

DSP/BIOS TextConf User's Guide

Literature Number: SPRU007C
November 2002



IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

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

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of that third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:
Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Read This First

About This Manual

DSP/BIOS gives developers of mainstream applications on Texas Instruments TMS320 DSP devices the ability to develop embedded real-time software. DSP/BIOS provides a small firmware real-time library and easy-to-use tools for real-time tracing and analysis.

This book 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 reference information about DSP/BIOS modules and properties discussed in this book.

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.
- ❑ <ccs_base_dir> indicates the directory in which Code Composer Studio was installed.

Trademarks

The Texas Instruments logo and Texas Instruments are registered trademarks of Texas Instruments. Trademarks of Texas Instruments include: TI, XDS, Code Composer, Code Composer Studio, Probe Point, Code Explorer, DSP/BIOS, RTDX, Online DSP Lab, TMS320, TMS320C54x, TMS320C55x, TMS320C62x, TMS320C64x, TMS320C67x, TMS320C5000, and TMS320C6000.

MS-DOS, Windows, and Windows NT are trademarks of Microsoft Corporation.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Solaris and SunOS are trademarks or registered trademarks of Sun Microsystems, Inc.

All other brand, product names, and service names are trademarks or registered trademarks of their respective companies or organizations.

Licences

The tconf command-line utility uses the Rhino open-source implementation of JavaScript. This is available at <http://www.mozilla.org/rhino>. The source code used by the tconf utility is available in the js.jar Java archive included with the utility. For licensing information about this JavaScript implementation, please see <http://www.mozilla.org/MPL>.

Contents

1 DSP/BIOS TextConf Overview	1-1
<i>This chapter compares the two methods for configuring DSP/BIOS programs and provides details about using DSP/BIOS TextConf.</i>	
1.1 DSP/BIOS Configuration Methods	1-2
1.2 An Overview of DSP/BIOS TextConf	1-6
1.3 Using DSP/BIOS TextConf for New Applications	1-10
1.4 Migrating Applications to DSP/BIOS TextConf	1-15
1.5 JavaScript Language Highlights	1-17
1.6 Command-Line Utility Reference	1-39
1.7 Example Scripts	1-43
2 DSP/BIOS TextConf Reference	2-1
<i>This chapter provides reference information about the Target Content Object Model.</i>	
2.1 Target Content Object Model Reference	2-2
2.2 DSP/BIOS Module and Instance Property Names	2-34
2.3 CSL Module and Instance Property Names	2-34

Figures

1-1	DSP/BIOS Configuration Methods	1-3
1-2	Target Content Object Model (TCOM).....	1-7
1-3	File Flow for DSP/BIOS TextConf.....	1-10
1-4	File Flow for DSP/BIOS TextConf Debugging	1-12
1-5	Rhino GUI Debugger Window	1-14
1-6	File Flow for cdbcmp Utility.....	1-15
2-1	Target Content Object Model (TCOM).....	2-2

Tables

1-1	Comparison of Portable Configuration Methods	1-31
2-1	Target Content Object Model Summary	2-2
2-2	Config Class Summary	2-4
2-3	Board Class Summary	2-8
2-4	Cpu Class Summary	2-13
2-5	Program Class Summary	2-18
2-6	Memory Class Summary	2-25
2-7	Extern Class Summary	2-27
2-8	Module Class Summary	2-28
2-9	Instance Class Summary	2-31
2-10	'C54x DMA Configuration Instance—DMA	2-35
2-11	'C54x DMA Resource Instance—HDMA	2-36
2-12	'C54x HDMA Pre-Created Instance Names	2-36
2-13	'C54x GPIO Configuration Instance—GPIO	2-37
2-14	'C54x MCBSP Configuration Instance—MCBSP	2-37
2-15	'C54x MCBSP Resource Instance—HMCBSP	2-38
2-16	'C54x HMCBSP Pre-Created Instance Names	2-38
2-17	'C54x PLL Configuration Instance—PLL	2-39
2-18	'C54x PLL Resource Instance—HPLL	2-39
2-19	'C54x HPLL Pre-Created Instance Names	2-39
2-20	'C54x Timer Configuration Instance—TIMER	2-39
2-21	'C54x Timer Resource Instance—HTIMER	2-39
2-22	'C54x HTIMER Pre-Created Instance Names	2-40
2-23	'C54x WDTIMER Configuration Instance—WDTIM	2-40
2-24	'C54x WDTIMER Resource Instance—HWDTIM	2-40
2-25	'C54x HWDTIM Pre-Created Instance Names	2-40
2-26	'C55x CHIP Configuration Instance—CHIP	2-40
2-27	'C55x DMA Configuration Instance—DMA	2-41
2-28	'C55x DMA Resource Module—HDMA	2-42
2-29	'C55x DMA Resource Instance—HDMA	2-42

2-30	'C55x HDMA Pre-Created Instance Names	2-42
2-31	'C55x EMIF Configuration Instance—EMIF	2-42
2-32	'C55x EMIF Resource Instance—HEMIF	2-43
2-33	'C55x HEMIF Pre-Created Instance Names	2-43
2-34	'C55x GPIO Configuration Instance—GPIO	2-44
2-35	'C55x MCBSP Configuration Instance—MCBSP	2-44
2-36	'C55x MCBSP Resource Instance—HMCBSP	2-45
2-37	'C55x HMCBSP Pre-Created Instance Names	2-45
2-38	'C55x PLL Configuration Instance—PLL	2-46
2-39	'C55x PLL Resource Instance—HPLL	2-46
2-40	'C55x HPLL Pre-Created Instance Names	2-46
2-41	'C55x PWR Configuration Instance—PWR	2-46
2-42	'C55x Real Time Clock Configuration Instance—RTC	2-47
2-43	'C55x Real Time Clock Resource Instance—RTCRES	2-48
2-44	'C55x RTCRES Pre-Created Instance Names	2-48
2-45	'C55x Timer Configuration Instance—TIMER	2-48
2-46	'C55x Timer Resource Instance—HTIMER	2-48
2-47	'C55x HTIMER Pre-Created Instance Names	2-48
2-48	'C55x USBRES Pre-Created Instance Names	2-49
2-49	'C55x WDTIMER Configuration Instance—WDTIM	2-49
2-50	'C55x WDTIMER Resource Instance—HWDTIM	2-49
2-51	'C6000 DMA Global Register Module—GDMA	2-49
2-52	'C6000 DMA Global Register Instance—GDMA	2-50
2-53	'C6000 DMA Configuration Instance—DMA	2-51
2-54	'C6000 DMA Resource Instance—HDMA	2-51
2-55	'C6000 HDMA Pre-Created Instance Names	2-51
2-56	'C6000 EDMA Configuration Instance—EDMA	2-52
2-57	'C6000 EDMA Resource Instance—HEDMA	2-52
2-58	'C6000 HEDMA Pre-Created Instance Names	2-53
2-59	'C6000 Parameter RAM Table Entry Instance—EdmaTable	2-54
2-60	'C6000 EMIF Configuration Instance—EMIF	2-54
2-61	'C6000 EMIF Resource Instance—HEMIF	2-55
2-62	'C6000 EMIFA Configuration Instance—EMIFA	2-55
2-63	'C6000 EMIFA Resource Instance—HEMIFA	2-56
2-64	'C6000 EMIFB Configuration Instance—EMIFB	2-56
2-65	'C6000 EMIFB Resource Instance—HEMIFB	2-56
2-66	'C6000 CSL Extern Declaration Module—ExternDecl	2-57
2-67	'C6000 CSL Extern Declaration Instance—ExternDecl	2-57
2-68	'C6000 MCBSP Configuration Instance—MCBSP	2-57

2-69	'C6000 MCBSP Resource Instance—HMCBSP	2-57
2-70	'C6000 HMCBSP Pre-Created Instance Names	2-58
2-71	'C6000 TCP Base Parameters—TCPBP	2-58
2-72	'C6000 TCP Configuration Instance—TCPBP	2-58
2-73	'C6000 TCP Resource Instance—HTCP	2-59
2-74	'C6000 TIMER Configuration Instance—TIMER	2-59
2-75	'C6000 TIMER Resource Instance—HTIMER	2-59
2-76	'C6000 HTIMER Pre-Created Instance Names	2-59
2-77	'C6000 VCP Base Parameters—VCPBP	2-60
2-78	'C6000 VCP Configuration Instance—VCPBP	2-60
2-79	'C6000 VCP Resource Instance—HVCP	2-60
2-80	'C6000 XBUS Configuration Instance—XBUS	2-61
2-81	'C6000 XBUS Resource Instance—HXBUS	2-61



DSP/BIOS TextConf Overview

This chapter compares the two methods for configuring DSP/BIOS programs and provides details about using DSP/BIOS TextConf.

Topic	Page
1.1 DSP/BIOS Configuration Methods	1-2
1.2 An Overview of DSP/BIOS TextConf	1-6
1.3 Using DSP/BIOS TextConf for New Applications	1-10
1.4 Migrating Applications to DSP/BIOS TextConf	1-15
1.5 JavaScript Language Highlights	1-17
1.6 Command-Line Utility Reference	1-39
1.7 Example Scripts	1-43

1.1 DSP/BIOS Configuration Methods

DSP/BIOS allows you to create and configure static objects for use by the DSP/BIOS API as part of your application design. DSP/BIOS provides the following methods of configuring your applications at design-time:

- ❑ **DSP/BIOS Configuration Tool (graphical configuration).** The DSP/BIOS Configuration Tool is supported in Code Composer Studio only for Microsoft Windows. Graphical configuration is described in the *DSP/BIOS User's Guide* and in the online help for the DSP/BIOS Configuration Tool.
- ❑ **DSP/BIOS TextConf (text-based configuration).** For this method, you use a text editor to write DSP/BIOS TextConf scripts. Command-line utilities run and generate these scripts. The scripts use the standard ECMAScript language (also known as JavaScript) and can be edited and manipulated as you would ordinary source code. The utilities are supported for Microsoft Windows and UNIX. This type of configuration is described in this document.

Both configuration methods generate source, header, and linker command files to be compiled and linked with your application.

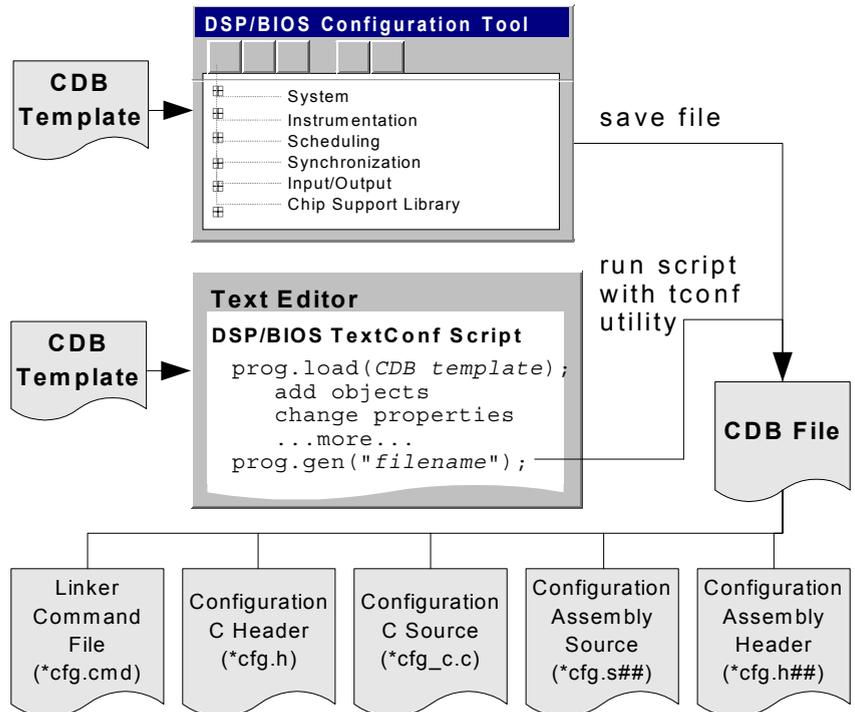
The DSP/BIOS API also supports dynamic creation of objects at run-time. Such objects are more flexible but increase performance overhead and code size.

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.

As the following diagram shows, both the DSP/BIOS Configuration Tool and DSP/BIOS TextConf generate a CDB (Configuration Data Base) file and source, header, and linker command files. The generated files in these processes are shown in gray. See Section 1.2.3, *Configuration File Types*, page 1-8 for a description of these generated files.

Figure 1-1. DSP/BIOS Configuration Methods



Both configuration methods use the same CDB template files. DSP/BIOS TextConf scripts are typically very short, because they define only the differences between the template objects and the objects used by the application. In contrast, CDB files define every object and property, even those defined in the template.

Both configuration methods allow you to set properties for both DSP/BIOS and Chip Support Library (CSL) modules. For information about DSP/BIOS module properties, see the *TMS320 DSP/BIOS API Reference Guide* for your platform. For information about CSL module properties, see the appropriate CSL documentation.

Both configuration methods have advantages in certain development environments. 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.

1.1.1 Why Use Graphical Configuration?

Using the DSP/BIOS Configuration Tool provides the following advantages over DSP/BIOS TextConf:

- ❑ 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 an additional scripting language.

1.1.2 Why Use DSP/BIOS TextConf?

Using DSP/BIOS TextConf provides the following advantages over the DSP/BIOS Configuration Tool graphical interface:

- ❑ Supported on both UNIX and Windows. (Graphical configuration is not supported on UNIX.)
- ❑ Supports a single configuration for systems containing multiple boards, processors, or programs.
- ❑ Uses the standard (ECMA-262) JavaScript language, which is C-like in syntax. As a result, design-time and run-time object creation statements are similar.
- ❑ DSP/BIOS uses scripts as source files. These are much smaller and easier to examine and maintain than the CDB files created by the DSP/BIOS Configuration Tool. With DSP/BIOS TextConf, the CDB files are still created, but they are not the original source files.
- ❑ Makes it easy to see which target-specific template was used as a starting point.
- ❑ Separates application-specific settings from other configuration settings, such as target-specific settings and ROM application settings. This makes it easy to port applications and modules to other target boards and platforms. It also makes it easier to maintain applications because you can quickly see which settings are made for target-specific reasons and which are made for application-specific reasons.
- ❑ Separates application-specific settings from DSP/BIOS configuration settings. This simplifies moving to a newer (or older) version of Code Composer Studio. In some cases, customers had to re-configure applications manually in order to upgrade when only the graphical configuration method was available.

- ❑ Allows you to modularize settings you use in all applications from application-specific settings. For example, if your applications all run on a target with minimal memory, all applications can include a single 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.

1.2 An Overview of DSP/BIOS TextConf

DSP/BIOS TextConf scripts contain statements in the JavaScript language. These statements are executed to perform design-time application configuration.

1.2.1 An Example TextConf Script

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 individual module and object. In contrast, the equivalent TextConf script contains only a few lines, because it defines only differences between template objects and the objects used by the application.

For example, the following statements are the TextConf script for a DSK6211 version of the Hello World example:

```
utils.loadPlatform("Dsk6211");
utils.getProgObjs(prog);

var trace = LOG.create("trace");
trace.bufLen = 32;
LOG_system.bufLen = 128;

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

This script loads a target-specific CDB file containing the target type, MEM objects and HWI objects and more. Loading this file is equivalent to choosing a template with the DSP/BIOS Configuration Tool. Since this is a separate statement, it is easy to make later changes to the target for the application.

The script also creates a LOG object called "trace" and sets the buffer length of two LOG objects.

After creating objects and setting properties, the script calls the prog.gen() method to generate both a CDB file and source, header, and linker command files to be used when compiling and linking the application.

1.2.2 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".

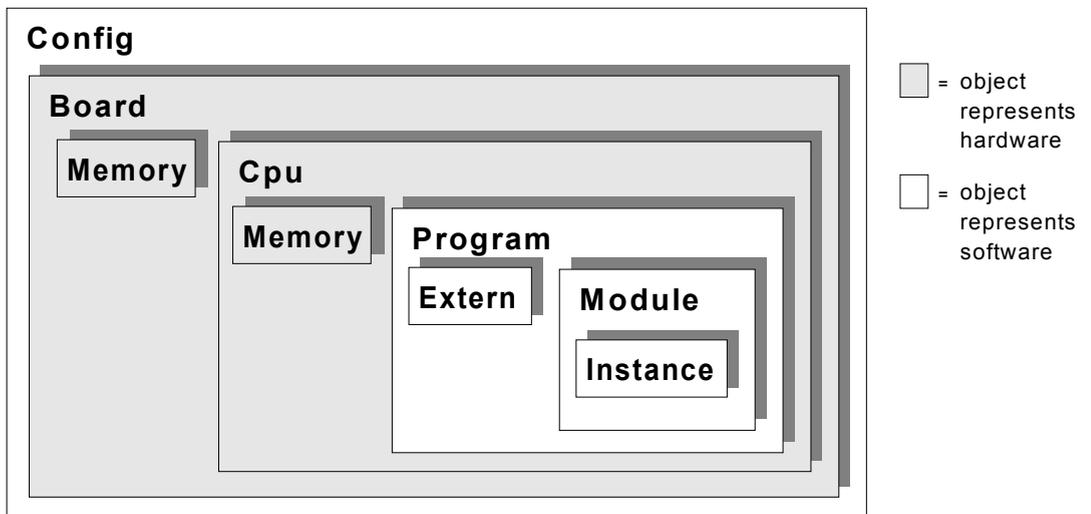
For DSP/BIOS TextConf, the object model is called the Target Content Object Model (TCOM) and the top-level object is the Config object.

The DOM model cannot be used with DSP/BIOS TextConf, 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 1-2. 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. This extends the graphical configuration object tree, which supports only one board, cpu, and program per configuration.

Several methods are provided for populating the hardware and software portions of the object model. If you have a single Board, Cpu, and Program, you can also use the `utils.getProgObjs()` method to define global variables for all objects, which allows you to reference individual objects by name without using the full object hierarchy notation.

See Section 2.1, *Target Content Object Model Reference*, page 2-2 for a list of the properties and methods of each of these object classes.

1.2.3 Configuration File Types

The following file types are source files for DSP/BIOS TextConf:

- ❑ **filename.tcf.** TextConf File (TCF) containing a DSP/BIOS TextConf script. If you follow certain optional conventions when naming this file, properties that define your board and CPU type are set automatically. See Section 1.5.7, *Configuration Coding Guidelines*, page 1-29 for details about these conventions.
- ❑ ***.tci.** TextConf Include (TCI) file. A script file included by the main *.tcf file or by another *.tci file. A different file extension is used for included files to support different handling of the main script and included scripts by program build utilities, such as gmake.
- ❑ ***.tcp.** TextConf Platform (TCP) file. A script file that acts as a starting point for applications by setting up target-specific objects. It defines the hardware-related TCOM objects and loads a configuration template. Typical TCF scripts begin by using the `utils.loadPlatform()` method to load a TCP script. TCP scripts are provided with DSP/BIOS TextConf for a number of common platforms.
- ❑ **target.cdb.** Configuration template files; also called a "seed files." For example, `evm62.cdb`. These files are provided with DSP/BIOS as starting points for configuring applications for various targets. They can be loaded into a script with the `prod.load()` method or by loading a platform (*.tcp) file.
- ❑ **tconfini.tcf.** DSP/BIOS TextConf startup file. See Section 1.5.9.1, *Startup Script Actions*, page 1-35 for more information.
- ❑ **tconflocal.tci.** Optional customizable startup file. See Section 1.5.9.1, *Startup Script Actions*, page 1-35 for more information.

The following files are generated by the DSP/BIOS Configuration Tool, the DSP/BIOS TextConf prog.gen() method, and the gconfgen command-line utility. In these filenames, "##" is a 2-digit target instruction set architecture (ISA—such as 55 or 64). It is generally recommended (but not required) that "program" should match the target program output filename in a Code Composer Studio project.

- ❑ **program.cdb.** Configuration Data Base (CDB) file. Stores configuration settings for use by the DSP/BIOS Configuration Tool graphical interface and by the DSP/BIOS Real-Time Analysis Tools.
- ❑ **programcfg_c.c.** Source file to define DSP/BIOS and Chip Support Library (CSL) structures and properties.
- ❑ **programcfg.h.** Includes DSP/BIOS module header files and declares external variables for objects in the configuration file.
- ❑ **programcfg.s##.** Assembly language source file for DSP/BIOS settings.
- ❑ **programcfg.h##.** Assembly language header file included by hellocfg.s##.
- ❑ **programcfg.cmd.** Linker command file.

In Code Composer Studio for Windows, you only need to add the generated CDB file and linker command file to your project.

1.2.4 Command-Line Utilities

The following command-line utilities are used with DSP/BIOS TextConf. For details about the command line syntax for these utilities, see Section 1.6, *Command-Line Utility Reference*, page 1-39.

- ❑ **tconf.** Runs DSP/BIOS TextConf scripts in one of three execution modes.
- ❑ **cdbcmp.** Compares one CDB file to the CDB template used to create it or compares two CDB files. Generates a DSP/BIOS TextConf script as output.
- ❑ **gconfgen.** Generates source, header, and linker command files from a CDB file.

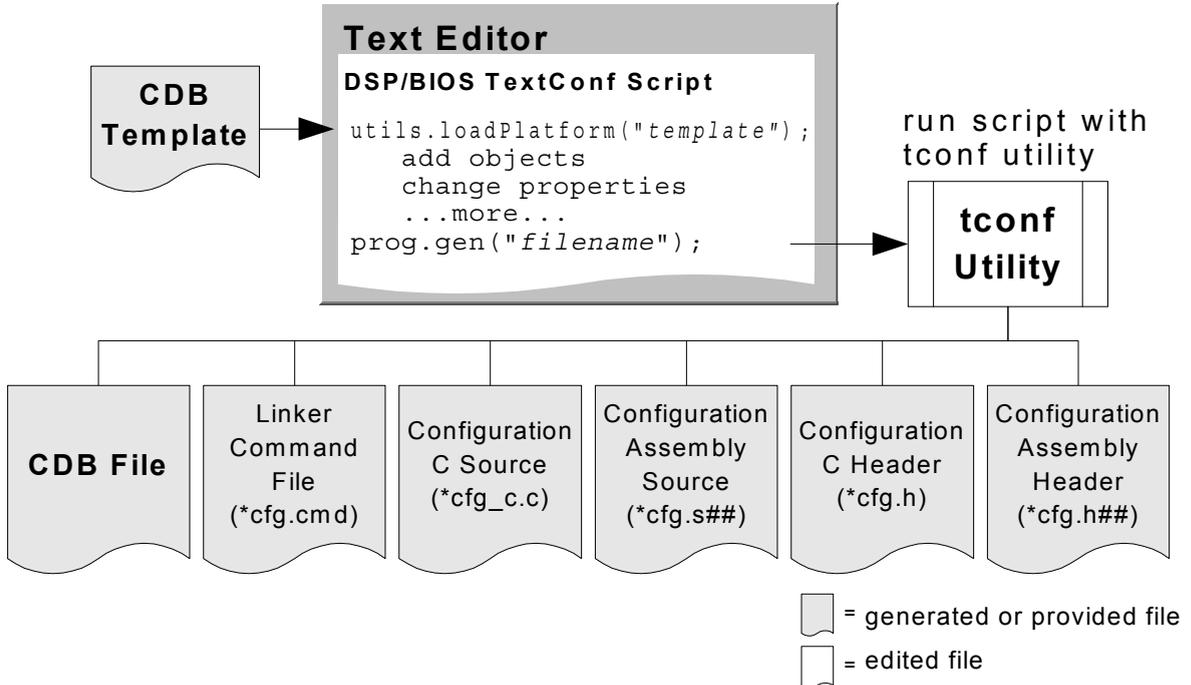
On Solaris, the tconf and cdbcmp utilities are installed in the <ccs_base_dir>/bin/utilities/tconf directory. On Windows, they are installed in the <ccs_base_dir>\bin\utilities\tconf folder. You may want to add this folder to your PATH variable so that you can run command-line utilities without specifying the full path to the utility each time.

The gconfgen utility is installed in <ccs_base_dir>/plugins/bios on Solaris and in <ccs_base_dir>\plugins\bios on Windows.

1.3 Using DSP/BIOS TextConf for New Applications

As the following diagram shows, typical DSP/BIOS TextConf scripts begin by loading a CDB template, contain statements that add objects and change properties, and then generate files to be used in building the application.

Figure 1-3. File Flow for DSP/BIOS TextConf



Use the procedures described in the following sections to write, test, and run DSP/BIOS TextConf scripts. If you have existing CDB files that you want to convert to scripts, see Section 1.4, *Migrating Applications to DSP/BIOS TextConf*, page 1-15.

1.3.1 Creating a Script for a New Application

To write a DSP/BIOS TextConf script from scratch, follow these steps:

- 1) Create a text file with an extension of .tcf.

If you follow certain optional conventions when naming this file, properties that define your board and CPU type are set automatically. See Section 1.5.7, *Configuration Coding Guidelines*, page 1-29 for details about these conventions.

- 2) Type the following line as the first statement in the file, where *Platform* matches the name of a *.tcp file that defines objects for your hardware platform. TCP scripts are provided with DSP/BIOS TextConf for a number of common platforms. You can create a custom TCP file if you are using custom hardware. TI supplies platform files for most boards supported in Code Composer Studio. These are located in tconf\include. See Section 1.5.10, *Creating a Platform File*, page 1-37 for creating platform files.

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

- 3) If your application will contain only one Board, Cpu, and Program, type the following statement. This method creates global variables that simplify references to Module and Instance objects.
- 4) Type the following line as the last statement in the file. It is generally recommended (but not required) that *program* should match the filename of your target output file. The prog.gen() method generates the appropriate CDB, source, header, and linker command files for use in building your application.

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

- 5) Add additional script statements to the file between the statements you typed in steps 3 and 4.

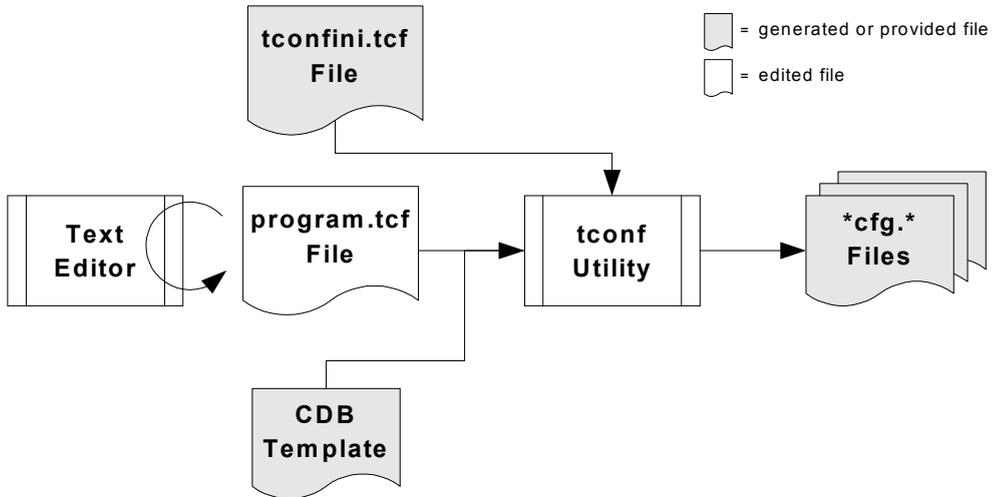
The method used to specifying the platform in this script is best for simple situations in which you are not testing programs on multiple platforms or expecting to port programs to other platforms. For more sophisticated ways to connect programs and platforms, see Section 1.5.8, *Specifying the Hardware Platform*, page 1-30.

utils.getProgObjs(program);

1.3.2 Debugging DSP/BIOS TextConf Scripts

As with other types of program development, creating a DSP/BIOS TextConf script is an iterative process. You edit the script with a text editor and test it with the tconf utility. As a result of testing, you may edit the script again.

Figure 1-4. File Flow for DSP/BIOS TextConf Debugging



The tconf utility provides three operation modes: the interactive debugging shell (Section 1.3.3, *Using the Interactive Debugging Shell*, page 1-12), the GUI debugger (Section 1.3.4, *Using the GUI Debugger*, page 1-14), and command-line mode (Section 1.6.1, *tconf Utility*, page 1-39). In any mode, if the script uses the prog.gen() method, the CDB, source, header, and linker command files are generated for use in building your application files. If the script uses the prog.save() method, only the CDB file is generated.

1.3.3 Using the Interactive Debugging Shell

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 the -g option.

The command-line syntax for the interactive debugging shell is as follows:

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

You can run a script from the interactive shell using the built-in `load()` method. For example:

```
% tconf
js> load("foo.tcf")
```

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. (In this example, the target platform is the DSK6211.)

```
% tconf
js> utils.loadPlatform("Dsk6211")
js> board_0 = config.boards()[0];
[object Board:board_0]
js> board_0.cpus()[0].attrs.cpuCore
6200
js> prog instanceof Program
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:

```
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 DSP/BIOS TextConf script; it is only available in the interactive shell.

The keywords `quit` and `exit` are reserved for future use in DSP/BIOS TextConf.

1.3.4 Using the GUI Debugger

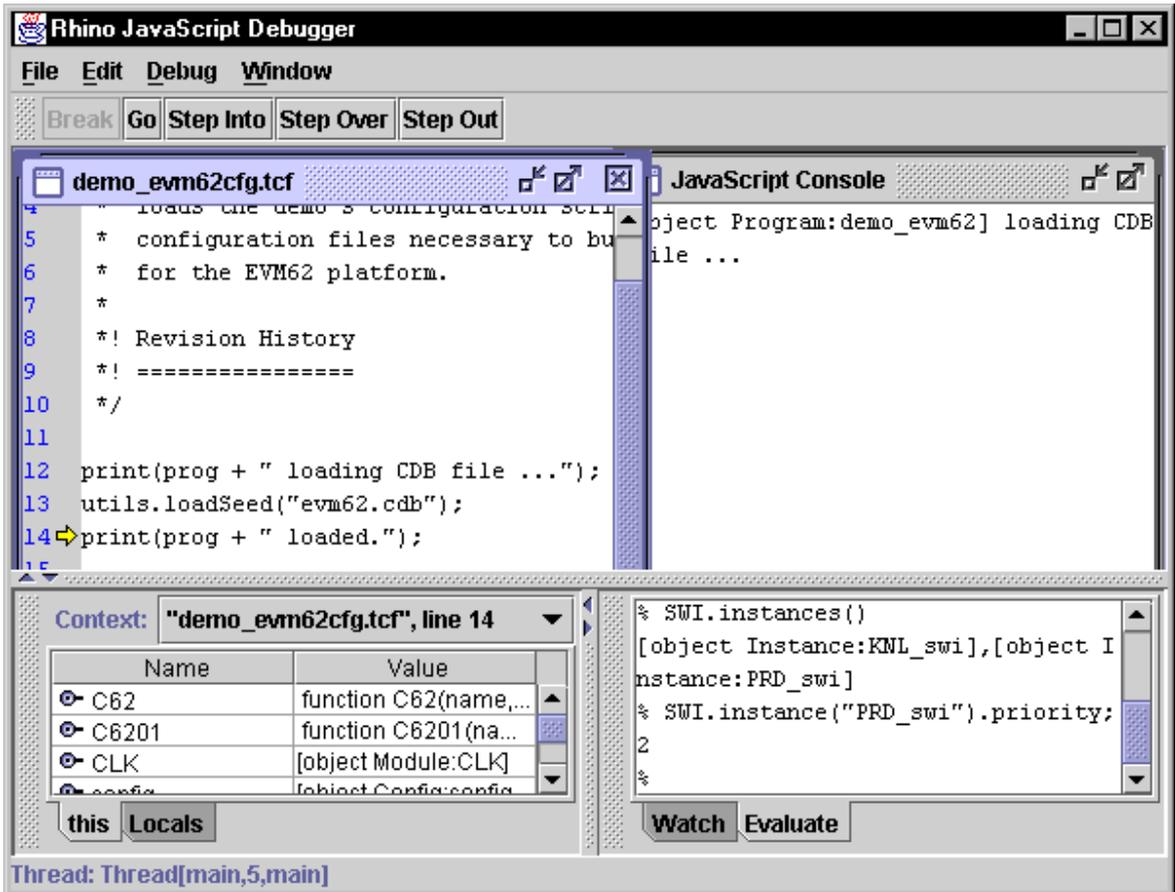
Rhino is an open-source implementation of JavaScript written entirely in Java (<http://www.mozilla.org/rhino>). It is installed on your workstation or PC along with the DSP/BIOS TextConf software.

Then, you can enter the Rhino debugger using the following command:

```
tconf -g
```

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 1-5. Rhino GUI Debugger Window



1.4 Migrating Applications to DSP/BIOS TextConf

If you have existing CDB files that you want to convert to scripts, use the procedures in this section. Use the procedures described in Section 1.3, *Using DSP/BIOS TextConf for New Applications*, page 1-10 to write, test, and run DSP/BIOS TextConf scripts.

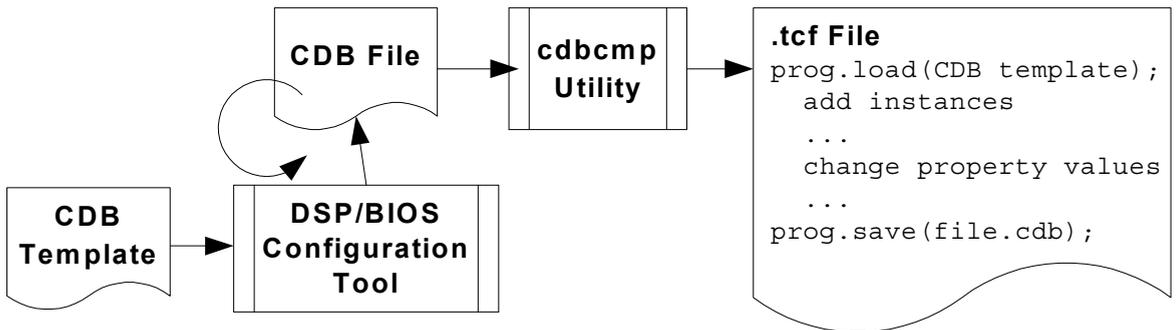
1.4.1 Creating a DSP/BIOS TextConf Script from a CDB File

If you have an existing CDB file, you can convert that file to a DSP/BIOS TextConf script using a command like the following:

```
cdbcmp demo.cdb > demo.tcf
```

This command finds the original CDB template used to create the demo.cdb file. It compares the demo.cdb file to the template and creates a script with statements to load the template file and to define objects and properties that are different from those in the template.

Figure 1-6. File Flow for cdbcmp Utility



Note: Order Dependencies in Generated Scripts

Some object classes have order dependencies, and you may need to modify the sequence of statements in the output script to get the script to run without errors. For example, an object may be referenced in a statement before it is created. Such errors are usually fairly easy to diagnose based on the JavaScript error and a visual inspection of the script. To correct such errors, move the statement that creates the object before the statement that references the object.

1.4.2 Comparing Configurations

To view changes made between two versions of a CDB file or to compare CDB files for two applications, use a command like the following:

```
cdbcmp v1/demo.cdb v2/demo.cdb
```

This command compares the two CDB file and creates a script with statements that would convert the first configuration to the second.

1.4.3 Merging Configurations

To merge changes to CDB files made by two developers, use commands like the following:

```
cdbcmp a/demo.cdb > demo_a.tcf  
cdbcmp b/demo.cdb > demo_b.tcf  
sdiff -o demo.tcf demo_a.tcf demo_b.tcf
```

These commands create two script files that define differences between the configurations and their templates. The `sdiff` UNIX command (and similar commands on UNIX and other platforms) allows you to merge the statements in the two DSP/BIOS TextConf script files without including duplications.

For example, configurations may need to be merged if one developer is working on a driver while another is working on thread scheduling.

1.5 JavaScript Language Highlights

This document does not provide details on the syntax of the JavaScript language. However, several concepts are important when using JavaScript for DSP/BIOS TextConf. This section provides an overview of such concepts. See Section 1.5.11, *JavaScript and Java References*, page 1-38 for JavaScript reference sources.

1.5.1 JavaScript 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.

1.5.2 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 DSP/BIOS TextConf.

- ❑ 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, DSP/BIOS TextConf 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 or UNIX machine. JavaScript code is never run on the target DSP.

1.5.3 Objects and Properties in JavaScript

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 `endian` property of a `Cpu` object:

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

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].endian
board_0.cpus() ["cpu_0"].endian
cpu_0.endian
```

You can use the `utils.getProgObjs()` method to create global variables that reference all Module and Instance objects after you load a configuration from a CDB file. 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 set up using a second parameter with the `utils.getProgObjs()` method.

```
gvars = {};
utils.getProgObjs(prog, gvars);
gvars.LOG_system
```

- ❑ Reference path set up using the `utils.getProgObjs()` method with no second parameter.

```
utils.getProgObjs(prog);
LOG_system
```

Most code examples in this document assume that the `utils.getProgObjs()` method was used with no second parameter to create global variables for all Module and Instance objects.

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 = 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:

```
alphanames = TSK.instances().sort()
```

1.5.3.1 *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, DSP/BIOS TextConf handles the properties of Module and Instance objects differently than those of other object classes.

In the DSP/BIOS Configuration Tool, the Property dialogs display these properties. Each field in these dialogs is mapped to a property name for use in DSP/BIOS TextConf scripts. The names are listed in Section 2.2, *DSP/BIOS Module and Instance Property Names*, page 2-34.

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. (The following examples assume the `utils.getProgObjs()` method was used to create global variables for all Module and Instance objects.)

```
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".

1.5.3.2 *Property Types*

Section 2.2, *DSP/BIOS Module and Instance Property Names*, page 2-34 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.** CDB files 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:

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

- ❑ **EnumInt.** Enumerated integer properties accept a set of valid integer values. These values are displayed in a drop-down list in the DSP/BIOS Configuration Tool.
- ❑ **EnumString.** Enumerated string properties accept a set of valid string values. These values are displayed in a drop-down list in the DSP/BIOS Configuration Tool.
- ❑ **Int16.** Integer properties hold 16-bit unsigned integer values. The value range accepted for a property may have additional limits.
- ❑ **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.

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

- ❑ **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:

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

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

1.5.3.3 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.

```
prog.module("SWI").create("audio");      /* OK */  
prog.module("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;  
}
```

1.5.4 Built-in Variable Arrays

DSP/BIOS TextConf 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.

1.5.4.1 Environment Array Variables

DSP/BIOS TextConf creates an array called "environment" and automatically defines a number of variables within that array. Additional variables may be added to the array using the -D option on the tconf command line. See Section 1.6.1, *tconf Utility*, page 1-39 for information about command-line options.

These 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"];
```

- ❑ `environment["config.tiRoot"]`. Contains the top-level directory location of the Code Composer Studio installation. If the installation directory can not be determined, this variable is not defined.
- ❑ `environment["config.rootDir"]`. Contains the directory location of the executable file for the `tconf` utility. This location is typically `<ccs_base_dir>\bin\utilities\tconf`. This variable is always defined.
- ❑ `environment["config.scriptName"]`. Contains the name of the script passed to the `tconf` utility on the command line. This variable is always defined. If no script was passed, this variable is set to an empty string (`""`).
- ❑ `environment["config.path"]`. Contains the set of directories used to locate DSP/BIOS TextConf components (including the `tconf` executable, the necessary import files, and DLLs). This variable is always defined. This path may be added to using the `-p` option on the `tconf` command line.
- ❑ `environment["config.importPath"]`. This variable may be defined as a search path using the `-D` option on the `tconf` command line. If it is defined, the `utils.importFile` function uses the specified search path to use to locate files. If this variable is not defined, `utils.importFile` looks in the current directory. (In either case, `utils.importFile` looks in the `environment["rootDir"]\include` directory after using the search path or the current directory.) For example:

```
tconf -Dconfig.importPath=d:\myproject\include
```

- ❑ `environment["config.compilerOpts"]`. This variable may define the compiler options used to build the program. The options that may be specified are `-me` (big endian), `-ml` (large data model) and `-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._54_
```

- ❑ `environment["config.platform"]`. A variable of this format may be defined using the `-D` option on the `tconf` command line. The variable should be set to one of the valid `board_types` for the `create()` method of the `config` object. If this variable is defined, it may be used

by the startup procedure to populate the hardware-related portion of the object model. For example:

```
tconf -Dconfig.platform=Dsk6211
```

1.5.4.2 *Argument Array Variables*

DSP/BIOS TextConf 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];
```

1.5.5 *File Interaction and I/O Methods*

DSP/BIOS TextConf provides the methods described in the following sub-sections for working with script files, configuration files, other files, and output to stdout.

Directory paths specified in JavaScript statements can use either "\" or "/" as a directory separator.

1.5.5.1 *Methods for Loading Scripts*

A DSP/BIOS TextConf script can load another script file. When a script file is loaded, any statements outside functions run, and the functions defined in the loaded script are available to be called by the script that loaded the file.

DSP/BIOS TextConf 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`. If the `config.importPath` variable was defined using the `-D` option on the `tconf` command line, `utils.importFile()` first looks in the directories in that search path. If the `config.importPath` variable was not defined, `utils.importFile()` looks first in the current directory. If the file is not found, `utils.importFile()` then looks in the `\include` subdirectory of the directory that contains the `tconf` executable.

- ❑ **utils.loadArch()**. A utility method that loads a CPU architecture file. This utility is typically used within a platform definition file when creating a `Cpu` object. For example:

```
/* Create new cpu object */
config.board("dsk5402").create("cpu_0",
    utils.loadArch("5402"));
```

- ❑ **utils.loadPlatform()**. A utility method that loads a platform definition file (`.tcp`). Such platform definition files contain scripts that create the `Config`, `Board`, and `Cpu` objects for the hardware platform. They also load the CDB template file for that platform. For example:

```
utils.loadPlatform("Dsk6211");
```

This function looks for the specified file in the same locations as the `utils.importFile()` method. A number of pre-defined platform definition files are provided in the `\include` subdirectory of the directory that contains the `tconf` executable. You can create custom platform definition files by using the provided files as examples.

1.5.5.2 Methods for Working with CDB Files

DSP/BIOS TextConf provides these methods for working with CDB files.

- ❑ **prog.load()**. A method of `Program` objects. This method reads the specified CDB file and populates the Target Content Object Model with the modules and instances named in the CDB file. It assumes the `Config`, `Board`, `Cpu`, and `Program` objects have already been created. Provide a relative path to the file from the current directory or the full path. For example:

```
prog.load("test/mytemplate.cdb");
```

Another way to load a CDB file is to use the `utils.loadPlatform()` method described in Section 1.5.5.1, *Methods for Loading Scripts*, page 1-24.

- ❑ **`utils.findSeed()`**. A utility method that returns the full path of the specified CDB file. This utility looks for the file in the `<ccs_base_dir>\c####\bios\include` directories. If it cannot find the file, it returns null. For example:

```
location = utils.findSeed("sim55.cdb");
```

- ❑ **`utils.getProgObjs()`**. A utility method that creates a global variable for every Module and Instance object in the specified Program object. Using this method simplifies the syntax required to reference 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
```

The first parameter for this method is the Program object for which you want to create variables. The second parameter is an optional variable name to act as a container for the set of global variables. If you omit the second parameter, the global variables are standalone variables with no container.

For example, these statements show the simple case in which there is one program object and the second parameter is omitted:

```
utils.getProgObjs(prog);  
LOG_system.bufLen = 128;
```

If two Program objects are referenced by the `prog_0` and `prog_1` variables, you can use statements like the following:

```
vars0 = {};  
vars1 = {};  
utils.getProgObjs(prog_0, vars0);  
utils.getProgObjs(prog_1, vars1);
```

```
vars0.LOG_system.bufLen = 512;  
vars1.LOG_system.bufLen = 128;
```

Most code examples in this document assume that the `utils.getProgObjs()` method was used with no second parameter to create global variables for all Module and Instance objects.

1.5.5.3 *Methods for File Manipulation*

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 DSP/BIOS TextConf, 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>.

1.5.5.4 *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);
}
```

1.5.6 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 the interactive debugging shell, `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.

1.5.6.1 *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("myApp");
}
else {
    print("An error has occurred.");
}
```

1.5.6.2 *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.");
}
```

1.5.7 Configuration Coding Guidelines

When using DSP/BIOS TextConf, we recommend using the following coding conventions.

- ❑ Name program-level scripts: *program_boardcfg.tcf*

where *program* is the name of the program, and *board* is the name of the target board. This allows the *tconfini.tcf* startup script to initialize the Board, Cpu, and Program objects correctly. For other ways to define the platform, see Section 1.5.8, *Specifying the Hardware Platform*, page 1-30.

- ❑ 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 configuration scripts into pieces that match the modules. This allows re-use of scripts wherever a module can be reused. Conceptually, each module now includes configuration script(s).
- ❑ Name module-level scripts: `mod.tci`
where `mod` is the module name. This enables one module's script to import another's with the `load()` method.
- ❑ Treat the main program as a module. Create a module-level script called `program.tci` that imports other scripts as necessary. This enables rapid porting to new boards by minimizing configuration script changes.
- ❑ Split module-level scripts into platform-dependent and platform-independent parts. This minimizes code duplication and simplifies porting to new platforms. The platform-independent part should include the appropriate platform dependent part. If no appropriate platform-dependent part exists, throw a meaningful exception.
- ❑ Name platform-dependent scripts: `mod_boardcfg.tci`
This prevents file name collisions with other modules.

1.5.8 Specifying the Hardware Platform

To configure programs with DSP/BIOS TextConf, you define both the hardware execution platform and the software components used in the program. This section describes methods for "binding" the program and the platform within DSP/BIOS TextConf scripts in portable ways. For information on creating a platform definition file (*.tcp), see Section 1.5.10, *Creating a Platform File*, page 1-37.

In all but the simplest development environments, it is important to write portable configuration scripts. Such scripts can be used without modification to re-target a program for a different platform. For example, an algorithm test may need to be run on both a simulator and a hardware development board. Portability is especially valuable when maintaining hundreds of tests in a regression test suite.

In order to create portable configuration scripts, you must separate the platform specification (that is, initialization of hardware objects) from the program's software configuration. Fortunately, DSP/BIOS TextConf supports a number of common mechanisms for creating re-usable configuration scripts. Such mechanisms include the ability to dynamically

load configuration scripts from other scripts, subroutines, environment variables, and user-modifiable startup scripts.

Several methods of creating portable scripts are presented in this section. Table 1-1 compares the advantages and disadvantages of each method. To choose a method, determine how many programs need to be maintained, how many platforms need to be supported, and whether platforms must be supported simultaneously by the same program. Also consider your build environment. Mature development environments (with regression test suites, support for multiple hardware generations, and multiple DSP programs) often use a combination of these methods.

Table 1-1. Comparison of Portable Configuration Methods

Platform Specification Method	Resulting Script is Portable	For Simultaneous Multi-Platform	For Large Suites of Programs	For Selected Platforms and Programs
<i>Specifying the Platform in the Script, page 1-31</i>	no	yes	no	yes
<i>Specifying the Platform on the Command Line, page 1-32</i>	yes	no	yes	no
<i>Specifying Platform and Directory on the Command Line, page 1-33</i>	yes	yes	yes	no
<i>Specifying the Platform in the Script File Name, page 1-33</i>	yes	yes	no	yes

1.5.8.1 Specifying the Platform in the Script

The most direct way to specify the execution platform is to explicitly create the hardware-related objects within the configuration script itself. For example, one can use statements like the following `hellocfg.tcf` example to create the `Config`, `Board`, and `Cpu` objects within a configuration script.

```
/* create platform-specific objects */
utils.loadPlatform("Dsk6211");

/* load platform-independent software config */
utils.importFile("hello.tci");

/* generate configuration files for the program */
prog.gen();
```

To port this configuration to a new execution platform, you would modify the board name literals that appear in this script. More sophisticated

platforms, such as multi-processor platforms, may require more extensive changes.

1.5.8.2 Specifying the Platform on the Command Line

While it is natural to expect to modify the configuration script when porting to alternative execution platforms as described in Section 1.5.8.1, *Specifying the Platform in the Script*, page 1-31, it may be inconvenient if you change platforms frequently.

Suppose, for example, that you have more developers than physical boards. To use a simulator as an alternative test platform, you would need to create two configuration scripts or repeatedly edit the configuration script to move between the simulator and real hardware. Creating two nearly-identical scripts results in a code maintenance problem; one must ensure that changes to one file are duplicated in the other. To avoid this problem, you need to "parameterize" the configuration script to support either platform. This allows all developers to use the same script.

The `tconf` utility allows you to specify the platform on the command line by defining the `config.platform` environment variable. This variable is available to the script executed in the environment array.

For example, to make the script in the previous section platform-independent, the configuration script can be re-written as follows:

```
utils.loadPlatform(environment["config.platform"]);
utils.importFile("hello.tci");
prog.gen();
```

Then, the following command line would generate the configuration for the `hello` example for the `Dsk6211` platform:

```
tconf -Dconfig.platform=Dsk6211 hellocfg.tcf
```

Because the `hellocfg.tcf` file no longer contains any hardware-specific information, you only need to change the `tconf` command line to port the example to an alternative platform.

This method makes it easy for multiple developers to use common sources to build for different platforms. This method also works well when a large number of programs need to run on a single platform. For example, suppose a regression test suite consists of a large number of programs that need to be re-targeted to a single execution platform. If the build system for this test suite is parameterized to pass the platform name to the `tconf` command lines, re-targeting the entire suite is as simple as naming the target platform in the build command line.

1.5.8.3 Specifying Platform and Directory on the Command Line

Notice that `hellocfg.tcf` generates the configuration files in the current working directory (typically the directory containing the `.tcf` file). Since the names of the configuration files are derived from the configuration script name, this script would overwrite the files for one platform with the files for another platform if a build system ran the script twice with different command line parameters.

To avoid overwriting files, a script can specify an output directory for the configuration files based on the platform parameter passed to the script. For example, the `hellocfg.tcf` configuration script can be re-written as follows:

```
utils.loadPlatform(environment["config.platform"]);
utils.importFile("hello.tci");
prog.gen(environment["config.platform"] + "/"
         + prog.name);
```

In this example, the generated configuration files are placed in a sub-directory of the current working directory. Thus, a single build system can generate configuration files for later execution on multiple platforms using this single script.

Notice that the slash (`/`) character is used to separate the directory name from the rest of the file name. Since this character works on both Windows and UNIX platforms, the script above can be used on either platform without modification.

1.5.8.4 Specifying the Platform in the Script File Name

The technique of specifying the platform with a command-line parameter works well when all programs can run on the full set of platforms. If, however, you maintain a collection of programs that can each be executed on different a subset of platforms, it is natural to create scripts with names that include both the program name and the platform name. This makes it easy to identify which configuration files relate to a specific platform and identify which platform can execute each generated program. More importantly, it also provides a simple way to specify which combinations of program and platform are valid.

The sample `tconflocal.tci` startup file examines the name of the configuration script. If it is of the form `program_platformcfg.tcf`, where the program name is `program` and the execution platform is specified as `platform`. Thus, the following `tconf` command lines generate the files necessary for the Dsk6211 and the C62x simulator platforms:

```
tconf hello_dsk6211cfg.tcf
```

```
tconf hello_sim62xxcfg.tcf
```

Note that, as with the command line method of determining the platform, the configuration script does not need any hardware-specific settings. To port to a new platform, you simply copy `hellocfg.tcf` to an appropriately named file such as, `hello_sim62xx.tcf`.

Note: Effects of `utils.loadPlatform()` on Startup Actions

The sample `tconflocal.tci` script does not load the CDB template for the specified platform. You can use the `utils.loadPlatform()` method to both initialize the TCOM object hierarchy and load the appropriate CDB template for the specified platform. If you use the `utils.loadPlatform()` the objects created by the startup script are overwritten.

By using different file names to specify the platform instead of using `tconf` command line options, program build tools (such as `make`) can automatically run `tconf` and build distinct executables for each valid combination of program and platform. Thus, adding support for a new platform requires no modifications to the makefiles, for instance.

This method works well when one must simultaneously support specific combinations of programs and platforms. However, in the regression test suite example above, this technique can be cumbersome. Adding a new platform for a test suite means creating a new configuration script file for every test.

Moreover, even if it were possible to run the test suite for each platform in parallel, the time to build all tests for all platforms and the disk space required might be prohibitively high. Thus, for pragmatic reasons, one often sequentially builds, runs, and deletes the executables one platform at a time. In this case, using the command line parameter method in Section 1.5.8.3, *Specifying Platform and Directory on the Command Line*, page 1-33 is preferable.

1.5.9 Platform Specification and the Startup Script

The platform-specification methods listed in Table 1-1 are all supported by the sample `tconflocal.tci` file provided with DSP/BIOS TextConf. That is, the `tconflocal.tci` file can initialize the TCOM hardware-related objects if the platform is specified using any of these methods.

When the `tconf` utility starts, it runs the `tconfini.tcf` startup script prior to the first line of any configuration script. This script performs necessary startup activities and should not be modified. The script uses the

utils.importFile function to search for a tconflocal.tci file. If a file with this name is found, this script is also run.

The startup script runs whenever you start the tconf utility. It runs in all three tconf operation modes (command-line, interactive debugging, and GUI debugging).

Note: Effects of utils.loadPlatform() on Startup Actions

The sample tconflocal.tci script does not load the CDB template for the specified platform. You can use the utils.loadPlatform() method to both initialize the TCOM object hierarchy and load the appropriate CDB template for the specified platform. If you use the utils.loadPlatform() the objects created by the startup script are overwritten.

1.5.9.1 Startup Script Actions

The tconfini.tcf startup script performs the following actions:

- 1) Loads the Target Content Object Model classes and constructors.
- 2) Defines a special load() function used to load scripts from within the tconfini.tcf file.
- 3) Loads the utils.tcf file, which contains a package of utility functions.
- 4) Loads the tconflocal.tci file if it exists on the search path. This file may be customized to define the specific Config, Board, Cpu, and Program objects needed by your application.
- 5) If it does not find a tconflocal.tci file, tconfini.tcf defines Config, Board, Cpu, and Program objects with the names “config_0”, “board_0”, “cpu_0”, and “prog_0”.

A sample version of the tconflocal.tci file is provided in the \include subdirectory of the directory containing the tconf executable file. This file performs the following actions. You can customize this file to meet your needs.

- 1) Defines a Config object called “config_0” and defines the global variable “config” to reference this object.
- 2) Attempts to determine the board type using the optional convention described in Section 1.5.9.2, *Optional Conventions for Initializing the Object Model*, page 1-36, and creates a Board object of that type. If it cannot determine the type, it creates a generic Board object. In either case, the Board object is called “board_0”.

- 3) Attempts to determine the CPU type and creates a Cpu object of that type. If it cannot determine the type, it creates a generic Cpu object. In either case, the Cpu object is called "cpu_0".
- 4) Attempts to determine the program name using the optional convention described in Section 1.5.9.2, *Optional Conventions for Initializing the Object Model*, page 1-36, and creates a Program object with the program name. If it cannot determine the name, it creates a generic Program object called "prog_0". It also sets program properties based on the properties of the Cpu object.

1.5.9.2 *Optional Conventions for Initializing the Object Model*

If you follow the optional conventions described in this section when naming your main .tcf script file and using the tconf command line, properties that define your board and CPU are set automatically.

These optional conventions simplify property definition if your application contains a single board, single CPU, and single program. They facilitate easy application migration to another board and CPU.

- ❑ **Board type:** You may use a -D option similar to the following on the tconf command to specify the board type:

```
tconf -Dconfig.platform=Dsk6211 hellocfg.tcf
```

If you do not use the -D option to define config.platform, you can give your main script file a name with the following form, where *board* is the board type and *program* matches the target executable filename created by Code Composer Studio:

```
[directory]program_boardcfg.tcf
```

For example, the following command line specifies a DSK62 board.

```
tconf demo_dsk62cfg.tcf
```

The following board types are supported for this convention: dsk54, dsk5416, dsk62, dsk67, evm54, evm55, evm62, sim54, sim55, sim62, and sim64.

- ❑ **Program name:** If a script is specified on the tconf command line, the program name is the filename minus any cfg suffix and the file extension. For example, if the command line is as follows, the Program object is called "myapp_dsk62":

```
tconf myapp_dsk62cfg.tcf
```

1.5.10 Creating a Platform File

TextConf platform files (*.tcp) provide definitions for the hardware execution platform. A platform file populates the hardware aspects of the Target Content Object Model (TCOM). See Section 2.1.1, *Target Content Object Model Quick Reference*, page 2-2 for details about the TCOM.

Platform files for boards supported by Code Composer Studio are provided in the tconf\include directory. The file naming convention is *Boardname.tcp*—for example, *Dsk6711.tcp*.

If you want to create a new platform file or understand the platform file for your board, the following list describes the steps performed within a platform file:

- 1) Create a comment heading and a tag line (for example, !DESCRIPTION 54XX!) for tools that read and display the platform files available for a given family of processors, such as 54xx. You may also want to include a revision history section.
- 2) Create a top-level Config object.
- 3) Instantiate a Board object, giving it the name of the board; for example, *dsk5402*.
- 4) Create a Cpu object by loading the CPU architecture definition; for example, 5402. This brings in the Cpu attributes, including the on-chip memory definitions and the DSP/BIOS hardware settings.
- 5) Define the external memory populated on the board.
- 6) Set the value for the clock oscillator on the board.
- 7) Set the PLL index. This is an index into an enumerated list of chip-specific hardware/software PLL multiplication factors.

The following example shows a typical platform file. The step numbers correspond to the items in the previous list.

```

Step 1: /* =====Dsk5402.tcp===== */
        /* !DESCRIPTION 54XX! */
        /* Dsk5402 with 64K SRAM and 256K FLASH */

Step 2: /* Create new config object */
        config = new Config("config_0");

Step 3: /* Create new board object */
        config.create("dsk5402");

Step 4: /* Create new cpu object */
        config.board("dsk5402").create("cpu_0",
                                         utils.loadArch("5402"));

```

Step 5:

```
/* 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
};
```

Step 6:

```
/* Define clock oscillator value on board for
   specified cpu in Mhz */
config.board("dsk5402").cpu("cpu_0").clockOscillator
    = 20.0000;
```

Step 7:

```
/* Define PLL index value used to compute speed
   of CPU */
config.board("dsk5402").pllIndex = 2;
```

1.5.11 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>

1.6 Command-Line Utility Reference

The following command-line utilities manipulate DSP/BIOS TextConf and CDB files. These utilities are supported on UNIX and Windows.

1.6.1 tconf Utility

Syntax

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

Options

- g Invoke the graphical script debugging interface.
- p <dir> Add the specified directory to the search path used to find files. The search path looks first in the current directory, then in the directory containing the tconf executable file, and then in any directory named using the -p option. See Section 1.5.4.1, *Environment Array Variables*, page 1-22 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. See Section 1.5.4.1, *Environment Array Variables*, page 1-22 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 1.5.4.2, *Argument Array Variables*, page 1-24 for details about the arguments array.
- h Display command-line syntax.

Description

tconf is a JavaScript execution utility. It can be used both for debugging and to output a CDB file and configuration files used to build applications.

On Solaris, the tconf file is in <ccs_base_dir>/bin/utilities/tconf. On Windows, tconf.exe is in <ccs_base_dir>\bin\utilities\tconf. You may want to add this directory to your PATH variable so that you can run tconf without specifying the full path to the utility each time.

For example, if the script or statement calls `prog.gen("myApp")`, `tconf` outputs a CDB file that corresponds to the script file or statements. It also generates the configuration files normally generated when you save a CDB file in the DSP/BIOS Configuration Tool.

For example, if the `demo.tcf` file contains the `prog.gen("demo");` statement, the following line generates a `demo.cdb` file and its associated generated files.

```
tconf demo.tcf
```

The `tconf` utility provides three operation modes:

- ❑ **Command-line mode.** If a script is listed on the command line, `tconf` processes the script without entering a debugging environment. If the script uses the `prog.save()` or `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 ...]
```

- ❑ **Interactive debugging shell.** If no script is listed on the command line, `tconf` enters interactive mode and reads and executes statements you type at the `js>` prompt. It echoes the results of print statements and expressions to your terminal window.

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

See Section 1.3.3, *Using the Interactive Debugging Shell*, page 1-12 for information about using the interactive debugging shell.

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

See Section 1.3.4, *Using the GUI Debugger*, page 1-14 for information about using the Rhino debugger.

Examples

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"]
```

1.6.2 cdbcmp Utility

Syntax

```
cdbcmp projname.cdb > projname.tcf
cdbcmp proj1.cdb proj2.cdb > proj_diffs.tcf
```

Description

The cdbcmp utility either compares one CDB file to the CDB template used to create it or compares two CDB files.

On Solaris, the cdbcmp.exe file is installed in the <ccs_base_dir>/bin/utilities/tconf directory. On Windows, it is installed in the <ccs_base_dir>\bin\utilities\tconf folder. You may want to add this directory to your PATH variable so that you can run cdbcmp without specifying the full path to the utility each time.

When used to compare a CDB file to its template, you do not need to specify the template; cdbcmp finds the template automatically. The cdbcmp utility generates a DSP/BIOS TextConf script that loads the template file, modifies the configuration to match the CDB file, and saves the resulting configuration as a CDB file.

When used to compare two CDB files, cdbcmp generates the script commands necessary to convert the settings in the first CDB file to the settings in the second CDB file. The cdbcmp utility generates a DSP/BIOS TextConf script that loads the first CDB file, modifies the configuration to match the second CDB file, and saves the resulting configuration as a CDB file.

1.6.3 gconfgen Utility

Syntax

```
gconfgen projname.cdb
```

Description

This command line utility reads a CDB file and generates the corresponding source, header, and linker command files. The CDB file may have been created with the DSP/BIOS Configuration Tool or the prog.gen() method in DSP/BIOS TextConf.

On Solaris, the gconfgen.exe file is installed in the <ccs_base_dir>/plugins/bios directory. On Windows, it is installed in the <ccs_base_dir>\plugins\bios folder.

If your DSP/BIOS TextConf script uses prog.gen(), you do not need to use this utility to generate files. If your DSP/BIOS TextConf script uses

`prog.save()`, only the CDB file is created; you should use the `gconfgen` utility to generate the corresponding files.

The `gconfgen` utility generates the following files:

- ❑ `programcfg_c.c`
- ❑ `programcfg.h`
- ❑ `programcfg.s##`
- ❑ `programcfg.h##`
- ❑ `programcfg.cmd`

See Section 1.2.3, *Configuration File Types*, page 1-8 for descriptions of these files.

Only the `program.cdb` and `programcfg.cmd` files need to be added to your Code Composer Studio project. The other files are then added automatically or included by other files already in the project.

1.7 Example Scripts

The following examples and the examples in Chapter Chapter 2, show a variety of ways to use DSP/BIOS TextConf.

1.7.1 Minimizing Application Code Size

If you are implementing programs on a 'C5000 platform, you might add a line like the following to all your applications to import a script you create that sets properties that minimize the size of the target code:

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

The following example script minimizes the code size footprint of a DSP/BIOS application:

```
/* don't use TSK threads */
TSK.USETSK = 0;

/* remove all default use of heaps */
MEM.SEGZERO = MEM_NULL;
MEM.MALLOCSEG = MEM_NULL;

/* loop through all MEM objects and remove any heap */
var memObjs = MEM.instances();
for (var i = 0; i < memObjs.length; i++ ) {
    if (memObjs[i] != MEM_NULL
        && memObjs[i].createHeap == 1) {
        memObjs[i].createHeap = 0;
    }
}

/* remove SYS stuff */
SYS.TRACESIZE = 0;
/* bind abort to user-defined error function */
SYS.ABORTFXN = prog.extern("error");
/* bind exit and error to the same function */
SYS.EXITFXN = prog.extern("error");
SYS.ERRORFXN = prog.extern("error");
/* bind SYS_putc to a fxn that does nothing */
SYS.PUTCFXN = prog.extern("FXN_F_nop");
```

1.7.2 Mailbox Example

The following script configures the mbxtest tutorial example. One advantage to using this script instead of the DSP/BIOS Configuration Tool is that the NWRITERS variable makes it easy to run a number of tests with different numbers of writer tasks.

```
utils.loadPlatform("Platform");
utils.getProgObjs(prog);

var NWRITERS = 3; /* number of writers to test */
var PRIORITY = 1; /* task priority in this test */

SYS.ABORTFXN = prog.decl("UTL_doAbort");
SYS.ERRORFXN = prog.decl("UTL_doError");
SYS.EXITFXN = prog.decl("UTL_halt");
SYS.PUTCFXN = prog.decl("UTL_doPutc");

var trace = LOG.create("trace");
trace.bufLen = 256;
trace.logType = "circular";

LOG_system.bufLen = 512;
LOG_system.logType = "fixed";

var reader0 = TSK.create("reader0");
reader0.priority = PRIORITY;
reader0.fxn = prog.extern("reader");

for (i = 0; i < NWRITERS; i++) {
    var writer = TSK.create("writer" + i);
    writer.priority = PRIORITY;
    writer.fxn = prog.extern("writer");
    writer.arg0 = i;
}

prog.gen();
```

DSP/BIOS TextConf Reference

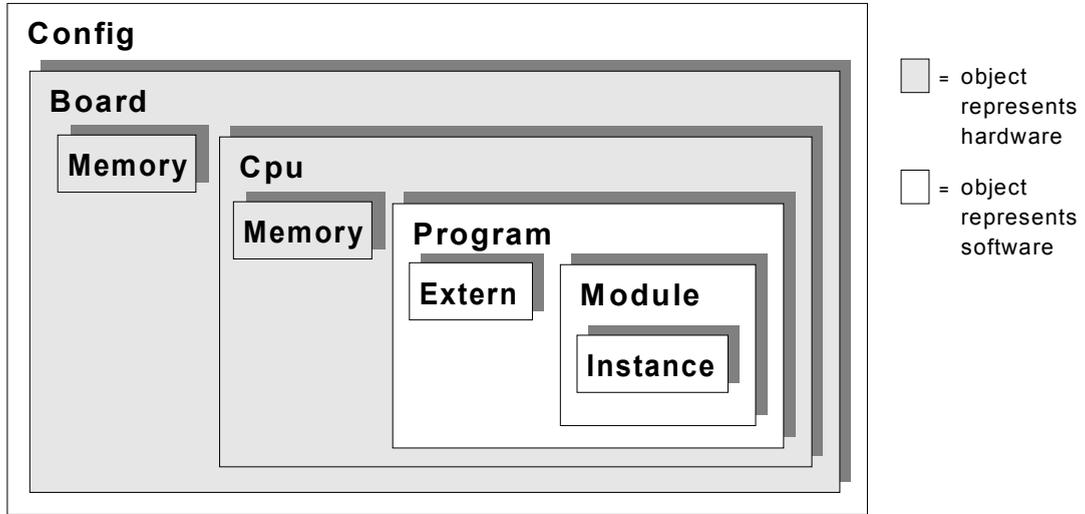
This chapter provides reference information about the Target Content Object Model.

Topic	Page
2.1 Target Content Object Model Reference	2-2
2.2 DSP/BIOS Module and Instance Property Names	2-34
2.3 CSL Module and Instance Property Names	2-34

2.1 Target Content Object Model 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 2-1. Target Content Object Model (TCOM)



2.1.1 Target Content Object Model Quick Reference

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 2-1. Target Content Object Model Summary

Object Type	Objects Contained	Methods	Properties	See Page
Config	Board	board() boards() create() destroy() warn()	hasReportedError name	Page 2–4
Board	Cpu Memory	cpu() cpus() create() destroy()	boardFamily boardRevision config mem[] name pllIndex	Page 2–8

Table 2-1. Target Content Object Model Summary (Continued)

Object Type	Objects Contained	Methods	Properties	See Page
Cpu	Program Memory	create() destroy() program() programs()	board clockOscillator endian id mem[] name attrs.cpuCore attrs.cpuFamily attrs.cpuNumber attrs.cpuCoreRevision attrs.dataWordSize attrs.minDataUnitSize attrs.minProgUnitSize	Page 2–13
Program	Module Extern	extern() externs() destroy() gen() get() load() module() modules() save()	cpu name prog.build.target.model.codeModel prog.build.target.model.dataModel prog.build.target.model.endian	Page 2–18
Memory	-- none --	-- none --	comment name space base len	Page 2–25
Extern	-- none --	-- none --	language name	Page 2–27
Module	Instance	create() instance() instances()	-defined in CDB- name program	Page 2–28
Instance	-- none --	destroy() references()	-defined in CDB- module name	Page 2–31

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.

2.1.2 Config Class

Table 2-2. 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.

```

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.

The sample `tconflocal.tci` script attempts to determine the board type and creates a single Board object called "board_0". See Section 1.5.9.1, *Startup Script Actions*, page 1-35 for details. You can use the `create()` method to create additional Board objects.

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.

Constructor files are currently provided in the `<ccs_base_dir>/bin/utilities/tconf/include` directory (Solaris) or `<ccs_base_dir>/bin/utilities\tconf\include` folder (Windows) for the following `board_types`: Dsk54, Dsk5416, Dsk62, Dsk67, Evm54, Evm55, Evm62, Sim54, Sim55, Sim62, and Sim64. You must load the appropriate constructor file before using the `create()` method in order for JavaScript to find the function contained by the constructor file.

Note: Platforms supported on Solaris

Code Composer Studio for Solaris supports only the 'C55x and 'C64x platforms.

For example, you can create an Evm62 Board object with the following statement:

```
utils.importFile("Evm62");  
board = config.create("board_0", "Evm62");
```

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:

```
/* create global context for configuration scripts */  
var config = new Config("config_0");  
  
/* create an EVM62 Board object within config */  
utils.importFile("Evm62");  
board_0 = config.create("board_0", "Evm62");  
  
/* create a generic Board object within config */  
board_1 = config.create("board_1");  
  
/* display number of DSPs on board_0 */  
print(board_0 + " has " + board.cpus().length + " DSPs");
```

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.

While you will probably not need to use the `destroy()` method when writing configuration scripts from scratch, the `destroy()` method is often needed in scripts created by the `cdbscmp` utility to compare two configuration files.

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 1.5.6, *Error Handling*, page 1-27 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 the interactive debugging shell, 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.

2.1.3 Board Class

Table 2-3. Board Class Summary

Object Type	Contains	Methods	Properties
Board	Cpu Memory	cpu() cpus() create() destroy()	boardFamily boardRevision config mem[] name pllIndex

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 startup script (see Section 1.5.9.1, *Startup Script Actions*, page 1-35). 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();

```

create() Method

Syntax: `create("cpu_name" [, "cpu_type"])`

Parameters: `cpu_name` Required name for new Cpu object.
`cpu_type` Optional sub-type of cpu.

Returns: new Cpu object, or null if error occurs

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

The sample `tconflocal.tci` script attempts to determine the Cpu type and creates a single Cpu object called "cpu_0". See Section 1.5.9.1, *Startup Script Actions*, page 1-35 for details. You can use the `create()` method to create additional Cpu objects.

The first parameter is the name to give the new Cpu object. The name must be unique among the Cpu objects for this Board. This parameter is required.

The second parameter defines the sub-type of cpu to create. This parameter is optional. If you provide a `cpu_type` that matches an existing JavaScript constructor function that has been loaded, that constructor runs to define properties for the Cpu object.

Constructor files are currently provided in the `<ccs_base_dir>/bin/utilities/tconf/include` directory (Solaris) or `<ccs_base_dir>/bin/utilities\tconf\include` folder (Windows) for the following `cpu_types`: C54, C5401, C5402, C5416, C55, C62, C6201, C6211, C64, C67, and C6711. You must load the appropriate constructor file before using the `create()` method in order for JavaScript to find the function contained by the constructor file.

Note: Platforms supported on Solaris

Code Composer Studio for Solaris supports only the 'C55x and 'C64x platforms.

For example, you can create an C54 Cpu object with the following statement:

```
utils.importFile("C54");
config.boards()[0].create("cpu_0", "C54");
```

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 initial Board object under config */
board = config.create("board_0");

/* create a C54 Cpu object under board */
utils.importFile("C54")
board.create("cpu_0", "C54");
```

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.

While you will probably not need to use the `destroy()` method when writing configuration scripts from scratch, the `destroy()` method is often needed in scripts created by the `cdbscmp` utility to compare two configuration files.

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 EVM62 */
boards.EVM62.destroy()
boards["EVM62"].destroy();
```

boardFamily Property	<p>The boardFamily property contains a string that identifies the type of board. Example strings are “evm62”, “dsk54”, and “sim55”.</p> <p>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.</p> <p>Example:</p> <pre> /* 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"); } </pre>
boardRevision Property	<p>The boardRevision property contains an optionally defined string that identifies the board revision number. Example strings are “1.0” and “2.1”.</p> <p>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.</p>
config Property	<p>The config property holds the Config object that contains the Board.</p> <p>Since there is only one Config object, this Config object contains all Boards in the configuration.</p> <p>This property is gettable only. It is set when the Board object is created.</p>
mem[] Array Property	<p>The mem[] array is used to access an array of Memory objects contained by this Board object. For more information, see Section 2.1.6, <i>Memory Class</i>, page 2-25.</p>
name Property	<p>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.</p> <p>Names of Board objects must be unique.</p>
pllIndex Property	<p>The pllIndex (lowercase PLL + “Index”) specifies the bit patterns of the clkmode pins on the board. This value is used to determine factors used to calculate the speed of the CPU, which is important for several other configuration properties. In addition, the clockOscillator property of the Cpu object is used when calculating the speed of the CPU.</p> <p>For example, the ‘C5402 DSK has three clkmode pins. If they are set (from left to right) as off, on, off, the bitmask would be 010 and the resulting pllIndex value is 2 (decimal).</p>

```
/* Define PLL index value used to compute CPU speed */  
config.board("dsk5402").pllIndex = 2;
```

This property is typically set only in the platform file. The range of valid values for the pllIndex is as follows for different targets:

- ❑ For C28: 0 to 31
- ❑ For C54x: 0 to 7
- ❑ For C55: 0 or 1
- ❑ For C6000: 0 to 3

2.1.4 Cpu Class

Table 2-4. Cpu Class Summary

Object Type	Contains	Methods	Properties
Cpu	Program Memory	create() destroy() program() programs()	board clockOscillator endian id mem[] name attrs.cpuCore attrs.cpuFamily attrs.cpuNumber 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 (see Section 1.5.9.1, *Startup Script Actions*, page 1-35). 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 sample tconflocal.tci script attempts to determine the program name and creates a single Program object. See Section 1.5.9.1, *Startup Script Actions*, page 1-35 for details. You can use the create() method to create additional Program objects.

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.

While you will probably not need to use the `destroy()` method when writing configuration scripts from scratch, the `destroy()` method is often needed in scripts created by the `cdncmp` utility to compare two configuration files.

Examples:

```
config.boards["EVM6201"].cpus["C6201"].destroy();
```

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.");
        }
    }
}
```

clockOscillator Property

The clockOscillator property of an object holds the value of the clock oscillator on the board in MHz. This property is typically set only in the platform file. In addition, the pllIndex property of the Board object is used when calculating the speed of the CPU.

Example:

```
/* Define clock oscillator value for CPU in MHz */
config.board("dsk5402").cpu("cpu_0").clockOscillator
= 20.0000;
```

endian Property

The endian property contains "big", "little", or undefined to indicate the byte addressing model used by the board and CPU.

id Property	<p>The id property specifies a unique id for this particular CPU on the board. This property is intended for future use.</p>
mem[] Array Property	<p>The mem[] array is used to access an array of Memory objects contained by this Cpu object. For more information, see Section 2.1.6, <i>Memory Class</i>, page 2-25.</p>
name Property	<p>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.</p> <p>Names of Cpu objects must be unique within the Board object that contains them.</p>
attrs.cpuCore Property	<p>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.</p> <p>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.</p>
attrs.cpuFamily Property	<p>The attrs.cpuFamily property contains the single-digit prefix for the Cpu platform followed by three zeros. Currently, it may be set to 2000, 5000, or 6000.</p> <p>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.cpuFamily property.</p>
attrs.cpuNumber Property	<p>The attrs.cpuNumber property contains the full four-digit number for the Cpu platform. The attrs.cpuNumber is the cpu core number; it identifies the core and a set of peripherals. Example values are 5416, 6201, and 6711. Together the attrs.cpuNumber and attrs.cpuCoreRevision uniquely identify a specific part number.</p> <p>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.cpuNumber property.</p>
attrs.cpuCoreRevision Property	<p>The attrs.cpuCoreRevision property contains an optional revision number of a particular Cpu part. Example values are 1.0, 2.1, and R2. Together the attrs.cpuNumber and attrs.cpuCoreRevision uniquely identify a specific part number.</p> <p>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.</p>

**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.

2.1.5 Program Class

Table 2-5. Program Class Summary

Object Type	Contains	Methods	Properties
Program	Module Extern	extern() externs() destroy() gen() get() load() module() modules() save()	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 methods for saving and loading CDB files. Loading a CDB file defines Module and Instance objects in the JavaScript environment. The create() method of a Program object cannot be used to create Module objects. Saving a CDB file and its generated file allows the settings made via DSP/BIOS TextConf to be linked with the program and used with the DSP/BIOS Real-Time Analysis Tools.

A default Program object is created by the startup script (see Section 1.5.9.1, *Startup Script Actions*, page 1-35). 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 1.5.3.3, *Namespace Management*, page 1-22 for details.

create() Method

Description: The only way to create a Module object is to load a CDB file with the prog.load() or 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.

While you will probably not need to use the `destroy()` method when writing configuration scripts from scratch, the `destroy()` method is often needed in scripts created by the `cdbscmp` utility to compare two configuration files.

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 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 DSP/BIOS TextConf script, you must create a CDB file and 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 a CDB file and all of the files normally generated when you save a CDB file with the DSP/BIOS Configuration Tool or when you use the `gconfggen` utility. See Section 1.2.3, *Configuration File Types*, page 1-8 for descriptions of the generated files.

It is generally recommended (but not required) that the `prog_name` match the output filename of your target program. For example, if your target program executable is `hello.exe`, use the following statement:

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

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

If 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.

Including the .CDB file extension in the `prog_name` parameter is optional. The `gen()` method stores the files it creates in your current directory.

In contrast to the `gen()` method, the `save()` method saves only the CDB file. It does not generate the other associated files.

Example: `prog.gen("myApp1");`

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 1.5.3.3, *Namespace Management*, page 1-22.

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 */
}
```

load() Method

Syntax: `load("cdb_file")`

Parameters: `cdb_file` Filename of CDB file to load objects from

Returns: `void`

Description: The `load()` method reads a CDB file and populates the Target Content Object Model with the Module, Instance, and Extern objects named in the CDB file. This method assumes that Config, Board, Cpu, and Program objects have been created and that `prog` is a global variable that references the Program object.

The `load()` method does not create global variables to reference each Module and Instance object. If you want such global variables to be created, use the `utils.getProgObjs()` method, which is described in Section 1.5.5.2, *Methods for Working with CDB Files*, page 1-25.

Example: `prog.load("evm62.cdb");`

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 1.5.3.3, *Namespace Management*, page 1-22.

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 + " ";
}
```

save() Method

Syntax: `save("prog_name");`

Parameters: `prog_name` Optional name of output CDB file.

Returns: True if successful; false if not successful

Description: The `save()` method saves a CDB file that corresponds to the current object model settings.

In contrast to the `gen()` method, the `save()` method saves only the CDB file. It does not generate the other associated files.

It is generally recommended (but not required) that the `prog_name` match the output filename of your target program. For example, if your target program executable is `hello.exe`, use the following statement:

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

If you omit the `prog_name` parameter, the default `prog_name` is the name of the Program object. If you specify a `prog_name` parameter, the CDB file uses that filename.

Including the `.CDB` file extension in the `prog_name` parameter is optional. The `save()` method stores the CDB file in your current directory.

Example: `prog.save("myAppl");`

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.cpuFamily == "5000") {
    /* C5000-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:

```
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:

```
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:

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

2.1.6 Memory Class

Table 2-6. 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
};
```

There are also no methods to get the name or names of the Memory objects contained by a Board or Cpu. Instead, a script should access the mem[] array within a Board or Cpu object.

Memory objects are typically created only in a platform file (*.tcp).

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.

2.1.7 Extern Class

Table 2-7. 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");
SYS.ABORTFXN = prog.extern("error");
```

The `extern()` method of the Program object (see Section 2.1.5, *Program Class*, page 2-18) 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 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.

2.1.8 Module Class

Table 2-8. Module Class Summary

Object Type	Contains	Methods	Properties
Module	Instance	create() instance() instances()	defined in CDB 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 load a CDB file with the `prog.load()` or `utils.loadPlatform()` method. Do not use the `create()` method of the Program object to create Module objects.

If the `utils.getProgObjs()` method has been called, a global variable is defined for each Module object. For example, DSP/BIOS contains modules named LOG, TSK, and MEM. These correspond to Module objects named LOG, TSK, and MEM. The corresponding global JavaScript variables are LOG, TSK, and MEM.

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

The examples in this section assume that the `utils.getProgObjs()` method was called (with the second parameter omitted) after a CDB file was 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 = PIP.create("input");
           inputPipe.notifyWriterFxn = prog.extern("writerFxn");
           inputPipe.notifyWriterArg0 = 0;
           inputPipe.bufAlign = 32;

           traceLog = LOG.create("trace");
           trace.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 = 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 = 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.

CDB Properties

Normally, all objects in a class have the same set of properties. However, in a CDB file, 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. For more details, see Section 1.5.3.1, *Module and Instance Property Names*, page 1-20.

Each CDB field name is mapped to a property name. In general, the properties of Module objects are in all uppercase letters. For example, "MEM.STACKSIZE". The names are listed in Section 2.2, *DSP/BIOS Module and Instance Property Names*, page 2-34. You can set and get these property values as you would properties of other object classes.

Example:

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

2.1.9 Instance Class

Table 2-9. Instance Class Summary

Object Type	Contains	Methods	Properties
Instance		destroy() references()	defined in CDB 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 CDB file 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 CDB file that has been loaded. All properties can be set, even those that are not writable in the DSP/BIOS Configuration Tool. If setting a property fails because of a rule defined in the CDB file 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."

The examples in this section assume that the `utils.getProgObjs()` method was called (with the second parameter omitted) after a CDB file was loaded.

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.

While you will probably not need to use the `destroy()` method when writing configuration scripts from scratch, the `destroy()` method is often needed in scripts created by the `cdcmp` utility to compare two configuration files.

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 = IDATA.references();
for (i = 0; i < refs.length; i++) {
    print(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 + " ";
}
```

CDB Properties

Normally, all objects in a class have the same set of properties. However, in a CDB file, 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. For more details, see Section 1.5.3.1, *Module and Instance Property Names*, page 1-20.

Each CDB field name is mapped to a property name. 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 list in Section 2.2, *DSP/BIOS Module and Instance Property Names*, page 2-34 of JavaScript names defined for the properties shown in CDB files. You can set and get these property values as you would properties of other object classes.

Example:

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

2.2 DSP/BIOS Module and Instance Property Names

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

- ❑ *TMS320C5000 DSP/BIOS Application Programming Interface Reference Guide* (literature number SPRU404E)
- ❑ *TMS320C6000 DSP/BIOS Application Programming Interface Reference Guide* (literature number SPRU403E)
- ❑ *TMS320C28x DSP/BIOS Application Programming Interface Reference Guide* (literature number SPRU625)

2.3 CSL Module and Instance Property Names

The tables in this section list the property names used in DSP/BIOS TextConf scripts for Chip Support Library (CSL) objects.

- ❑ The name shown after the dash (—) in each table title is the name to use for the Module object in DSP/BIOS TextConf scripts. For example, to create a McBSP Configuration instance:

```
var mcbSPCfg_0 = McBSP.create("mcbSPCfg0");
```

- ❑ The Description column shows the property field labels displayed in the DSP/BIOS Configuration Tool for CSL objects.
- ❑ The TextConf Name column shows the property names to use in DSP/BIOS TextConf scripts to refer to these properties. You can set and get these property values as you would properties of other object classes. For example, the following statement sets the Open Handle to McBSP property of a McBSP resource instance:

```
myMcBSP_0.mcbSPHandleEnable = 1;
```

- ❑ The Type column shows how the value is stored. For information about the data types, see Section 1.5.3.1, *Module and Instance Property Names*, page 1-20.
- ❑ The names of any Instance objects that are pre-defined for a module by the configuration templates are listed in a table after the properties of that Instance type.

Refer to the CSL documentation for descriptions of these properties.

2.3.1 TMS320C54x Properties

Table 2-10. 'C54x DMA Configuration Instance—DMA

Description	TextConf Name	Type
Channel Priority (0x0000 or 0x0001)	dmaDmprecDprcAdv	Numeric
Global Reload Register Usage in Autoinit Mode (AUTOIX: 0x0000 or 0x0001)	dmaAutoixAdv	Numeric
Transfer Mode Control Register (DMMCR)	dmaDmmcr	Numeric
Sync Event and Frame Count Register (DMSFC)	dmaDmsfc	Numeric
Source Address Format	dmaDmsrcFormatAdv	EnumString
Source Address Register (DMSRC) - Numeric	dmaDmsrcNumericAdv	Numeric
Source Address Register (DMSRC) - Symbolic	dmaDmsrcSymbolicAdv	String
Destination Address Format	dmaDmdstFormatAdv	EnumString
Destination Address Register (DMDST) - Numeric	dmaDmdstNumericAdv	Numeric
Destination Address Register (DMDST) - Symbolic	dmaDmdstSymbolicAdv	String
Element Count Register (DMCTR)	dmaDmctrAdv	Numeric
Global Source Address Format	dmaDmgsaFormatAdv	EnumString
Global Source Address Reload Register (DMGSA) - Numeric	dmaDmgsaNumericAdv	Numeric
Global Source Address Reload Register (DMGSA) - Symbolic	dmaDmgsaSymbolicAdv	String
Global Destination Address Format	dmaDmgdaFormatAdv	EnumString
Global Destination Address Reload Register (DMGDA) - Numeric	dmaDmgdaNumericAdv	Numeric
Global Destination Address Reload Register (DMGDA) - Symbolic	dmaDmgdaSymbolicAdv	String
Global Element Count Reload Register (DMGCR)	dmaDmgcrAdv	Numeric
Global Frame Count Reload Register (DMGFR)	dmaDmgfrAdv	Numeric
Extended Source Data Page Register (DMSRCDP) - Numeric	dmaDmsrcdpNumeric	Numeric
Extended Source Data Page Register (DMSRCDP) - Symbolic	dmadmsrcdpSymbolic	String
Extended Destination Data Page Register (DMDSTDP) - Numeric	dmaDmdstdpNumeric	Numeric
Extended Destination Data Page Register (DMDSTDP) - Symbolic	dmaDmdstdpSymbolic	String

Table 2-10. 'C54x DMA Configuration Instance—DMA (Continued)

Description	TextConf Name	Type
Source Program Page Address Register (DMSRCP) - Numeric	dmaDmsrcpNumeric	Numeric
Source Program Page Address Register (DMSRCP) - Symbolic	dmaDmsrcpSymbolic	String
Destination Program Page Address Register (DMDSTP) - Numeric	dmaDmdstpNumeric	Numeric
Destination Program Page Address Register (DMDSTP) - Symbolic	dmaDmdstpSymbolic	String
Element Address Index Register 0 (DMIDX0)	dmaDmidx0Adv	Int16
Frame Address Index Register 0 (DMFRI0)	dmaDmfri0Adv	Int16
Element Address Index Register 1 (DMIDX1)	dmaDmidx1Adv	Int16
Frame Address Index Register 1 (DMFRI1)	dmaDmfri1Adv	Int16

Table 2-11. 'C54x DMA Resource Instance—HDMA

Description	TextConf Name	Type
Open Handle to DMA	dmaEnableHandle	Bool
Specify Handle Name	dmaHandleName	String
Enable pre-initialization	dmaEnablePreInit	Bool
Pre-initialize	dmaPreInit	Reference

Table 2-12. 'C54x HDMA Pre-Created Instance Names

DMA0
DMA1
DMA2
DMA3
DMA4
DMA5

Table 2-13. 'C54x GPIO Configuration Instance—GPIO

Description	TextConf Name	Type
Select IODIR0 as IO0	gpiolo0dir	EnumString
Select IODIR1 as IO1	gpiolo1dir	EnumString
Select IODIR2 as IO2	gpiolo2dir	EnumString
Select IODIR3 as IO3	gpiolo3dir	EnumString

Table 2-14. 'C54x MCBSP Configuration Instance—MCBSP

Description	TextConf Name	Type
Serial Port Control Register 1	mcbSpSpcr1	Numeric
Serial Port Control Register 2	mcbSpSpcr2	Numeric
Receive Control Register 1	mcbSpRcr1	Numeric
Receive Control Register 2	mcbSpRcr2	Numeric
Transmit Control Register 1	mcbSpXcr1	Numeric
Transmit Control Register 2	mcbSpXcr2	Numeric
Sample Rate Generator Register 1	mcbSpSrgr1	Numeric
Sample Rate Generator Register 2	mcbSpSrgr2	Numeric
Multichannel Control Register 1	mcbSpMcr1	Numeric
Multichannel Control Register 2	mcbSpMcr2	Numeric
Pin Control Register	mcbSpPcr	Numeric
Receive Channel Enable Register Partition A	mcbSpRceraAdv	Numeric
Receive Channel Enable Register Partition B	mcbSpRcerbAdv	Numeric
Receive Channel Enable Register Partition C	mcbSpRcercAdv	Numeric
Receive Channel Enable Register Partition D	mcbSpRcerdAdv	Numeric
Receive Channel Enable Register Partition E	mcbSpRcereAdv	Numeric
Receive Channel Enable Register Partition F	mcbSpRcerfAdv	Numeric
Receive Channel Enable Register Partition G	mcbSpRcergAdv	Numeric

Table 2-14. 'C54x MCBSP Configuration Instance—MCBSP (Continued)

Description	TextConf Name	Type
Receive Channel Enable Register Partition H	mcbSPRcerhAdv	Numeric
Transmit Channel Enable Register Partition A	mcbSPXceraAdv	Numeric
Transmit Channel Enable Register Partition B	mcbSPXcerbAdv	Numeric
Transmit Channel Enable Register Partition C	mcbSPXcercAdv	Numeric
Transmit Channel Enable Register Partition D	mcbSPXcerdAdv	Numeric
Transmit Channel Enable Register Partition E	mcbSPXcereAdv	Numeric
Transmit Channel Enable Register Partition F	mcbSPXcerfAdv	Numeric
Transmit Channel Enable Register Partition G	mcbSPXcergAdv	Numeric
Transmit Channel Enable Register Partition H	mcbSPXcerhAdv	Numeric

Table 2-15. 'C54x MCBSP Resource Instance—HMCBSP

Description	TextConf Name	Type
Open Handle to McBSP	mcbSPHandleEnable	Bool
Specify Handle Name	mcbSPHandleName	String
Enable pre-initialization	mcbSPEnablePreInit	Bool
Pre-initialize	mcbSPPreInit	Reference

Table 2-16. 'C54x HMCBSP Pre-Created Instance Names

McBSP0
McBSP1
McBSP2

Table 2-17. 'C54x PLL Configuration Instance—PLL

Description	TextConf Name	Type
PLL Counter Value (PLLCOUNT) [0 - 255]	pllClkmdPllcount	Int16
PLL Multiplier	pllPllmulRatio	EnumString
PLL Multiplier (PLLMUL)	pllClkmdPllmul	EnumInt
CLKOUT Output Divide Factor	pllDivideFactor	EnumString

Table 2-18. 'C54x PLL Resource Instance—HPLL

Description	TextConf Name	Type
Enable Configuration of PLL	pllEnablePreinit	Bool
Pre-initialize	pllPreInit	Reference

Table 2-19. 'C54x HPLL Pre-Created Instance Names

PLL0

Table 2-20. 'C54x Timer Configuration Instance—TIMER

Description	TextConf Name	Type
Timer Control Register	timerTcr	Numeric
Timer Period Register	timerPrdAdv	Numeric
Timer Secondary Control Register (TSCR)	timerTscr	Numeric

Table 2-21. 'C54x Timer Resource Instance—HTIMER

Description	TextConf Name	Type
Open Handle to Timer	timerHandleEnable	Bool
Specify Handle Name	timerHandleName	String
Enable pre-initialization	timerEnablePreInit	Bool
Pre-initialize	timerPreInit	Reference

Table 2-22. 'C54x HTIMER Pre-Created Instance Names

Timer0
Timer1

Table 2-23. 'C54x WDTIMER Configuration Instance—WDTIM

Description	TextConf Name	Type
Timer Control Register	wdtimerTcr	Numeric
Timer Period Register	wdtimerPrdAdv	Numeric
Timer Secondary Control Register (TSCR)	wdtimerTscr	Numeric

Table 2-24. 'C54x WDTIMER Resource Instance—HWDTIM

Description	TextConf Name	Type
Enable Configuration of Watchdog Timer	wdtimerHandleEnable	Bool
Pre-initialize	wdtimerPreInit	Reference

Table 2-25. 'C54x HWDTIM Pre-Created Instance Names

WDTim0

2.3.2 TMS320C55x Properties

Table 2-26. 'C55x CHIP Configuration Instance—CHIP

Description	TextConf Name	Type
Parallel Port Mode	chipXbsrPPMode	EnumString
Serial Port1 Mode	chipXbsrSp1Mode	EnumString
Serial Port 2 Mode	chipXbsrSp2Mode	EnumString

Table 2-27. 'C55x DMA Configuration Instance—DMA

Description	TextConf Name	Type
Set Manually	dmaSetManually	Bool
Source Destination Register (CSDP)	dmaCsdp	Numeric
Control Register (CCR)	dmaCcr	Numeric
Interrupt Control Register (CICR)	dmaCicr	Numeric
Source Space	dmaSrcSpaceAdv	EnumString
Source Address Format	dmaSrcAddrFormatAdv	EnumString
Lower Source Address (CSSA_L)- Numeric (Byte Address)	dmaCsslNumeric	Numeric
Lower Source Address (CSSA_L) - Symbolic (Byte Address)	dmaCsslSymbolic	String
Upper Source Address (CSSA_U) - Numeric (Byte Address)	dmaCsslauNumeric	Numeric
Upper Source Address (CSSA_U) - Symbolic (Byte Address)	dmaCsslauSymbolic	String
Destination Space	dmaDstSpaceAdv	EnumString
Destination Address Format	dmaDstAddrFormatAdv	EnumString
Lower Destination Address (CDSA_L)- Numeric (Byte Address)	dmaCdslNumeric	Numeric
Lower Destination Address (CDSA_L) - Symbolic (Byte Address)	mdmaCdslSymbolic	String
Upper Destination Address (CDSA_U) - Numeric (Byte Address)	mdmaCdslauNumeric	Numeric
Upper Destination Address (CDSA_U) - Symbolic (Byte Address)	mdmaCdslauSymbolic	String
Element Number (CEN)	dmaCenAdv	Numeric
Frame Number (CFN)	dmaCfnAdv	Numeric
Frame Index (CFI)	dmaCfiAdv	Numeric
Source Frame Index (CSFI)	dmaCfiSrcAdv	Numeric
Destination Frame Index (CDFI)	dmaCfiDstAdv	Numeric
Element Index (CEI)	dmaCeiAdv	Numeric
Source Element Index (CSEI)	dmaCeiSrcAdv	Numeric
Destination Element Index (CDEI)	dmaCeiDstAdv	Numeric

Table 2-28. 'C55x DMA Resource Module—HDMA

Description	TextConf Name	Type
PG1.0 Compatibility Mode Select	dmaModeSelect	EnumString
Generate Global DMA Configuration and Function Call	dmaGenerateGlobalCall	Bool
Set Manually	dmaSetManually	Bool
Global Control Register (GCR)	dmaGcr	Numeric
Enable DMA clocks during IDLE (AUTOGATE)	dmaGcrAutogate	Bool

Table 2-29. 'C55x DMA Resource Instance—HDMA

Description	TextConf Name	Type
Open Handle to DMA	dmaHandleEnable	Bool
Specify Handle Name	dmaHandleName	String
Enable pre-initialization	dmaEnablePreInit	Bool
Pre-initialize	dmaPreInit	Reference

Table 2-30. 'C55x HDMA Pre-Created Instance Names

DMA0
DMA1
DMA2
DMA3
DMA4
DMA5

Table 2-31. 'C55x EMIF Configuration Instance—EMIF

Description	TextConf Name	Type
Configure Manually	emifManuallConfigure	Bool
Global Control Register	emifGcr	Numeric
Global Reset Register	emifGrr	Numeric

Table 2-31. 'C55x EMIF Configuration Instance—EMIF (Continued)

Description	TextConf Name	Type
CE0 Space Control Register 1	emifCe0scr1	Numeric
CE0 Space Control Register 2	emifCe0scr2	Numeric
CE0 Space Control Register 3	emifCe0scr3	Numeric
CE1 Space Control Register 1	emifCe1scr1	Numeric
CE1 Space Control Register 2	emifCe1scr2	Numeric
CE1 Space Control Register 3	emifCe1scr3	Numeric
CE2 Space Control Register 1	emifCe2scr1	Numeric
CE2 Space Control Register 2	emifCe2scr2	Numeric
CE2 Space Control Register 3	emifCe2scr3	Numeric
CE3 Space Control Register 1	emifCe3scr1	Numeric
CE3 Space Control Register 2	emifCe3scr2	Numeric
CE3 Space Control Register 3	emifCe3scr3	Numeric
SDRAM Control Register 1	emifSdcr1	Numeric
SDRAM Period Register	emifSdperiodAdv	Numeric
SDRAM Initialization Register	emifSdinitAdv	Numeric
SDRAM Control Register 2	emifSdcr2	Numeric

Table 2-32. 'C55x EMIF Resource Instance—HEMIF

Description	TextConf Name	Type
Enable pre-initialization	emifEnablePreInit	Bool
Pre-initialize	emifPreInit	Reference

Table 2-33. 'C55x HEMIF Pre-Created Instance Names

hEMIF0

Table 2-34. 'C55x GPIO Configuration Instance—GPIO

Description	TextConf Name	Type
Configure Non-multiplexed GPIO	gpioConfigure	Bool
Select IODIR0 as IO0	gpiolo0dir	EnumString
Select IODIR1 as IO1	gpiolo1dir	EnumString
Select IODIR2 as IO2	gpiolo2dir	EnumString
Select IODIR3 as IO3	gpiolo3dir	EnumString
Select IODIR4 as IO4	gpiolo4dir	EnumString
Select IODIR5 as IO5	gpiolo5dir	EnumString
Select IODIR6 as IO6	gpiolo6dir	EnumString
Select IODIR7 as IO7	gpiolo7dir	EnumString

Table 2-35. 'C55x MCBSP Configuration Instance—MCBSP

Description	TextConf Name	Type
Set Manually	mcbSPManualSet	Bool
Serial Port Control Register 1	mcbSPsPcr1	Numeric
Serial Port Control Register 2	mcbSPsPcr2	Numeric
Receive Control Register 1	mcbSPRcr1	Numeric
Receive Control Register 2	mcbSPRcr2	Numeric
Transmit Control Register 1	mcbSPXcr1	Numeric
Transmit Control Register 2	mcbSPXcr2	Numeric
Sample Rate Generator Register 1	mcbSPSgr1	Numeric
Sample Rate Generator Register 2	mcbSPSgr2	Numeric
Multichannel Control Register 1	mcbSPMcr1	Numeric
Multichannel Control Register 2	mcbSPMcr2	Numeric
Pin Control Register	mcbSPPcr	Numeric
Receive Channel Enable Register Partition A	mcbSPRceraAdv	Numeric

Table 2-35. 'C55x MCBSP Configuration Instance—MCBSP (Continued)

Description	TextConf Name	Type
Receive Channel Enable Register Partition B	mcbbspRcerbAdv	Numeric
Receive Channel Enable Register Partition C	mcbbspRcercAdv	Numeric
Receive Channel Enable Register Partition D	mcbbspRcerdAdv	Numeric
Receive Channel Enable Register Partition E	mcbbspRcereAdv	Numeric
Receive Channel Enable Register Partition F	mcbbspRcerfAdv	Numeric
Receive Channel Enable Register Partition G	mcbbspRcergAdv	Numeric
Receive Channel Enable Register Partition H	mcbbspRcerhAdv	Numeric
Transmit Channel Enable Register Partition A	mcbbspXceraAdv	Numeric
Transmit Channel Enable Register Partition B	mcbbspXcerbAdv	Numeric
Transmit Channel Enable Register Partition C	mcbbspXcercAdv	Numeric
Transmit Channel Enable Register Partition D	mcbbspXcerdAdv	Numeric
Transmit Channel Enable Register Partition E	mcbbspXcereAdv	Numeric
Transmit Channel Enable Register Partition F	mcbbspXcerfAdv	Numeric
Transmit Channel Enable Register Partition G	mcbbspXcergAdv	Numeric
Transmit Channel Enable Register Partition H	mcbbspXcerhAdv	Numeric

Table 2-36. 'C55x MCBSP Resource Instance—HMCBSP

Description	TextConf Name	Type
Open Handle to McBSP	mcbbspHandleEnable	Bool
Specify Handle Name	mcbbspHandleName	String
Enable pre-initialization	mcbbspEnablePreInit	Bool
Pre-initialize	mcbbspPreInit	Reference

Table 2-37. 'C55x HMCBSP Pre-Created Instance Names

hMCBSP0
hMCBSP1
hMCBSP2

Table 2-38. 'C55x PLL Configuration Instance—PLL

Description	TextConf Name	Type
PLL Response After Idle (IAI)	pllClkmdlai	EnumString
Response to Loss of PLL Core Lock (IOB)	pllClkmdlob	EnumString
PLL Multiply Value (PLL_MULT)	pllClkmdPllmult	Numeric
PLL Divide Value (PLL_DIV)	pllClkmdPlldiv	EnumString

Table 2-39. 'C55x PLL Resource Instance—HPLL

Description	TextConf Name	Type
Enable Configuration of PLL	pllEnablePreInit	Bool
Pre-initialize	pllPreInit	Reference

Table 2-40. 'C55x HPLL Pre-Created Instance Names

PLL0

Table 2-41. 'C55x PWR Configuration Instance—PWR

Description	TextConf Name	Type
Enable Pre-initialization	pwrConfigPwr	Bool
Clock Generator Disable	pwrClockgenPwrDwnMode	Bool
CACHE Disable	pwrCachePwrDwnMode	Bool
CPU Disable	pwrCpuPwrDwnMode	Bool
DMA Disable	pwrDmaPwrDwnMode	Bool
EMIF Disable	pwrEmifPwrDwnMode	Bool
McBsp 0 Disable	pwrMcbasp0PwrDwnMode	Bool
McBsp 1 Disable	pwrMcbasp1PwrDwnMode	Bool
McBsp 2 Disable	pwrMcbasp2PwrDwnMode	Bool

Table 2-41. 'C55x PWR Configuration Instance—PWR (Continued)

Description	TextConf Name	Type
Timer 0 Disable	pwrTimer0PwrDwnMode	Bool
Timer 1 Disable	pwrTimer1PwrDwnMode	Bool
I2C Disable	pwrI2cPwrDwnMode	Bool
Clockout Pin Disable	pwrClockoutPwrDwnMode	Bool
Disable CLKMEM	pwrClkmemPwrDwnMode	Bool
Oscillator Disable	pwrOscPwrDwnMode	Bool

Table 2-42. 'C55x Real Time Clock Configuration Instance—RTC

Description	TextConf Name	Type
Set Manually	rtcSetManually	Bool
Seconds Register (RTCSEC)	rtcRtcsecAdv	Numeric
Seconds Alarm Register (RTCSECA)	rtcRtcsecaAdv	Numeric
Minutes Register (RTCMIN)	rtcRtcminAdv	Numeric
Minutes Alarm Register (RTCMINA)	rtcRtcminaAdv	Numeric
Hour Register (RTCHOUR)	rtcRtchourAdv	Numeric
Hour Alarm Register (RTCHOURA)	rtcRtchouraAdv	Numeric
Day of the Week Register (RTCDAYW)	rtcRtcdaywAdv	Numeric
Day of the Month Register (RTCDAYM)	rtcRtcdaymAdv	Numeric
Month Register (RTCMONTH)	rtcRtcmonthAdv	Numeric
Year Register (RTCYEAR)	rtcRtcyearAdv	Numeric
Periodic Interrupt Selection Register (RTCPINTR)	rtcRtcpintrAdv	Numeric
Interrupt Enable Register (RTCINTEN)	rtcRtcintenAdv	Numeric

Table 2-43. 'C55x Real Time Clock Resource Instance—RTCRES

Description	TextConf Name	Type
Enable RTC Configuration	rtcCfgEnable	Bool
Pre-initialize	rtcPreInit	Reference

Table 2-44. 'C55x RTCRES Pre-Created Instance Names

RTC0

Table 2-45. 'C55x Timer Configuration Instance—TIMER

Description	TextConf Name	Type
Set Manually	timerSetManually	Bool
Timer Control Register (TCR)	timerTcr	Numeric
Timer Period Register (PRD)	timerPrdAdv	Numeric
Timer Prescalar Register (PRSC)	timerPrsc	Numeric

Table 2-46. 'C55x Timer Resource Instance—HTIMER

Description	TextConf Name	Type
Open Handle to Timer	timerHandleEnable	Bool
Specify Handle Name	timerHandleName	String
Enable pre-initialization	timerEnablePreInit	Bool
Pre-initialize	timerPreInit	Reference

Table 2-47. 'C55x HTIMER Pre-Created Instance Names

TIMER0
TIMER1

Table 2-48. 'C55x USBRES Pre-Created Instance Names

USB0

Table 2-49. 'C55x WDTIMER Configuration Instance—WDTIM

Description	TextConf Name	Type
WD Timer Control Register (WDTCR)	wdtimWdtrAdv	Numeric
WD Timer Period Register (WDPRD)	wdtimWdprdAdv	Numeric
WD Timer Secondary Control Register (WDTCR2)	wdtimWdtr2Adv	Numeric

Table 2-50. 'C55x WDTIMER Resource Instance—HWDTIM

Description	TextConf Name	Type
Enable Configuration of Watchdog Timer	wdtimEnablePreInit	Bool
Pre-initialize	wdtimPreInit	Reference

2.3.3 TMS320C6000 Properties

Table 2-51. 'C6000 DMA Global Register Module—GDMA

Description	TextConf Name	Type
Pre-allocation Global Address Reload Register A	DMA_PRE_ALLOC_GBLADDRA	Bool
Pre-allocation Global Address Reload Register B	DMA_PRE_ALLOC_GBLADDRB	Bool
Pre-allocation Global Address Reload Register C	DMA_PRE_ALLOC_GBLADDRC	Bool
Pre-allocation Global Address Reload Register D	DMA_PRE_ALLOC_GBLADDRD	Bool
Pre-allocation Global Index Register A	DMA_PRE_ALLOC_GBLIDXA	Bool
Pre-allocation Global Index Register B	DMA_PRE_ALLOC_GBLIDXB	Bool
Pre-allocation Global Count Reload Register A	DMA_PRE_ALLOC_GBLCNTA	Bool
Pre-allocation Global Count Reload Register B	DMA_PRE_ALLOC_GBLCNTB	Bool

Table 2-51. 'C6000 DMA Global Register Module—GDMA (Continued)

Description	TextConf Name	Type
DMA Global Register ID	DMA_HANDLE_NAME	String
Enable Pre-Initialization	DMA_ENABLE_PRE_INIT	Bool
Pre-Initialize with	DMA_PRE_INIT	Reference

Table 2-52. 'C6000 DMA Global Register Instance—GDMA

Description	TextConf Name	Type
Global Address Reload Register A Format	dmaGbladdrAFormatAdv	EnumString
Reload Register A - Numeric	dmaGbladdrANumericAdv	Numeric
Reload Register A - Symbolic	dmaGbladdrASymbolicAdv	String
Global Address Reload Register B Format	dmaGbladdrBFormatAdv	EnumString
Reload Register B - Numeric	dmaGbladdrBNumericAdv	Numeric
Reload Register B - Symbolic	dmaGbladdrBSymbolicAdv	String
Global Address Reload Register C Format	dmaGbladdrCFormatAdv	EnumString
Reload Register C - Numeric	dmaGbladdrCNumericAdv	Numeric
Reload Register C - Symbolic	dmaGbladdrCSymbolicAdv	String
Global Address Reload Register D Format	dmaGbladdrDFormatAdv	EnumString
Reload Register D - Numeric	dmaGbladdrDNumericAdv	Numeric
Reload Register D - Symbolic	dmaGbladdrDSymbolicAdv	String
Global Index Register A	dmaGblidxAAdv	Numeric
Global Index Register B	dmaGblidxBAdv	Numeric
Global Count Reload Register A	dmaGblcntAAdv	Numeric
Global Count Reload Register B	dmaGblcntBAdv	Numeric

Table 2-53. 'C6000 DMA Configuration Instance—DMA

Description	TextConf Name	Type
Primary Control Register	dmaPrictl	Numeric
Secondary Control Register	dmaSecctl	Numeric
Source Address Format	dmaSrcAddrFormatAdv	EnumString
Source Address - Numeric	dmaSrcAddrNumericAdv	Numeric
Destination Address Format	dmaDstAddrFormatAdv	EnumString
Destination Address - Numeric	dmaDstAddrNumericAdv	Numeric
Transfer Counter Format	dmaXfrcntFormatAdv	EnumString
Transfer Counter - Numeric	dmaXfrcnt	Numeric

Table 2-54. 'C6000 DMA Resource Instance—HDMA

Description	TextConf Name	Type
Open DMA Channel	dmaHandleEnable	Bool
Handle	dmaHandleName	String
Enable Pre-Initialization	dmaEnablePreInit	Bool
Pre-Initialize with	dmaPreInit	Reference

Table 2-55. 'C6000 HDMA Pre-Created Instance Names

DMA_Channel0
 DMA_Channel1
 DMA_Channel2
 DMA_Channel3

Table 2-56. 'C6000 EDMA Configuration Instance—EDMA

Description	TextConf Name	Type
Option	edmaOptions	Numeric
Source Address Format	edmaSrcAddrFormatAdv	EnumString
Source Address - Numeric	edmaSrcAddrNumericAdv	Numeric
Transfer Counter Format	edmaTransferCounterFormatAdv	EnumString
Transfer Counter - Numeric	edmaTransferCounterNumeric	Numeric
Destination Address Format	edmaDstAddrFormatAdv	EnumString
Destination Address - Numeric	edmaDstAddrNumericAdv	Numeric
Index Format	edmaIndexFormatAdv	EnumString
Index register - Numeric	edmaIndexNumeric	Numeric
Element Count Reload and Link Address	edmaEcrldLinkAddr	Numeric

Table 2-57. 'C6000 EDMA Resource Instance—HEDMA

Description	TextConf Name	Type
Open EDMA Channel	edmaHandleEnable	Bool
Handle	edmaHandleName	String
Enable Pre-Initialization	edmaEnablePreInit	Bool
Pre-Initialize with	edmaPreInit	Reference
Enable Selected Channel	edmaEnableChannel	Bool

Table 2-58. 'C6000 HEDMA Pre-Created Instance Names

EDMA_Channel0_DSPINT
EDMA_Channel1_TINT0
EDMA_Channel2_TINT1
EDMA_Channel3_SDINT
EDMA_Channel4_EXTINT4_GPINT4
EDMA_Channel5_EXTINT5_GPINT5
EDMA_Channel6_EXTINT6_GPINT6
EDMA_Channel7_EXTINT7_GPINT7
EDMA_Channel8_TCC8_GPINT0
EDMA_Channel9_TCC9_GPINT1
EDMA_Channel10_TCC10_GPINT2
EDMA_Channel11_TCC11_GPINT3
EDMA_Channel12_XEVT0
EDMA_Channel13_REVT0
EDMA_Channel14_XEVT1
EDMA_Channel15_REVT1
EDMA_Channel16
EDMA_Channel17_XEVT2
EDMA_Channel18_REVT2
EDMA_Channel19_TINT2
EDMA_Channel20_SDINTB
EDMA_Channel21_PCI
EDMA_Channel22
EDMA_Channel23
EDMA_Channel24
EDMA_Channel25
EDMA_Channel26
EDMA_Channel27
EDMA_Channel28_VCPREVT
EDMA_Channel29_VCPXEVT
EDMA_Channel30_TCPREVT
EDMA_Channel31_TCPXEVT
EDMA_Channel32_UREVT
EDMA_Channel33
EDMA_Channel34
EDMA_Channel35
EDMA_Channel36
EDMA_Channel37
EDMA_Channel38
EDMA_Channel39
EDMA_Channel40_UXEVT
EDMA_Channel41
EDMA_Channel42

Table 2-58. 'C6000 HEDMA Pre-Created Instance Names

EDMA_Channel43
 EDMA_Channel44
 EDMA_Channel45
 EDMA_Channel46
 EDMA_Channel47
 EDMA_Channel48_GPINT8
 EDMA_Channel49_GPINT9
 EDMA_Channel50_GPINT10
 EDMA_Channel51_GPINT11
 EDMA_Channel52_GPINT12
 EDMA_Channel53_GPINT13
 EDMA_Channel54_GPINT14
 EDMA_Channel55_GPINT15
 EDMA_Channel56
 EDMA_Channel57
 EDMA_Channel58
 EDMA_Channel59
 EDMA_Channel60
 EDMA_Channel61
 EDMA_Channel62
 EDMA_Channel63

Table 2-59. 'C6000 Parameter RAM Table Entry Instance—EdmaTable

Description	TextConf Name	Type
Allocate Parameter RAM Table	edmaAllocPramTable	Bool
Allocate Table Number	edmaTableNumber	Numeric
Enable Pre-Initialization	edmaEnablePreInit	Bool
Pre-Initialize with	edmaPreInit	Reference

Table 2-60. 'C6000 EMIF Configuration Instance—EMIF

Description	TextConf Name	Type
Global Control Reg. (GBLCTL)	emifGblctl	Numeric
CE0 Space Control Reg. (Cectl0)	emifCectl0	Numeric
CE1 Space Control Reg. (Cectl1)	emifCectl1	Numeric
CE2 Space Control Reg. (Cectl2)	emifCectl2	Numeric

Table 2-60. 'C6000 EMIF Configuration Instance—EMIF (Continued)

Description	TextConf Name	Type
CE3 Space Control Reg. (Cectl3)	emifCectl3	Numeric
SDRAM Control Reg.(SDCTL)	emifSdctl	Numeric
SDRAM Timing Reg.(SDTIM)	emifSdtim	Numeric
SDRAM Extended Reg.(SDEXT)	emifSdext	Numeric

Table 2-61. 'C6000 EMIF Resource Instance—HEMIF

Description	TextConf Name	Type
Enable Pre-Initialization	emifEnablePreInit	Bool
Pre-Initialize with	emifPreInit	Reference

Table 2-62. 'C6000 EMIFA Configuration Instance—EMIFA

Description	TextConf Name	Type
Global Control Reg. (GBLCTL)	emifaGblctl	Numeric
CE0 Space Control Reg. (Cectl0)	emifaCectl0	Numeric
CE1 Space Control Reg. (Cectl1)	emifaCectl1	Numeric
CE2 Space Control Reg. (Cectl2)	emifaCectl2	Numeric
CE3 Space Control Reg. (Cectl3)	emifaCectl3	Numeric
SDRAM Control Reg.(SDCTL)	emifaSdctl	Numeric
SDRAM Timing Reg.(SDTIM)	emifaSdtim	Numeric
SDRAM Extended Reg.(SDEXT)	emifaSdext	Numeric
CE0 Space Secondary Control Reg. (Cesec0)	emifaCesec0	Numeric
CE1 Space Secondary Control Reg. (Cesec1)	emifaCesec1	Numeric
CE2 Space Secondary Control Reg. (Cesec2)	emifaCesec2	Numeric
CE3 Space Secondary Control Reg. (Cesec3)	emifaCesec3	Numeric

Table 2-63. 'C6000 EMIFA Resource Instance—HEMIFA

Description	TextConf Name	Type
Enable Pre-Initialization	emifaEnablePreInit	Bool
Pre-Initialize with	emifaPreInit	Reference

Table 2-64. 'C6000 EMIFB Configuration Instance—EMIFB

Description	TextConf Name	Type
Global Control Reg. (GBLCTL)	emifbGblctl	Numeric
CE0 Space Control Reg. (Cectl0)	emifbCectl0	Numeric
CE1 Space Control Reg. (Cectl1)	emifbCectl1	Numeric
CE2 Space Control Reg. (Cectl2)	emifbCectl2	Numeric
CE3 Space Control Reg. (Cectl3)	emifbCectl3	Numeric
SDRAM Control Reg.(SDCTL)	emifbSdctl	Numeric
SDRAM Timing Reg.(SDTIM)	emifbSdtim	Numeric
SDRAM Extended Reg.(SDEXT)	emifbSdext	Numeric
CE0 Space Secondary Control Reg. (Cesec0)	emifbCesec0	Numeric
CE1 Space Secondary Control Reg. (Cesec1)	emifbCesec1	Numeric
CE2 Space Secondary Control Reg. (Cesec2)	emifbCesec2	Numeric
CE3 Space Secondary Control Reg. (Cesec3)	emifbCesec3	Numeric

Table 2-65. 'C6000 EMIFB Resource Instance—HEMIFB

Description	TextConf Name	Type
Enable Pre-Initialization	emifbEnablePreInit	Bool
Pre-Initialize with	emifbPreInit	Reference

Table 2-66. 'C6000 CSL Extern Declaration Module—ExternDecl

Description	TextConf Name	Type
Enter header filename between quotes	HEADER_FILENAME	String

Table 2-67. 'C6000 CSL Extern Declaration Instance—ExternDecl

Description	TextConf Name	Type
CSL Symbol Type (ex: Uint32)	bufType	String
Symbol Name (ex: BuffA)	bufName	String
Symbol Specification	bufSpec	EnumString

Table 2-68. 'C6000 MCBSP Configuration Instance—MCBSP

Description	TextConf Name	Type
Serial Port Control Reg. (SPCR)	mcbSPSPcr	Numeric
Receiver Control Reg. (RCR)	mcbSPRcr	Numeric
Transmitter Control Reg. (XCR)	mcbSPXcr	Numeric
Sample-Rate Generator Reg. (SRGR)	mcbSPSrgR	Numeric
Multichannel Control Reg. (MCR)	mcbSPMcr	Numeric
Receiver Channel Enable(RCER)	mcbSPRcer	Numeric
Transmitter Channel Enable(XCER)	mcbSPXcer	Numeric
Pin Control Reg. (PCR)	mcbSPPcr	Numeric

Table 2-69. 'C6000 MCBSP Resource Instance—HMCBSP

Description	TextConf Name	Type
Open MCBSP Port	mcbSPHandleEnable	Bool
Handle	mcbSPHandleName	String
Enable Pre-Initialization	mcbSPEnablePreInit	Bool
Pre-Initialize with	mcbSPPreInit	Reference

Table 2-70. 'C6000 HMCBSP Pre-Created Instance Names

Mcbbsp_Port0
 Mcbsp_Port1
 Mcbsp_Port2

Table 2-71. 'C6000 TCP Base Parameters—TCPBP

Description	TextConf Name	Type
Decoder Standard (3GPP/IS2000)	tcpBaseParamStandard	EnumString
Code Rate (RATE)	tcpBaseParamRate	EnumString
Frame length (FL)	tcpBaseParamFrameLen	Int16
Prolog Size (P)	tcpBaseParamProlSize	Int16
Maximum Iteration (MAXIT)	tcpBaseParamMaxIt	Int16
Snr Threshold (SNR)	tcpBaseParamSnrTh	Int16
Output Paramters Read Flag (OUTF)	tcpBaseParamOutFlag	EnumString
Interleaver Write Flag (INTER)	tcpBaseParamInterFlag	EnumString

Table 2-72. 'C6000 TCP Configuration Instance—TCPBP

Description	TextConf Name	Type
Input control register 0 (IC0)	tcplc0	Numeric
Input control register 1 (IC1)	tcplc1	Numeric
Input control register 2 (IC2)	tcplc2	Numeric
Input control register 3 (IC3)	tcplc3	Numeric
Input control register 4 (IC4)	tcplc4	Numeric
Input control register 5 (IC5)	tcplc5	Numeric

Table 2-73. 'C6000 TCP Resource Instance—HTCP

Description	TextConf Name	Type
Enable Parameters Setting	tcpEnableParams	Bool
Pre-Initialize with	tcpBaseParamInit	Reference
Output TCP Params ConfigName (ex: tcpParam)	tcpParamConfigName	EnumString
Set TCP Params Values to the IC Config. Obj	tcpSetParamEnable	Bool
Enable Pre-Initialization	tcpEnablePreInit	Bool
Pre-Initialize with	tcpPreInit	Reference

Table 2-74. 'C6000 TIMER Configuration Instance—TIMER

Description	TextConf Name	Type
Control Register (CTL)	timerCtlAdv	Numeric
Period Register (PRD)	timerPrdAdv	Numeric
Counter Register (CNT)	timerCntAdv	Numeric

Table 2-75. 'C6000 TIMER Resource Instance—HTIMER

Description	TextConf Name	Type
Open Timer Device	timerHandleEnable	Bool
Handle	timerHandleName	String
Enable Pre-Initialization	timerEnablePreInit	Bool
Pre-Initialize with	timerPreInit	Reference

Table 2-76. 'C6000 HTIMER Pre-Created Instance Names

Timer_Device0
Timer_Device1
Timer_Device2

Table 2-77. 'C6000 VCP Base Parameters—VCPBP

Description	TextConf Name	Type
Code Rate	vcpBaseParamRate	EnumString
Constraints Length	vcpBaseParamConstrLen	EnumString
Frame length (F)	vcpBaseParamFrameLen	Int16
Yamamoto Threshold (YAMTH)	vcpBaseParamYamth	Int16
Max State Index Iteration (IMAXI)	vcpBaseParamIndexMaxIter	Int16
Output Hard Decision (SDHD)	vcpBaseParamDecisionType	EnumString
Output Paramters Read Flag (OUTF)	vcpBaseParamOutFlag	EnumString

Table 2-78. 'C6000 VCP Configuration Instance—VCPBP

Description	TextConf Name	Type
Input control register 0 (IC0)	vcplc0	Numeric
Input control register 1 (IC1)	vcplc1	Numeric
Input control register 2 (IC2)	vcplc2	Numeric
Input control register 3 (IC3)	vcplc3	Numeric
Input control register 4 (IC4)	vcplc4	Numeric
Input control register 5 (IC5)	vcplc5	Numeric

Table 2-79. 'C6000 VCP Resource Instance—HVCP

Description	TextConf Name	Type
Enable Parameters Setting	vcpEnableParams	Bool
Pre-Initialize with	vcpBaseParamInit	Reference
Output VCP Params ConfigName (ex: vcpParam)	vcpParamConfigName	EnumString
Set VCP Params Values to the IC Config. Obj	vcpSetParamEnable	Bool
Enable Pre-Initialization	vcpEnablePreInit	Bool
Pre-Initialize with	vcpPreInit	Reference

Table 2-80. 'C6000 XBUS Configuration Instance—XBUS

Description	TextConf Name	Type
Global Control Register(XBGC)	xbusXbgc	Numeric
XCE0 Space Control Register(XCectl0)	xbusXcectl0	Numeric
XCE1 Space Control Register(XCectl1)	xbusXcectl1	Numeric
XCE2 Space Control Register(XCectl2)	xbusXcectl2	Numeric
XCE3 Space Control Register(XCectl3)	xbusXcectl3	Numeric
XBUS HPI Control Register(XBHC)	xbusXbhc	Numeric
XBUS Internal Master Address Register(XBIMA)	xbusXbimaAdv	Numeric
XBUS External Address Register(XBEA)	xbusXbeaAdv	Numeric

Table 2-81. 'C6000 XBUS Resource Instance—HXBUS

Description	TextConf Name	Type
Enable Pre-Initialization	xbusEnablePreInit	Bool
Pre-Initialize with	xbusPreInit	Reference



Index

A

- Arg data type 1-20
- arguments array 1-24
- array
 - arguments 1-24
 - environment 1-22
 - methods 1-20
 - of objects 1-18, 1-19
 - properties 1-19
- assembly header file 1-9
- assembly source file 1-9

B

- base property
 - Memory object 2-25
- big endian 2-15, 2-24
- Board object 2-8
 - defined by startup script 1-35
 - initializing 1-36
- board property
 - Cpu object 2-15
- board() method
 - Config object 2-4
- boardFamily property
 - Board object 2-11
- boardRevision property
 - Board object 2-11
- boards() method
 - Config object 2-4
- Boolean values 1-20
- bracket ([]) notation 1-18
- branching 1-5
- build mechanisms 1-31

C

- C header file 1-9
- C source file 1-9
- C54x properties for CSL 2-35
- C55x properties for CSL 2-40

- C6000 properties for CSL 2-49
- catch keyword 1-29
- catching exceptions 1-29
- CDB file 1-3, 1-8
 - comparing 1-41
 - generating project files 1-41, 2-20
 - loading 1-25, 2-22
 - migrating to TCF 1-15
 - saving 2-23
 - size 1-6
- CDB properties
 - Instance object 2-33
 - Module object 2-30
- cdbcmp utility 1-15, 1-41
- CHIP properties
 - 'C55x 2-40
- Chip Support Library properties 2-34
- clkmode pins 2-11
- clockOscillator property
 - Cpu object 2-15
- CMD file 1-9
- code size
 - minimizing 1-43
- codeModel property
 - Program object 2-24
- command-line mode 1-40
- command-line utilities 1-9, 1-39
- comment property
 - Memory object 2-25
- comparing configurations 1-16
- comparison on floats 1-21
- compilerOpts 1-23, 2-24
- Config object 2-4
 - defined by startup script 1-35
- config property
 - Board object 2-11
- config.rootDir variable 1-23
- config.scriptName variable 1-23
- config.tiRoot variable 1-23
- configuration methods 1-2
- configurations
 - comparing 1-16
 - merging 1-16
- containers 1-18

- conventions
 - coding 1-29
 - object model initialization 1-36
- Cpu object 2-13
 - defined by startup script 1-35
- cpu property
 - Program object 2-24
- CPU speed 2-11
- cpu() method
 - Board object 2-8
- cpuCore property
 - Cpu object 2-16
- cpuCoreRevision property
 - Cpu object 2-16
- cpuFamily property
 - Cpu object 2-16
- cpuNumber property
 - Cpu object 2-16
- cpus() method
 - Board object 2-8
- create() method
 - Board object 2-9
 - Config object 2-5
 - Cpu object 2-13
 - Instance object 2-31
 - Module object 2-28
 - Program object 2-18
- creating scripts 1-11
- CSL Extern Declaration properties
 - 'C6000 2-57
- CSL properties 2-34

D

- D option 1-39, 1-40
- data types 1-17, 1-20
 - Arg 1-20
 - Boolean 1-20
 - EnumInt 1-21
 - EnumString 1-21
 - Extern 1-21
 - Int16 1-21
 - Int32 1-21
 - Numeric 1-21
 - Reference 1-21
 - String 1-21
 - word size 2-17
- dataModel property
 - Program object 2-24
- dataWordSize property
 - Cpu object 2-17
- debugging 1-12
 - GUI debugger 1-14
 - interactive shell 1-12
- decimal values 1-21

- dependencies
 - on objects 2-32
 - statement order 1-15
- design-time configuration 1-2
- destroy() method
 - Board object 2-10
 - Config object 2-6
 - Cpu object 2-14
 - Instance object 2-31
 - Program object 2-19
- differences between configurations 1-16
- directory path 1-24, 1-39
- DMA Global Register properties
 - 'C6000 2-49, 2-50
- DMA properties
 - 'C54x 2-35, 2-36
 - 'C55x 2-41, 2-42
 - 'C6000 2-51
- Document Object Model (DOM) 1-7, 1-17
- documentation, other 1-38
- dot (.) notation 1-18
- DSP/BIOS 1-2
- DSP/BIOS Configuration Tool 1-2
 - advantages 1-4
- DSP/BIOS TextConf 1-3
 - advantages 1-4
 - for new applications 1-10
 - migrating existing applications 1-15
- dynamic objects 1-2

E

- ECMA-262 1-4, 1-17
- EDMA properties
 - 'C6000 2-52
- EMIF properties
 - 'C55x 2-42, 2-43
 - 'C6000 2-54, 2-55, 2-56
- endian property
 - Cpu object 2-15
 - Program object 2-24
- enumerated data type 1-21
- EnumInt data type 1-21
- EnumString data type 1-21
- environment array 1-22, 1-39, 1-40
- error handling 1-27
- errors 1-28
- examples 1-43
 - hello world 1-6
 - minimizing code size 1-43
- exceptions 1-28
 - catching 1-29
 - throwing 1-28
- exit keyword 1-13
- exit status 1-28

- exiting from tconf 1-13
- Extern data type 1-21
- Extern Declaration properties
 - 'C6000 2-57
- Extern object 1-21, 2-27
 - creating 2-19
- extern() method
 - Program object 2-19
- externs() method
 - Program object 2-20

F

- family of CPU 2-16
- far model 2-24
- file services 1-18, 1-26
- files
 - assembly header 1-9
 - assembly source 1-9
 - C header 1-9
 - C source 1-9
 - CDB 1-8, 1-15
 - CMD 1-9
 - naming 1-29
 - startup 1-8
 - TCF 1-8
 - TCI 1-8
 - TCP 1-8
 - template 1-8
- findSeed() method 1-26
- floating values 1-21
- function names 1-21

G

- gconfgen utility 1-41
- gen() method 1-41
 - Program object 2-20
 - when to use 1-11
- get() method
 - Program object 2-21
- getProgObjs() method 1-8, 1-11, 1-26
- global variables 1-18
 - on command line 1-39, 1-40
- GPIO properties
 - 'C54x 2-37
 - 'C55x 2-44
- graphical configuration 1-2
- GUI debugger 1-14
 - command line 1-40
- guidelines
 - coding 1-29
 - object model initialization 1-36

H

- hardware specification 1-30
- hasReportedError property
 - Config object 2-7
- header files 1-9
- hierarchy of objects 1-7

I

- id property
 - Cpu object 2-16
- importFile() method 1-25
- importPath 1-23
- include file 1-8
- initializing the object model 1-36
- Instance object 2-31
 - CDB properties 1-20
- instance() method
 - Module object 2-29
- instanceof operator 1-22
- instances() method
 - Module object 2-29
- Int16 data type 1-21
- Int32 data type 1-21
- interactive debugging shell 1-12
 - command line 1-40

J

- Java 1-18
 - documentation 1-38
 - Rhino written in 1-14
- java.io package 1-27
- JavaScript 1-4
 - documentation 1-38
 - language issues 1-17
 - misconceptions 1-17
 - overview 1-17
 - Rhino interpreter 1-14

L

- large model 2-24
- len property
 - Memory object 2-26
- linker command file 1-9
- little endian 2-15, 2-24
- LiveConnect 1-27
- load() method 1-13, 1-24
 - Program object 2-22

- loadArch() method 1-25
- loadPlatform() method 1-11, 1-25
- long integer 1-21
- looping 1-5
- loosely-typed language 1-17

M

- MCBSP properties
 - 'C54x 2-37, 2-38
 - 'C55x 2-44, 2-45
 - 'C6000 2-57
- Memory object 2-25
- merging configurations 1-16
- methods 1-18
- Microsoft Windows
 - configuration methods 1-2
- migration 1-4, 1-15
- minDataUnitSize property
 - Cpu object 2-17
- minProgUnitSize property
 - Cpu object 2-17
- modularization 1-5, 1-30
- Module object 2-28
 - CDB properties 1-20
- module property
 - Instance object 2-32
- module() method
 - Program object 2-22
- modules() method
 - Program object 2-23
- multiple boards 1-8
 - creating objects 2-5
- multiple CPUs 1-8
 - creating objects 2-9
- multiple programs 1-8
 - creating objects 2-13

N

- name property
 - Board object 2-11
 - Config object 2-7
 - Cpu object 2-16
 - Extern object 2-27
 - Memory object 2-26
 - Module object 2-30
 - Program object 2-24
- names
 - namespace 1-22
 - TCF file 1-29
 - TCI file 1-29
 - variables 1-17

- namespace 1-22
 - get() Method 2-21
- naming conventions
 - files 1-29, 1-36
 - properties 1-20
- near model 2-24
- number of CPU 2-16
- Numeric data type 1-21

O

- object
 - as return value or parameter 1-19
 - hierarchy 1-7
 - model initialization 1-36
- object-orientation 1-18
- operation modes 1-12, 1-40
- order dependencies 1-15
- order of objects in array 1-20

P

- Parameter RAM Table properties
 - 'C6000 2-54
- path 1-23
 - adding directory to 1-39
 - separators 1-24
- PATH variable
 - running cdbcmp 1-41
 - running tconf 1-39
- platform file 1-8
- platform specification 1-30
- platform-dependent scripts 1-30
- platform-independent scripts 1-30
- PLL properties
 - 'C54x 2-39
 - 'C55x 2-46
- pllIndex property
 - Board object 2-11
- pointers 1-17
- portable scripts 1-30
- porting 1-4, 1-15
- print() method 1-13, 1-27
 - Rhino GUI 1-14
- Program object 2-18
 - defined by startup script 1-35
 - initializing 1-36
- program() method
 - Cpu object 2-14
- programs() method
 - Cpu object 2-15
- project
 - adding files to 1-9

properties 1-18
 naming conventions 1-20
 of Modules and Instances 1-20
 PWR properties
 'C55x 2-46

Q

quit command 1-13

R

Reference data type 1-21
 references to objects 1-19
 references() method
 Instance object 2-32
 reserved keywords 1-13
 revision number of CPU 2-16
 Rhino 1-14, 1-40
 rootDir variable 1-23
 RTC properties
 'C55x 2-47
 RTCRES properties
 'C55x 2-48
 running a script 1-39

S

save() method
 Program object 2-23
 script
 creating 1-11
 file 1-8
 generating from CDB files 1-41
 running 1-39
 scripting languages 1-7
 scriptName variable 1-23
 scripts
 portable 1-30
 search path 1-39
 seed file 1-8
 small model 2-24
 source files 1-9
 space property
 Memory object 2-26
 speed of CPU 2-11
 startup file 1-8
 startup script 1-34
 static objects 1-2
 stderr location 1-28
 stdout location 1-27
 String data type 1-21

T

Target Content Object Model (TCOM) 1-7, 1-18
 class containers 1-18
 diagram 1-7
 quick reference 2-2
 TCF file 1-8
 creating from CDB 1-15
 creating from scratch 1-11
 generating from CDB file 1-41
 naming 1-29, 1-36
 TCI file 1-8
 loading 1-24
 naming 1-29
 tconf utility 1-39
 exit status 1-28
 operation modes 1-12
 tconfini.tcf file 1-8, 1-34
 tconflocal.tci file 1-8, 1-35
 TCP file 1-8
 template file
 differences from 1-15, 1-41
 template files 1-8
 testing 1-12
 text-based configuration 1-2
 TextConf 1-3
 advantages 1-4
 for new applications 1-10
 migrating existing applications 1-15
 throw keyword 1-29
 throwing exceptions 1-28
 TIMER properties
 'C54x 2-39
 'C55x 2-48
 'C6000 2-58, 2-59, 2-60
 tiRoot variable 1-23
 TMS320C54x properties for CSL 2-35
 TMS320C55x properties for CSL 2-40
 TMS320C6000 properties for CSL 2-49
 true/false values 1-20
 try keyword 1-29

U

UNIX
 configuration methods 1-2
 USBRES properties
 'C55x 2-49
 utilities 1-9
 cdbcmp 1-41
 gconfgen 1-41
 tconf 1-39
 utils.findSeed() method 1-26
 utils.getProgObjs() method 1-8, 1-26

utils.importFile() method 1-25
utils.loadArch() method 1-25
utils.loadPlatform() method 1-25
utils.tcf file 1-35
 methods provided 1-25, 1-26

V

variable names 1-17
 environment array 1-22
variable types 1-17, 1-20

W

warn() method 2-7

warnings 1-27
 enabling 1-39
WDTIMER properties
 'C54x 2-40
 'C55x 2-49
Windows
 configuration methods 1-2
word size 2-17
writing scripts 1-11

X

XBUS properties
 'C6000 2-61