to perform design-time (static) application configuration.

❏ If you already have a CDB-based configuration, you need to convert that configuration to a Tconf script. Please read Section A.3, *Converting from CDB Configurations* for instructions.

❏ If you already have a Tconf script for a version of DSP/BIOS prior to DSP/BIOS 5.0, read Section A.3, *Converting from CDB Configurations* for changes you may need to make to your scripts.

This section shows how to use a text editor to create a Tconf script that configures a simple application that prints "Hello World!" to a LOG object named "trace". The source file hello.c of the application is as follows:

```
#include <std.h>
#include <log.h>
#include "hellocfg.h"

/* ======== main ======== */
Void main()
{
    LOG_printf(&trace, "Hello World!");

    /* fall into DSP/BIOS idle loop */
    return;
}
```

The CDB file for the hello application is about 500 KB. Examining this configuration with the DSP/BIOS Configuration Tool would involve browsing through each module and object. In contrast, the equivalent Tconf script contains only a few lines, because it defines only differences between the default DSP/BIOS configuration and the objects used by the application.

To write a Tconf script for an application, follow these steps:

1) **Create a text file with an extension of .tcf.**
   In this example the name of the script is hello.tcf. It is not required that the application source files and the Tconf script have the same base name, but this naming convention simplifies the scripts and their maintenance.

2) **Load a platform.**
   A typical Tconf script begins by loading a platform. In this example, the loaded platform is dsk6416, one of the TI-supplied platforms. Later, in Chapter 4, *Tconf Platform Files*, we describe how you can create and use your own customized platforms.

   Loading a platform defines the target device, external and internal memory objects, various DSP/BIOS default objects, and more.

   ```
   utils.loadPlatform("ti.platforms.dsk6416");
   ```

   If you need to port an application to another platform, the platform name in the utils.loadPlatform() method is the only part of the hello.tcf script you need to change.

3) **Add statements to create objects and set their properties.**
   For this application, we first enable components of the DSP/BIOS kernel that are required for this application. See Section 3.4, *Enabling DSP/BIOS Components* for information on enabling and disabling components of the DSP/BIOS kernel.

   ```
   bios.enableRealTimeAnalysis(prog);
   bios.enableRtdx(prog);
   ```

   Then, we create the "trace" LOG object, which is referred to in hello.c. We also set its size and the type of the log. The last statement in this section sets the size of LOG_system, the system LOG object.

   ```
   var trace;
   trace         = bios.LOG.create("trace");
   trace.bufLen  = 1024;
   trace.logType = "circular";

   bios.LOG_system.bufLen = 512;
   ```

4) **Type the following lines at the end of the file.**

   ```
   // !GRAPHICAL_CONFIG_TOOL_SCRIPT_INSERT_POINT!

   if (config.hasReportedError == false) {
       prog.gen();
   }
   ```

The comment indicates the location for the DSP/BIOS Configuration Tool to insert Tconf statements to match your configuration settings.

The prog.gen() method generates the appropriate CDB, source, header, and linker command files for use in building your application. Section 2.1, *Running a Tconf Script* describes all these generated files. One of the generated files is the hellocfg.h header file, which is included in hello.c. This header file defines the trace variable, which is used in the LOG_printf function call.

The error check prevents an attempt to generate files if any errors occur when running the configuration script.

As a result of these steps, we have the following complete script for the hello application:

```
/* Load the DSK6416 platform. */
utils.loadPlatform("ti.platforms.dsk6416");

/* Enable needed DSP/BIOS features */
bios.enableRealTimeAnalysis(prog);
bios.enableRtdx(prog);

/* Create and initialize a LOG object */
var trace;
trace         = bios.LOG.create("trace");
trace.bufLen  = 1024;
trace.logType = "circular";

/* Set the buffer length of LOG_system buffer */
bios.LOG_system.bufLen = 512;

// !GRAPHICAL_CONFIG_TOOL_SCRIPT_INSERT_POINT!

if (config.hasReportedError == false) {
    prog.gen();
}
```

Chapter 2

Running Tconf Scripts

This chapter describes how to run Tconf scripts.

Topic **Page**

2.1 Running a Tconf Script

Tconf scripts are run by the tconf command-line utility. This utility is available on Solaris, Linux, and Microsoft Windows.

The tconf executable file is located in the xdctools subfolder of the DSP/BIOS installation folder (BIOS_INSTALL_DIR\xdctools). You may want to add this folder to your PATH variable so that you can run tconf without specifying the full path to the utility each time. (See the SetupGuide.html file in the DSP/BIOS installation folder for information about setting the PATH.)

To run the configuration script hello.tcf we developed in Section 1.3, *Creating a Tconf Script*, and to generate files that you compile with the source files of your application, type the following command on your command line:

```
tconf -Dconfig.importPath="C:/dspbios/bios_5_20/packages" hello.tcf
```

2.1.1 Generated Files

When a Tconf script executes successfully, or more specifically when the prog.gen() method is called, Tconf generates a set of files to be compiled with your source code.

The names of these generated files depend upon the argument supplied to prog.gen(). In our example script, we invoked prog.gen() with no argument. If no argument is supplied, the base name for the generated files defaults to the base name of the executed Tconf script—in this example, "hello".

If a string argument is supplied to prog.gen(), that string becomes the base filename for generated files. Using a string argument with prog.gen() is not supported if you are building projects with CCStudio. An alternate way to specify output filenames is to set the config.programName property to the filename string you want.

The following files are generated by the DSP/BIOS Configuration Tool, the Tconf prog.gen() method. In these filenames, "##" is a 2-digit target instruction set architecture (ISA—such as 55 or 64), and *program* is the base name of the Tconf script (hello in our example):

❑ **<program>cfg_c.c.** Source file to define DSP/BIOS structures and properties.

❑ **<program>cfg.h.** Includes DSP/BIOS module header files and declares external variables for objects in the configuration file.

- **<program>cfg.s##.** Assembly source file for DSP/BIOS settings. Since in our example we loaded dsk6416 platform, based on 64 architecture, the name of this file is hellocfg.s64.
- **<program>cfg.h##.** Assembly language header file included by programcfg.s##. In our example, the name of this file is hellocfg.h64.
- **<program>cfg.cmd.** Linker command file.
- **<program>.cdb.** Configuration Data Base (CDB) file. Read-only file. No longer used as a source file.

2.2 The tconf Command-Line Utility

The previous section described the simplest and the most frequent usage of the Tconf command-line utility. This section gives a more detailed overview of the Tconf utility's options and environment variables.

Syntax

```
tconf [-h] [-g] [-p <dir>] [-Dname=value]
      [-js <js options ...>] [script [args ...]]
```

Options

-g	Invoke the Rhino JavaScript debugging tool starting at the beginning of the application's TCF file. Within the Rhino debugger, Break on Exception and Break on Function Entry are enabled.
-g=i	Invoke the Rhino JavaScript debugging tool starting at the beginning of the tconfini.tcf initialization script. Within the Rhino debugger, only Break on Exception is enabled. As a result, if you click Run, the script runs to conclusion without stopping unless an exception occurs.
-b	Run in batch mode. If there is no TCF file specified on the command line, simply exit rather than starting the interactive shell.
-p <dir>	The preferred method for specifying the search path is the -Dconfig.importPath option. (If you choose to use -p instead, the -p option adds the specified folder to the search path used to find internal Tconf files. The search path looks first in the current folder, then in the folder containing the tconf executable file, and then in any folder named using the -p option. See Section 2.2.1, *Environment Array Variables*, page 2-5 for information about how the search path is used.)
-Dname=value	Define variables that can be examined in the script via the global environment array. You can define multiple variables by using the –D option multiple times. The gconf.exe command-line also supports this -D option. See Section 2.2.1, *Environment Array Variables*, page 2-5 for details about the environment array.
-js	Separate run-time options from JavaScript shell options. JavaScript shell options include:

-w	Enable warning reporting.
-f file	Run script in the specified file.

script	Specify a script to run.
args …	Specify arguments to pass to the script via the global arguments array. See Section 2.2.2, *Argument Array Variables*, page 2-7 for details about the arguments array.
-h	Display command-line syntax.

Tconf provides several built-in arrays of variables that are set automatically or based on options in the tconf command line. These arrays are the environment[] array and the arguments[] array.

2.2.1 Environment Array Variables

Tconf creates an array called "environment" and automatically defines a number of variables within that array and sets the initial values for some of them. These variables may also be set by using the -D option on the tconf command line.

Automatically set variables can be used by scripts to obtain information about file names, file locations, and the hardware platform. For example, the following statement gets the name of the script file passed to the tconf utility on the command line.

```
myScript = environment["config.scriptName"];
```

The following variables are automatically part of the environment array.

❏ `environment["config.importPath"]`. This variable defines search locations that Tconf uses to find various files, including platform files and imported scripts. The platform files supplied with DSP/BIOS are located in BIOS_INSTALL_DIR\packages. This folder is added to config.importPath during the Tconf initialization, so in most cases you do not need to set the value of this variable. However, if you create your own platform files or Tconf scripts to be included by other Tconf scripts, and those files are located elsewhere, you should set config.importPath to point to the location of new files.

  For example, if you created your customized platforms in the d:/platforms folder, you would set config.importPath to d:/platforms as follows:

```
 tconf -Dconfig.importPath="d:/platforms" hello.tcf
```

  The command above adds d:/platforms to the beginning of the list of the searched directories, but it does not remove any of the directories already in config.importPath. If you need to add more than one folder to config.importPath, separate them with semicolons (;). For example, this command adds two directories to config.importPath:

```
tconf -Dconfig.importPath="c:/include;d:/platforms" hello.tcf
```

Note that forward slashes (/) must be used on the tconf command line; backslashes (\ ) are not permitted.

The -Dconfig.importPath option can also be specified in Code Composer Studio on the DspBiosBuilder tab of the Build Options dialog.

- `environment["config.rootDir"]`. Contains the folder location of the executable file for the tconf utility. This location is typically BIOS_INSTALL_DIR\xdctools. This variable is always available within a script.
- `environment["config.scriptName"]`. Contains the name of the script passed to the tconf utility on the command line. This variable is always available within a script. If no script was passed, this variable is set to an empty string ("").
- `environment["config.path"]`. Contains the set of directories used to locate internal Tconf components (including the tconf executable and necessary DLLs). This variable is always available within a script. This path may be added to using the -p option on the tconf command line.
- `environment["config.compilerOpts"]`. This variable may define the compiler options used to build the program. The options that may be specified are as follows:
  - -me (big endian)
  - -ml (large data model)
  - --memory_model=huge
  - -mf (far code model)

  If this variable is defined, it sets a corresponding property of the Program object. For example, the following specifies that the program is compiled in big-endian mode:

  ```
  tconf -Dconfig.compilerOpts="-me"
  ```
- `environment["config._`*arch*`_"]`. A variable of this format may be defined using the -D option on the tconf command line, where *arch* may be 28, 54, 55, 62, 64, or 67. If such a variable is defined, it specifies the CPU architecture. Since the CPU is specified by the variable name, the variable need not be set to a value. For example:

  ```
  tconf -Dconfig._55_
  ```

Together, the config.compilerOpts and config.*_arch_* variables support the creation of portable Tconf scripts. The parts of the script that depend upon the compiler options and the architecture can read these variables and configure DSP/BIOS accordingly. For example:

```
if (environment["config._55_"]) {
   if environment["config.compilerOpts"]=="-ml") {
      bios.GBL.MEMORYMODEL = "LARGE";
   }
}
```

- ❏ `environment["config.tiRoot"]`. This variable was used in the previous Tconf releases. However, it is deprecated and will not be supported in subsequent releases.

You can also define additional environment variables and access them from the script. This command line defines three global variables for use within Tconf. The third variable is defined as an empty string.

```
tconf  -Dvar1=value1 -Dvar2=value2 -Dvar3
```

To access these variables within tconf, use the following expressions:

```
environment["var1"]
environment["var2"]
environment["var3"]
```

2.2.2 Argument Array Variables

Tconf creates an array called "arguments" and automatically stores in it arguments passed to the script on the tconf command line. These variables can be used to modify the behavior of a script depending on the command line used to run it.

For example, suppose a command line like the following is used:

```
tconf myscript.tcf 4 2 1
```

The following statements could then be used in myscript.tcf to set variables used when creating various DSP/BIOS objects:

```
numOfTasksToCreate = arguments[0];
numOfReaders = arguments[1];
numOfWriters = arguments[2];
```

2.3 Tconf Operation Modes

The Tconf utility provides the following three operation modes:

❑ DSP/BIOS Configuration Tool. See Chapter 6.

❑ command-line mode. See Section 2.3.1.

❑ GUI debugger. See Section 2.3.2.

❑ interactive mode. See Section 2.3.3.

2.3.1 Command Line Mode

If a script is listed on the command line, as we did in Section 2.1, *Running a Tconf Script* for hello.tcf, Tconf processes the script without entering a JavaScript shell or a GUI debugger.

If the script uses the prog.gen() method, configuration files are generated as a result of running the script. This mode is used for automated program build processes.

The full command-line syntax for this mode is:

```
tconf [-p <dir>] [-Dname=value] [-js <jsshell opts>] script [args ...]
```

Please notice, that the script filename must be supplied.

2.3.2 The GUI Script Debugger

If the -g option is used on the command line, tconf opens the Rhino GUI debugger. Rhino is an open-source implementation of JavaScript written entirely in Java (http://www.mozilla.org/rhino).

The full command-line syntax for the GUI debugger is:

```
tconf -g[=i] [-p <dir>] [-Dname=value] [-js <jsshell opts>] [script [args ...]]
```

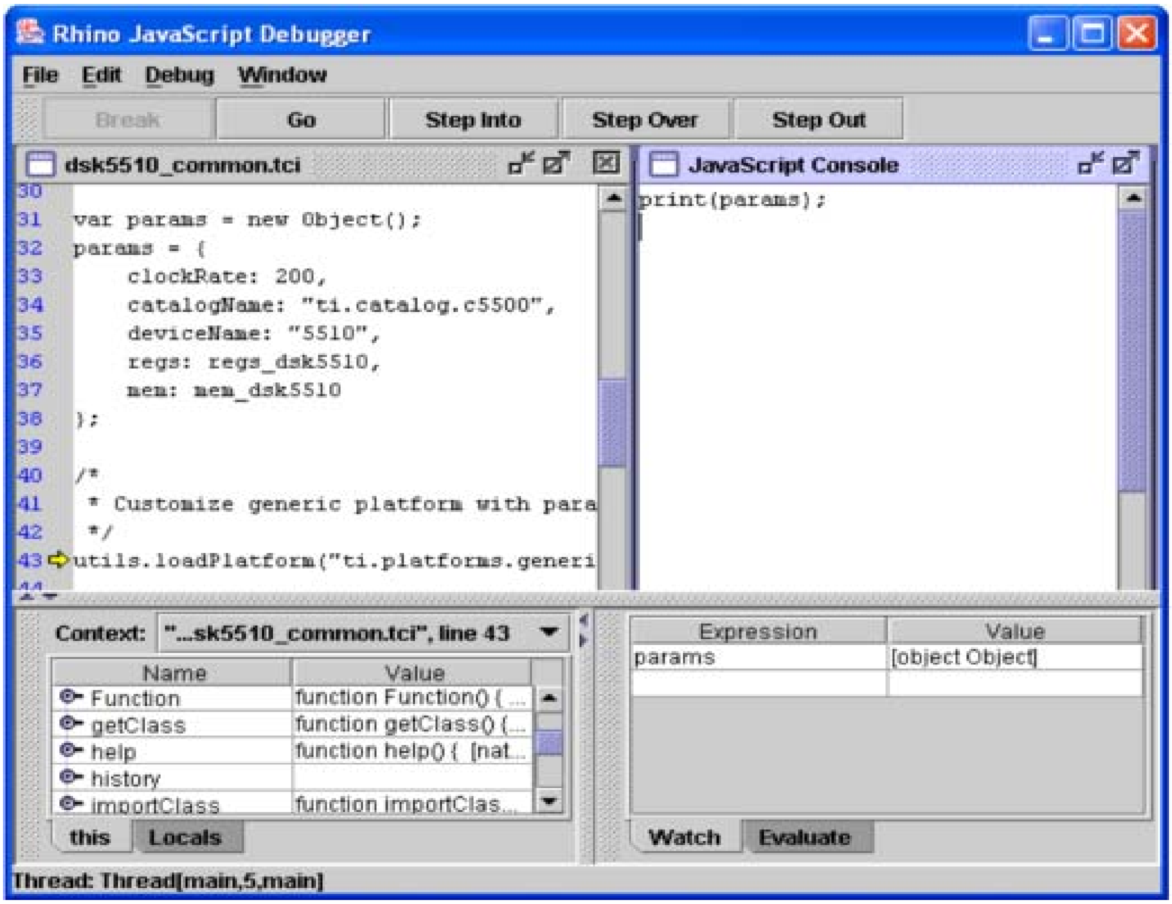
You can use the Rhino debugger to step through the execution of a TCF file. To start this debugger, do either of the following:

❑ Set the Graphical debugger option (-g) in the Debug category of the DspBiosBuilder tab of the **Project->Build Options** window. The Rhino debugger will open when you build the project in CCStudio.

❑ Right-click the script name in the Project Manager, then select the **DSP/BIOS Config->Run in Graphical Debugger** pop-up menu option. (The Text Edit option in the same menu allows you to open the Tconf script in a text editor, instead of using the DSP/BIOS Configuration Tool environment.)

In the Rhino environment, you can use **File->Run** to run a script file. Output from the print() statement is displayed in the JavaScript Console window. You can Step Into and Step Over script functions. This debugger also allows you to watch variables, evaluate arbitrary expressions, and view the current context for the "this" variable and local variables.

Figure 2-1 Rhino GUI Debugger Window



```
var params = new Object();
params = {
    clockRate: 200,
    catalogName: "ti.catalog.c5500",
    deviceName: "5510",
    regs: regs_dsk5510,
    mem: mem_dsk5510
};

/*
 * Customize generic platform with para
 */
utils.loadPlatform("ti.platforms.generi
```

```
print(params);
```

Context: "...sk5510_common.tci", line 43

Name	Value
Function	function Function() { ...
getClass	function getClass() {...
help	function help() { [nat...
history	
importClass	function importClas...

Expression	Value
params	[object Object]

Thread: Thread[main,5,main]

Here are some important hints for using the Rhino debugger:

- ❑ The **Debug** menu contains three check boxes: Break on Exception, Break on Function Enter, and Break on Function Return. If the -g option is used on the tconf command line, Break on Exception and Break on Function Enter are enabled within the debugger. Only Break on Exception is enabled if you use the -g=i option.
- ❑ If you use -g on the command line, the debugger automatically runs the initialization file and breaks at the start of the application's TCF file. If you use -g=i on the command line, the debugger opens initially

to the start of the tconfini.tcf initialization file. With the -g=i option, if you click Run without creating a breakpoint, the script runs to completion without breaks unless an exception occurs.

- When Break on Exception is enabled, non-fatal errors are displayed in exception dialog boxes as they occur.
- Break on Function Enter and Break on Function Return cause the debugger to stop at entry and exit of each JavaScript function. You may want to deselect these options if you just want to run to a specific breakpoint you have set.
- You can set a breakpoint by clicking in the gray column next to the line number of the script. You can only set breakpoints on lines that contain executable statements.
- Choose **Windows->Console** to open the Console window, which receives standard out and standard error. The Tconf script errors seen in the DSP/BIOS Configuration Tool or the CCStudio Build window are shown in the Console window.
- We recommend that you set a breakpoint at the following error check in your TCF file to see any displayed messages in the Console window before the debugger finishes running the script.

```
if (config.hasReportedError == false) {
    prog.gen();
}
```

- Use **Windows->Tile** or **Windows->Cascade** to open windows for the main TCF script, all its included TCI files, and internal Tconf files.
- The Rhino debugger allows you to browse and view Tconf objects, however the list is not always clear or complete. You can add print() statements to the TCF script. The results of print() statements are displayed in the Console window.

2.3.3 Interactive Tconf

If no script is listed on the command line, Tconf enters the interactive JavaScript shell and reads and executes statements you type at the js> prompt. It echoes the results of print statements and expressions to your terminal window.

The full command-line syntax for interactive Tconf is:

```
tconf [-p <dir>] [-Dname=value] [-js <jsshell opts>]
```

The tconf utility provides an interactive JavaScript debugging shell. You enter the interactive shell if you use the tconf command without specifying a script or using either the -g or -b option.

Once you enter interactive mode, you can run a script from the interactive shell using the built-in utils.importFile() method. For example:

```
% tconf
js> utils.importFile("hello.tcf")
```

The result of this statement are generated files, just as if the script were executed from the command line. However, after the execution ends, you are still in the shell.

Alternatively, instead of loading a script, you can create a configuration by simply typing commands. For each line or group of lines that constitutes a complete expression, complete statement, or complete statement block, the debugging shell displays the result on the next line. For example, a portion of a debugging session might look like the following:

```
% tconf
js> utils.loadPlatform("ti.platforms.dsk6416");
[object Program:prog_0]
js> bios.enableRealTimeAnalysis(prog);
js> bios.enableRtdx(prog);
js> var trace;
js> trace = bios.LOG.create("trace");
[object Instance:trace]
js> prog.gen();
true
```

You can also print the value of an expression using the print() method:

```
js> textvar = "hello world";
js> print(textvar);
```

To load the contents of a script file into the JavaScript environment, use a command like the following:

```
js> load("filename.tci");
```

Any statements in the loaded file that are not contained within a function run when the file is loaded. Functions in the loaded file become available for execution by other statements.

To exit from the interactive shell, type `quit` or press CTRL+C. The `quit` command cannot be executed in a Tconf script; it is only available in the interactive shell. The keywords `quit` and `exit` are reserved for future use in Tconf.

Chapter 3

Tconf Language and Object Model

This chapter describes the Tconf language, the object model it uses, and some extensions to JavaScript available in Tconf.

Topic **Page**

3.1 JavaScript Language Highlights

Tconf scripts contain statements in the JavaScript language. These statements are executed to perform design-time application configuration.

This document does not provide details on the syntax of the JavaScript language. However, several concepts are important when using JavaScript for Tconf. This section provides an overview of such concepts. See Section 3.1.3, *JavaScript and Java References*, page 3-3 for JavaScript reference sources.

3.1.1 Language Overview

JavaScript syntax, operators, and flow-control statements are similar to those in the C language. C programmers can easily read JavaScript. It includes if, else, switch, break, for, while, do, and return statements.

JavaScript is a loosely-typed language. Variables in JavaScript are more flexible than variables in C or Java. Variables do not need to be explicitly declared, and the same variable can alternately store any data type. These types are number, string, Boolean value, array, object, function (which is actually an object itself), and null. Operators automatically convert values between data types as necessary.

Variables can be local to a function or global to the entire JavaScript environment. Variable and object names may not contain spaces or punctuation other than "_" or "$". In addition, variable and object names can include numbers but must not begin with a number.

JavaScript does not have pointers and does not deal with memory addresses.

3.1.2 Common Misconceptions About JavaScript

If you've used JavaScript before, you have probably added scripts to a web page. It's important to clear up misconceptions some programmers may have about JavaScript when used outside the context of web pages:

❑ JavaScript is a general-purpose, cross-platform programming language. While it was developed for use in web-browsers, it has a number of features that make it useful for application configuration. It is easy to learn and use, the syntax is similar to C, it is object-oriented, and it is widely documented.

❑ JavaScript is standardized. The language is also called ECMAScript, and the ECMA-262 standard defines the language (see http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM). The basic syntax and semantics of the language are stable and standardized.

- When you use JavaScript in a web page, the objects you use are defined by the Document Object Model (DOM). These objects include window, document, form, and image. The DOM is not part of the JavaScript standard; nor is the DOM part of Tconf.
- Other object models can be defined for use with JavaScript. Instead of the DOM, DSP/BIOS provides the Target Content Object Model (TCOM), with object classes that include Board, Cpu, and Module.
- JavaScript is not a part of Java. These are two different languages that have similar names for historical marketing reasons. However, Tconf does allow scripts to call Java functions to provide file services. JavaScript itself does not provide file services for security reasons on web browsers.
- DSP/BIOS runs JavaScript only on the host PC, UNIX, or Linux machine. JavaScript code is never run on the target DSP.

3.1.3 JavaScript and Java References

This document does not provide details on the syntax of the JavaScript language or on the Java packages that can be used. For reference information, we recommend the following sources:

- *JavaScript, The Definitive Guide, 3rd Edition*, David Flanagan; O'Reilly 1998
- ECMA-262 standard:
  http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM
- Rhino JavaScript interpreter: http://www.mozilla.org/rhino
- Java 2 SDK: http://java.sun.com/j2se/1.3/docs
- java.io package:
  http://java.sun.com/j2se/1.3/docs/api/java/io/package-summary.html

3.2 The Target Content Object Model (TCOM)

Modern scripting languages separate the language syntax from the object model. This division is true of such languages as VBScript, JavaScript, and TCL. The major benefit of this division is that the script language can be standardized independently from its application domain.

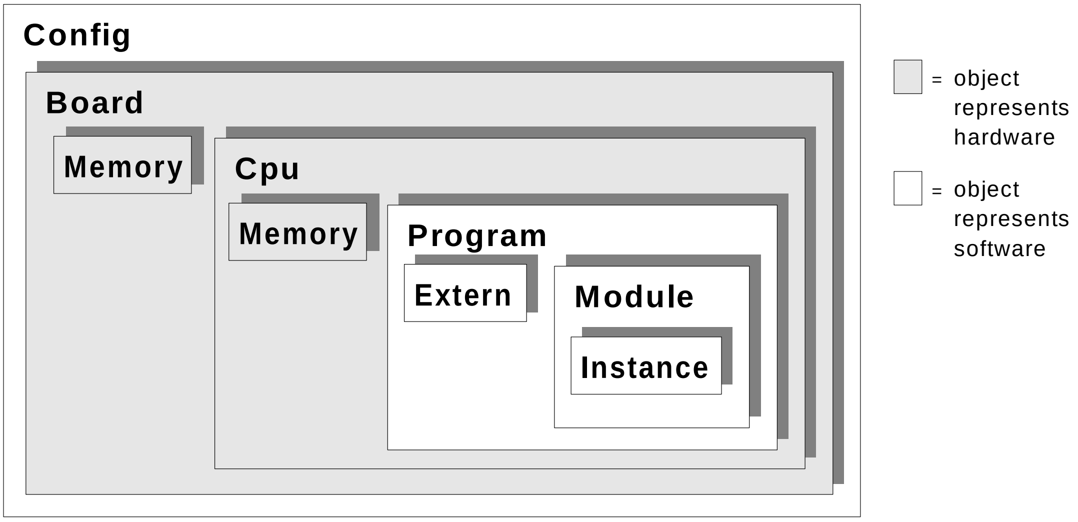
Object models typically define a single top-level object designed to allow navigation via an object hierarchy to all other objects. For example, in a web browser, the object model is called the Document Object Model (DOM) and the top-level object is the "window".

For Tconf, the object model is called the Target Content Object Model (TCOM) and the top-level object is the Config object.

The DOM model cannot be used with Tconf, and the TCOM cannot be used in a web page.

As with the DOM, the TCOM is a hierarchy of “container” objects. These container objects may contain zero or more child objects. For example, within each Program object, there is a Module container that contains an array of Module objects. The TCOM object hierarchy is shown in the following diagram.

Figure 3-1 Target Content Object Model (TCOM)



The top-level Config object contains the entire configuration. Each object class has methods and properties. The entire object tree can be navigated by JavaScript statements.

Notice that a configuration can contain multiple Board objects, Boards can contain multiple Cpu objects, and Cpu objects can contain multiple Program objects. Several methods are provided for populating the hardware and software portions of the object model.

The examples in this document and the examples supplied with DSP/BIOS deal only with configurations with only one Board object, only one Cpu object, and only one Program object. This simplifies configuration scripts, so that the users rarely need to directly access the hardware-specific portions of TCOM. However, for completeness, we describe both portions of TCOM here.

See Section 5.1, *Target Content Object Model Quick Reference*, page 5-2 for a list of the properties and methods of each of these object classes.

3.3 Methods for Loading Other Scripts

A Tconf script can load another script file. When a script file is loaded, any statements that are outside any function are executed. The functions defined in the loaded script are available to be called by the script that loaded the file.

Directory paths specified in JavaScript statements can use either "\\" or "/" as a directory separator. (Directory paths on the tconf command line must use "/".)

Tconf provides the following methods for loading script files:

❑ **load().** An extension to JavaScript that runs JavaScript statements in any file. The file path and full filename must be specified. For example:

```
load("..\\..\\project\\includes\\file.tci");
```

or

```
load("../../project/includes/file.tci");
```

❑ **utils.importFile().** A utility method that attempts to find and load the specified file using a search path. For example:

```
utils.importFile("minFootprint");
```

If you do not specify a file extension, this function looks for the specified file with an extension of .tci. The search sequence used by Tconf is as follows:

a) Any directories specified for config.importPath (in the order specified)

b) Current folder

c) BIOS_INSTALL_DIR\xdctools\include

d) BIOS_INSTALL_DIR\xdctools\packages

The last two locations contain files for internal use that should not be modified or added to.

- **utils.loadPlatform().** A utility method that loads a platform definition. See Chapter 4, Tconf Platform Files for details about platform files.

  In addition to setting platform-specific properties, the utils.loadPlatform() method creates a namespace called "bios" that can be used to shorten references to Module and Instance objects. For example, the standard syntax to reference the bufLen property of the LOG_system object is:

  ```
  prog.module("LOG").instance("LOG_system").bufLen
  ```

  Within the "bios" namespace, Modules and Instances can be referenced directly. For example:

  ```
  bios.LOG_system.bufLen = 128;
  ```

3.4 Enabling DSP/BIOS Components

It is important to note that the utils.loadPlatform() method loads only the minimal set of DSP/BIOS components. Heaps, tasks, real-time analysis, and RTDX are disabled after a platform is loaded. If any of the disabled components is needed, it must be explicitly enabled.

The preferred way to enable or disable DSP/BIOS components is by calling methods from the "bios" namespace:

```
bios.enableRealTimeAnalysis(prog);   // enables RTA
bios.enableMemoryHeaps(prog);        // enables heaps
bios.enableRtdx(prog);               // enables RTDX
bios.enableTskManager(prog);         // enables tasks

bios.disableRealTimeAnalysis(prog);  // disables RTA
bios.disableMemoryHeaps(prog);       // disables heaps
bios.disableRtdx(prog);              // disables RTDX
bios.disableTskManager(prog);        // disables tasks
```

The "prog" variable refers to a Program object from the TCOM. This variable is set by the Tconf environment during initialization.

Alternatively, DSP/BIOS components can be enabled and disabled by directly setting the properties of DSP/BIOS modules.

```
bios.GBL.ENABLEINST = true;     // enables RTA
bios.MEM.NOMEMORYHEAPS = false; // enables heaps
bios.RTDX.ENABLERTDX = true;    // enables RTDX
bios.TSK.ENABLETSK = true;      // enables task

bios.GBL.ENABLEINST = false;    // disables RTA
bios.MEM.NOMEMORYHEAPS = true;  // disables heaps
bios.RTDX.ENABLERTDX = false;   // disables RTDX
bios.TSK.ENABLETSK = false;     // disables tasks
```

If you enable heaps in one or more memory segments, you need to explicitly set the configuration parameters that reference memory segments with heaps. For example, the property MEM.BIOSOBJSEG of the MEM module defines the memory segments for DSP/BIOS objects created at run-time. That parameter is initially set to MEM_NULL. After heaps are enabled as shown in the previous example and the segment MEM_DYN (for example) has a heap enabled, MEM.BIOSOBJSEG still points to MEM_NULL. It has to be explicitly set as follows to use the MEM_DYN heap:

```
bios.MEM.BIOSOBJSEG = prog.get("MEM_DYN");
```

Similarly, MEM.MALLOCSEG and TSK.STACKSEG need to be set explicitly in order to use heaps and tasks.

3.5 Configuration Coding Guidelines

When using Tconf, we recommend using the following coding conventions.

❏ There is one TCF script per application. That script has the same name as the application. For example, if the main source file is hello.c and the executable is hello.out, the name of the main configuration script should be hello.tcf.

❏ Use a file extension of .tci for scripts included by the main script. A different file extension is recommended for included files to support different handling of the main script and included scripts by program build utilities, such as gmake.

❏ Split the main configuration script into platform-dependent and platform-independent pieces. This simplifies porting to new platforms, since only a platform-dependent part needs to be changed.

- ❑ Further determine and define as separate files the pieces of the main script common for many applications. This minimizes code duplication.
- ❑ Create .tci files from the identified parts of the main configuration script. The main configuration script includes these .tci files.

The examples supplied with DSP/BIOS have TCF scripts organized according to these guidelines.

See *DSP/BIOS Tconf Language Coding Standards* (SPRAA67), which is included in the DSP/BIOS installation, for lexical coding conventions recommended for use with Tconf.

3.6 Object and Property Naming and Referencing

JavaScript is object-oriented. The object model is separate from the JavaScript language, but object handling syntax is part of the language.

Objects have properties to define their characteristics. Such properties are actually variables local to the object. You access properties using the dot (.) notation. For example, use `config.hasReportedError` to refer to the hasReportedError property of the Config object.

Objects also have methods that define actions the object can perform. Methods are also accessed using the dot notation. For example, `config.destroy()` deletes the Config object. Such methods are actually functions that are local to the object.

The Target Content Object Model (TCOM) defines object classes that contain an array of zero or more objects. For example, within each Board object, there is a cpu container that contains an array of Cpu objects. You can use the bracket ([ ]) notation or the name of an object to reference an individual object. For example, these notations all reference the clockRate property of a Cpu object:

```
config.boards()[0].cpus()[0].clockRate
config.boards()["board_0"].cpus()["cpu_0"].clockRate
```

If global variables have been declared for board_0 and cpu_0, then the following additional expressions reference the same property:

```
board_0.cpus()[0].clockRate
board_0.cpus()["cpu_0"].clockRate
cpu_0.clockRate
```

While the clockRate property and other properties from the hardware portions of the TCOM can still be accessed using all the notations mentioned here, the preferred way for setting these properties is through the parameters of the generic platform. The generic platform and its parameters are described in Section 4.2, *Creating Custom Platform Files*, page 4-4.

The utils.loadPlatform() method creates a namespace with variables to reference all Module and Instance objects. This simplifies object references as shown by the following references to the LOG_system instance:

❏ Full reference path:

```
config.boards()[0].cpus()[0].programs()[0].module("LOG").instance("LOG_system")
```

❏ Reference path using the `prog` variable automatically created to reference the first Program object:

```
prog.module("LOG").instance("LOG_system")
```

❏ Reference path using the "bios" namespace created by the utils.loadPlatform() method.

```
bios.LOG_system
```

The examples in this document and in DSP/BIOS almost exclusively refer to Module and Instance objects through the "bios" namespace.

Many methods expect an object as a parameter or return an object. When an object is assigned to a variable, that variable internally contains a reference to the object. Objects are not copied when they are assigned; they are stored in one place and referenced by variables. Thus, if multiple variables reference an object, changes to the object made via one variable affect the same object when referenced by another variable.

Some methods return an array of objects. Standard array properties, such as length, can be used with arrays of objects. For example, this statement gets the number of objects in the TSK.instances() array:

```
numtasks = bios.TSK.instances().length
```

These statements create a string listing the names of all Module objects:

```
list = "";
modules = prog.modules();
for (i = 0; i < modules.length; i++) {
    list += modules[i].name + " ";
}
```

The order of objects created within a container array is undefined. You may use JavaScript's array sorting methods, such as join(), sort(), and reverse(), to sort lists of objects. For example, this statement sets a variable to an array of Instance objects with their names in ASCII order:

```
alphatasks = bios.TSK.instances().sort()
```

3.6.1 Module and Instance Property Names

Normally, all objects in a class have the same set of properties. However, each type of Module and Instance object has a different set of properties. Therefore, Tconf handles the properties of Module and Instance objects differently than those of other object classes.

The names of the properties are listed in the *DSP/BIOS Application Programming Interface Reference Guide* for your platform.

You can set and get these property values as you would with properties of other object classes. For example, the following statement sets the size of the LOG_system buffer.

```
bios.LOG_system.bufLen = 16;
```

In general, property names of Module objects are in all uppercase letters. For example, "MEM.STACKSIZE". Property names of Instance objects begin with a lowercase word. Subsequent words have their first letter capitalized. For example, "TSK_idle.stackSize".

3.6.2 Namespace Management

A namespace is the context within which all variables must have unique names. Program objects define a global namespace for all objects contained within the Program object. As a result, all Module, Instance, and Extern objects within a Program object must have unique names.

For example, if the first statement is performed, the second statement fails because the name "audio" is already used.

```
bios.SWI.create("audio");     /* OK */
bios.PIP.create("audio");     /* fails */
```

Any object in a namespace can be retrieved by name. This simplifies object lookup in scripts. For example, these statements look for an object named "audio" and check to see whether it is an Instance object before modifying a property.

In the following example, “instanceof” is a JavaScript operator that returns true if the object is of the specified class. “Instance” is the name of a class.

```
audio = prog.get("audio");
if (audio instanceof Instance) {
    audio.priority = 1;
}
```

3.7 Property Types

The *DSP/BIOS Application Programming Interface Reference Guide* for your platform lists the type of value expected for each property and identifies properties used only for certain DSP platforms. Most types are automatically converted to and from the corresponding JavaScript type.

❏ **Arg.** Arg properties hold arguments to pass to program functions.

❏ **Bool.** DSP/BIOS configurations store Boolean (true/false) values as 1 for true and 0 for false. JavaScript handles both Boolean and integer values. You may use JavaScript to assign either a true value or an integer 1 value to a Boolean Module or Instance property in order to set it to true. Do not set a Boolean value to the quoted string "true" or "false".

  For example, both of these statements disable use of the CLK manager to drive the PRD tick:

```
bios.PRD.USECLK = 0;
bios.PRD.USECLK = false;
```

❏ **EnumInt.** Enumerated integer properties accept a set of valid integer values.

❏ **EnumString.** Enumerated string properties accept a set of valid string values.

❏ **Extern.** Properties that hold function names use the Extern type. In order to provide a function label, use an Extern object (for "external declaration") in JavaScript. All Extern objects within a Program object must have unique names.

  Extern objects may be defined as asm, C, or C++ language symbols. The default language is C.

For example, the following statements create Extern objects for program functions or get the specified object if it already exists. They assign the object to the specified property.

```
bios.task0.fxn = prog.extern("audioFxn", "C");
bios.SYS.ABORTFXN = prog.extern("error");
```

❑ **Int16.** Integer properties hold 16-bit unsigned integer values. The value range accepted for a property may have additional limits.

❑ **Int32.** Long integer properties hold 32-bit unsigned integer values. The value range accepted for a property may have additional limits.

❑ **Numeric.** Numeric properties hold either 32-bit signed or unsigned values or decimal values, as appropriate to the property. When comparing non-integer values, use sufficient digits after the decimal point to match the actual value stored as a Numeric value. For example, if the value of myFloat is 3.456789, the following comparison would evaluate as false:

```
if (myFloat == 3.4568) {  /* FALSE */
   ...
}
```

❑ **Reference.** Properties that reference other objects contain an object reference. For example, properties that specify a MEM segment reference an Instance object contained by the MEM Module object. The following statement gets a reference to a MEM Instance and assigns it to the SWI Object Memory property:

```
bios.SWI.OBJMEMSEG = bios.MEM.instance("EDATA");
```

❑ **String.** String properties hold text strings.

3.8 File Manipulation with Java

For security reasons, JavaScript does not provide any file services. In a web browser, the lack of file services prevents most forms of file access on your computer. In Tconf, file services are provided through the Rhino JavaScript interpreter via LiveConnect. The implementation provides unrestricted use of the java.io package.

Calls to the java.io library from a script look just like JavaScript function calls. Only the function called is written in Java. For example, these statements return the path to a file if it exists:

```
var file = new java.io.File(fileName);
if (file.exists()) {
   return (file.getPath());
}
```

For documentation of the java.io package, see version 1.3.1 of the Java 2 SDK documentation at http://java.sun.com/j2se/1.3/docs. In particular, see the java.io page at http://java.sun.com/j2se/1.3/docs/api/java/io/package-summary.html.

3.9 The print() Method

The print() method is an extension to JavaScript that sends the result of the expression passed to the method to the stdout location. Within the Rhino environment, output from the print() statement is displayed in the JavaScript Console window.

In this example, if any array of objects has been assigned to obj, these statements print a list of the objects in the array.

```
for (var i in obj) {
  print("obj." + i + " = " + obj[i])
}
```

This example uses the print() method to get an array of Board objects and print a list of all the Board objects:

```
boards = config.boards();

for (i = 0; i < boards.length; i++) {
    print("board[" + i + "] = " + board[i].name);
}
```

3.10 Error Handling

Three levels of errors are reported by the host configuration objects. From least to most significant, the levels are:

❑ **Warning.** Probable but unconfirmed error, action completed.

  Warnings are disabled by default, but can be enabled with the config.warn() method or the –w command-line switch. Warnings are written to the stderr location if they are enabled.

❑ **Error.** Confirmed error, action failed.

  The error status of a script is tracked by the config.hasReportedError property. Error messages are always written to the stderr location.

❑ **Exception.** Confirmed error, action failed, non-local return.

  Scripts can throw exceptions. Exceptions thrown by a script or TCOM object can be caught in a script. Uncaught exceptions cause a script to terminate execution. Exceptions are always written to the stderr location, even if they are caught by a script.

In interactive tconf, stderr messages are shown as separate lines without the js> command prompt. In the GUI debugger, stderr messages are shown in the JavaScript Console window.

The exit status from the tconf utility is 0 (success) unless a script specified on the command line could not be run (for example, because the file was not found). If the script runs and results in errors, the tconf exit status is non-zero.

3.10.1 More About Errors

If an error occurs, the config.hasReportedError property is set to true. A script can check this property to determine whether one or more errors has occurred. Error messages are always written to the stderr location.

The following example uses the config.hasReportedError property to determine whether an output configuration file should be generated.

```
if (config.hasReportedError == false) {
  prog.gen();
}
else {
  print("An error has occurred.");
}
```

3.10.2 More About Exceptions

To throw an exception, scripts use the "throw" keyword. This example throws an exception if the lowest-priority task is not the TSK_idle task. The exception goes to stderr.

```
function increasingPri(a, b)
{
    return(a.priority - b.priority);
}

tasklist = prog.module("TSK").instances();
tasklist.sort(increasingPri);

if (tasklist[0].name != "TSK_idle") {
    throw new Error("Idle task should be lowest priority!");
}
```

To catch an exception, a script can use a “try-catch” block. The syntax for such a block is as follows:

```
try {
    // something that might throw an exception //
}
catch (e) {
    // e is the error object thrown //
}
```

For example, the following statements attempt to load a JavaScript file. If the file does not exist, an exception is thrown. When the exception is caught, a message is sent to stderr and the script continues executing. If this script did not catch the exception, the script would terminate execution when the exception occurred.

```
try {
  fileName = prog.name + "_test.tci";
  load(fileName);
}
catch (e) {
  throw new Error(e + "\nNo " + fileName + " file.");
}
```

Chapter 4

Tconf Platform Files

This chapter describes how Tconf scripts should specify the platform to use and how Tconf scripts interact with other files and with operating system issues.

Important: This manual is for use with DSP/BIOS 5.30. See Appendix A for conversion information.

Topic **Page**

4.1 Using TI-Supplied Platform Files

If you use the DSP/BIOS Configuration Tool as described in Chapter 6 to select a platform and create the initial configuration, the reference to a platform file is created automatically in your script.

You can skip the rest of this chapter unless you intend to write TCF scripts from scratch using a text editor or you will need to create custom platform configuration files.

As shown in Section 1.3, *Creating a Tconf Script*, a Tconf script typically starts with a call to the utils.loadPlatform() method. The method loads a platform file using a logical naming convention that matches a partial file path. For example:

```
utils.loadPlatform("ti.platforms.dsk6416");
```

A platform configuration contains Tconf statements that specify board parameters from a software standpoint—for example, the CPU, clock speed, CPU-specific registers, and external memory size and start address.

DSP/BIOS provides a number of platform configuration files. You typically do not need to edit platform configuration files; you simply reference a platform with the utils.loadPlatform() method in your TCF script.

The platform files shipped with DSP/BIOS are located in the following location:

```
BIOS_INSTALL_DIR/packages/ti/platforms/<boardname>/Platform.tci
```

where <*boardname*> is the name of a board such as dsk6416. All provided platform files have a filename of Platform.tci; the folder that contains a particular file identifies it.

The logical platform name used in the utils.loadPlatform() method must match the physical folder location of the desired Platform.tci file. For example, if the <*boardname*> above is dsk6416, the logical platform name is ti.platforms.dsk6416, which matches the /ti/platforms/dsk6416 location.

After you load a platform, you have loaded a minimal DSP/BIOS configuration. One element of that configuration is the memory segments. The table of memory segments for all platforms is provided in Section B.1, *Platform Memory Configurations*.

If you want to change the memory configuration, you should do it in the Platform.tci file you use, not in the application's TCF script or with the DSP/BIOS Configuration Tool.

4.1.1 Referencing a Platform File with utils.loadPlatform()

The search sequence use by utils.loadPlatform() to locate platform files is as follows:

1) User-specified locations, if any, in config.importPath.
2) Default locations in config.importPath, including the current folder.
3) BIOS_INSTALL_DIR\xdctools\include
4) BIOS_INSTALL_DIR\xdctools\packages

If the argument to utils.loadPlatform() contains names separated by a period (.), each name represents a subfolder. The resulting relative path is appended to the items in the preceding search sequence during the attempt to locate the Platform.tci file. For example, suppose the following statement is at the beginning of a script:

```
utils.loadPlatform("ti.platforms.dsk6416")
```

Tconf appends /ti/platforms/dsk6416 to each item in its search sequence, and is therefore able to find the Platform.tci file in the folder BIOS_INSTALL_DIR/packages/ti/platforms/dsk6416/ when you specify that BIOS_INSTALL_DIR/packages should be in the default search path.

The BIOS_INSTALL_DIR/packages folder is automatically added to the search path if you are using CCStudio to build a project or the DSP/BIOS Configuration Tool to save a configuration. You only need to explicitly add BIOS_INSTALL_DIR to the search path if you are running tconf from the command line or a batch file.

Previous versions of DSP/BIOS contained platform specification files with an extension of *.tcp. This convention has been replaced with the use of Platform.tci files stored in platform-specific directories. In previous versions of DSP/BIOS, you used the utils.loadPlatform method to load platform files with filenames of <*Platform_name*>.tcp. For example, utils.loadPlatform("Dsk6416") would load the Dsk6416.tcp file. Tconf currently supports this syntax by locating a ti/platforms/dsk6416 folder containing a Platform.tci file. A message is provided that indicates that this syntax is deprecated and may not be supported in the future.

4.2 Creating Custom Platform Files

If you have a custom board, you can create your own platform configuration and use it just as the supplied TI platform configurations. This saves time by allowing you to define DSP, memory, and clock settings for your hardware once and then reuse the configuration for each application. This encapsulation of board parameters also facilitates the writing of configuration scripts that are portable to other boards.

DSP/BIOS provides more than ten platform configuration template files in the BIOS_INSTALL_DIR\packages\mycompany\platforms\myboard folder. These templates allow you to create a new platform configuration by changing a few values.

To create a new platform definition using a template, follow these steps:

1) Open the readme.txt file in the BIOS_INSTALL_DIR\packages\mycompany\platforms\myboard folder. This file helps you select the right template based on the DSP used on your board. Find your DSP in the "Devices" column and determine the corresponding configuration template from the "Platforms" column.

2) In the same folder, use a text editor to open the TCI file that has a filename that matches the recommended configuration template.

3) Choose File->Save As to save this template to a separate TCI file. (The name doesn't matter at this point. You will change it later.)

4) Edit the mem_ext, device_regs, and params structures to match the external memory, register, and CPU settings for your board. Follow the instructions in the file's comments.

   - The mem_ext array should contain memory objects that describe external memories on your board that can be accessed by DSP. See the MEM Object Properties topic in the online help or API Reference Guide for your platform for more information.

   - The device_regs structure contains platform-specific parameters. See Section 4.3, *Setting Platform Params* for more information.

   - In the params structure, you only need to change only two properties. The first, clockRate, must match the rate of the CPU clock. The second, deviceName, must match the name of the DSP on your board (as listed in the Devices column in the readme.txt file from Step 1). See Section 4.3, *Setting Platform Params* for more information.

5) Save your changes and close the TCI file.

6) If your platform will be widely distributed (for example, if you are a board vendor), see Section 4.2.1, *Creating a Platform for External Distribution* from this point on. Otherwise, continue to the next step.

7) In Windows Explorer, browse to the BIOS_INSTALL_DIR\packages\myplatforms folder. Create a new folder and give it the name of your board. For example, osk2420.

8) Copy your edited configuration template file to this new folder, and rename the copied file Platform.tci.

9) Open the Platform.tci file and edit the "!NAME!" tag. Change it to use the name of your board. This must exactly match the name you gave the new folder in Step 7. For example, if Platform.tci is in BIOS_INSTALL_DIR\packages\myplatforms\osk2420, the !NAME! tag line must read as follows. (Don't uncomment the !NAME! and !DESCRIPTION! tags.)

```
 *  !NAME!  myplatforms.osk2420
```

   Alternately, you could name the platform "osk2420" and add packages\myplatforms to your config.importPath definition.

10) Edit the text description in the "!DESCRIPTION..." comment. Do not change the text between the two exclamation marks (for example, !DESCRIPTION 55XX!). Write a description to be displayed when this platform is highlighted in the New DSP/BIOS Configuration dialog. For example:

```
*  !DESCRIPTION 55XX!  OSK2420 as configured by our group
```

11) Save and close the Platform.tci file.

12) In CCStudio, choose File->New->DSP/BIOS Configuration.

13) Click the Browse Platforms button and browse to the BIOS_INSTALL_DIR\packages\myplatforms folder. You should see the folder named for your board in the list. Then, click OK.

   You should see your board listed in the New DSP/BIOS Configuration dialog. If it is not listed, be sure the Browse Platforms location points to the folder above the platform-named folder you created. Also check the tags in the Platform.tci file against the instructions above.

14) Select the platform and click OK to begin creating your configuration.

Alternately, you can use the content created in Step 4 to define the platform in the application's TCF script by copying the edited configuration template to the beginning of your application's script. Examples that define a platform in this manner are provided with the DSP/BIOS examples for simulators. See the

BIOS_INSTALL_DIR\packages\ti\bios\examples\basic\bigtime\sim6416\bigtime.tcf file for an example. This method is best used when a single application will use this platform configuration.

4.2.1 Creating a Platform for External Distribution

If your platform definition will be widely distributed (for example, if you are a board vendor), you should follow these special rules for storing and editing your Platform.tci file so that namespace conflicts will not occur between your platforms and other vendor's platforms.

1) First, create your platform file by following Steps 1 through 5 in the general platform creation instructions.
2) In Windows Explorer, browse to the BIOS_INSTALL_DIR\packages folder.
3) Create a new folder with the name of your company. For example, "boardDepot".
4) Within this company-named folder, create a folder called "platforms".
5) Within the "platforms" folder, create a new folder and give it the same name as your board. For example, osk2420.
6) Copy your edited configuration template file to the folder named for your board and rename the copied file Platform.tci.
7) Open the Platform.tci file and edit the "!NAME!" tag. Change it to the full logical name of your platform. This must exactly match the path from your company-named folder to the board-named folder. For example, if Platform.tci is in BIOS_INSTALL_DIR\packages\boardDepot\platforms\osk2420, the !NAME! tag line must read as follows. (Don't uncomment the !NAME! and !DESCRIPTION! tags.)

```
 *  !NAME!  boardDepot.platforms.osk2420
```

8) Edit the text description in the "!DESCRIPTION..." comment. Do not change the text between the two exclamation marks (for example, !DESCRIPTION 55XX!). Write a description to be displayed when this platform is highlighted in the New DSP/BIOS Configuration dialog. For example:

```
*  !DESCRIPTION 55XX!  OSK2420 as configured by BoardDepot
```

9) Save and close the Platform.tci file.
10) In CCStudio, choose File->New->DSP/BIOS Configuration.

11) Click the Browse Platforms button and browse to the BIOS_INSTALL_DIR\packages folder. You should see the folder named for your company in the list. Click OK.

    You should see your board listed in the appropriate tab of the New DSP/BIOS Configuration dialog. If it is not, make sure the Browse Platforms location points to the folder above your company-named folder. Also check the tags in the Platform.tci file against the instructions above.

12) When you distribute your Platform.tci file, you should ensure it is installed in the same location within the DSP/BIOS distribution. If you use another location, you must instruct users to browse to that location with this Gconf dialog. The most important part of these steps is that the !NAME! tag logical name must match the folder tree levels below the location users browse to (with backslashes replaced by periods).

4.3 Setting Platform Params

Most of the work done by a Platform.tci file is to set up the "params" JavaScript object. A sample params declaration looks like this:

```
params = {
    clockRate:      600,
    catalogname:    "ti.catalog.c6000",
    devicename:     "6416",
    regs:           device_regs,
    mem:            mem_ext
};
```

The params object has the following properties.

❑ **clockRate.** Float. Specifies the CPU’s clock rate in MHz. This property must be set.

❑ **deviceName.** String. Specifies the name of the DSP in use. This property must be set. The list of devices and their configurations is provided in Section B.3, *Device Memory Configurations*. Devices listed there are valid values for this property.

❑ **catalogName.** String. Specifies the catalog for the DSP. This property must be set. The following TI catalogs are supplied with DSP/BIOS: "ti.catalog.c2800", "ti.catalog.c5400", "ti.catalog.c5500" (includes OMAPs), and "ti.catalog.c6000".

- **regs.** An object with target-dependent properties listed in Table 4-1. This property is required for the devices listed in Table 4-1, and can be left unset for other devices. For example:

```
device_regs = {
   l2Mode: "4-way cache (32k)"
};
```

- **mem.** An array that describes the external memory. This property is optional. For example:

```
var mem_ext = [];

mem_ext[0] = {
    comment: "Defines space for DSP's off-chip memory",
    name:    "EXT_1",
    base:    0x80000000,
    len:     0x01000000,
    space:   "code/data"
};
```

Tconf does not check the consistency of the "regs" and "mem" settings. You must ensure that the properties of the off-chip part of the memory map correspond to the values in the platform definition.

Table 4-1 Target-Dependent Properties of regs

DSP	Properties	Description
28x	Int pllcr	The four least significant bits of this register (pllcr[3:0]) define the clocking ratio between pins X1/XCLKIN and X2, and the system clock. Valid values for this register and further descriptions can be found in Section 3.8 of the data manual for the TMS320F2812 DSP (SPRS174).
54x DM270 DM310 DM320	Int clkmd;	The clkmd property defines the behavior of the clock generator. The clock generator is described in Section 3.10 of the data manual for any C54x device (for example, SPRS095 for C5416). The DSP/BIOS configuration parameter GBL.CLKMD, which corresponds to the register clkmd, is mentioned in Chapter 2 of the *C5000 DSP/BIOS API Reference Guide* (SPRU404).
54x DM270 DM310 DM320	Int pmst;	The Processor Mode Status (PMST) register is used in DSP/BIOS configurations mainly to define memory map. This register is described in Section 3.5 of the data manual for any C54x device (for example, SPRS095 for C5416), and in Chapter 2 of the *C5000 DSP/BIOS API Reference Guide* (SPRU404).

Table 4-1 Target-Dependent Properties of regs

DSP	Properties	Description
54x DM270 DM310 DM320	Int swwsr, swcr;	The Software Wait-State Register (SWWSR) and the Software Wait Control Register (SWCR) control the software-programmable wait-state generator. These registers are described in Section 3.6 of the data manual for any C54x device (for example, SPRS095 for C5416), and in Chapter 2 of the *C5000 DSP/BIOS API Reference Guide* (SPRU404).
54x DM270 DM310 DM320	Int bscr;	The Bank Switching Control Register (BSCR) controls programmable bank-switching logic. This register is described in Section 3.6 of the data manual for any C54x device (for example, SPRS095 for C5416), and in Chapter 2 of the *C5000 DSP/BIOS API Reference Guide* (SPRU404).
5501 5502 DA295 DA300	Int pllm, plldiv0;	The pllm and plldiv0 registers define the behavior of the system clock generator on the C5502. The pllm register multiplies and the plldiv0 register divides the input clock. The combination of pllm and plldiv0 can generate outputClock = (pllm/plldiv0)*inputClock. They are described in the Section 3.9 of the datasheet for the C5502 (SPRS166).
5561	Int st3_55;	st3_55 is the status register. Its main purpose in DSP/BIOS configuration is to define the memory map on the C5561 device. This register is described in the Chapter 2 of the *C55xx CPU Reference Guide* (SPRU371).
5503 5507 5509A 5510A DA255	Int clkmd;	Defines the behavior of the clock generator. The DSP/BIOS configuration parameter GBL.CLKMD, which corresponds to the register clkmd, is mentioned in Chapter 2 of the *C5000 DSP/BIOS API Reference Guide* (SPRU404).
6x1x DRI300 DM64x 64+	String l2Mode;	Defines the mode for the L2 cache. This property corresponds to the GBL.C621XCCFGL2MODE parameter on C621x and C671x devices, and the GBL.C641XCCFGL2MODE parameter on the C641x devices. It is described in Chapter 2 of the *C6000 DSP/BIOS API Reference Guide* (SPRU403).
64+	String l1PMode, String l1DMode	Defines the mode for the L1 program cache and L1 data cache. The valid values are "0k", "4k", "8k", "16k", and "32k".

After the "params" object is set up, the last statement of Platform.tci invokes the utils.loadPlatform() method as follows:

```
/* Customize generic platform with parameters above. */
utils.loadPlatform("ti.platforms.generic", params);
```

When the first argument to utils.loadPlatform() is "ti.platforms.generic", the params object is required as the second argument.

The following subsections provide example Platform.tci files for various devices.

4.3.1 Example for ’C2812

```
var mem_ext = [];
mem_ext[0] = {
    comment: "Defines space for the DSP's off-chip memory",
    name:      "SRAM",
    base:      0x80000,
    len:  0x10000,
    space:     "data"
};

var device_regs = {
    pllcr: 0xa
};

var params = {
    clockRate: 150;
    catalogname:"ti.catalog.c2800",
    devicename:"2812",
    regs: device_regs,
    mem: mem_ext
};

utils.loadPlatform("ti.platforms.generic", params);
```

4.3.2 Example for ’C5416

```
var mem_ext = [];
mem_ext[0] = { comment: "External program memory 0",
               name:"EPROG0",
               base:0x18000,
               len:  0x8000,
               space: "code"
};
mem_ext[1] = {  comment: "External program memory 1",
               name:"EPROG1",
               base:0x28000,
               len:  0x8000,
               space: "code"
};
mem_ext[2] = {  comment: "External program memory 2",
               name:"EPROG2",
               base:0x38000,
               len:  0x8000,
               space: "code"
};
mem_ext[3] = {  comment: "Interrupt Vectors",
               name:"VECT",
               base:0xff80,
               len:  0x0080,
               space: "code"
};

var device_regs = {
    clkmd: 0x9007,
    pmst:  0xffa0,
    swwsr: 0x4492,
    swcr:  0x0,
    bscr:  0xa002
};
var params = {
    clockRate: 160;
    catalogname:"ti.catalog.c5400",
    devicename:"5416",
    regs: device_regs,
    mem: mem_ext
};

utils.loadPlatform("ti.platforms.generic", params);
```

4.3.3 Example for ’C5510

```
var mem_5510 = [];
mem_5510[0] = { name:   "SDRAM",
                base:   0x50000,    /* byte address */
                len:    0x3b0000,   /* length in bytes */
                space:  "code/data"
};

var regs_5510 = {
   clkmd: 0x2cd0,
};

var params = {
   clockRate: 200;
   catalogname:"ti.catalog.c5500",
   devicename:"5510",
   regs: regs_5510,
   mem: mem_5510
};

utils.loadPlatform("ti.platforms.generic", params);
```

4.3.4 Example for OMAP 1510

```
var params = {
    clockRate: 120,
    catalogname:"ti.catalog.c5500",
    devicename:"1510",
};

utils.loadPlatform("ti.platforms.generic", params);
```

4.3.5 Example for ’C6416

```
var device_regs ={
    l2mode: "4-way cache (0k)"
}

var params = {
    clockRate: 600,
    devicename:"6416",
    catalogname:"ti.catalog.c6000",
    regs: device_regs
}

utils.loadPlatform("ti.platforms.generic", params);
```

4.3.6 Example for ’C6713

```
var mem_ext = [];
mem_ext[0] = { name:   "SDRAM",
               base:   0x80000000,
               len:    0x00800000,
               space:  "code/data"
};

var device_regs = {
    l2Mode: "SRAM"
};

var params = {
    clockRate: 225.0000,
    catalogname:"ti.catalog.c6000",
    devicename:"6713",
    regs: device_regs,
    mem: mem_ext
};

utils.loadPlatform("ti.platforms.generic", params);
```

4.3.7 Example for ’C64+ Devices

```
var mem_ext = [];

mem_ext[0] = {
    comment: "256Mbytes of the DSP's off-chip memory",
    name: "EXT",
    base: 0x80000000,
    len:  0x10000000,
    space: "code/data"
};
var device_regs = {
    l1PMode: "32k",
    l2DMode: "32k",
    l2Mode: "0k"
};
var params = { clockRate: 600,
               catalogName: "ti.catalog.c6000",
               deviceName: "DM420",
               regs: device_regs,
               mem: mem_ext
};
utils.loadPlatform("ti.platforms.generic", params);
```

4.3.8 Example for ’C67+ Devices

```
var mem_ext = [];

mem_ext[0] = {
    comment: "Defines space for DSP's SDRAM off-chip memory",
    name: "EXT0",
    base: 0x80000000,
    len:  0x01000000,
    space: "code/data"
};
mem_ext[1] = {
    comment: "Defines space for DSP's SRAM off-chip memory",
    name: "EXT1",
    base: 0x90000000,
    len:  0x01000000,
    space: "code/data"
};
var params = {   clockRate: 300,
                 catalogName: "ti.catalog.c6000",
                 deviceName: "6727",
                 mem: mem_ext
};
utils.loadPlatform("ti.platforms.generic", params);
```

4.4 Using Custom Platform Files

To use a custom platform located in the mycompany/platforms/myboard tree, place the following statement at the beginning of your *.tcf script. If you have renamed mycompany or myboard, change this statement as needed.

```
utils.loadPlatform("mycompany.platforms.myboard");
```

If, for any reason, you cannot store your platform files under BIOS_INSTALL_DIR/packages, you need to add the path to your platforms by setting config.importPath. For example, if your c:/BIOS-5-20/packages/myplatforms folder contains a subfolder called mycompany/board1 containing a Platform.tci file, you would set your config.importPath as follows:

```
tconf -Dconfig.importPath="c:BIOS-5-20/packages/myplatforms" hello.tcf
```

Your script should then load your platform file as follows:

```
utils.loadPlatform("mycompany.board1");
```

Chapter 5

Tconf Object Model Reference

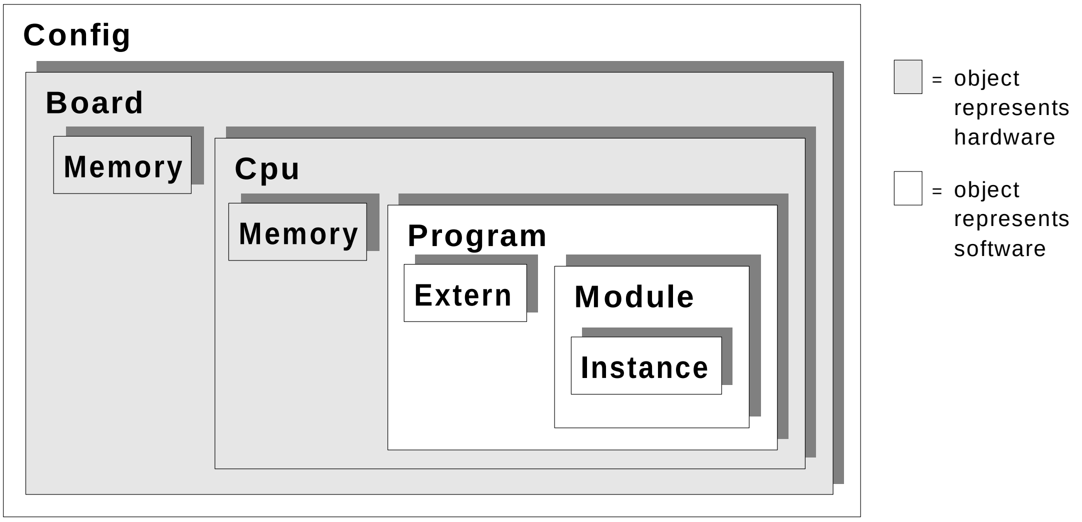
This chapter provides reference information about the Target Content Object Model.

Topic **Page**

5.1 Target Content Object Model Quick Reference

The Target Content Object Model (TCOM) is a hierarchy of “container” objects. These container objects may contain zero or more child objects. For example, within each Module object, there is a container that contains a set of Instance objects. The TCOM is shown in the following diagram.

Figure 5-1 Target Content Object Model (TCOM)



This table summarizes the methods and properties of the objects in the Target Content Object Model. For details, see the sections on each class.

Table 5-1 Target Content Object Model Summary

Object Type	Objects Contained	Methods	Properties	See Page

Config	Board	board()
		boards()
		create()
		destroy()
		warn()
			hasReportedError
			name
				Page 5–4

Board	Cpu
	Memory
		cpus()
		create()
		destroy()
		getMemoryMap()
			boardFamily
			boardRevision
			config
			name
				Page 5–8

Table 5-1 Target Content Object Model Summary (Continued)

Object Type	**Objects Contained**	**Methods**	**Properties**	**See Page**

Cpu	Program
	Memory
		create()
		destroy()
		getMemoryMap()
		program()
		programs()
			board
			clockRate
			deviceName
			id
			name
			attrs.cpuCore
			attrs.cpuCoreRevision
			attrs.dataWordSize
			attrs.minDataUnitSize
			attrs.minProgUnitSize
				Page 5–11

Program	Module
	Extern
		extern()
		externs()
		destroy()
		gen()
		get()
		module()
		modules()
			cpu
			name
			prog.build.target.model.codeModel
			prog.build.target.model.dataModel
			prog.build.target.model.endian
				Page 5–16

Memory	-- none --	-- none --	comment name space base len	Page 5–22
Extern	-- none --	-- none --	language name	Page 5–23

Module	Instance	create()
		instance()
		instances()
			-defined in API Reference-
			name
			program
				Page 5–24

Instance	-- none --	destroy()
		references()
			-defined in API Reference-
			module
			name
				Page 5–28

Note that the create() and destroy() methods act on different objects. While the create() methods create a child object for the specified object, the destroy() methods destroy the specified object itself.

5.2 Config Class

Table 5-2 Config Class Summary

Object Type	Contains	Methods	Properties

Config	Board	board()
		boards()
		create()
		destroy()
		warn()
			hasReportedError
			name

The Config object is the top-level container for an entire system configuration. Each configuration has one and only one Config object. The Config object has methods and properties for debugging, error handling, and host configuration memory management.

A default Config object and a global variable called "config" are automatically created by the startup script. Should you ever need to create a Config object explicitly, use a statement similar to the following:

```
/* create global context for configuration scripts */
var config = new Config("config_0");
```

board() Method

Syntax: board("*name*")

Parameters:

name	Name of object to get. Required.

Returns: object, or null if error occurs

Description: The board() method returns the Board object specified by the name parameter.

If there is no Board object with the specified name, board() returns null.

boards() Method

Syntax: boards()

Parameters: none

Returns: Array of all Board objects contained in the Config object

Description: The boards() method gets an array of all the Boards contained within the Config object.

This method is performed within the generic platform definition. Application scripts should not need to use this method.

Example:

```
/* get an array of all boards in config */
boards = config.boards();

/* print a list of the names of all boards in config */
for (i = 0; i < boards.length; i++) {
    print("board[" + i + "] = " + board[i].name);
}
```

create() Method

Syntax: create("*board_name*" [, "*board_type*"])

Parameters:

board_name	Required name for new Board object.
board_type	Optional sub-type of board.

Returns: new Board object, or null if error occurs

Description: The config.create() method creates a new Board object within the Config object.

This method is performed within the generic platform definition. Application scripts should not need to use this method.

The first parameter is the name to give the new Board object. The name must be unique among the boards. This parameter is required.

The second parameter defines the sub-type of board to create. This parameter is optional. If you provide a board_type that matches an JavaScript constructor function that has been loaded, that constructor runs to define properties for the Board object and to create the standard Cpu object for the board.

The order of objects created within a container array is undefined. You may use JavaScript's array sorting methods to get sorted lists of objects.

destroy() Method

Syntax: destroy()

Parameters: none

Returns: true if successful; false if failed

Description: The destroy() method destroys the specified Config object.

This method fails and returns false if the object is either referenced by another object or contains objects.

Notice that while the create() method creates an object one layer lower in the hierarchy than the object whose method is used, the destroy() method deletes the actual object whose method is used.

You will probably not need to use the destroy() method when writing configuration scripts from scratch.

Examples:

```
/* Fails if config contains a board */
config.destroy();
/* So, destroy the previously created board */
board.destroy();
/* Succeeds if config is now empty and unreferenced */
config.destroy();
```

warn() Method

Syntax: warn()

Parameters: true or false

Returns: Previous warning setting (true or false)

Description: The warn() method enables and disables warnings.

Warnings are disabled by default, but can be enabled with the warn() method or the –w command-line switch. See Section 3.10, *Error Handling*, page 3-14 for information about warnings, errors, and exceptions.

If you enable warnings, you will notice that the Rhino interpreter provides a warning if the "var" keyword is omitted from a variable declaration. You can ignore these messages. Omitting the "var" keyword is permitted by the standard and is common practice in JavaScript.

In command-line mode, warnings are written to the stderr location. In interactive tconf, warnings are shown as separate lines without the js> command prompt. In the GUI debugger, warnings are shown in the JavaScript Console window.

Example:

```
config.warn(true);
```

hasReportedError Property

The hasReportedError property contains a Boolean value that indicates whether any error or exception has occurred during the current session.

This property is gettable only. It is initially set to false and becomes true if an error or exception occurs. This property is never reset to false during a session.

Warnings do not affect the value of this property.

Example:

```
if (config.hasReportedError == true) {
   print("Error has occurred");
}
```

name Property

The name property of an object holds the name of that object. This property is gettable only. It is set when the object is created.

There is only one Config object, so its name is unique by definition.

5.3 Board Class

Table 5-3 Board Class Summary

Object Type	Contains	Methods	Properties

Board	Cpu
	Memory
		cpu()
		cpus()
		destroy()
		getMemoryMap()
			boardFamily
			boardRevision
			config
			name

A configuration may contain one or more Board objects. Board objects may contain one or more Cpu and Memory objects. Board objects have properties for storing information about the board hardware used.

A default Board object is created by the utils.loadPlatform() method. Additional Board objects can be created with the config.create() method.

cpu() Method

Syntax: cpu("*name*")

Parameters:

name	Name of object to get. Required.

Returns: object, or null if error occurs

Description: The cpu() method returns the Cpu object specified by the name parameter.

If there is no object with the specified name in the specified Board object, cpu() returns null.

cpus() Method

Syntax: cpus()

Parameters: none

Returns: Array of all Cpu objects contained in the specified Board object

Description: The cpus() method gets an array of all the cpus contained within the Board object.

Example:

```
/* get board containing this cpu */
var cpu = config.boards()[0].cpus()[0];
var board = cpu.board;

/* get all cpus on this board */
var cpus = board.cpus();
```

destroy() Method

Syntax: destroy()

Parameters: none

Returns: True if successful; false if failed.

Description: The destroy() method destroys the specified object.

This method fails and returns false if the object is either referenced by another object or contains objects.

Notice that while the create() method creates an object one layer lower in the hierarchy than the object whose method is used, the destroy() method deletes the actual object whose method is used.

You will probably not need to use the destroy() method when writing configuration scripts from scratch.

Examples:

```
/* Fails if config contains a board */
config.destroy();
/* So, destroy the previously created board */
board.destroy();
/* Succeeds if config is now empty and unreferenced */
config.destroy();

/* Two ways to destroy a board named DSK6416 */
boards.DSK6416.destroy()
boards["DSK6416"].destroy();
```

getMemoryMap() Method

Syntax: getMemoryMap()

Parameters: none

Returns: The full physical memory map of the platform, including on-chip and off-chip segments.

Description: This method returns the full physical memory map of the platform. This map is an array of both on-chip and off-chip memory segments. The map is returned as an array of Memory objects.

The board.getMemoryMap() function for a particular board is defined in the platform definition. For an example, see the generic platform file. In the supplied platform definitions, on-chip memory is before off-chip memory in the array.

boardFamily Property

The boardFamily property contains a string that identifies the type of board. Example strings are “evm62”, “dsk54”, and "sim55".

This property is gettable only. It is set if the board_type argument to the config.create() method matches a constructor function, and that constructor function sets the boardFamily property.

Example:

```
/* load platform-dependent configuration info */
try {
    utils.importFile("dss_" + prog.cpu.board.boardFamily);
}
catch (e) {
    throw new Error(e + "\nDSS doesn't support the '" +
        prog.cpu.board.boardFamily + "' board");
}
```

boardRevision Property

The boardRevision property contains an optionally defined string that identifies the board revision number. Example strings are "1.0" and "2.1".

This property is gettable only. It is set if the board_type argument to the config.create() method matches a constructor function, and that constructor function sets the boardRevision property.

config Property

The config property holds the Config object that contains the Board.

Since there is only one Config object, this Config object contains all Boards in the configuration.

This property is gettable only. It is set when the Board object is created.

name Property

The name property of an object holds the name of that object. This property is gettable only. It is set when the object is created.

Names of Board objects must be unique.

5.4 Cpu Class

Table 5-4 Cpu Class Summary

Object Type	Contains	Methods	Properties

Cpu	Program
	Memory
		create()
		destroy()
		getMemoryMap()
		program()
		programs()
			board
			clockRate
			deviceName
			id
			name
			attrs.cpuCore
			attrs.cpuCoreRevision
			attrs.dataWordSize
			attrs.minDataUnitSize
			attrs.minProgUnitSize

A Board object may contain one or more Cpu objects. Cpu objects may contain one or more Program and one or more Memory objects. Cpu objects have properties for storing information about the Cpu type and memory handling behavior.

Configurations for multi-core DSPs should have a single Cpu object. Configurations for boards with multiple DSPs should have multiple Cpu objects.

A default Cpu object is created by the startup script. Additional Cpu objects can be created with the create() method of a Board object.

create() Method

Syntax: create("*prog_name*"])

Parameters: prog_name Required name for new Program object.

Returns: new Program object, or null if error occurs

Description: The create() method for a Cpu object creates a new Program object within the Cpu object.

The parameter is the name to give the new Program object. The name must be unique among the Program objects for this Board and within the Program object's namespace. This parameter is required.

The order of objects created within a container array is undefined. You may use JavaScript's array sorting methods to get sorted lists of objects.

Example:

```
/* create a C54 Cpu object for the board */
utils.importFile("C54");
config.boards()[0].create("cpu_0", "C54");

/* create a Program object for the default Cpu */
config.boards()[0].cpus()[0].create("myApp");

/* create "short-cut" for program config scripts */
var prog = config.boards()[0].cpus()[0].programs()[0];
```

destroy() Method

Syntax: destroy()

Parameters: none

Returns: true if successful; false if failed

Description: The destroy() method destroys the specified object.

This method fails and returns false if the object is either referenced by another object or contains objects.

Notice that while the create() method creates an object one layer lower in the hierarchy than the object whose method is used, the destroy() method deletes the actual object whose method is used.

You will probably not need to use the destroy() method when writing configuration scripts from scratch.

Examples:

```
config.boards["DSK6416"].cpus["C6416"].destroy();
```

getMemoryMap() Method

Syntax: getMemoryMap()

Parameters: none

Returns: The on-chip memory map of the platform.

Description: This method returns the on-chip memory map of the platform. The map is returned as an array of Memory objects.

The cpu.getMemoryMap() function for a particular cpu is defined in the *tci file for the particular DSP. For an example, see the C5402.tci file.

program() Method

Syntax: program("*name*")

Parameters: name — Name of Program object to get. Required.

Returns: object, or null if error occurs

Description: The program() method returns the Program object specified by the name parameter.

If there is no object with the specified name in the Cpu object, program() returns null.

programs() Method

Syntax: programs()

Parameters: none

Returns: Array of all Program objects contained in the specified Cpu object

Description: The programs() method gets an array of all the Program objects contained within the specified Cpu object.

Example:

```
/* create "short-cut" for program config scripts */
var prog = config.boards()[0].cpus()[0].programs()[0];
```

board Property

The board property holds the Board object that contains the Cpu object.

This property is gettable only. It is set when the Cpu object is created.

Examples:

```
utils.importFile("myApp_" + prog.cpu.board.boardFamily);

function checkMIPS(cpu) {
    /* get board containing this cpu */
    var board = cpu.board;
    /* get all cpus on this board */
    var cpus = board.cpus();
    var MIPS = cpu.MIPS;

    for (var i = 0; i < cpus.length; i++) {
        /* check all cpus against cpu.MIPS */
        if (cpus[i].MIPS != MIPS) {
            throw new Error("All " + board.name +
               " Cpus must run at the same rate.");
        }
    }
}
```

clockRate Property

The clockRate property of an object holds the value of the clock rate of the board in MHz. This property is typically set in the platform definition.

Example:

```
/* Define clock rate for CPU in MHz */
config.board("dsk5402").cpu("cpu_0").clockRate = 100.0;
```

deviceName Property

The deviceName property specifies a name for this particular CPU on the board. The name may be a number, such as 5402 or 6416. Some deviceNames also contain letters, such as DM640 and OMAP1510.

id Property

The id property specifies a unique id for this particular CPU on the board. This property is intended for future use.

name Property

The name property of an object holds the name of that object. This property is gettable only. It is set when the object is created.

Names of Cpu objects must be unique within the Board object that contains them.

attrs.cpuCore Property

The attrs.cpuCore property contains the two-digit Cpu platform followed by two zeros. Currently, it may be set to one of the following: 2800, 5400, 5500, 6200, 6400, or 6700.

This property is gettable only. It is set if the cpu_type argument to the Board object's create() method matches a constructor function, and that constructor function sets the attrs.cpuCore property.

attrs.cpuCoreRevision Property

The attrs.cpuCoreRevision property contains an optional revision number of a particular Cpu part. Example values are 1.0, 2.1, and R2.

This property is gettable only. It is set if the cpu_type argument to the Board object's create() method matches a constructor function, and that constructor function sets the attrs.cpuCoreRevision property.

attrs.dataWordSize Property

The attrs.dataWordSize property contains the size of a word (int) on this Cpu in 8-bit units. On 'C5000 platforms, attrs.dataWordSize is two 8-bit units. On 'C6000 platforms, attrs.dataWordSize is four 8-bit units.

This property is gettable only. It is set if the cpu_type argument to the Board object's create() method matches a constructor function, and that constructor function sets the attrs.dataWordSize property.

Example:

In this example, the application's data frame size (FRAME_SIZE) is measured in 16-bit samples. However, DSP/BIOS pipe objects (DSS_rxPipe) have frame sizes measured in the platform-dependent word size. So, the attrs.dataWordSize property (in 8-bit units) is used to convert from the application's frame size to the DSP/BIOS frame size.

```
var FRAME_SIZE = 64;           /* in 16-bit units */
var WORD_SIZE  = prog.cpu.attrs.dataWordSize;
                                /* in 8-bit units */

/* convert appl frame size to platform word size */
DSS_rxPipe.framesize = (2 * FRAME_SIZE) / WORD_SIZE;
DSS_rxPipe.numframes = 2;
```

So, on 'C5000 platforms, DSS_rxPipe.framesize equals (2 * 64) / 2, or 64. On 'C6000 platforms, DSS_rxPipe.framesize equals (2 * 64) / 4, or 32.

attrs.minDataUnitSize Property

The attrs.minDataUnitSize property contains the size of the smallest addressable data value (in 8-bit units). On 'C5000 platforms, the attrs.minDataUnitSize is two 8-bit units. On 'C6000 platforms, the attrs.minDataUnitSize is one 8-bit unit.

This property is gettable only. It is set if the cpu_type argument to the Board object's create() method matches a constructor function, and that constructor function sets the attrs.minDataUnitSize property.

attrs.minProgUnitSize Property

The attrs.minProgUnitSize property contains the size of the smallest addressable program value (in 8-bit units). On 'C54x platforms, the attrs.minProgUnitSize is two 8-bit units. On 'C55x platforms, the attrs.minProgUnitSize is one 8-bit unit. On 'C6000 platforms, the attrs.minProgUnitSize is one 8-bit unit.

This property is gettable only. It is set if the cpu_type argument to the Board object's create() method matches a constructor function, and that constructor function sets the attrs.minProgUnitSize property.

5.5 Program Class

Table 5-5 Program Class Summary

Object Type	Contains	Methods	Properties

Program	Module
	Extern
		extern()
		externs()
		destroy()
		gen()
		get()
		module()
		modules()
			cpu
			name
			prog.build.target.model.codeModel
			prog.build.target.model.dataModel
			prog.build.target.model.endian

A Cpu object may contain one or more Program objects. Program objects may contain one or more Module objects. Program objects may also contain an array of Extern (external declaration) objects. Program objects have properties for storing information about the program compilation model.

Program objects also have a method for generating files to be used in building the application. Generating files allows the settings made via Tconf to be linked with the program and used with the DSP/BIOS Real-Time Analysis Tools.

The create() method of a Program object cannot be used to create Module objects.

A default Program object is created by the startup script. This startup script also creates a global variable called "prog" that references this object. Additional Program objects can be created with the create() method of a Cpu object.

Program objects define a namespace within which all objects must have unique names. See Section 3.6.2, *Namespace Management*, page 3-10 for details.

create() Method

Description: The only way to create a Module object is to load a platform definition or CDB file with the utils.loadPlatform() method. Do not use the create() method of the Program object to create Module objects.

destroy() Method

Syntax: destroy()

Parameters: none

Returns: true if successful; false if failed

Description: The destroy() method destroys the specified object.

This method fails and returns false if the object is either referenced by another object or contains objects.

Notice that while the create() method creates an object one layer lower in the hierarchy than the object whose method is used, the destroy() method deletes the actual object whose method is used.

You will probably not need to use the destroy() method when writing configuration scripts from scratch.

extern() Method

Syntax: extern("name", "language")

Parameters:

name	Name of Extern object to create or get. Required.
language	Optional language for which to declare this symbol

Returns: Extern object created or specified

Description: In order to specify a function name as the value of a Module or Instance property, you must create an Extern object (for "external declaration"). All Extern objects within a Program object must have unique names.

If no Extern object exists with the specified name, the extern() method creates and returns a new Extern object. If an Extern object already exists with the specified name, the extern() method returns the object.

The optional language parameter allows you to specify whether the symbol should be defined as an asm, C, or C++ symbol. If no language is specified, the default is C.

You do not need to use an underscore prefix in a Tconf script for the names of any Extern objects you create.

Examples:

```
myTask.fxn = prog.extern("myTaskFxn", "C");
mySwi.fxn = prog.extern("mySwiFxn", "asm");
SYS.ABORTFXN = prog.extern("error");
```

externs() Method

Syntax: externs()

Parameters: none

Returns: Array of all Extern objects contained in the Program object

Description: The externs() method gets an array of all the Extern objects contained within the specified Program object.

Example: The following statements print a list of the Extern objects contained by a Program:

```
externs = prog.externs();
for (var i = 0; i < externs.length; i++)
  print(externs[i].name);
}
```

gen() Method

Syntax: gen("prog_name");

Parameters:

prog_name	Optional name of output application.

Returns: True if successful; false if not successful

Description: After you have created a Tconf script, you must create its generated files.

On Windows, you must also add the CDB file to your Code Composer Studio project. Then, you can build your DSP/BIOS application with Code Composer Studio. The CDB file also makes configuration information available to the DSP/BIOS Real-Time Analysis Tools.

The gen() methods generates the files needed to build the configuration into the application. See Section 2.1.1, *Generated Files*, page 2-2 for descriptions of the generated files.

If you are going to build your project with Code Composer Studio, use the prog.gen() method as follows without specifying an output filename.

```
prog.gen();
```

If you do not plan to build in Code Composer Studio, you can optionally use the prog_name argument to specify an output filename for the generated files. It is generally recommended that the filename match your target program .out filename. For example, if your target program executable is hello.out, use the following statement:

```
prog.gen("hello");
```

The prog_name parameter can also specify a folder location to contain the generated files.

When you omit the prog_name parameter, the default prog_name is the name of the Program object.

If you specify a prog_name parameter, all generated files begin with that prefix. The "cfg" suffix is appended to the filename for all generated files, and the appropriate file extensions all are added to all files.

The gen() method stores the files it creates in your current folder.

Example:

```
prog.gen("myAppl");
```

get() Method

Syntax: get("*name*")

Parameters:

name	Name of object to get. Required.

Returns: object, or null if error occurs

Description: The get() method returns the object specified by the name parameter.

The get() method can return any object in the namespace of the object for which it is called. For example, you can use the get() method for a Program object to get any Module (such as LOG), Instance object (such as LOG_system), or Extern object. In contrast, the module() method can return only Module objects and the instance() method can return only Instance objects. For more information about namespaces, see Section 3.6.2, *Namespace Management*, page 3-10.

If there is no object with the specified name in the namespace of the container whose get() method is used, get() returns null.

Example: In this example, “instanceof” is a JavaScript operator that returns true if the object is of the specified class. “Instance” is the name of a class.

```
/* lookup existing object named "audio" */
audio = prog.get("audio");

/* if audio is an Instance object */
if (audio instanceof Instance) {
    audio.priority = 1;       /* set its priority */
}
```

module() Method

Syntax: module("*name*")

Parameters:

name	Name of Module object to get. Required.

Returns: object, or null if error occurs

Description: The module() method returns the object specified by the name parameter.

If there is no object with the specified name in the Program object, module() returns null.

The get() method can return any object in the namespace of the Program object for which it is called. For example, you can use the get() method for a Program object to get any Module (such as LOG) or Instance object (such as LOG_system). In contrast, the module() method can return only Module objects. For more information about namespaces, see Section 3.6.2, *Namespace Management*, page 3-10.

modules() Method

Syntax: modules()

Parameters: none

Returns: Array of all Module objects contained in the specified Program object

Description: The modules() method gets an array of all the Module objects contained within the specified Program object.

Example:

```
list = "";
modules = prog.modules();
for (i = 0; i < modules.length; i++) {
    list += modules[i].name + " ";
}
```

cpu Property The cpu property holds the Cpu object that contains the Program object.

This property is gettable only. It is set when the Program object is created.

Example:

```
if (prog.cpu.attrs.cpuCore == "5500") {
   /* C5500-specific statements */
}
```

name Property The name property of an object holds the name of that object. This property is gettable only. It is set when the object is created. Names of Program objects must be unique within the Cpu object that contains them.

codeModel Property The prog.build.target.model.codeModel property contains "near" or "far" to indicate the code addressing model used by the program. On 'C6000 platforms, the value is always "far". On all other platforms, the default is "near".

This property is set automatically if you use the utils.loadPlatform() method. To set this property to "far", you may use the following –D option on the tconf command line:

```
tconf -Dconfig.compilerOpts="-mf"
```

Example:

```
bios.GBL.CALLMODEL =
    prog.build.target.model.codeModel;
```

dataModel Property

The prog.build.target.model.dataModel property contains "small" or "large" to indicate the data addressing model used by the program. The default is "small" on all platforms.

This property is set automatically if you use the utils.loadPlatform() method. To set this property to "large", you may use the following –D option on the tconf command line:

```
tconf -Dconfig.compilerOpts="-ml"
```

Example:

```
bios.GBL.MEMORYMODEL =
    prog.build.target.model.dataModel;
```

endian Property

The prog.build.target.model.endian property contains "little" or "big" to indicate the byte addressing model used by the program. The default is "little" on all platforms.

This property is set automatically if you use the utils.loadPlatform() method. To set this property to "big", you may use the following –D option on the tconf command line:

```
tconf -Dconfig.compilerOpts="-me"
```

Example:

```
bios.GBL.ENDIANMODE = prog.build.target.model.endian;
```

5.6 Memory Class

Table 5-6 Memory Class Summary

Object Type	Contains	Methods	Properties
Memory		-- none --	base comment len name space

A Board or Cpu object may contain one or more Memory objects. Memory objects do not contain any objects. Memory objects represent memory on the board or CPU.

There is no method to create a Memory object. Instead, Memory objects are defined as elements in a mem[] array. For example:

```
/*  Define external memory on board */
config.board("dsk5402").mem = [];

config.board("dsk5402").mem[0] = {
    comment: "External Program Memory",
    name:      "EPROG",
    space:      "code",
    base:      0x8000,
    len:  0x7f80
};

config.board("dsk5402").mem[1] = {
    comment: "External Data Memory",
    name:      "EDATA",
    space:      "data",
    base:      0x8000,
    len:  0x8000
};
```

The getMemoryMap() method of the Board and Cpu objects allows you to get the array of defined Memory objects.

Memory objects are typically created only in a platform definition.

base Property

The base property holds the location of the base of the memory segment. It is typically specified using a hex value.

comment Property

The comment property holds a text description about the memory segment.

len Property

The len property holds the length of the memory segment. It is typically specified using a hex value.

name Property

The name property of an object holds the name of that object. This property is gettable only. It is set when the object is created. Names of Memory objects must be unique within the Board or Cpu object that contains them.

space Property

The space property specifies the type of memory space as a string. It may be "code", "data", "code/data", or any other value appropriate for the platform.

5.7 Extern Class

Table 5-7 Extern Class Summary

Object Type	Contains	Methods	Properties
Extern		-- none --	language name

A Program object may contain one or more Extern objects. Extern objects do not contain any objects.

Extern objects represent external declarations made in program code that need to be referenced in the configuration. The following example statements create Extern objects:

```
myTask.fxn = prog.extern("myTaskFxn", "C");
mySwi.fxn = prog.extern("mySwiFxn", "asm");
bios.SYS.ABORTFXN = prog.extern("error");
```

The extern() method of the Program object (see Section 5.5, *Program Class*, page 5-16) creates a new Extern object only if none exists with the specified name. If an Extern object already exists with the specified name, the extern() method returns the object. The externs() method of the Program object gets an array of all Extern objects contained within the specified Program object.

language Property

The language property of an object identifies the language in which the name is declared. It may be "C", "C++", or "asm". This property is gettable only. It is set when the object is created. The default is "C".

name Property

The name property of an object holds the name of that object. An underscore prefix is not needed in Tconf scripts for the names of any Extern objects. This property is gettable only. It is set when the object is created. Names of Extern objects must be unique within the Program object that contains them.

5.8 Module Class

Table 5-8 Module Class Summary

Object Type	Contains	Methods	Properties

Module	Instance	create()
		instance()
		instances()
			defined in API Reference
			name
			program

A Program object may contain one or more Module objects. Module objects may contain one or more Instance objects. Module objects represent a target module within a single program.

The only way to create a Module object is to use the utils.loadPlatform() method. Do not use the create() method of the Program object to create Module objects.

The utils.loadPlatform() method defines a variable for each Module object within the "bios" namespace. For example, DSP/BIOS contains modules named LOG, TSK, and MEM. These correspond to Module objects named LOG, TSK, and MEM. To access them in a script, use bios.LOG, bios.TSK, and bios.MEM.

Module objects have properties that are specific to the type of module and are defined within the platform definition that has been loaded.

create() Method

Syntax: create("*instance_name*"])

Parameters: instance_name Required name for new Instance object.

Returns: new Instance object, or null if error occurs

Description: The create() method for a Module creates a new Instance object within the Module object.

The parameter is the name to give the new Instance object. The name must be unique among the Module, Instance, and Extern objects for this program. This parameter is required.

The order of objects created within a container array is undefined. You may use JavaScript's array sorting methods to get sorted lists of objects.

Examples:

```
inputPipe = bios.PIP.create("input");
inputPipe.notifyWriterFxn = prog.extern("writerFxn");
inputPipe.notifyWriterArg0 = 0;
inputPipe.bufAlign = 32;

traceLog = bios.LOG.create("trace");
traceLog.buflen = 32;
```

instance() Method

Syntax: instance("*name*")

Parameters:

name	Name of object to get. Required.

Returns: object, or null if error occurs

Description: The instance() method returns the Instance object specified by the name parameter.

If there is no object with the specified name in the Module, instance() returns null.

Note that while individual objects within any container object may be referred to as "instances," there is also a specific object class called "Instance," which is the child of the Module class. Thus, the instance() method of the Module class returns an "Instance object." Because of the potential for confusion, this document refers to individual objects that are not of the "Instance" class as "objects," not as "object instances" or "instances."

Example:

```
log = bios.LOG.instance("LOG_system");
```

instances() Method

Syntax: instances()

Parameters: none

Returns: Array of all objects contained within this object

Description: The instances() method returns an array of all the Instance objects contained in the Module object whose method is used. This allows scripts to loop through all the instances.

Note that while individual objects within any container object may be referred to as "instances," there is also a specific object class called "Instance," which is the child of the Module class. Thus, the instances() method of the Module class returns an array of "Instance objects."

Because of the potential for confusion, this document refers to individual objects that are not of the "Instance" class as "objects," not as "object instances" or "instances."

Example:

```
/* loop through all MEM objects and remove any heaps */
var memObjs = bios.MEM.instances();
for (var i = 0; i < memObjs.length; i++ ) {
  /* can't remove MEM_NULL heap */
  if (memObjs[i] != MEM_NULL
         && memObjs[i].createHeap == 1) {
    memObjs[i].createHeap = 0;
  }
}
```

name Property

The name property of an object holds the name of that object. This property is gettable only. It is set when the object is created.

Names of Module objects must be unique within the namespace of the Program object that contains them. Program objects define a namespace that includes all Extern, Module, and Instance objects contained by the Program object. Extern, Module, and Instance objects within two different Program objects can have duplicate names.

The names of Extern, Module, and Instance objects are the same as their C identifiers.

Example:

```
/* assemble a list of the module names in prog */
list = "";
modules = prog.modules();
for (i = 0; i < modules.length; i++) {
    list += modules[i].name + " ";
}
```

program Property

The program property holds the Program object that contains the Module object. This property is gettable only. It is set when the Module object is created.

API Properties

Normally, all objects in a class have the same set of properties. However, each module and each instance type has a different set of properties. Therefore, properties for Module and Instance objects are handled differently than those of other object classes.

Refer to the following reference guides for lists of property names used in Tconf scripts for DSP/BIOS Module and Instance objects:

❑ *TMS320C5000 DSP/BIOS Application Programming Interface Reference Guide* (literature number SPRU404)

❑ *TMS320C6000 DSP/BIOS Application Programming Interface Reference Guide* (literature number SPRU403)

❑ *TMS320C28x DSP/BIOS Application Programming Interface Reference Guide* (literature number SPRU625)

In general, the properties of Module objects are in all uppercase letters. For example, "MEM.STACKSIZE". See the *DSP/BIOS Application Programming Interface Reference Guide* for your platform for property names to use in Tconf scripts. You can set and get these property values as you would properties of other object classes.

Example:

```
bios.GBL.CALLMODEL =
   prog.build.target.model.codeModel;
bios.CLK.MICROSECONDS = 25000;
```

5.9 Instance Class

Table 5-9 Instance Class Summary

Object Type	Contains	Methods	Properties

Instance		destroy()
		references()
			defined in API Reference
			module
			name

A Module object may contain one or more Instance objects. Instance objects do not contain any objects. Instance objects represent a single target object.

Loading a platform definition defines Module and Instance objects in the JavaScript environment. The create() method of a Module object can also be used to create Instance objects.

Instance objects have properties that are specific to the type of module that contains them and are defined within the platform definition that has been loaded. If setting a property fails because of a rule defined for setting that property, an error is reported but no exception is thrown.

Note that while individual objects within any container object may be referred to as "instances," there is also a specific object class called "Instance," which is the child of the Module class. Thus, the instances() method of the Module class returns an array of "Instance objects." Because of the potential for confusion, this document refers to individual objects that are not of the "Instance" class as "objects," not as "object instances" or "instances."

create() Method

Instance objects cannot contain other objects, therefore the create() method of an Instance object fails and returns an error.

destroy() Method

Syntax: destroy()

Parameters: none

Returns: true if successful; false if failed

Description: The destroy() method destroys the specified object.

This method fails and returns false if the object is either referenced by another object or contains objects.

Notice that while the create() method creates an object one layer lower in the hierarchy than the object whose method is used, the destroy() method deletes the actual object whose method is used.

You will probably not need to use the destroy() method when writing configuration scripts from scratch,.

references() Method

Syntax: references()

Parameters: none

Returns: Array of all objects that directly reference this object

Description: The references() method returns an array of objects that directly reference the object whose method is used. Scripts can use the returned array to attempt to delete referring objects or to display meaningful errors.

Example:

```
/* display list of all objects that reference IDATA */
refs = bios.IDATA.references();
for (i = 0; i < refs.length; i++) {
    print(bios.IDATA.name +
        " is referenced by " + refs[i].name);
}
```

module Property The module property holds the Module object that contains the Instance object. This property is gettable only. It is set when the Instance object is created.

Example:

```
thread_type = myThread.module.name;
```

name Property The name property of an object holds the name of that object. This property is gettable only. It is set when the object is created.

Names of Instance objects must be unique within the namespace of the Program object that contains them. Program objects define a namespace that includes all Extern, Module, and Instance objects contained by the Program object. Extern, Module, and Instance objects within two different Program objects can have duplicate names.

The names of Extern, Module, and Instance objects are the same as their C identifiers.

Example:

```
/* assemble a list of the module names in prog */
list = "";
modules = prog.modules();
for (i = 0; i < modules.length; i++) {
    list += modules[i].name + " ";
}
```

API Properties

Normally, all objects in a class have the same set of properties. However, each DSP/BIOS module and each instance type has a different set of properties. Therefore, properties for Module and Instance objects are handled differently than those of other object classes.

Refer to the following reference guides for lists of property names used in Tconf scripts for DSP/BIOS Module and Instance objects:

❑ *TMS320C5000 DSP/BIOS Application Programming Interface Reference Guide* (literature number SPRU404)

❑ *TMS320C6000 DSP/BIOS Application Programming Interface Reference Guide* (literature number SPRU403)

❑ *TMS320C28x DSP/BIOS Application Programming Interface Reference Guide* (literature number SPRU625)

In general, properties of Instance objects begin with a lowercase word. Subsequent words have their first letter capitalized. For example, "TSK_idle.stackSize".

See the *DSP/BIOS Application Programming Interface Reference Guide* for your platform for property names to use in Tconf scripts. You can set and get these property values as you would properties of other object classes.

Example:

```
trace = bios.LOG.create("trace");
trace.bufLen = 32;
trace.logType = "circular";
```

Chapter 6

The DSP/BIOS Configuration Tool (Gconf)

This chapter describes use of the DSP/BIOS Graphical Configuration Tool.

Detailed help information for the DSP/BIOS Configuration Tool is provided in the DSP/BIOS online help. This chapter provides a summary of how to use the DSP/BIOS Configuration Tool and some specific information relating to its interaction with Tconf.

Topic **Page**

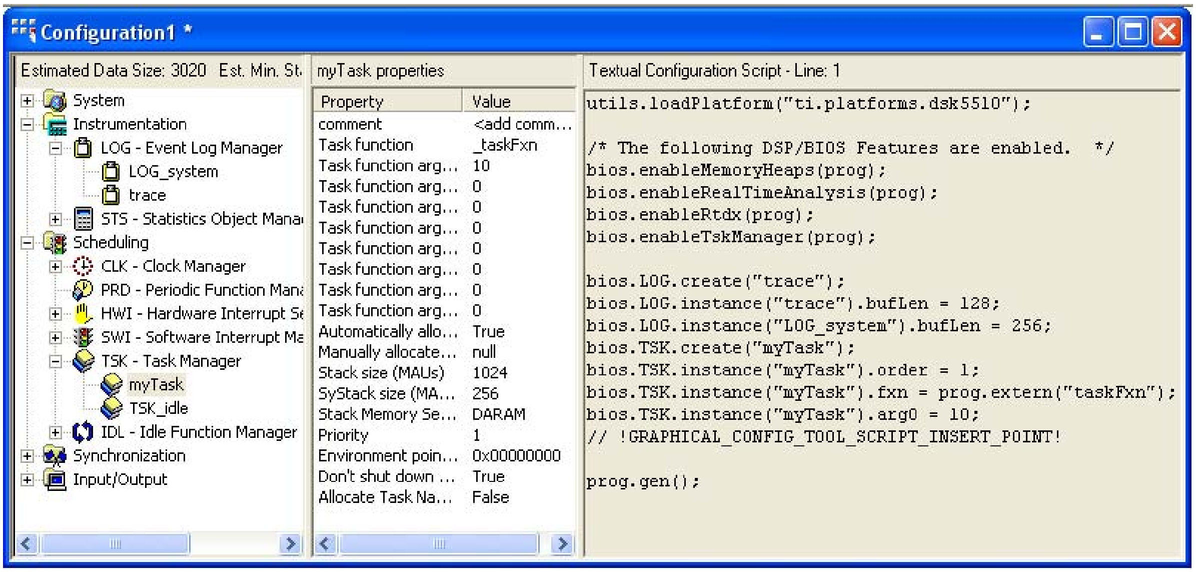
6.1 Tconf Pane in the Graphical Editor

The DSP/BIOS Configuration Tool (Gconf) provides a graphical editor for Tconf scripts.

In previous versions of DSP/BIOS, the DSP/BIOS Configuration Tool worked with CDB files as source files. It now treats TCF files as source files, making Tconf the basis for DSP/BIOS configuration whether you edit them with the graphical tool or a text editor.

The new right pane of the DSP/BIOS Configuration Tool acts as a "macro recorder" for Tconf. Changes you make in the object tree and property dialogs are reflected in the script.

Figure 6-1 DSP/BIOS Configuration Tool



Configuration1 *

Estimated Data Size: 3020 Est. Min. St

```
+ System
- Instrumentation
    - LOG - Event Log Manager
        LOG_system
        trace
    + STS - Statistics Object Mana
- Scheduling
    + CLK - Clock Manager
      PRD - Periodic Function Mana
    + HWI - Hardware Interrupt Se
    + SWI - Software Interrupt Ma
    - TSK - Task Manager
        myTask
        TSK_idle
    + IDL - Idle Function Manager
+ Synchronization
+ Input/Output
```

myTask properties

Property	Value
comment	<add comm...
Task function	_taskFxn
Task function arg...	10
Task function arg...	0
Task function arg...	0
Task function arg...	0
Task function arg...	0
Task function arg...	0
Task function arg...	0
Task function arg...	0
Automatically allo...	True
Manually allocate...	null
Stack size (MAUs)	1024
SyStack size (MA...	256
Stack Memory Se...	DARAM
Priority	1
Environment poin...	0x00000000
Don't shut down ...	True
Allocate Task Na...	False

Textual Configuration Script - Line: 1

```
utils.loadPlatform("ti.platforms.dsk5510");

/* The following DSP/BIOS Features are enabled.  */
bios.enableMemoryHeaps(prog);
bios.enableRealTimeAnalysis(prog);
bios.enableRtdx(prog);
bios.enableTskManager(prog);

bios.LOG.create("trace");
bios.LOG.instance("trace").bufLen = 128;
bios.LOG.instance("LOG_system").bufLen = 256;
bios.TSK.create("myTask");
bios.TSK.instance("myTask").order = 1;
bios.TSK.instance("myTask").fxn = prog.extern("taskFxn");
bios.TSK.instance("myTask").arg0 = 10;
// !GRAPHICAL_CONFIG_TOOL_SCRIPT_INSERT_POINT!

prog.gen();
```

In the **left pane** of the DSP/BIOS Configuration Tool, you use the right-click menu to create and delete objects, set properties of modules and objects, and set priorities of threads.

The **center pane** shows property names and values or thread priorities.

The **right pane** shows the Tconf script. It shows the statements for the changes you make in the left pane. You cannot edit script statements in the right pane, but you can use the right-click menu to add blank lines and comments to the script.

This tool is designed to run in Code Composer Studio. It can also be run standalone by running the gconf.exe executable, which is located in BIOS_INSTALL_DIR\packages\ti\bios\config\gconf\bin.

6.2 Tconf Integration with the DSP/BIOS Configuration Tool

When you write a Tconf script with a text editor, error checking is done when you run the script (for example, by building with CCStudio).

In contrast, when you use the DSP/BIOS Configuration Tool, error checking is done initially when it opens a TCF script and then for each change you make to the configuration. The exception is the MEM objects, whose properties are validated when you save the configuration. (Internally, the tool holds the old CDB file model in memory to check configurations for correctness.

Internally, the tool operates on a CDB object model that is created when the TCF file is executed. The CDB objects contain the validation rules that are used to check each graphical configuration action.

The DSP/BIOS Configuration Tool can open legacy CDB files in read-only mode for browsing older configurations or convert them from CDB files to TCF scripts. This behavior is not available in some versions of CCStudio; however, it can be performed when running the gconf.exe program outside the CCStudio environment.

6.2.1 Limitations of Tconf Integration

The DSP/BIOS Configuration Tool has the following limitations regarding the integration of Tconf with graphical configuration:

❑ You cannot edit script statements in the right pane. (You can add comments and blank lines for readability using the right-click menu.)

❑ You can only see and modify the top-level TCF file. Any TCI files that are imported affect the configuration in the left pane but are not visible in the right pane.

❑ A TCF script must contain the following items in either the top-level TCF script or an included TCI file:

  ■ A call to utils.loadPlatform that specifies the logical platform file to load.

  ■ A location or insertion marker to indicate where new Tconf statements are to be added.

  ■ A call to prog.gen() to generate the output files.

❑ You cannot use a filename in the prog.gen() method to modify the output filenames for generated files. An alternative is to set the config.programName property, which can specify a unique name/location for generated files for project configurations (for example, Debug, Release, and Custom).

❑ Double-clicking a TCF file in Windows Explorer to open gconf.exe in standalone mode is likely to result in problems locating include files and platform files. The best way to run the DSP/BIOS Configuration Tool is within CCStudio using a project file. The recommended way to for launch gconf.exe in standalone mode is to create a Windows shortcut that specifies gconf.exe command-line options.

❑ The DspBiosBuilder tab in the CCStudio **File->Build Options** dialog allows you to set options for the project. It passes these options to gconf.exe via the tcfopts.dat file, which it writes to the project folder. See the help for the DspBiosBuilder tab for more details.

❑ Not all Tconf command-line options are supported by the DSP/BIOS Configuration Tool. Only the -D<name>=<value> options are supported. Your script can be written to make a number of choices that depend upon the -D options passed to it.

6.2.2 Prog.gen() Method Argument Rules

The prog.gen() method at the end of a TCF script for a project to be built with CCStudio should not use a parameter to specify the filenames of the generated files. However, a parameter is actually allowed if it does not affect the output filenames. That is, if the parameter matches the name of current TCF file without the ".tcf" extension. For example, if the TCF file is called bigtime.tcf, the prog.gen method may be as follows:

```
prog.gen(bigtime);
```

6.2.3 Insertion Marker Rules

A TCF script must contain the insertion marker shown below in order for the DSP/BIOS Configuration Tool to successfully insert script statements.

```
// !GRAPHICAL_CONFIG_TOOL_SCRIPT_INSERT_POINT!
```

If this marker is not present, the DSP/BIOS Configuration Tool places one above the "prog.gen()" statement when you perform the first edit operation.

6.3 DSP/BIOS Configuration Tool Menu Operations

You can use menu commands within the DSP/BIOS Configuration Tool to perform the actions described in the following sections. These descriptions provide more information about the interaction with Tconf than the online help descriptions.

File->Open

You can open both CDB and TCF files with the DSP/BIOS Configuration Tool. Note that when running the tool inside some versions of CCStudio, you cannot open CDB files. In such cases, you can run the gconf.exe program outside the CCStudio environment to open a CDB file.

To open a file, choose **File->Open**.

❑ **TCF files.** When you open a TCF file, the DSP/BIOS Configuration Tool looks for a Tconf search path using the following priority order:

  ■ The command line to gconf.exe. An easy way to use this is to create a desktop shortcut and to specify command line arguments for the shortcut.

  ■ A file named tcfopts.dat in the same folder as the TCF file. This file is created when you initially save a TCF file with the DSP/BIOS Configuration Tool, or if you use the DspBiosBuilder tab in the CCStudio Build Options dialog.

  ■ If no other search path is specified, a dialog prompts you for one. The default is the "packages" folder above the folder that contains gconf.exe.

❑ **CDB files.** When the DSP/BIOS Configuration Tool opens a CDB file, it asks if you want to convert the CDB. If you choose to convert, you are prompted to browse to the location of the CCStudio installation. The tool then runs the cdb2tcf conversion utility. In some cases you may be prompted to correct errors found by cdb2tcf. The TCF file produced will contain an "in place" generic platform definition.

In CCStudio, you can also right-click on a TCF file in a Project View to choose from a DSP/BIOS Config right-click menu that allows you to use Graphical Edit, Text Edit, or Run in Graphical Debugger.

An error occurs if you attempt to open a TCF file that does not contain a valid configuration. Valid configurations must load a platform and run the prog.gen() method.

File->New

To create a new TCF-based configuration, follow these steps:

1) Choose **File->New->DSP/BIOS Configuration**. This opens the New DSP/BIOS Configuration dialog, which allows you to select a platform and enable DSP/BIOS features.
2) Select the tab for your DSP family.
3) Select the platform file for your DSP board.

   When you highlight a template, a description is shown to the right of the dialog. You can click the buttons above the description to view the templates with large icons, small icons, or a list that includes file sizes and modification dates.
4) Select the DSP/BIOS features you want to enable in your configuration. All the features listed are disabled by default. Leaving a feature disabled significantly reduces the code size, memory usage, and other resource usage of the resulting application. See the online help for this dialog for details about the options.
5) Click OK to open the new configuration.

DSP/BIOS provides platform files for common boards produced by Texas Instruments. Path information to the platform you select is stored in a tcfopts.dat file in the same folder as the TCF file when you save it.

File->Save

Whenever you save a Tconf script, the script is run and files are generated from it. These files are described in Section 2.1.1, *Generated Files*. Additional files are created or copied when you use File->Save As.

Errors may be reported when you save a TCF file if you made changes to MEM objects. MEM object locations and sizes are not verified for overlaps on the fly. This allows you to make changes to the sizes and locations of several objects without encountering numerous errors within the DSP/BIOS Configuration Tool. Instead, any errors that remain after your changes are reported when you save the configuration.

File->Save As

As with File->Save, the Tconf script is run and files are generated from it. These files are described in Section 2.1.1, *Generated Files*.

In addition, a file called tcfopts.dat is generated the first time you save a configuration in a particular folder.

If you use File->Save As to save a configuration in a different folder, the tcfopts.dat file and any TCI files in the original folder are copied to the new folder so that they will be available to the TCF in its new location. In addition, relative paths are resolved to absolute paths in the TCF script.

These steps are necessary to move the Tconf script with its context intact. Even with these measures, there are cases where errors will occur, such as if an included TCI script contains a relative path reference.

Edit->Copy and Edit->Paste

These commands copy and paste the object using the clipboard.

These commands are useful if you want to create several similar objects of a particular type. You can only cut, copy, and paste objects within a single module. You cannot paste an object into a different module's folder. You cannot cut, copy, or paste a module manager.

Copying an object to the clipboard replaces the contents previously stored there. You are prompted for an object name when you paste an object, since objects must have unique names.

You can use the CTRL+C and CTRL+V keyboard shortcuts to copy and paste text within a property dialog.

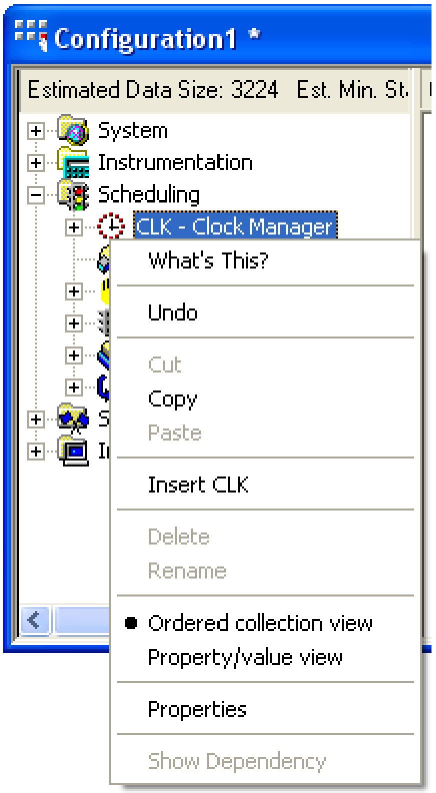
View->Toolbar and View->Status Bar

These commands are available only if you run gconf.exe as a standalone program. Within CCStudio, the DSP/BIOS Configuration Tool shares the CCStudio standard toolbar, and status information is shown at the top of the configuration window.

In standalone mode, the DSP/BIOS Configuration Tool has its own toolbar and a status bar at the bottom of the window. You can use these commands to hide or redisplay these items.

Object->Insert

Select a module and use this command to insert an object into the module. You can also right-click on a module and choose the Insert command from the right-click menu. You are prompted to type a name for the object when you insert it.



Inserting an object adds a statement using the create() method to the TCF script at the insertion point. For example:

```
bios.LOG.create("myLOG");
```

Object->Delete

Select an object and use this command to delete the object. You can also right-click on an object and choose the Delete command from the right-click menu.

If you delete an object that you created during this session, the Tconf statement that creates it and any statements that set its properties are simply deleted from the script.

However, if you delete an object that was created during a previous session, a Tconf statement that destroys the object is added at the insertion point. For example:

```
bios.TSK.instance("TSK1").destroy();
```

Object->Rename

Select an object and use this command to rename the object. You are prompted for a new name. You can also right-click on an object and choose the Rename command from the right-click menu.

If you rename an object that you created during this session, the Tconf statement that creates it and any statements that set its properties are deleted from the script, and new statements are added at the insertion point to create the object and set its properties.

However, if you rename an object that was created during a previous session, a Tconf statement that destroys the object is added at the insertion point, and new statements are added to recreate the object and set its properties. For example:

```
bios.TSK.instance("TSK1").destroy();
bios.TSK.create("renamedTSK");
bios.TSK.instance("renamedTSK").priority = 3;
```

Object->Properties

This command opens the Property dialog for the selected object. You can also right-click on an object and choose the Properties command from the right-click menu. When you click OK, statements are added to the script at the insertion point to set the properties you changed. For example:

```
bios.LOG.instance("LOG1").bufLen = 512;
```

There are no properties for the top level folders (for example, System).

Object->Show Dependency

This command opens a dialog that lists other objects that depend on this object. For example, using this command on a MEM object lists other objects that use this MEM object. Using this command on an HWI object lists the interrupt sources related to this object.

Ordered Collection vs. Property/Value Views

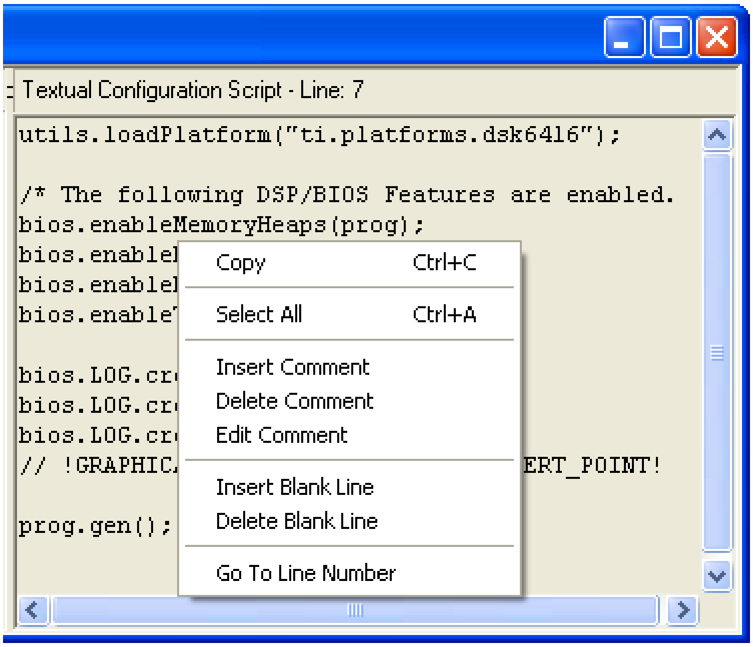
By default, the center pane shows the priorities and execution order for threads in the Scheduling category.

To change the default, right-click on a module in the Scheduling category and select Property/value view. The center pane will now list properties and values for that module manager.

Comments and Blank Lines in TCF Pane

Although you cannot edit the Tconf statements within the DSP/BIOS Configuration Tool, you can add comment lines and blank lines.

To add a comment, left-click on the line you want to place a comment above. Then, right-click and select Insert Comment from the right-click menu. Type your comment text in the dialog box, and click OK.



```
utils.loadPlatform("ti.platforms.dsk6416");

/* The following DSP/BIOS Features are enabled.
bios.enableMemoryHeaps(prog);
bios.enablel
bios.enablel
bios.enable

bios.LOG.cr
bios.LOG.cr
bios.LOG.cr
// !GRAPHIC                        ERT_POINT!

prog.gen();
```

You can delete a comment by left-clicking on the comment line, right-clicking, and selecting Delete Comment from the right-click menu. You can also choose to edit a comment line.

Similarly, you can add a blank line to a TCF script by left-clicking on the line you want to place a blank line above. Then, right-click and select Insert Blank Line from the right-click menu.

To edit the statements in a Tconf script, save and close the script in the DSP/BIOS Configuration Tool. Then, open the script with a text editor. You can later reopen the script with the DSP/BIOS Configuration Tool. You may encounter messages when you reopen the script with the DSP/BIOS Configuration Tool if any edited statements generate errors or warnings.

6.4 The Gconf.ini File

The DSP/BIOS Configuration Tool (gconf.exe) saves information to the gconf.ini file in BIOS_INSTALL_DIR\packages\ti\bios\config\gconf\bin when it exits:

- **Settings.** The last positions and sizes of the Hierarchy (tree view) pane, the Object Properties pane and the Script pane. You should not attempt to edit these values.
- **Old seed path.** When you open a legacy CDB file, a dialog box prompts for the seed path of the legacy CDB being opened. This is typically CCS_INSTALL_DIR\<ISA>\bios\include. This dialog is always displayed when you open a legacy CDB file. The default is the most recently used Old seed path stored in gconf.ini.
- **Current seed path.** You can add this setting can to gconf.ini by manually editing the file. It is used to find the base CDB file to perform an integrity check. The default location is the BIOS_INSTALL_DIR\packages\ti\bios\config\cdb folder.
- **Platforms.** When you create a new TCF file, you can use the Browse Platforms button in the File->New dialog to point to a platform folder other than the default. The default is BIOS_INSTALL_DIR\packages. The gconf.ini file stores the last folder you browsed for platforms for use in future sessions.

This is a sample gconf.ini file:

```
[Settings]
HierarchyPanePosition=0000000000000000FD0000008A01000088
ObjectPanePosition=0000000000000000B50000009F01000055
ScriptPanePosition=00000000000000003401000008A010000C0

[Old seed path]
TMS320C62XX=C:\CCStudio_v3.1\C6000\bios\include

[Current seed path]
TMS320C62XX=C:\bios_5_20\packages\ti\bios\config\cdb

[Platforms]
Search Directory=C:\bios_5_20\packages
```

6.5 Gconf.exe Command Line

You can run the DSP/BIOS Configuration Tool (gconf.exe) from a command line. For example, you might create a desktop shortcut, a batch file, or a makefile to run it.

The syntax is:

```
gconf.exe script_name.tcf [-tcfopts="opts"]
```

When you run gconf.exe from a command line, you can use the -D<name>=<value> pairs supported for the Tconf command line. To specify such pairs, use the -tcfopts option as follows:

```
-tcfopts="-Dconfig.importPath=.;..;../common;C:/bios_5_20/packages; -Dfoo=bar"
```

Any relative paths are relative to the location of the TCF file being opened.

If you use a command inside a Windows shortcut, you must place a backslash (\) before each quote mark ("). For example:

```
-tcfopts=\"-Dconfig.importPath=.;..;../common;C:/bios_5_20/packages; -Dfoo=bar\"
```

Here are examples of full gconf.exe command lines:

```
gconf.exe C:\bios_5_20\packages\ti\bios\examples\basic\clk\dsk6713\clk.tcf
-tcfopts=\"-Dconfig.importPath=.;..;../../../common; C:/bios_5_20/packages;\"
```

6.6 Error Handling

If you open a TCF file that generates errors, those errors are reported in a dialog box. Examples of problems that may be detected are:

- ❏ File doesn’t match the requirements for editing in the DSP/BIOS Configuration Tool. For example, it does not have a utils.loadPlatform method or a prog.gen method.
- ❏ A JavaScript error occurred. For example, a property may be set to an invalid value or a syntax error may occur.
- ❏ The file may be invalid. For example, it may be a zero-length file or not a JavaScript file.
- ❏ The TCI scripts imported by the TCF may not be locatable.

If you see a dialog that reports errors, click the Copy to Clipboard button so that you can paste the errors somewhere you can refer to them later. Then, open the TCF file in a text editor and correct the problems.

Appendix A

Updating DSP/BIOS Configurations

This appendix describes how to convert application configurations created with previous versions of DSP/BIOS.

Topic **Page**

A.1 Overview

In previous versions of DSP/BIOS, the DSP/BIOS Configuration Tool was the only mechanism for creating and storing configuration information. Such configurations were stored in CDB files. In DSP/BIOS releases beginning with DSP/BIOS 5.0, configurations are stored in Tconf script files, but you can still use the DSP/BIOS Configuration Tool to create and modify your configurations.

The cdb2tcf conversion tool is provided to help you convert existing CDB-based configurations to new script-based configurations. With cdb2tcf, you generate Tconf scripts that configure DSP/BIOS for use with your applications. The cdb2tcf utility is available for Windows, Linux and Solaris. Additionally, the cdb2tcf tool can be run by opening a CDB file when running the DSP/BIOS Configuration Tool in standalone mode.

If you already have Tconf configuration scripts, this appendix also describes how to edit existing files to use them with newer DSP/BIOS versions (5.0 and later).

A.2 The cdb2tcf Utility

Syntax

```
cdb2tcf [-h] [-a <n>] [-d]
          [-i <custom_tci_file> <custom_cdb_file>]
          [-l <logfile>] app.cdb
```

Options

-h Displays help information and exits.

-a <n> Sets the information verbosity levels of the generated script, including comments and references to unused DSP/BIOS elements. Valid values for this option are:

0. (default) No comments in the generated TCF script.

1. Comments about old values that have been changed are provided in the TCF script.

-d Create a TCI file (a file to be imported by a TCF script) containing only the differences between a configuration template and your configuration, rather than a complete TCF script. This option is typically used if you created your own configuration template from a TI-supplied configuration template, and then you developed multiple application configurations based on your template. If you want to maintain such configurations, you can first run cdb2tcf with '-d' to create a TCI file with a description of your configuration template, and then run cdb2tcf with the option '-i' for each of your application configurations. Each generated TCF script includes the TCI file instead of repeating the same set of instructions. See Section A.3.2, *Converting a Custom Base Seed* for an example.

-i <custom tci file> <custom cdb file>
This option ensures that a <custom tci_file>, generated by a prior execution of cdb2tcf using the -d option, is included in the generated TCF script. See Section A.3.2, *Converting a Custom Base Seed* for an example.

-l <log file> The name of an optional log file. The log file contains messages about the progress of the conversion, and possible warnings and errors. If the log file is not specified, messages are sent to standard output.

Return Code

Returns zero if the conversion succeeds, non-zero if error.

Description

The cdb2tcf utility converts an application's CDB file to a Tconf TCF script. The generated TCF script has the same name as the application CDB file (for example, sample.cdb results in sample.tcf).

The cdb2tcf utility looks for the BIOS_INSTALL_DIR environment variable to determine to which DSP/BIOS version to use to convert configurations. If BIOS_INSTALL_DIR is not set, cdb2tcf assumes that you want to convert to the DSP/BIOS version from which the cdb2tcf executable is run.

CDB file basics:

An application's CDB file is generated from a configuration template. A configuration template contains configuration information that represents a platform with some additional DSP/BIOS settings switched on. You can determine the configuration template that was used to create your configuration by opening the CDB file in a text editor and looking at the third line of the file. An application's configuration extends the configuration template settings with additional DSP/BIOS objects and program configuration parameters. CDB files may also contain CSL mapping declarations.

The cdb2tcf utility generates a TCF script that produces the same configuration as the original CDB file. It separates the configuration into the following sections so that you can easily reuse or extend your configuration. Comments identify the sections. The sections may contain in-line statements or include separate .tci files.

- ❏ **Minimum DSP/BIOS configuration.** Loads the initial configuration using the ti.platforms.generic platform with the parameters appropriate for the chip configuration. This configures the minimal set of DSP/BIOS modules possible.
- ❏ **Base configuration.** Adds statements to extend the generic platform configuration to match the configuration template selected when the DSP/BIOS Configuration Tool was originally used to create the original configuration.
- ❏ **Custom configuration (if options -i and -d were used).** If a custom configuration template was used as the basis for creating the configuration, this optional section extends the TI configuration to match the custom template.
- ❏ **Application configuration.** This section configures objects used by the application. This might include such things as DSP/BIOS tasks and logs.
- ❏ **CSL configuration (optional).** If the application CDB file contains CSL configuration, these are placed in an isolated in a csl.c file. CSL configuration is no longer part of DSP/BIOS configuration.

Examples

```
cdb2tcf app.cdb
```

This command generates the full application TCF script from the application CDB file.

```
cdb2tcf -d customseed.cdb
```

This command generates a customseed.tci file with the list of changes a user made in Gconf when the file customseed.cdb was originally created.

```
cdb2tcf -i customseed.tci customseed.cdb app.cdb
```

This command generates the full application app.tcf script which includes customseed.tci and the differences between customseed.cdb and app.cdb.

A.3 Converting from CDB Configurations

If your existing CDB file was created using one of the configuration templates (base seeds) provided with Code Composer Studio, cdb2tcf can convert that file to a new DSP/BIOS compatible Tconf script. Your command line needs to supply the path to your application's CDB file.

The resulting Tconf script is created in the current folder.

A.3.1 Example for Base Seed Conversion

In the following examples, it is assumed that the application configuration file app.cdb is in your current folder, and cdb2tcf is in your $PATH. To create app.tcf, type the following command on the command line:

```
cdb2tcf app.cdb
```

The original configuration file, app.cdb is saved as app.cdb.bak. This is done to avoid having the original configuration file rewritten, because when tconf executes the generated script, it rewrites app.cdb as an intermediate step. If the file app.tcf already exists, cdb2tcf exits without overwriting it and displays an error message.

If cdb2tcf displays the following (or similar) error message:

```
"Error: Seed file C:/CCStudio_v3.2/bios_5_30_00_03/packag-
es/ti/bios/config/update/4.82.50/c64xx.cdb cannot be found
```

Your should check to see if the directory identified in the error exists. If not, unzip the file update.zip, which is available in the "update" directory. Check to see whether the missing file is created when you unzip the update.zip file. Once the directory and file exist, try using cdb2tcf again.

The created Tconf script consists of several parts separated by comments so that you can easily distinguish the purpose of each part.

❑ **Load generic platform with specific parameters.** First, the script loads the generic platform with parameters derived from the appropriate fields in app.cdb. The first statement in the script enables the use of old memory names. See Section 4.2, *Creating Custom Platform Files*, page 4-4 for more information.

❑ **Enable DSP/BIOS components.** The generic platform enables only the minimal subset of DSP/BIOS components, which means that heaps, tasks, real-time analysis, and RTDX are disabled. The base seeds enabled all DSP/BIOS components. Therefore, the second part of the script enables all DSP/BIOS components.

❑ **Apply user changes.** The third part of the script makes the changes you made to the original base seed.

```
/* ========= app.tcf ========= */
/* load generic platform with specific parameters */
environment["ti.bios.oldMemoryNames"] = true;
var params = {};
params.clockRate = 140;
params.deviceName = "5510";
params.catalogName = "ti.catalog.c5500";
params.regs = {};
params.regs.clkmd = 9106;
utils.loadPlatform("ti.platforms.generic", params);

/* enable DSP/BIOS components */
bios.GBL.ENABLEINST = true;
bios.MEM.NOMEMORYHEAPS = false;
bios.RTDX.ENABLERTDX = true;
bios.HST.HOSTLINKTYPE = "RTDX";
bios.TSK.ENABLETSK = true;
bios.DARAM.createHeap = true;

/* apply user changes */
bios.RTDX.MODE = "Simulator";
...
prog.gen();
```

A.3.2 Converting a Custom Base Seed

If you created your own custom configuration template (base seed) from one of the base seeds available in CCS, and if you used that custom base seed as a starting point for many different applications, cdb2tcf allows you to create a *.tci file that corresponds to that custom base seed. Using that file, you can:

❏ Convert CDB-based configurations to Tconf configuration, but retain the separation between code that defines your custom configuration and the application code.

❏ Continue developing new applications starting from the custom configuration.

To convert your CDB files to Tconf scripts in this case, you need to invoke cdb2tcf twice. First, you use cdb2tcf to create a script that corresponds to your custom configuration. Then, you create a configuration script relative to the custom configuration. To do this, follow these steps:

1) Invoke cdb2tcf with the -d option and one argument. The argument is the path to your custom base seed. For example, to create the custom.tci script, type the following on the command line:

   ```
   cdb2tcf -d custom.cdb
   ```

   The generated script does not load any platform nor does it contain the statement that generates the source files from the configuration. It contains only the statements that correspond to the difference between the original TI base seed and your custom base seed.

2) Next, invoke cdb2tcf again. This time, use a command similar to the following to specify that the configuration for your application is derived from custom.cdb, and that you want to import the file custom.tci generated previously. The file to be imported is specified using the option -i. This command creates a script file called app.tcf.

   ```
   cdb2tcf -i custom.tci custom.cdb app.cdb
   ```

The generated script contains the same parts as the app.tcf file in Section A.3.1, *Example for Base Seed Conversion*, except for the statement that imports custom.tci. The custom seed configuration is included after the platform is loaded and the DSP/BIOS components are enabled:

```
/* ========= app.tcf ========= */
/* load generic platform with specific parameters */
environment["ti.bios.oldMemoryNames"] = true;
var params = {};
params.clockRate = 720;
params.deviceName = "6416";
params.catalogName = "ti.catalog.c6000";
params.regs = {};
params.regs.l2Mode = "4-way cache (0k)";
utils.loadPlatform("ti.platforms.generic", params);

/* enable DSP/BIOS components */
bios.GBL.ENABLEINST = true;
bios.MEM.NOMEMORYHEAPS = false;
bios.RTDX.ENABLERTDX = true;
bios.HST.HOSTLINKTYPE = "RTDX";
bios.TSK.ENABLETSK = true;
bios.GBL.ENDIANMODE = "little";
bios.GBL.C641XCONFIGUREL2 = true;
bios.ISRAM.createHeap = true;
bios.ISRAM.heapSize = 0x8000;

/* import the custom seed */
utils.importFile("custom.tci");

/* apply user changes */
bios.CLK.TIMERSELECT = "Timer 1";
...
prog.gen();
```

If, instead, you do not want to continue using a separate custom configuration template, you can generate the new application configuration directly:

```
cdb2tcf -d app.cdb
```

The resulting app.tcf has the three parts described in Section A.3.1, *Example for Base Seed Conversion*, except that changes made to the base seed to create custom.cdb and changes from custom.cdb to app.cdb are grouped together in the third part of the script.

A.4 Converting from Existing Tconf Configurations

If you used Tconf with a previous release of DSP/BIOS, you may need to make some changes so that your scripts continue to function with DSP/BIOS.

A.4.1 Changes to the loadPlatform() Method

In earlier versions of DSP/BIOS, the loadPlatform() method expected its first argument to be a platform name with the first character capitalized, for example "Dsk5510".

We now recommend that new applications use the new syntax for loading platforms. The new syntax is documented in Section 4.1, *Using TI-Supplied Platform Files*, page 4-2. It requires that the name of the platform includes the whole path from the folder specified in config.importPath, for example:

```
utils.loadPlatform("ti.platforms.dsk5510");
```

DSP/BIOS still supports the old loadPlatform() syntax. For example, if your script calls "loadPlatform("Dsk5510")", the loadPlatform() method finds and loads the Platform.tci file in the ti/platforms/dsk5510 folder. Tconf provides a warning to indicate that you are using a deprecated form of loadPlatform(). The Platform.tci files internally use the new form of loadPlatform() with the appropriate parameters.

This release provides Platform.tci files for most boards supported in previous releases of DSP/BIOS. If you wrote a custom .tcp file, you cannot use it with DSP/BIOS. Translating it to a Platform.tci file that uses the generic platform is fairly straightforward. See Section 4.2, *Creating Custom Platform Files*, page 4-4. You can use your new .tci file in three ways:

❑ Copy your custom Platform.tci file to a new folder and continue to use the loadPlatform() method. Your new folder should be named something like *mycompany*/platforms/*boardname* and you should include the root for this folder in your tconf importPath. For example, if the folder is c:/local/mycompany/platforms/my5510, you should use tconf -Dconfig.importPath=”c:/local”.

  For example, if your original script used this statement:

  ```
  utils.loadPlatform("My5510");
  ```

  You should change it to:

  ```
  utils.loadPlatform(“mycompany.platforms.my5510”);
  ```

- ❏ Copy your .tci file to a common folder and change your scripts to use “importFile” instead of “loadPlatform”. In the example below, you would change the name of the tci file to Dsk5510.tci. Note that your common folder must be in the Tconf import path (for example, tconf -Dconfig.importPath="c:/commmondir;...").

  For example, if your original script used this statement:

  ```
  utils.loadPlatform("My5510");
  ```

  You should change it to:

  ```
  utils.importFile("My5510");
  ```

- ❏ **Not Recommended.** Copy your custom Platform.tci file to the ti/platforms/my5510 folder. If you do this, your scripts will not need to change. However, this is not recommended, since it is better to keep user scripts and code outside standard DSP/BIOS installation directories.

A.4.2 New Memory Configurations and Names

Some memory parameters (mostly names) have been changed in DSP/BIOS versions 5.0 and later. However, in order to support older scripts, DSP/BIOS allows the use of both memory naming sets through the flag ti.bios.oldMemoryNames. Note that the deprecated/*.tci files described in the previous section use the oldMemoryNames flag to allow existing scripts to work.

There are two ways to activate the support for the old memory configurations:

- ❏ When you run tconf, use the -D option to set the flag:

  ```
  tconf -Dti.bios.oldMemoryNames app.tcf
  ```

- ❏ At the beginning of the configuration script add the following statement:

  ```
  environment["ti.bios.oldMemoryNames"] = true;
  ```

For the scripts that use ‘C670x/’C671x devices and the corresponding seeds 6xxx.cdb and c6x1x.cdb, setting the ti.bios.oldMemoryNames flag is not necessary, since the memory configurations for these devices have not changed.

For a list of platforms for which the memory configurations changed, refer to Section B.1, *Platform Memory Configurations* for new configurations and to Section B.2, *Deprecated Platform Memory Configurations* for old memory configurations.

A.4.3 Changes to the loadSeed() Method

The loadSeed() function has been deprecated. The number of seed files provided with DSP/BIOS has been reduced to the minimum number needed to support Tconf. You can continue to use loadSeed() if your script uses one of the remaining seed files. Check the ti/bios/config/cdb folder for the available seed files. If the seed file you are using is no longer provided, you must modify your script to use the importFile() method with a .tci file that you will need to create.

You can replace the utils.loadSeed() statement with utils.importFile() where the import file contains the results from the following steps. (DSK6711 is used in this example; replace it with your platform as needed.)

1) Replace this statement:

   ```
   utils.loadSeed("dsk6711.cdb");
   ```

   with the following:

   ```
   utils.importFile("dsk6711.tci");
   ```

2) Copy the dsk6711.cdb file to a temp or working folder.

3) Open the dsk6711.cdb file with a text editor to find which seed this file was derived from. The seed version is shown in the first or second line of the file. For example, if you see a line of the form “!# c6211.cdb 4.90.270”, the original seed file is c6211.cdb.

4) Use the following command to generate the dsk6711.tcf file.

   ```
   cdb2tcf C:\ti\c6000\bios\include\c6211.cdb dsk6711.cdb
   ```

5) Open the dsk6711.tcf file with a text editor, and remove the “prog.gen()” statement from this file.

6) Rename this file “dsk6711.tci”.

7) Copy dsk6711.tci to a common folder. Include this common folder in the importPath for tconf so that utils.importFile() can find this file.

Appendix B

Configurations for Supported Platforms and Devices

This appendix provides lists of the platforms and devices supported by DSP/BIOS and their memory configurations.

Topic **Page**

B.1 Platform Memory Configurations

The platforms supplied with DSP/BIOS bring the following memory segments into a configuration:

Platform	Internal memory		External memory
ezdsp2808	name:	H0SARAM	
	base:	0xa000	
	size:	0x2000	
	space:	code	
	name:	BOOTROM	
	base:	0x3ff000	
	size:	0xfc0	
	space:	code	
	name:	FLASH	
	base:	0x3e8000	
	size:	0x10000	
	space:	code	
	name:	MSARAM	
	base:	0x0	
	size:	0x800	
	space:	data	
	name:	LSARAM	
	base:	0x8000	
	size:	0x2000	
	space:	data	
	name:	OTP	
	base:	0x3d7800	
	size:	0x400	
	space:	code	
	name:	PIEVECT	
	base:	0xd00	
	size:	0x100	
	space:	data	

Platform	**Internal memory**		**External memory**	
ezdsp2812, sim28xx	name:	H0SARAM	name:	SRAM
	base:	0x3f8000	base:	0x100000
	size:	0x2000	size:	0x10000
	space:	code	space:	data
	name:	BOOTROM		
	base:	0x3ff000		
	size:	0xfc0		
	space:	code		
	name:	FLASH		
	base:	0x3d8000		
	size:	0x20000		
	space:	code		
	name:	MSARAM		
	base:	0x0		
	size:	0x800		
	space:	data		
	name:	LSARAM		
	base:	0x8000		
	size:	0x2000		
	space:	data		
	name:	OTP		
	base:	0x3d7800		
	size:	0x400		
	space:	code		
	name:	PIEVECT		
	base:	0xd00		
	size:	0x100		
	space:	data		

Platform	Internal memory		External memory
sim2810	name:	H0SARAM	
	base:	0x3f8000	
	size:	0x2000	
	space:	code	
	name:	BOOTROM	
	base:	0x3ff000	
	size:	0xfc0	
	space:	code	
	name:	FLASH	
	base:	0x3e8000	
	size:	0x10000	
	space:	code	
	name:	MSARAM	
	base:	0x0	
	size:	0x800	
	space:	data	
	name:	LSARAM	
	base:	0x8000	
	size:	0x2000	
	space:	data	
	name:	OTP	
	base:	0x3d7800	
	size:	0x400	
	space:	code	
	name:	PIEVECT	
	base:	0xd00	
	size:	0x100	
	space:	data	

Platform	Internal memory		External memory	
dsk5402	name:	D_SPRAM	name:	EDATA
	base:	0x60	base:	0x8000
	size:	0x1a	size:	0x7000
	space:	data	space:	data
	name:	CSLREGS	name:	EPROG
	base:	0x7a	base:	0x8000
	size:	0x2	size:	0x7f80
	space:	data	space:	code
	name:	BIOSREGS		
	base:	0x7c		
	size:	0x4		
	space:	data		
	name:	D_DARAM		
	base:	0x80		
	size:	0x1f80		
	space:	data		
	name:	P_DARAM		
	base:	0x2000		
	size:	0x2000		
	space:	code		
	name:	VECT		
	base:	0xff80		
	size:	0x80		
	space:	code		
dsk5416	name:	D_SPRAM		
	base:	0x60		
	size:	0x1a		
	space:	data		
	name:	CSLREGS		
	base:	0x7a		
	size:	0x2		
	space:	data		
	name:	BIOSREGS		
	base:	0x7c		
	size:	0x4		
	space:	data		

Platform	Internal memory		External memory
dsk5416 (cont.)	name:	D_DARAM03	
	base:	0x80	
	size:	0x7000	
	space:	data	
	name:	D_DARAM47	
	base:	0x8000	
	size:	0x8000	
	space:	data	
	name:	P_DARAM03	
	base:	0x7080	
	size:	0xf00	
	space:	code	
	name:	VECT	
	base:	0x7f80	
	size:	0x80	
	space:	code	
	name:	P_ROM	
	base:	0xc000	
	size:	0x3f00	
	space:	code	
	name:	P_DARAM47	
	base:	0x18000	
	size:	0x8000	
	space:	code	
	name:	P_SARAM03	
	base:	0x28000	
	size:	0x8000	
	space:	code	
	name:	P_SARAM47	
	base:	0x38000	
	size:	0x8000	
	space:	code	

Platform	**Internal memory**		**External memory**	
evm5471	name:	D_SPRAM	name:	EDATA
	base:	0x60	base:	0x8000
	size:	0x1a	size:	0x8000
	space:	data	space:	data
	name:	CSLREGS	name:	EPROG
	base:	0x7a	base:	0x8000
	size:	0x2	size:	0x7f80
	space:	data	space:	code
	name:	BIOSREGS	name:	VECT
	base:	0x7c	base:	0xff80
	size:	0x4	size:	0x80
	space:	data	space:	code
	name:	D_DARAM		
	base:	0x80		
	size:	0x1f80		
	space:	data		
	name:	D_APIDARAM		
	base:	0x2000		
	size:	0x2000		
	space:	data		
	name:	D_SARAM0		
	base:	0x4000		
	size:	0x2000		
	space:	data		
	name:	D_SARAM1		
	base:	0x6000		
	size:	0x2000		
	space:	data		
	name:	P_DARAM		
	base:	0x80		
	size:	0x1f80		
	space:	code		
	name:	P_APIDARAM		
	base:	0x2000		
	size:	0x2000		
	space:	code		
	name:	P_SARAM0		
	base:	0x4000		
	size:	0x2000		
	space:	code		

Platform	Internal memory		External memory	
sim54xx	name:	D_SPRAM	name:	EDATA
	base:	0x60	base:	0x4000
	size:	0x1a	size:	0xb000
	space:	data	space:	data
	name:	CSLREGS	name:	EPROG
	base:	0x7a	base:	0x4000
	size:	0x2	size:	0xbf80
	space:	data	space:	code
	name:	BIOSREGS	name:	VECT
	base:	0x7c	base:	0xff80
	size:	0x4	size:	0x80
	space:	data	space:	code
	name:	D_DARAM		
	base:	0x80		
	size:	0x1f80		
	space:	data		
	name:	P_DARAM		
	base:	0x2000		
	size:	0x2000		
	space:	code		
evm5509, evm5509A	name:	DARAM	name:	SDRAM
	base:	0x60	base:	0x20000
	size:	0x7fa0	size:	0x1e0000
	space:	code/data	space:	code/data
	name:	SARAM	name:	FLASH
	base:	0x8000	base:	0x200000
	size:	0x17f80	size:	0x100000
	space:	code/data	space:	code/data
	name:	VECT		
	base:	0x1ff80		
	size:	0x80		
	space:	code/data		
dsk5510	name:	DARAM	name:	SDRAM
	base:	0x60	base:	0x28000
	size:	0x7fa0	size:	0x1d8000
	space:	code/data	space:	code/data
	name:	SARAM1	name:	FLASH
	base:	0x8000	base:	0x200000
	size:	0x8000	size:	0x40000
	space:	code/data	space:	code/data

Platform	Internal memory		External memory
dsk5510 (cont.)	name:	SARAM	
	base:	0x10000	
	size:	0x10000	
	space:	code/data	
	name:	SARAM2	
	base:	0x20000	
	size:	0x7f80	
	space:	code/data	
	name:	VECT	
	base:	0x27f80	
	size:	0x80	
	space:	code/data	
sim55xx	name:	DARAM	
	base:	0x60	
	size:	0x7fa0	
	space:	code/data	
	name:	SARAM1	
	base:	0x8000	
	size:	0x8000	
	space:	code/data	
	name:	SARAM	
	base:	0x10000	
	size:	0x10000	
	space:	code/data	
	name:	SARAM2	
	base:	0x20000	
	size:	0x7f80	
	space:	code/data	
	name:	VECT	
	base:	0x27f80	
	size:	0x80	
	space:	code/data	
teb5561	name:	DARAM	
	base:	0x60	
	size:	0x7fa0	
	space:	data	
	name:	SARAM	
	base:	0x8000	
	size:	0x7f80	
	space:	code/data	

Platform	Internal memory		External memory	
teb5561 (cont.)	name:	SARAM1		
	base:	0x10000		
	size:	0x10000		
	space:	data		
	name:	SARAM2		
	base:	0x20000		
	size:	0x10000		
	space:	data		
	name:	VECT		
	base:	0xff80		
	size:	0x80		
	space:	code/data		
	name:	SHRAM		
	base:	0x7c0000		
	size:	0x3f800		
	space:	code/data		
sdbTitan	name:	SARAM	name:	SDRAM
	base:	0x8000	base:	0x200000
	size:	0x7f80	size:	0x200000
	space:	code/data	space:	code/data
	name:	DARAM01		
	base:	0x80		
	size:	0x3f80		
	space:	code/data		
	name:	VECT		
	base:	0xff80		
	size:	0x80		
	space:	code/data		
	name:	DARAM23		
	base:	0x4000		
	size:	0x4000		
	space:	data		

Platform	Internal memory		External memory	
innovator1510, h2omap1610, h3omap1710, h4omap2420	name:	DARAM		
	base:	0x60		
	size:	0x7fa0		
	space:	code/data		
	name:	SARAM		
	base:	0x8000		
	size:	0xbf80		
	space:	code/data		
	name:	VECT		
	base:	0x13f80		
	size:	0x80		
	space:	code/data		
osk5912	name:	DARAM	name:	SDRAM
	base:	0x60	base:	0x200000
	size:	0x7fa0	size:	0x080000
	space:	code/data	space:	code/data
	name:	SARAM		
	base:	0x8000		
	size:	0xbf80		
	space:	code/data		
	name:	VECT		
	base:	0x13f80		
	size:	0x80		
	space:	code/data		
dsk6211	name:	IRAM	name:	SDRAM
	base:	0x0	base:	0x80000000
	size:	0x10000	size:	0x400000
	space:	code/data	space:	code/data
sim62xx	name:	IPRAM	name:	SBSRAM
	base:	0x0	base:	0x40000
	size:	0x10000	size:	0x40000
	space:	code	space:	code/data
	name:	IDRAM	name:	SDRAM0
	base:	0x80000000	base:	0x2000000
	size:	0x10000	size:	0x400000
	space:	data	space:	code/data
			name:	SDRAM1
			base:	0x3000000
			size:	0x400000
			space:	code/data

Platform	Internal memory		External memory	
dsk6416,	name:	IRAM	name:	SDRAM
teb6416,	base:	0x0	base:	0x80000000
sim64xx	size:	0x100000	size:	0x1000000
	space:	code/data	space:	code/data
evmDM642	name:	IRAM	name:	SDRAM
	base:	0x0	base:	0x80000000
	size:	0x40000	size:	0x2000000
	space:	code/data	space:	code/data
dsk6711,	name:	IRAM	name:	SDRAM
sim67xx	base:	0x0	base:	0x80000000
	size:	0x10000	size:	0x1000000
	space:	code/data	space:	code/data
dsk6713	name:	IRAM	name:	SDRAM
	base:	0x0	base:	0x80000000
	size:	0x40000	size:	0x800000
	space:	code/data	space:	code/data
padk6727	name:	IRAM	name:	SDRAM
	base:	0x10000000	base:	0x80000000
	size:	0x40000	size:	0x8000000
	space:	code/data	space:	code/data
dsk6455	name:	IRAM	name:	DDR2
	base:	0x800000	base:	0x20000000
	size:	0x200000	size:	0x10000000
	space:	code/data	space:	code/data
dskTCI6482	name:	IRAM	name:	DDR
	base:	0x800000	base:	0x20000000
	size:	0x200000	size:	0x10000000
	space:	code/data	space:	code/data
sim64Pxx	name:	IRAM	name:	DDR
	base:	0x800000	base:	0x80000000
	size:	0x200000	size:	0x10000000
	space:	code/data	space:	code/data

Platform	Internal memory		External memory	
evmDM420	name:	IRAM	name:	DDR2
	base:	0x11800000	base:	0x80000000
	size:	0x10000	size:	0x10000000
	space:	code/data	space:	code/data
	name:	ARM_RAM		
	base:	0x10008000		
	size:	0x4000		
	space:	data		
	name:	L1DSRAM		
	base:	0x11f04000		
	size:	0xc000		
	space:	data		
	name:	IMCOP		
	base:	0x11100000		
	size:	0x1f400		
	space:	data		
evmDM6446	name:	IRAM	name:	DDR2
	base:	0x11800000	base:	0x80000000
	size:	0x10000	size:	0x10000000
	space:	code/data	space:	code/data
	name:	ARM_RAM		
	base:	0x10008000		
	size:	0x4000		
	space:	data		
	name:	L1DSRAM		
	base:	0x11f04000		
	size:	0xc000		
	space:	data		
	name:	VICP		
	base:	0x11100000		
	size:	0x1f400		
	space:	data		
sdp2430	name:	IRAM	name:	DDR
	base:	0x10800000	base:	0x80000000
	size:	0x10000	size:	0x8000000
	space:	code/data	space:	code/data
	name:	L1DSRAM		
	base:	0x10f04000		
	size:	0xc000		
	space:	data		

B.2 Deprecated Platform Memory Configurations

In the DSP/BIOS releases before 5.0, some TI-supplied platforms had different memory configurations. Please refer to this table, if you had scripts that load platforms using the old platform names. For example, "Dsk6416" instead of a new name "ti.platforms.dsk6416".

Platform	**Internal memory**		**External memory**	
dsk5402	name:	USERREGS	name:	EDATA
	base:	0x60	base:	0x8000
	size:	0x1a	size:	0x8000
	space:	data	space:	data
	name:	CSLREGS	name:	EPROG
	base:	0x7a	base:	0x8000
	size:	0x2	size:	0x7f80
	space:	data	space:	code
	name:	BIOSREGS		
	base:	0x7c		
	size:	0x4		
	space:	data		
	name:	IDATA		
	base:	0x80		
	size:	0x1f80		
	space:	data		
	name:	IPROG		
	base:	0x2000		
	size:	0x2000		
	space:	code		
	name:	VECT		
	base:	0xff80		
	size:	0x80		
	space:	code		

Platform	Internal memory		External memory
dsk5416	name:	USERREGS	
	base:	0x60	
	size:	0x1a	
	space:	data	
	name:	CSLREGS	
	base:	0x7a	
	size:	0x2	
	space:	data	
	name:	BIOSREGS	
	base:	0x7c	
	size:	0x4	
	space:	data	
	name:	IDATA	
	base:	0x80	
	size:	0x7000	
	space:	data	
	name:	DARAM47	
	base:	0x8000	
	size:	0x8000	
	space:	data	
	name:	IPROG	
	base:	0x7080	
	size:	0xf00	
	space:	code	
	name:	VECT	
	base:	0x7f80	
	size:	0x80	
	space:	code	
	name:	SARAM03	
	base:	0x28000	
	size:	0x8000	
	space:	code	
	name:	SARAM47	
	base:	0x38000	
	size:	0x8000	
	space:	code	

Platform	Internal memory		External memory	
evm5471	name:	USERREGS	name:	EDATA
	base:	0x60	base:	0x8000
	size:	0x1a	size:	0x8000
	space:	data	space:	data
	name:	CSLREGS	name:	EPROG
	base:	0x7a	base:	0x6000
	size:	0x2	size:	0x9f80
	space:	data	space:	code
	name:	BIOSREGS		
	base:	0x7c		
	size:	0x4		
	space:	data		
	name:	IDATA		
	base:	0x80		
	size:	0x1f80		
	space:	data		
	name:	APIRAM1		
	base:	0x2000		
	size:	0x1800		
	space:	data		
	name:	APIRAM2		
	base:	0x3800		
	size:	0x800		
	space:	data		
	name:	IDATA1		
	base:	0x6000		
	size:	0x2000		
	space:	data		
	name:	IPROG		
	base:	0x4000		
	size:	0x2000		
	space:	code		
	name:	VECT		
	base:	0xff80		
	size:	0x80		
	space:	code		

Platform	Internal memory		External memory	
sim54xx	name:	USERREGS	name:	EDATA
	base:	0x60	base:	0x8000
	size:	0x1a	size:	0x4000
	space:	data	space:	data
	name:	CSLREGS	name:	EPROG
	base:	0x7a	base:	0xc000
	size:	0x2	size:	0x3f80
	space:	data	space:	code
	name:	BIOSREGS		
	base:	0x7c		
	size:	0x4		
	space:	data		
	name:	IDATA		
	base:	0x80		
	size:	0x1f80		
	space:	data		
	name:	IPROG		
	base:	0x2000		
	size:	0x2000		
	space:	code		
	name:	VECT		
	base:	0xff80		
	size:	0x80		
	space:	code		

Platform	Internal memory		External memory	
evm5509	name:	VECT	name:	SDRAM
	base:	0x80	base:	0x20000
	size:	0x80	size:	0x1d8000
	space:	code/data	space:	code/data
	name:	DARAM	name:	FLASH
	base:	0x100	base:	0x200000
	size:	0x7f00	size:	0x100000
	space:	code/data	space:	code/data
	name:	SARAM_A		
	base:	0x8000		
	size:	0x8000		
	space:	code/data		
	name:	SARAM		
	base:	0x10000		
	size:	0x8000		
	space:	code/data		
	name:	SARAM_B		
	base:	0x18000		
	size:	0x8000		
	space:	code/data		
dsk5510	name:	VECT	name:	SDRAM
	base:	0x80	base:	0x28000
	size:	0x80	size:	0x1d8000
	space:	code/data	space:	code/data
	name:	DARAM	name:	FLASH
	base:	0x100	base:	0x200000
	size:	0x7f00	size:	0x40000
	space:	code/data	space:	code/data
	name:	SARAM_A		
	base:	0x8000		
	size:	0x8000		
	space:	code/data		
	name:	SARAM		
	base:	0x10000		
	size:	0x10000		
	space:	code/data		
	name:	SARAM_B		
	base:	0x20000		
	size:	0x8000		
	space:	code/data		

Platform	Internal memory		External memory
teb5561	name: base: size: space:	VECT 0x80 0x80 code/data	
	name: base: size: space:	DARAM 0x100 0x7f00 data	
	name: base: size: space:	SARAM 0x8000 0x7f80 code/data	
	name: base: size: space:	SARAM1 0x10000 0x10000 data	
	name: base: size: space:	SARAM2 0x20000 0x10000 data	
	name: base: size: space:	SHRAM 0x7c0000 0x3f800 code/data	
innovator1510	name: base: size: space:	VECT 0x80 0x80 code/data	
	name: base: size: space:	DARAM 0x100 0x7f00 code/data	
	name: base: size: space:	SARAM 0x8000 0xc000 code/data	
	name: base: size: space:	PDROM 0x7fc000 0x4000 code/data	

Platform	**Internal memory**		**External memory**	
sim55xx	name:	VECT		
	base:	0x80		
	size:	0x80		
	space:	code/data		
	name:	DARAM		
	base:	0x100		
	size:	0x7f00		
	space:	code/data		
	name:	SARAM_A		
	base:	0x8000		
	size:	0x8000		
	space:	code/data		
	name:	SARAM		
	base:	0x10000		
	size:	0x10000		
	space:	code/data		
	name:	SARAM_B		
	base:	0x20000		
	size:	0x8000		
	space:	code/data		
sim62xx	name:	IPRAM		
	base:	0x0		
	size:	0x10000		
	space:	code		
	name:	IDRAM		
	base:	0x80000000		
	size:	0x10000		
	space:	data		
dsk6416, teb6416	name:	ISRAM	name:	SDRAM
	base:	0x0	base:	0x80000000
	size:	0x100000	size:	0x1000000
	space:	code/data	space:	code/data
evmDM642	name:	ISRAM	name:	SDRAM
	base:	0x0	base:	0x80000000
	size:	0x40000	size:	0x2000000
	space:	code/data	space:	code/data
sim64xx	name:	ISRAM		
	base:	0x0		
	size:	0x100000		
	space:	code/data		

B.3 Device Memory Configurations

The following table lists devices supported in DSP/BIOS and their memory configurations. Please consult release notes for any additional devices and their configurations.

The 54xx memory configurations are valid for the following values of the relevant bits in the PMST register: MP/MC=0, OVLY=1, DROM=1. Depending on the values of the PMST register in your configuration, some listed memory segments may not be a part of your configuration.

The 55xx and OMAP memory configurations are valid for the following values of the relevant bits in the ST3 register: MP/MC=0. Depending on the values of the ST3 register in your configuration, some listed memory segments may not be a part of your configuration.

The 6x memory configurations assume that no internal memory is used for L2 cache. For 64+ devices, both L1P and L1D cache are set to maximum size.

Device Name	Memory configuration	
TNETV1050, TNETV1055	name:	SARAM
	base:	0x8000
	size:	0x7f80
	space:	code/data
	name:	DARAM01
	base:	0x80
	size:	0x3f80
	space:	code/data
	name:	VECT
	base:	0xff80
	size:	0x80
	space:	code/data
	name:	DARAM23
	base:	0x4000
	size:	0x4000
	space:	data

Device Name	Memory configuration	
2801	name:	BOOTROM
	base:	0x3fc000
	size:	0x3fc0
	space:	code
	name:	FLASH
	base:	0x3f4000
	size:	0x4000
	space:	code
	name:	MSARAM
	base:	0x0
	size:	0x800
	space:	data
	name:	LSARAM
	base:	0x8000
	size:	0x1000
	space:	data
	name:	OTP
	base:	0x3d7800
	size:	0x400
	space:	code
	name:	PIEVECT
	base:	0xd00
	size:	0x100
	space:	data
2806	name:	BOOTROM
	base:	0x3fc000
	size:	0x3fc0
	space:	code
	name:	FLASH
	base:	0x3f0000
	size:	0x8000
	space:	code
	name:	MSARAM
	base:	0x0
	size:	0x800
	space:	data

Device Name	**Memory configuration**	
2806 (cont.)	name:	OTP
	base:	0x3d7800
	size:	0x400
	space:	code
	name:	L1SARAM
	base:	0x9000
	size:	0x1000
	space:	code
	name:	PIEVECT
	base:	0xd00
	size:	0x100
	space:	data
	name:	L0SARAM
	base:	0x8000
	size:	0x1000
	space:	data
2808	name:	H0SARAM
	base:	0xa000
	size:	0x2000
	space:	code
	name:	BOOTROM
	base:	0x3ff000
	size:	0xfc0
	space:	code
	name:	FLASH
	base:	0x3e8000
	size:	0x10000
	space:	code
	name:	MSARAM
	base:	0x0
	size:	0x800
	space:	data
	name:	LSARAM
	base:	0x8000
	size:	0x2000
	space:	data

Device Name	Memory configuration	
2808 (cont.)	name:	OTP
	base:	0x3d7800
	size:	0x400
	space:	code
	name:	PIEVECT
	base:	0xd00
	size:	0x100
	space:	data
2810	name:	H0SARAM
	base:	0x3f8000
	size:	0x2000
	space:	code
	name:	BOOTROM
	base:	0x3ff000
	size:	0xfc0
	space:	code
	name:	FLASH
	base:	0x3e8000
	size:	0x10000
	space:	code
	name:	MSARAM
	base:	0x0
	size:	0x800
	space:	data
	name:	LSARAM
	base:	0x8000
	size:	0x2000
	space:	data
	name:	OTP
	base:	0x3d7800
	size:	0x400
	space:	code
	name:	PIEVECT
	base:	0xd00
	size:	0x100
	space:	data

Device Name	Memory configuration	
2811, 2812	name:	H0SARAM
	base:	0x3f8000
	size:	0x2000
	space:	code
	name:	BOOTROM
	base:	0x3ff000
	size:	0xfc0
	space:	code
	name:	FLASH
	base:	0x3d8000
	size:	0x20000
	space:	code
	name:	MSARAM
	base:	0x0
	size:	0x800
	space:	data
	name:	LSARAM
	base:	0x8000
	size:	0x2000
	space:	data
	name:	OTP
	base:	0x3d7800
	size:	0x400
	space:	code
	name:	PIEVECT
	base:	0xd00
	size:	0x100
	space:	data
5401	name:	D_SPRAM
	base:	0x60
	size:	0x1a
	space:	data
	name:	CSLREGS
	base:	0x7a
	size:	0x2
	space:	data
	name:	BIOSREGS
	base:	0x7c
	size:	0x4
	space:	data

Device Name	**Memory configuration**	
5401 (cont.)	name:	D_DARAM
	base:	0x1000
	size:	0x1000
	space:	data
	name:	D_ROM
	base:	0xf000
	size:	0xf00
	space:	data
	name:	P_DARAM
	base:	0x2000
	size:	0x1000
	space:	code
	name:	P_ROM
	base:	0xf000
	size:	0xf00
	space:	code
	name:	VECT
	base:	0xff80
	size:	0x80
	space:	code
5402	name:	D_SPRAM
	base:	0x60
	size:	0x1a
	space:	data
	name:	CSLREGS
	base:	0x7a
	size:	0x2
	space:	data
	name:	BIOSREGS
	base:	0x7c
	size:	0x4
	space:	data
	name:	D_DARAM
	base:	0x80
	size:	0x1f80
	space:	data
	name:	D_ROM
	base:	0xf000
	size:	0xf00
	space:	data

Device Name	Memory configuration	
5402 (cont.)	name:	P_DARAM
	base:	0x2000
	size:	0x2000
	space:	code
	name:	P_ROM
	base:	0xf000
	size:	0xf00
	space:	code
	name:	VECT
	base:	0xff80
	size:	0x80
	space:	code
5402A	name:	D_SPRAM
	base:	0x60
	size:	0x1a
	space:	data
	name:	CSLREGS
	base:	0x7a
	size:	0x2
	space:	data
	name:	BIOSREGS
	base:	0x7c
	size:	0x4
	space:	data
	name:	D_DARAM
	base:	0x80
	size:	0x1f80
	space:	data
	name:	P_DARAM
	base:	0x2000
	size:	0x2000
	space:	code
	name:	P_ROM
	base:	0xc000
	size:	0x3f00
	space:	code
	name:	VECT
	base:	0xff80
	size:	0x80
	space:	code

Device Name	Memory configuration	
5404	name:	D_SPRAM
	base:	0x60
	size:	0x1a
	space:	data
	name:	CSLREGS
	base:	0x7a
	size:	0x2
	space:	data
	name:	BIOSREGS
	base:	0x7c
	size:	0x4
	space:	data
	name:	D_DARAM
	base:	0x80
	size:	0x1f80
	space:	data
	name:	D_ROM
	base:	0xc000
	size:	0x4000
	space:	data
	name:	P_DARAM
	base:	0x2000
	size:	0x2000
	space:	code
	name:	P_ROM0
	base:	0x8000
	size:	0x7f00
	space:	code
	name:	P_ROM1
	base:	0x18000
	size:	0x8000
	space:	code
	name:	VECT
	base:	0xff80
	size:	0x80
	space:	code

Device Name	Memory configuration	
5405	name:	D_ROM
	base:	0xf000
	size:	0xf00
	space:	data
	name:	P_ROM
	base:	0xf000
	size:	0xf00
	space:	code
	name:	CSLREGS
	base:	0x7a
	size:	0x2
	space:	data
	name:	VECT
	base:	0xff80
	size:	0x80
	space:	code
	name:	P_DARAM
	base:	0x2000
	size:	0x2000
	space:	code
	name:	D_DARAM
	base:	0x80
	size:	0x1f80
	space:	data
	name:	D_SPRAM
	base:	0x60
	size:	0x1a
	space:	data
	name:	BIOSREGS
	base:	0x7c
	size:	0x4
	space:	data
5407	name:	D_SPRAM
	base:	0x60
	size:	0x1a
	space:	data

Device Name	Memory configuration	
5407 (cont.)	name:	CSLREGS
	base:	0x7a
	size:	0x2
	space:	data
	name:	BIOSREGS
	base:	0x7c
	size:	0x4
	space:	data
	name:	D_DARAM04
	base:	0x6000
	size:	0x4000
	space:	data
	name:	D_ROM
	base:	0xc000
	size:	0x4000
	space:	data
	name:	P_DARAM02
	base:	0x80
	size:	0x5f80
	space:	code
	name:	P_ROM0
	base:	0x6000
	size:	0x9f00
	space:	code
	name:	VECT
	base:	0xff80
	size:	0x80
	space:	code
	name:	P_ROM1
	base:	0x18000
	size:	0x8000
	space:	code
	name:	P_ROM2
	base:	0x28000
	size:	0x8000
	space:	code
	name:	P_ROM3
	base:	0x38000
	size:	0x6000
	space:	code

Device Name	Memory configuration	
5409	name:	D_SPRAM
	base:	0x60
	size:	0x1a
	space:	data
	name:	CSLREGS
	base:	0x7a
	size:	0x2
	space:	data
	name:	BIOSREGS
	base:	0x7c
	size:	0x4
	space:	data
	name:	D_DARAM
	base:	0x80
	size:	0x3f80
	space:	data
	name:	D_ROM
	base:	0xc000
	size:	0x3f00
	space:	data
	name:	P_DARAM
	base:	0x4000
	size:	0x4000
	space:	code
	name:	P_ROM
	base:	0xc000
	size:	0x3f00
	space:	code
	name:	VECT
	base:	0xff80
	size:	0x80
	space:	code

Device Name	Memory configuration	
5409A	name:	P_ROM
	base:	0xc000
	size:	0x3f00
	space:	code
	name:	CSLREGS
	base:	0x7a
	size:	0x2
	space:	data
	name:	VECT
	base:	0xff80
	size:	0x80
	space:	code
	name:	P_DARAM
	base:	0x4000
	size:	0x4000
	space:	code
	name:	D_DARAM
	base:	0x80
	size:	0x3f80
	space:	data
	name:	D_SPRAM
	base:	0x60
	size:	0x1a
	space:	data
	name:	BIOSREGS
	base:	0x7c
	size:	0x4
	space:	data

Device Name	**Memory configuration**	
5410	name:	D_SPRAM
	base:	0x60
	size:	0x1a
	space:	data
	name:	CSLREGS
	base:	0x7a
	size:	0x2
	space:	data
	name:	BIOSREGS
	base:	0x7c
	size:	0x4
	space:	data
	name:	D_DARAM
	base:	0x80
	size:	0x1f80
	space:	data
	name:	D_SARAM2
	base:	0x8000
	size:	0x8000
	space:	data
	name:	P_SARAM1
	base:	0x2000
	size:	0x6000
	space:	code
	name:	P_SARAM2
	base:	0x18000
	size:	0x8000
	space:	code
	name:	VECT
	base:	0xff80
	size:	0x80
	space:	code
	name:	P_ROM
	base:	0xc000
	size:	0x3f80
	space:	code

Device Name	Memory configuration	
5410A	name:	D_DARAM03
	base:	0x80
	size:	0x3f80
	space:	data
	name:	CSLREGS
	base:	0x7a
	size:	0x2
	space:	data
	name:	VECT
	base:	0xff80
	size:	0x80
	space:	code
	name:	D_SPRAM
	base:	0x60
	size:	0x1a
	space:	data
	name:	P_DARAM47
	base:	0x18000
	size:	0x8000
	space:	code
	name:	P_ROM
	base:	0xc000
	size:	0x3f00
	space:	code
	name:	P_DARAM03
	base:	0x4000
	size:	0x4000
	space:	code
	name:	BIOSREGS
	base:	0x7c
	size:	0x4
	space:	data
	name:	D_DARAM47
	base:	0x8000
	size:	0x8000
	space:	data
5416	name:	D_DARAM03
	base:	0x80
	size:	0x7000
	space:	data

Device Name	**Memory configuration**	
5416 (cont.)	name:	CSLREGS
	base:	0x7a
	size:	0x2
	space:	data
	name:	VECT
	base:	0x7f80
	size:	0x80
	space:	code
	name:	D_SPRAM
	base:	0x60
	size:	0x1a
	space:	data
	name:	P_DARAM47
	base:	0x18000
	size:	0x8000
	space:	code
	name:	P_ROM
	base:	0xc000
	size:	0x3f00
	space:	code
	name:	P_DARAM03
	base:	0x7080
	size:	0xf00
	space:	code
	name:	P_SARAM47
	base:	0x38000
	size:	0x8000
	space:	code
	name:	BIOSREGS
	base:	0x7c
	size:	0x4
	space:	data
	name:	D_DARAM47
	base:	0x8000
	size:	0x8000
	space:	data
	name:	P_SARAM03
	base:	0x28000
	size:	0x8000
	space:	code

Device Name	Memory configuration	
5420	name:	P_SARAM3
	base:	0x18000
	size:	0x8000
	space:	code
	name:	P_SARAM2
	base:	0xc000
	size:	0x3f80
	space:	code
	name:	CSLREGS
	base:	0x7a
	size:	0x2
	space:	data
	name:	P_SARAM1
	base:	0x4000
	size:	0x3f80
	space:	code
	name:	D_SARAM2
	base:	0x8000
	size:	0x4000
	space:	data
	name:	D_DARAM0
	base:	0x80
	size:	0x3f80
	space:	data
	name:	VECT
	base:	0x7f80
	size:	0x80
	space:	code
	name:	D_SPRAM
	base:	0x60
	size:	0x1a
	space:	data
	name:	BIOSREGS
	base:	0x7c
	size:	0x4
	space:	data

Device Name	Memory configuration	
5470, 5471	name:	P_SARAM2
	base:	0x8000
	size:	0x7f80
	space:	code
	name:	D_APIDARAM
	base:	0x2000
	size:	0x2000
	space:	data
	name:	CSLREGS
	base:	0x7a
	size:	0x2
	space:	data
	name:	P_SARAM1
	base:	0x6000
	size:	0x2000
	space:	code
	name:	D_SARAM2
	base:	0xc000
	size:	0x4000
	space:	data
	name:	P_SARAM0
	base:	0x4000
	size:	0x2000
	space:	code
	name:	D_SARAM1
	base:	0x6000
	size:	0x2000
	space:	data
	name:	D_SARAM0
	base:	0x4000
	size:	0x2000
	space:	data
	name:	VECT
	base:	0xff80
	size:	0x80
	space:	code
	name:	D_DARAM
	base:	0x80
	size:	0x1f80
	space:	data
	name:	D_SPRAM
	base:	0x60
	size:	0x1a
	space:	data

Device Name	Memory configuration	
5470, 5471 (cont.)	name:	P_DARAM
	base:	0x80
	size:	0x1f80
	space:	code
	name:	BIOSREGS
	base:	0x7c
	size:	0x4
	space:	data
	name:	P_APIDARAM
	base:	0x2000
	size:	0x2000
	space:	code
54CST	name:	D_ROM
	base:	0xc000
	size:	0x4000
	space:	data
	name:	P_DARAM0
	base:	0x80
	size:	0x5f80
	space:	code
	name:	P_ROM
	base:	0x6000
	size:	0x9f80
	space:	code
	name:	CSLREGS
	base:	0x7a
	size:	0x2
	space:	data
	name:	VECT
	base:	0xff80
	size:	0x80
	space:	code
	name:	D_DARAM
	base:	0x6000
	size:	0x4000
	space:	data

Device Name	**Memory configuration**	
54CST (cont.)	name:	D_SPRAM
	base:	0x60
	size:	0x1a
	space:	data
	name:	BIOSREGS
	base:	0x7c
	size:	0x4
	space:	data
DM270	name:	CSLREGS
	base:	0x7a
	size:	0x2
	space:	data
	name:	VECT
	base:	0x7f80
	size:	0x80
	space:	code
	name:	D_SARAM
	base:	0x8000
	size:	0x4000
	space:	data
	name:	D_DARAM
	base:	0x80
	size:	0x3f80
	space:	data
	name:	D_SPRAM
	base:	0x60
	size:	0x1a
	space:	data
	name:	P_ROM2
	base:	0xc000
	size:	0x3f00
	space:	code
	name:	P_ROM1
	base:	0x8000
	size:	0x4000
	space:	code
	name:	D_ROM
	base:	0xc000
	size:	0x4000
	space:	data

Device Name	Memory configuration	
DM270 (cont.)	name:	P_DARAM
	base:	0x4000
	size:	0x3f80
	space:	code
	name:	P_SARAM
	base:	0x1c000
	size:	0x4000
	space:	code
	name:	BIOSREGS
	base:	0x7c
	size:	0x4
	space:	data
DM310	name:	CSLREGS
	base:	0x7a
	size:	0x2
	space:	data
	name:	IMAGE_BUFFER_AC
	base:	0x8000
	size:	0x2000
	space:	data
	name:	VECT
	base:	0x7f80
	size:	0x80
	space:	code
	name:	IMAGE_BUFFER_B
	base:	0xa000
	size:	0x2000
	space:	data
	name:	D_SPRAM
	base:	0x60
	size:	0x1a
	space:	data
	name:	D_DARAM
	base:	0x80
	size:	0x3f80
	space:	data
	name:	D_ROM
	base:	0xc000
	size:	0x4000
	space:	data

Device Name	**Memory configuration**	
DM310 (cont.)	name:	P_ROM
	base:	0xc000
	size:	0x4000
	space:	code
	name:	P_DARAM
	base:	0x3f80
	size:	0x4000
	space:	code
	name:	P_SARAM
	base:	0x18000
	size:	0x8000
	space:	code
	name:	BIOSREGS
	base:	0x7c
	size:	0x4
	space:	data
DM320	name:	CSLREGS
	base:	0x7a
	size:	0x2
	space:	data
	name:	VECT
	base:	0x7f80
	size:	0x80
	space:	code
	name:	D_SARAM
	base:	0x8000
	size:	0x4000
	space:	data
	name:	D_SPRAM
	base:	0x60
	size:	0x1a
	space:	data
	name:	D_DARAM
	base:	0x80
	size:	0x3f80
	space:	data
	name:	D_ROM
	base:	0xc000
	size:	0x4000
	space:	data

Device Name	**Memory configuration**	
DM320 (cont.)	name:	P_ROM
	base:	0x8000
	size:	0x7f00
	space:	code
	name:	P_DARAM
	base:	0x4000
	size:	0x3f80
	space:	code
	name:	P_SARAM
	base:	0x1c000
	size:	0x4000
	space:	code
	name:	BIOSREGS
	base:	0x7c
	size:	0x4
	space:	data
1510, 1610, 5903, 5905, 5910, 5912	name:	SARAM
	base:	0x8000
	size:	0xbf80
	space:	code/data
	name:	DARAM
	base:	0x60
	size:	0x7fa0
	space:	code/data
	name:	PDROM
	base:	0x7fc000
	size:	0x4000
	space:	code/data
	name:	VECT
	base:	0x13f80
	size:	0x80
	space:	code/data

Device Name	Memory configuration	
1710, 2420	name:	SARAM
	base:	0x8000
	size:	0xbf80
	space:	code/data
	name:	DARAM
	base:	0x60
	size:	0x7fa0
	space:	code/data
	name:	PDROM
	base:	0x7f8000
	size:	0x8000
	space:	code/data
	name:	VECT
	base:	0x13f80
	size:	0x80
	space:	code/data
5501	name:	ROM
	base:	0x7fc000
	size:	0x4000
	space:	code/data
	name:	DARAM
	base:	0x60
	size:	0x3f20
	space:	code/data
	name:	VECT
	base:	0x3f80
	size:	0x80
	space:	code/data
5502	name:	ROM
	base:	0x7fc000
	size:	0x4000
	space:	code/data
	name:	DARAM
	base:	0x60
	size:	0x7f20
	space:	code/data
	name:	VECT
	base:	0x7f80
	size:	0x80
	space:	code/data

Device Name	Memory configuration	
5503	name:	ROM
	base:	0x7f8000
	size:	0x8000
	space:	code/data
	name:	DARAM
	base:	0x60
	size:	0x7f20
	space:	code/data
	name:	VECT
	base:	0x7f80
	size:	0x80
	space:	code/data
5507	name:	ROM
	base:	0x7f8000
	size:	0x8000
	space:	code/data
	name:	SARAM
	base:	0x8000
	size:	0x7f80
	space:	code/data
	name:	DARAM
	base:	0x60
	size:	0x7fa0
	space:	code/data
	name:	VECT
	base:	0xff80
	size:	0x80
	space:	code/data

Device Name	Memory configuration	
5509, 5509A, DA255	name:	ROM
	base:	0x7f8000
	size:	0x8000
	space:	code/data
	name:	SARAM
	base:	0x8000
	size:	0x17f80
	space:	code/data
	name:	DARAM
	base:	0x60
	size:	0x7fa0
	space:	code/data
	name:	VECT
	base:	0x1ff80
	size:	0x80
	space:	code/data
5510, 5510A	name:	ROM
	base:	0x7fc000
	size:	0x4000
	space:	code/data
	name:	SARAM
	base:	0x10000
	size:	0x10000
	space:	code/data
	name:	DARAM
	base:	0x60
	size:	0x7fa0
	space:	code/data
	name:	SARAM2
	base:	0x20000
	size:	0x7f80
	space:	code/data
	name:	SARAM1
	base:	0x8000
	size:	0x8000
	space:	code/data
	name:	VECT
	base:	0x27f80
	size:	0x80
	space:	code/data

Device Name	Memory configuration	
5561	name:	ROM
	base:	0x7ff800
	size:	0x800
	space:	code/data
	name:	SARAM
	base:	0x8000
	size:	0x7f80
	space:	code/data
	name:	DARAM
	base:	0x60
	size:	0x7fa0
	space:	data
	name:	SARAM2
	base:	0x20000
	size:	0x10000
	space:	data
	name:	SARAM1
	base:	0x10000
	size:	0x10000
	space:	data
	name:	VECT
	base:	0xff80
	size:	0x80
	space:	code/data
	name:	SHRAM
	base:	0x7c0000
	size:	0x3f800
	space:	code/data
2430, 3430	name:	IRAM
	base:	0x10800000
	size:	0x10000
	space:	code/data
	name:	L1DSRAM
	base:	0x10f04000
	size:	0xc000
	space:	data

Device Name	Memory configuration	
5944, 5946, 5948	name:	SARAM
	base:	0x8000
	size:	0xbf80
	space:	code/data
	name:	DARAM
	base:	0x80
	size:	0x7f80
	space:	code/data
	name:	PDROM
	base:	0x7fc000
	size:	0x4000
	space:	code/data
	name:	VECT
	base:	0x13f80
	size:	0x80
	space:	code/data
DA295, DA300	name:	ROM
	base:	0x7f8000
	size:	0x8000
	space:	code/data
	name:	DARAM
	base:	0x60
	size:	0x17f20
	space:	code/data
	name:	VECT
	base:	0x17f80
	size:	0x80
	space:	code/data
6201, 6204, 6205, 6701	name:	IPRAM
	base:	0x0
	size:	0x10000
	space:	code
	name:	IDRAM
	base:	0x80000000
	size:	0x10000
	space:	data

Device Name	**Memory configuration**	
6202	name:	IPRAM
	base:	0x0
	size:	0x40000
	space:	code
	name:	IDRAM
	base:	0x80000000
	size:	0x20000
	space:	data
6203, 6203B	name:	IPRAM
	base:	0x0
	size:	0x60000
	space:	code
	name:	IDRAM
	base:	0x80000000
	size:	0x80000
	space:	data
6211, 6211B, 6711, 6711B, 6712	name:	IRAM
	base:	0x0
	size:	0x10000
	space:	code/data
6410, DM640, DM641	name:	IRAM
	base:	0x0
	size:	0x20000
	space:	code/data
6411, 6412, 6413, 6713, DM642	name:	IRAM
	base:	0x0
	size:	0x40000
	space:	code/data
6414, 6415, 6416	name:	IRAM
	base:	0x0
	size:	0x100000
	space:	code/data
6418, DRI300	name:	IRAM
	base:	0x0
	size:	0x80000
	space:	code/data
6722	name:	IRAM
	base:	0x10000000
	size:	0x20000
	space:	code/data

Device Name	Memory configuration	
DA700, DA705, DA707, DA710, 6726, 6727	name:	IRAM
	base:	0x10000000
	size:	0x40000
	space:	code/data
6455, TCI6482	name:	IRAM
	base:	0x800000
	size:	0x200000
	space:	code/data
TCI6486	name:	LL2RAM
	base:	0x800000
	size:	0x98000
	space:	code/data
	name:	SL2RAM
	base:	0x200000
	size:	0xc0000
	space:	code/data
DM415, DM420, DM421, DM425, DM426	name:	ARM_RAM
	base:	0x10008000
	size:	0x4000
	space:	data
	name:	IRAM
	base:	0x11800000
	size:	0x10000
	space:	code/data
	name:	L1DSRAM
	base:	0x11f04000
	size:	0xc000
	space:	data
	name:	IMCOP
	base:	0x11100000
	size:	0x1f400
	space:	data

Device Name	Memory configuration	
DM6443, DM6446	name:	ARM_RAM
	base:	0x10008000
	size:	0x4000
	space:	data
	name:	IRAM
	base:	0x11800000
	size:	0x10000
	space:	code/data
	name:	L1DSRAM
	base:	0x11f04000
	size:	0xc000
	space:	data
	name:	VICP
	base:	0x11100000
	size:	0x1f400
	space:	data

Index