

XDC Consumer User's Guide

Literature Number: SPRUEX4
July 2007



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 the 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.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products

Amplifiers	amplifier.ti.com
Data Converters	dataconverter.ti.com
DSP	dsp.ti.com
Interface	interface.ti.com
Logic	logic.ti.com
Power Mgmt	power.ti.com
Microcontrollers	microcontroller.ti.com
Low Power Wireless	www.ti.com/lpw

Applications

Audio	www.ti.com/audio
Automotive	www.ti.com/automotive
Broadband	www.ti.com/broadband
Digital Control	www.ti.com/digitalcontrol
Military	www.ti.com/military
Optical Networking	www.ti.com/opticalnetwork
Security	www.ti.com/security
Telephony	www.ti.com/telephony
Video & Imaging	www.ti.com/video
Wireless	www.ti.com/wireless

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

Read This First

About This Manual

The eXpress DSP Components (XDC, pronounced "ex-dee-see") is a standard for providing reusable software components that are optimized for real-time embedded systems.

This document is a user's guide for consumers of XDC-based packages, including XDC itself.

How to Use this Manual

Chapter 1 describes XDC and provides a general introduction.

Chapter 2 describes the steps to creating and building applications that use XDC in more detail.

Chapter 3 describes the tools provided with XDC that apply to XDC consumers.

Chapter 4 describes the XDC runtime packages that apply to XDC consumers.

Chapter 5 describes the XDC configuration syntax.

Appendix A lists compiler options for various targets.

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.

Here is a sample program listing:

```
#include <xdc/runtime/System.h>

int main(){
    System_printf("Hello World!\n");
    return (0);
}
```

- 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.

Related Documentation From Texas Instruments

- *XDC Release Notes* ([XDC_INSTALL_DIR/release_notes.html](#)). Includes information about software version, upgrades and compatibility, host and target device support, validation, and known issues.
- *XDC Getting Started Guide* ([XDC_INSTALL_DIR/doc/XDC_Getting_Started_Guide.pdf](#)). Includes steps for installing and validating the installation. Provides a quick introduction to XDC using a "hello world" application.
- "CDOC" reference documentation. Contains full reference information about all installed packages and their modules, APIs, XDC configuration, data structures, etc. See Section 1.6.1 for how to use this online system.
- *XDC Producer User's Guide* (SPRUEX5). Provides information about creating XDC packages.
- *XDC Documentation List* ([XDC_INSTALL_DIR/doc/index.html](#)). Provides links to several other XDC documents.

Related Documentation

You can use the following books to supplement this reference guide:

The C Programming Language (second edition), by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice-Hall, Englewood Cliffs, New Jersey, 1988

Programming in C, Kochan, Steve G., Hayden Book Company

Programming Embedded Systems in C and C++, by Michael Barr, Andy Oram (Editor), published by O'Reilly & Associates; ISBN: 1565923545, February 1999

Real-Time Systems, by Jane W. S. Liu, published by Prentice Hall; ISBN: 013099651, June 2000

Principles of Concurrent and Distributed Programming (Prentice Hall International Series in Computer Science), by M. Ben-Ari, published by Prentice Hall; ISBN: 013711821X, May 1990

American National Standard for Information Systems-Programming Language C X3.159-1989, American National Standards Institute (ANSI standard for C); (out of print)

Trademarks

The Texas Instruments logo and Texas Instruments are registered trademarks of Texas Instruments. Trademarks of Texas Instruments include: TI, Code Composer, Code Composer Studio, DSP/BIOS, SPOX, TMS320, TMS320C54x, TMS320C55x, TMS320C62x, TMS320C64x, TMS320C67x, TMS320C28x, TMS320C5000, TMS320C6000 and TMS320C2000.

Windows is a registered trademark of Microsoft Corporation.

Linux is a registered trademark of Linus Torvalds.

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

Contents

1	About XDC	1-1
	<i>This chapter provides an overview of eXpress Dsp Components (XDC).</i>	
1.1	What is XDC?	1-2
1.2	XDC Terminology	1-3
1.3	How to Use XDC: The Quick Tour	1-5
1.4	The Package Path	1-6
1.5	Setting Your PATH Definition	1-8
1.6	Getting More Information	1-9
2	Basic Steps to Using XDC	2-1
	<i>This chapter provides a more detailed look at the steps to using XDC.</i>	
2.1	Overview of the Development Steps	2-2
2.2	Configuring the Application	2-2
2.3	Writing C Code	2-7
2.4	Processing the Configuration	2-14
2.5	Compiling and Linking	2-19
3	XDC Tools	3-1
	<i>This chapter provides command-line syntax and examples for the tools provided with XDC.</i>	
3.1	Overview of the Tools	3-2
3.2	The cdoc Tool	3-3
3.3	The configuro Tool	3-5
3.4	The path Tool	3-6
3.5	The repoman Tool	3-8
4	XDC Runtime Modules	4-1
	<i>This chapter provides introductory information about the XDC runtime modules.</i>	
4.1	Overview of the Runtime Modules	4-2
4.2	XDC Boot Sequence and Control Points	4-2
4.3	System Module	4-6
4.4	Memory Segments and Sections	4-10
4.5	Memory Allocation and Heaps	4-13
4.6	Timestamp Module	4-17
4.7	Gate Interface and Implementations	4-19
4.8	Diagnostics and Logs	4-21
4.9	Types Module	4-33

5 XDC Configuration 5-1
This chapter provides more information about XDC configuration syntax.

5.1 More About the XDC Script Language 5-2

5.2 More About the xs Command 5-5

5.3 JavaScript Language 5-13

5.4 XDC Script Methods 5-15

A Target Compiler Options A-1
This appendix lists the compiler options used by default for the various targets you can select.

About XDC

This chapter provides an overview of eXpress Dsp Components (XDC).

Topic	Page
1.1 What is XDC?	1-2
1.2 XDC Terminology	1-3
1.3 How to Use XDC: The Quick Tour.....	1-5
1.4 The Package Path.....	1-6
1.5 Setting Your PATH Definition	1-8
1.6 Getting More Information	1-9

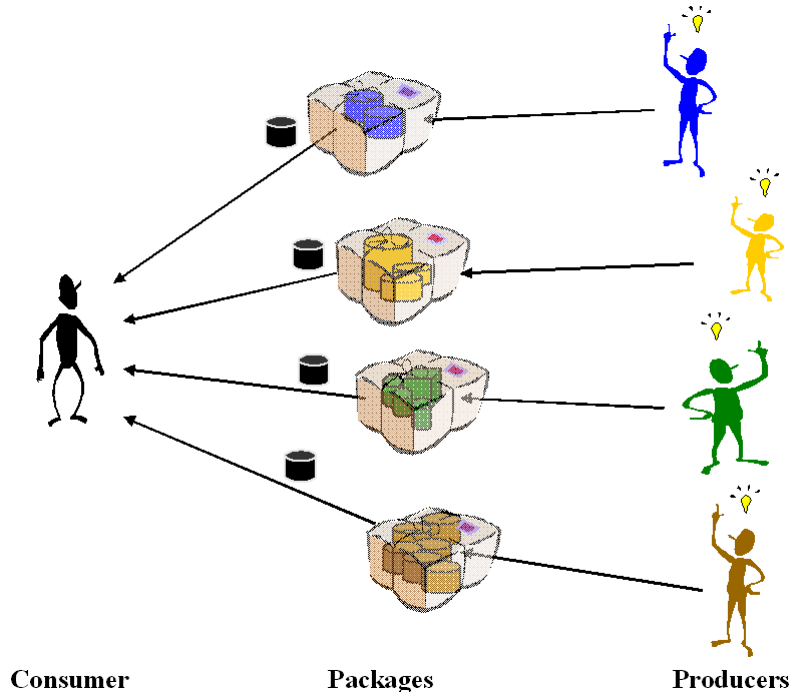
1.1 What is XDC?

The eXpress DSP Components (XDC, pronounced "ex-dee-see") is a standard for providing reusable software components, called "packages", that are optimized for real-time embedded systems.

XDC includes tools and standards for API development, static configuration, and packaging. XDC components have hardware-neutral formal interfaces, are configurable offline to optimize memory and performance, and support custom automation in the development environment via a scripting language.

The main benefit of XDC is that it standardizes the delivery of target content and makes it easier for target content to be included in applications.

The users of XDC are divided into developers we call "consumers" and "producers." Consumers integrate target content packages—DSP algorithms, device drivers, TCP/IP stacks, real-time OSes, and so on—into their own applications. Producers create the packages used by consumers.



An XDC "package" is a named collection of files that form a unit of versioning, update, and delivery from a producer to a consumer. Each package is embodied as a specially-named directory (and its contents) within a file system. Packages are the focal point for managing content throughout its life-cycle. All packages are built, tested, released, and deployed as a unit.

An XDC "repository" is simply a directory that contains packages. The dots in the name of a package, interface, or module refer to its location within the repository. For example the `ti.sysbios.knl.Task` module would be located at `ti/sysbios/knl` with respect to a repository directory named in the XDC "package path". The package path is simply a list of repositories containing packages installed by the user.

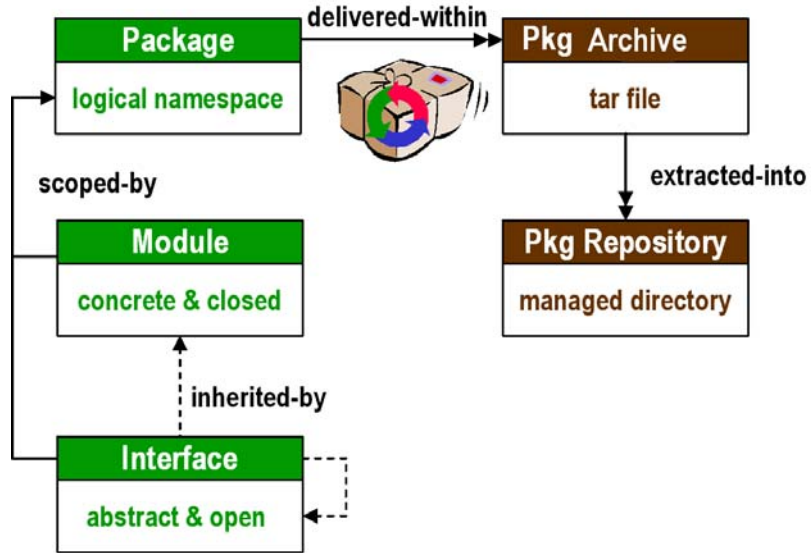
This document is intended for use by consumers. However, producers must also be familiar with the concepts in this document in order to move beyond it and understand how to create packages. (Also, producers often also function as consumers, for example, to use a third-party RTOS when creating a TCP/IP stack.) After reading this document, producers should move on to the *XDC Producer User's Guide* (SPRUEX5).

1.2 XDC Terminology

XDC uses a number of terms that you will need to be familiar with:

- **Packages.** These serve as general-purpose containers for modules and interfaces as well as other software artifacts. Packages are the focal point for managing content throughout its life-cycle. All packages are built, tested, released, and deployed as a unit.
- **Modules.** These encapsulate a related set of types and functions. They have both an external specification and a concrete internal implementation. A module optionally manages a single instance type, which is analogous to a C++ class.

- **Interfaces.** These are effectively "abstract modules". They have a specification without an implementation. Other modules and interfaces can inherit the specification. An interface defines a collection of related types, constants, variables, and functions. A single C (and/or asm) header file defines the interface.



- **Repository.** A directory in which one or more packages are installed. Repositories can contain only one version of a package. So, side-by-side installations of two different versions of a package require two repositories. The user has complete control over the number and names of repositories.
- **Package Path.** An ordered sequence of repositories that are searched when locating a package's file. Like the PATH environment variable used to locate commands, the package path is set by the user and allows source code to reference package files without using absolute paths. Simple adjustments of the package path can be used to quickly switch between different versions of one or more packages.
- **Target content.** Software bound into an application program executing on a particular hardware platform.
- **Meta content.** Host-based content that plays an active role in the design-time configuration as well as the run-time analysis of target programs.
- **Client applications.** The application that consumes packages and calls functions in interfaces to perform application actions.

1.3 How to Use XDC: The Quick Tour

Using an XDC package requires the introduction of a configuration step to the conventional compile/link cycle. You can easily add this configuration step to existing build flows using XDC's "configuro" command-line tool. For more about integrating configuration into your build flow, see the *XDC Getting Started Guide* (XDC_INSTALL_DIR/doc/XDC_Getting_Started_Guide.pdf).

Follow the steps in the first chapter of the *XDC Getting Started Guide* to install XDC and test the installation.

Follow the steps in the second chapter of the *XDC Getting Started Guide* to get a quick introduction to the process of creating applications that use XDC-based content. The main steps in that process are:

- 1) Configuring the application. See Section 2.2 for more information.
- 2) Writing C code. See Section 2.3 for more information.
- 3) Processing the configuration for your target and platform. See Section 2.4 for more information.
- 4) Compiling and linking the application. See Section 2.5 for more information.

The next chapter of this manual (*XDC Consumer User's Guide*) goes into more detail than the *XDC Getting Started Guide* about each of these development steps.

1.4 The Package Path

XDC packages often need to reference files contained in other packages. Files contained in a package are always found along the "Package Path," which is simply a semicolon-separated list of package "repositories". A repository is simply a directory that contains installed packages.

The Package Path is always defined as follows:

```
XDCPATH;XDC_INSTALL_DIR/packages;^
```

In this definition, XDCPATH is a semicolon-separated set of repositories specified by the user. The XDC_INSTALL_DIR/packages is the repository of packages included with the XDC tools. The "^" denotes the current package's repository (if there is a current package). For most tools, the current package is defined as the package named by a package.xdc file in the current working directory. If no such file exists, then any repository name containing the "^" character expands to an empty string "".

Because XDC_INSTALL_DIR/packages is always part of the package path, the XDC tools can automatically find XDC's own packages. However, if you have other software that uses XDC packaging (such as DSP/BIOS), you need to set the XDCPATH environment variable to reference any repositories containing packages you require.

All source files, whether they are part of a package or your application, should always reference files contained in a package by prefixing the file name with the package's directory name. For example, to include the Task.h header in the ti.sysbios.knl package, your source files should `#include <ti/sysbios/knl/Task.h>`.

When compiling your sources, you should add a `-I` option for each repository named in the Package Path in the same order that they appear in the Package Path. Following this pattern allows you to easily add/remove/override packages without having to update your build system. You can simply install (or remove) a package into one of the repositories, add (or remove) the reference from your sources, and rebuild.

Since virtually all build tools support an option similar to `-I`, the pattern works for linker command files, assembly language files, XDC scripts, and makefiles.

1.4.1 Managing the XDCPATH Portion of the Package Path

In the Package Path, you control the XDCPATH portion. This prefix is a string of ';' separated directories that contain packages.

XDCPATH may be specified as an environment variable or on the command line of any tool that uses the Package Path. See the documentation of the particular command-line tool to find the appropriate option.

IMPORTANT NOTE: All directories referenced in XDCPATH, whether in an environment variable or a command line, must use **forward slashes (/)** as separators instead of the Windows-standard backslash (\). For example, use "c:/os;c:/drivers;c:/framework".

Relative paths in XDCPATH reference directories relative to the package being built rather than the directory from which the command was invoked. Thus, a relative path refers to a different repository for each package used.

Note: It is usually a mistake to use a relative path in the XDCPATH prefix.

It is possible, however, to use the '^' character in the XDCPATH definition to refer to the "current package's" repository. So, if you have a repository that is always in a fixed location relative to all of your package's repositories, it is possible to create a single XDCPATH setting that works for all your packages and does not include any absolute paths.

Suppose, for example, that your build system places all prerequisite packages in an "imports" repository prior to building the packages in a "src" repository and the imports and src repositories are sibling directories in the file system. The following XDCPATH setting is sufficient to build all packages in the src repository.

```
set XDCPATH=^/../../imports
```

Multiple versions of the same package can appear along the Package Path. The Package Path can name multiple repositories that can contain a package directory with the same name. When searching for a package, the first repository that contains a directory matching the package's name is used. Thus, even if two packages with the same name appear in the package path, only one will ever be found-the first one in the order specified in the package path.

Thus, it is possible to quickly update selected packages by adding an "updates" repository at the front of XDCPATH. If the updates are successful, you can "permanently" remove the old versions; otherwise you can quickly revert to the previous versions by simply removing the updates repository from XDCPATH.

Here is a Linux command example for setting XDCPATH:

```
export XDCPATH="/opt/embedded/os;/opt/embedded/drivers"
```

If you are using Microsoft Windows, see Section 1.5 for how to set environment variables.

XDC configuration scripts mention packages by name, not by location. This makes scripts portable to new directory locations.

If you do not want to specify a directory using the XDCPATH environment variable, you can use the `--xdcpath` option on the command line for all the `xs` command-line tools.

Alternately, you can place the `--xdcpath` command-line option in a text file, say `mysettings.txt`, and specify the file on the command line of `xs` commands. This is useful in makefiles, which can have a dependency on the text file. The `xs` command line would reference the file as shown in this example:

```
xs @mysettings.txt
```

1.5 Setting Your PATH Definition

XDC does not use the `PATH` environment variable directly. However, your operating system or shell does—`PATH` defines where the operating system looks for command line programs. You may want to include the top-level XDC directory in your `PATH` to simplify invoking the XDC command-line tools.

Setting the `PATH` definition makes it easy to run XDC's `xs` command, which is used to run a number of tools. If you do not want to add the XDC directory to your `PATH`, you can use one of the following alternate ways of running the `xs` command:

- ☐ You can invoke `xs` from the command line by using the full path `<xdc_install_dir>/xs`.
- ☐ You can make an alias to the `xs` or `xs.exe` executable.

On Microsoft Windows, the `PATH` is managed by Windows itself (in Windows 2000 and Windows XP) rather than by the `autoexec.bat` or `autoexec.nt` files used in previous versions of Windows.

You can create aliases using the `doskey` command from the Windows `cmd.exe` shell as shown in this example:

```
doskey xdc=<xdc_install_dir>/xs.exe $*
```


To change the Windows PATH (and other Windows environment variables), follow these steps:

- 1) Right-click My Computer and choose **Properties**.
- 2) In the System Properties window, choose the **Advanced** tab.
- 3) In the Advanced tab, click the **Environment Variables** button.
- 4) Highlight the Path variable in the "Systems variables" area and click **Edit**.
- 5) Add the directory in which you installed XDC to the end of the current Path definition. Use a semicolon to separate the locations in the list. For example, you might add the following to your existing Path definition:

```
;c:\xdc_2_95
```

- 6) Click **OK** in the Environment Variables window and **OK** in the System Properties window.
- 7) Reopen any Command Prompt or MS-DOS windows you want to use with XDC.

1.6 Getting More Information

You can read the following additional documents to learn more about XDC:

- ☐ *XDC Release Notes* (XDC_INSTALL_DIR/release_notes.html). Includes information about software version, upgrades and compatibility, host and target device support, validation, and known issues.
- ☐ *XDC Getting Started Guide* (XDC_INSTALL_DIR/doc/XDC_Getting_Started_Guide.pdf). Includes steps for installing and validating the installation. Provides a quick introduction to XDC using a "hello world" application.
- ☐ "CDOC" reference documentation. Contains full reference information about all installed packages and their modules, APIs, XDC configuration, data structures, etc. See Section 1.6.1 for how to use this online system.
- ☐ *XDC Producer User's Guide* (SPRUEX5). Provides information about creating XDC packages.
- ☐ *XDC Documentation List* (XDC_INSTALL_DIR/doc/index.html). Provides links to several other XDC documents.

1.6.1 Using the CDOC Reference Help System

To open the CDOC reference help system, you can choose CDOC from the XDC group in the All Programs section of the Windows **Start** menu.

Alternatively, issue the following command from the command prompt—for example by opening the Microsoft Windows Start menu and choosing **Run**.

```
xs xdc.tools.cdoc.sg
```

You see the "cdoc" window with a tree view of the XDC packages available to you. All of these packages provide documentation that can be displayed by the CDOC viewer.

Click "+" next to a repository to expand its list of packages. Click "+" next to a package name to see the list of modules it provides. You can further expand the tree to see a list of the functions provided by a module. Double-click on a package or module to see its reference information.

Notice the icons in the upper-right corner of the window. The following icons are useful:



View C interface view of documentation.



View XDC interface view of documentation.



Close all page tabs.

For each topic you view, there is a tab across the top of the page area. You can use these to quickly return to other pages you have viewed. You can also use the arrows next to "Views" to move backward and forward in your history of page views.

To close a page and remove its tab, click the X on the tab.

Basic Steps to Using XDC

This chapter provides a more detailed look at the steps to using XDC.

Topic	Page
2.1 Overview of the Development Steps	2-2
2.2 Configuring the Application	2-2
2.3 Writing C Code	2-7
2.4 Processing the Configuration	2-14
2.5 Compiling and Linking	2-19

2.1 Overview of the Development Steps

To expand on the quick tour provided in the *XDC Getting Started Guide* this chapter expands on each of the steps in the development process.

Creating applications that use XDC-based software packages follows a development cycle that builds on traditional C programming techniques. XDC packages standardize the delivery of TI and 3rd party software, easing the integration of such software.

Using an XDC package adds a configuration step to the conventional compile/link cycle. The configuration processing is easily added to existing build flows. XDC provides a number of tools that simplify the task of integrating and using XDC-based packages.

The steps for creating an application that uses XDC-based content are as follows:

- 1) Configuring the application. See Section 2.2.
- 2) Writing C code. See Section 2.3.
- 3) Processing the configuration for your target and platform. See Section 2.4.
- 4) Compiling and linking the application. See Section 2.5.

As with the standard development cycle, the focus is on writing C code. The other steps are simple. XDC provides tools to support the additional configuration steps.

2.2 Configuring the Application

An application that uses XDC packages needs a configuration file. This is a file containing script statements that statically configure how the application uses XDC packages. Your C program code can do additional dynamic configuration, but the XDC configuration defines the starting point.

The XDC configuration serves the following purposes:

- ☐ Specifies the packages to use and the static objects to create.
- ☐ Performs integrity checks between specified and dependent packages.
- ☐ Sets options for modules and objects to change their default behavior.

The configuration file is typically located in the same directory as your main C code file. It can have any name. The name should not contain spaces, and lowercase names are recommended.

As you develop applications, the configuration step is one you may revisit as you add functionality to your C code or tune the functionality of the packages used by your application for optimal performance.

Use a text editor to create the configuration file, and save the file with an extension of `.cfg`. For example, your file might be called `mycfg.cfg`. When you save the file, be sure to save it as plain text (not as a Microsoft Word or other word processor file).

XDC configuration files use JavaScript (EcmaScript) syntax. For more information about configuration syntax, see Chapter 5. For details about the packages, modules, objects, and properties you can configure, see the CDOC reference help system (see Section 1.6.1).

The following subsections provide an overview of the basic statement types you will use for most configurations.

2.2.1 Using a Module

The `xdc.useModule` command enables an application to use a particular module. For example, this command enables use of the `xdc.runtime.System` module:

```
var System = xdc.useModule("xdc.runtime.System");
```

(The package path allows XDC to find packages, so all you specify is the name of the package within a repository referenced in the package path.)

The generalized syntax is:

```
var Mod = xdc.useModule("pkg.name.Mod");
```

Your C code file will also need to identify modules referenced in C code by including their header files. For example:

```
#include <pkg/name/Mod.h>
```

Notice the similarity in referencing modules between JavaScript and C. This is made possible by adding a `-I` option for each repository in the package path to the C compiler's command line. A similar technique can be used with assembly language files, linker command files, and even GNU makefiles.

2.2.2 Setting a Module Parameter Value

Your XDC configuration scripts can set properties for modules and for instances you have created for a module.

The generalized syntax for setting a parameter of a module in an XDC configuration script is:

```
Mod.cfgParam1 = value1;
```

This example enables the Error module, which handles run-time errors. It sets the Policy parameter to return errors to the calling function.

```
var Err = xdc.useModule("xdc.runtime.Error");  
Err.Policy = Err.UNWIND;
```

Warning: We normally would use "Error" as the name of the variable. But since JavaScript already has a Error class defined, creating an Error variable here would overwrite JavaScript's built-in Error class and prevent other parts of the configuration from throwing Error exceptions.

2.2.3 Defining Static Constants

Program.global is a special object that allows you to define global variables and constants in a configuration script and access these values from C/C++.

Suppose, for example, that your configuration script contains the following line:

```
Program.global.MAXCHANNELS = 2;
```

You can access the MAXCHANNELS parameter from your C/C++ files as follows:

```
#include <xdc/cfg/global.h>  
  
void main()  
{  
    printf("MAXCHANNELS = %d\n", MAXCHANNELS);  
}
```

See the xdc.cfg.Program.global topic in the CDOC reference help system (see Section 1.6.1) for information about how to compile sources that use xdc/cfg/global.h.

2.2.4 Static Object Instance Creation

Each module can manage at most one instance type. For example, suppose the Task module of the ti.sysbios.knl package allows you to create Task objects. You can create objects statically (in an XDC configuration file) or dynamically (in C code).

The syntax to create such an object statically is as follows.

```
var Task = xdc.useModule("ti.sysbios.knl.Task");

Program.global.tsk0 = Task.create("&twoArgsFxn");
```

The tsk0 name is the name of the object created within the Program.global namespace.

The "&twoArgsFxn" is a module-specific argument to the create() method. In this case, it specifies an externally defined C function named "twoArgsFxn" that the Task thread should run.

To use tsk0 from your C/C++ code, follow the same pattern used in the previous section to access MAXCHANNELS. For example:

```
#include <xdc/cfg/global.h>           /* declares tsk0 */
#include <ti/sysbios/knl/Task.h>      /* declares Task functions */
...
Task_setPri(tsk0);
```

These additional examples create instances for other modules:

```
Program.global.heap0 = HeapBuf.create(heapBufParams);
Program.global.swi0 = Swi.create('&swi0Fxn', swiParams);
Program.global.clk0 = Clock.create("&clk0Fxn", 5, clkParams);
```

In general, the create method of any module has zero or more required parameters, and the last parameter is a "structure" of instance parameters that is optionally passed to the create method; if it is not passed (as in the Task example above), module-specific defaults are used. Each module's online reference documentation describes these defaults; see Section 1.6.1 for more information about how to view a module's reference documentation.

2.2.5 Setting an Instance Parameter Value

You can set the parameters of instances you create. Typically, this is done by setting parameters in a Params structure that is specific to the module. Then, you pass the Params structure to the create() method for the module (sometimes along with some other arguments). The instance created has those parameters.

For example, the following statements enable the xdc.runtime.Memory and ti.sysbios.heaps.HeapBuf module. Then, a variable of type HeapBuf.Params is declared. This is the Params structure defined by the ti.sysbios.heaps.HeapBuf module. Statements fill in the values of the blockSize, numBlocks, and align fields of the structure. Then, a HeapBuf instance is created using those parameters and the heap is assigned to be the default heap instance for the Memory module.

```
var Memory = xdc.useModule('xdc.runtime.Memory');
var HeapBuf = xdc.useModule('ti.sysbios.heaps.HeapBuf');

/* Create a heap using ti.bios.HeapBuf */
var heapBufParams = new HeapBuf.Params();
heapBufParams.blockSize = 128;
heapBufParams.numBlocks = 100;
heapBufParams.align = 8;
Program.global.heap0 = HeapBuf.create(heapBufParams);

/* Use heap0 as the default heap */
Memory.defaultHeapInstance = Program.global.heap0;
```

2.2.6 Finding Configuration Options

Configuration parameters are described in a module's CDOC online help. Look for the “config” keyword in CDOC to find the properties you can set for the module and instances of the object it describes.

The section with the heading “Module-wide configuration parameters” lists parameters you can set for the entire module. The heading “Per-instance configuration parameters” lists parameters you can set for an instance or in the create() method for the module.

2.2.7 Property Types

The CDOC online help lists the type of value expected for each configuration parameter. Since JavaScript is an untyped language, variables can refer to different types at different points in a script and most types are automatically converted as needed. For example:

```
var tmp = 1;           /* tmp is the numeric variable 1 */
tmp = "tmp = " + tmp; /* now tmp is the string "tmp = 1" */
```

However, all properties of XDC objects are declared with a fixed type and all assignments to these properties are checked at runtime. If an inappropriate assignment is made, a runtime exception is thrown and the script terminates at the point that the assignment is made. This runtime type checking helps catch programmer errors. For example, the `defaultHeapSize` property of the `xdc.runtime.Memory` module is declared to be an "int". The following script terminates on the second assignment to `defaultHeapSize` with a fatal error:

```
var Memory = xdc.useModule("xdc.runtime.Memory");
Memory.defaultHeapSize = 64 * 1024; /* ok*/
Memory.defaultHeapSize = "64K";    /* ERROR! must be an int */
```

Similarly, do not set a Boolean value to the quoted string "true" or "false".

2.3 Writing C Code

XDC establishes C coding conventions that make it easier to integrate XDC-based content. After you learn these conventions, you will be able to integrate any modules supplied within XDC packages.

2.3.1 Using Modules

A module encapsulates a related set of types and functions. A module is analogous to a C++ class and can optionally manage a single instance type.

To use a module in C code, you must `#include` its header file. For example:

```
#include <xdc/runtime/Memory.h>
```

Suppose you want to use the `printf` method supplied by the `xdc.runtime.System` module. As we saw in Section 2.2.1, first you use this XDC configuration statement in your application's configuration script:

```
var System = xdc.useModule("xdc.runtime.System");
```

Then you should add the following statements to your C code:

```
#include <xdc/runtime/System.h>
:
System_printf("Hello World!");
```

Notice that the package name in the `#include` statement was not used in the name of System's `printf` method. In fact, all XDC modules provide two names for each module identifier: a "short" name without the package name (e.g., `System_printf`) and a "long" name that includes the package name (e.g., `xdc_runtime_System_printf`). So, the code above could also have been written as follows.

```
#include <xdc/runtime/System.h>
:
xdc_runtime_System_printf("Hello World!");
```

Short names are much easier to read but may conflict with another method's short name. For example, the `printf` method of a hypothetical `ti.wlan.System` module. Since conflicts are rare, we use the short names whenever possible.

In the event of a conflict, you can disable the short names for a specific module by defining the symbol `<modName>__nolocalnames` prior to including the module's header. For example:

```
#define xdc_runtime_System__nolocalnames
#include <xdc/runtime/System.h>
#include <ti/wlan/System.h>
:
System_printf("Hello world"); /* call ti_wlan_System_printf() */
xdc_runtime_System_printf("Hello World!");
```

The same technique is used to avoid conflicts with the base types defined in `xdc/std.h`. Suppose, for example, your code defines a Boolean type named `Bool` that conflicts with the same named type in `xdc/std.h`. Simply define the symbol `xdc__nolocalnames` and all identifiers from `xdc/std.h` will have the "xdc_" prefix. For example:

```
#define xdc__nolocalnames
#include <xdc/std.h>

typedef unsigned char Bool; /* define a local boolean type */
:
xdc_Bool flag1; /* flag1 is an XDC boolean variable */
Bool flag2; /* flag2 is a local boolean variable */
```

As rule, all identifiers in a module's interface have the prefix `<pkg>_<Mod>_`. Shorter names are defined for all identifiers with just the `<Mod>_` prefix. In the event that a module's short names conflict with other identifiers, you can `#define <pkg>_<Mod>__nolocalnames` to disable these short names.

Finally, all identifiers with the prefix `<pkg>_<Mod>_` are defined in the `<pkg/Mod>.h` header file.

2.3.2 Naming Conventions

XDC uses a set of naming conventions and suggests that you and third-party providers follow these same conventions. The benefits of these conventions are as follows:

- 1) Package names can be distinguished from module names. For example, `ti.bios.utils` is a package, but `ti.bios.Utils` is a module in the `ti.bios` package.
- 2) Modifiable entities can be distinguished from constants. For example, `System.DONE` is a constant, but `System.maxPasses` is a module configuration parameter.
- 3) Interfaces can be distinguished from concrete modules. For example, `xdc.IPlatform` is an interface, but `xdc.Platform` is a module.
- 4) Types can be distinguished from variables and functions. For example, `Arg` and `Startup_InitFxn` are types, but `arg` and `Startup_firstFxn` are parameters or variables.

All identifiers follow these naming conventions:

- ☐ Package names are all lowercase.

```
xdc.runtime
```

- ☐ Module names begin with an uppercase letter.

```
xdc.runtime.Memory
```

- ☐ Interface names begin with an uppercase "I".

```
IHeap and IGateProvider
```

- ☐ Type names begin with an uppercase letter. The uppercase applies to the first letter after any prefix (for example, "Handle" in `Task_Handle`).

```
Arg and String          /* types */
xdc_String
```

- ❑ Functions, variables, and parameter names begin with a lowercase letter after any module prefix. Camel case is used where needed to delimit multi-word names.

```
Diags_setMask()      /* function */
Log_system           /* global variable */
```

- ❑ Multiple-word identifiers are separated by uppercase letters, not an underscore.

```
Mod_twoWords         /* multi-word global variable */
```

- ❑ Constants are in all uppercase after the module or package prefix. If multiple words are required, an underscore delimits the words.

```
Diags_INTERNAL       /* constant */
Types_STATIC_POLICY  /* two-word constant name */
```

- ❑ Structure field names and configuration parameters begin with a lowercase letter.

```
attrs.count          /* structure field name */
System_maxAtexitHandlers /* configuration parameter */
```

2.3.3 Runtime Object Instance Creation

Modules that manage instances provide C functions to create and delete the type of object managed by that module.

create() function. The function for dynamically creating an instance is named `<Mod>_create()`. The `<Mod>_create()` function returns a handle to the new instance or `NULL` if the function fails. The handle returned by `<Mod>_create()` is of type `<Mod>_Handle`. The type of the instance itself is `<Mod>_Object`.

```
#include <ti/sysbios/knl/Task.h>
#include <xdc/runtime/System.h>
#include <xdc/runtime/Error.h>
:
Error_Block eb;
Task_Handle task;
:
task = Task_create(NULL, &eb);
if (task == NULL) {
    System_abort("task create failed!\n");
}
```

The first argument to a function that deals with an instance for that module is a `<Mod>_Handle`. (The exception is the `delete()` function, which takes a pointer to a `<Mod>_Handle`.)

```
Task_setPri(task, Task_MAXPRI); /* set priority of task thread */
Task_delete(&task);           /* delete task object, set task to NULL */
```

The `create()` function for a module always take a parameter that is a pointer to a structure of type `<Mod>_Params`. If you set this parameter to `NULL` (as in the previous example), the defaults are used for all the object's parameters.

For all modules with instance creation parameters, you can get the default instance parameter values by calling `<Mod>_Params_init()`. For example:

```
HeapMin_Params heapParams;
HeapMin_Handle heap;

/* initialize all instance parameters to their defaults */
HeapMin_Params_init(&heapParams);

/* set specific parameters */
heapParams.size = 0x10000;

/* create instance */
heap = HeapMin_create(&heapParams, NULL);
```

The `create()` function always takes a last parameter that is a pointer to an `Error_Block`. The `Error_Block` structure is defined by the `xdc.runtime.Error` module. If this parameter is `NULL`, errors will cause the application to terminate (via `xdc_runtime_System_abort()`) without returning to the caller.

delete() function. The function for dynamically deleting an instance is named `<Mod>_delete()`. This function returns `Void`, which means it cannot directly return a failure status. If it is successful, it sets the pointer to a `<Mod>_Handle` that was passed to it to `NULL`. This has the benefit of preventing the application from inadvertently using the handle variable after the instance has been deleted.

The following example shows the `create()` and `delete()` functions for a Semaphore module. In this case, we create a Semaphore instance with default parameters and no error block (so any error will abort the application).

```
Semaphore_Handle sem; /* declare handle to semaphore */

sem = Semaphore_create(NULL, NULL); /* create semaphore */
...
Semaphore_delete(&sem);              /* delete semaphore */
```

The following example illustrates how to create a module (Task) instance with parameters other than the defaults.

```
Task_Handle myTask;      /* declare handle to task */
Task_Params params;      /* declare task creation attributes */

Task_Params_init(&params); /* init task params to defaults */
/* set only attributes of interest */
params.stacksize = 1024 * sizeof (Int);
myTask = Task_create(fxn, &params, NULL); /* create task */
```

2.3.4 XDC Datatypes

XDC defines the following base datatypes for use in C code in xdc/std.h. You should use these types in application code that uses XDC modules or is intended to be portable to multiple targets. Most datatypes have names that are similar to the corresponding C type, but with TitleCase used for differentiation and abbreviation of unsigned and long. The underlying sizes may be different on various targets.

For more information about these types as well as predefined macros that facilitate the creation of portable C modules see the CDOC reference help for the xdc package (see Section 1.6.1).

Table 2–1 XDC Datatypes for C Programs

XDC Type	C99 Type	Comments
Bits16	uint16_t	
Bits32	uint32_t	
Bits8	uint8_t	Supported only for 'C55x+byte mode and for all 'C6000 and TMS470 targets
Bool	unsigned short	1 for true and 0 for false or use TRUE and FALSE
Char	char	
Double	double	
Float	float	
Fxn	int (*)()	function pointer
IArg	intptr_t	
Int	int	
Int16	int_least16_t or int_fast16_t	The target may opt to use either the int_fast or the int_least types for these definitions.

XDC Type	C99 Type	Comments
Int32	int_least32_t or int_fast32_t	
Int8	int_least8_t or int_fast8_t	
LDouble	long double	
LLong	long long	May be supported only as long on some targets
Long	long	
Ptr	void*	data pointer
Short	short	
SizeT	size_t	
String	char*	null-terminated string
UArg	uintptr_t	holds arguments to pass to functions
UChar	unsigned char	
UInt	unsigned int	
UInt16	uint_least16_t or uint_fast16_t	The target may opt to use either the uint_fast or the uint_least types.
UInt32	uint_least32_t or uint_fast32_t	
UInt8	uint_least8_t or uint_fast8_t	
ULLong	unsigned long long	May be supported only as unsigned long on some targets
ULong	unsigned long	
Uns	unsigned int	
UShort	unsigned short	
VaList	va_list	

2.4 Processing the Configuration

In this step, you use the "configuro" tool provided with XDC to process your application's configuration file. Processing the file generates a compiler.opt file to be used when you compile the application and a linker.cmd file to be used when you link the application.

Before a configuration script can be processed, you need to specify the "target" compiler and the hardware "platform" required to run your application. In general terms:

- ❑ A **target** identifies a specific compiler and an ISA and runtime model supported by the compiler. For example, the TI 'C6000 compiler for the 'C64+ ISA running in big-endian mode.
- ❑ A **platform** identifies the hardware execution environment as seen by your application. For example, a DM6446 EVM with 64 MB of DDR2 external memory.

XDC uses simple string names to identify targets and platforms.

2.4.1 Choosing a Target

The following target strings are recognized by XDC. See Section A.1 for a list showing the compiler options generated for various targets.


To see the latest list of targets, open the CDOC online documentation (Section 1.6.1). Select the  (XDC) view. You can expand the ti.targets and gnu.targets lists.

Table 2–2 TI Targets

ti.targets.TMS470	ti.targets.C28	ti.targets.C62
ti.targets.TMS470_big_endian	ti.targets.C28_large	ti.targets.C62_big_endian
ti.targets.MSP430	ti.targets.C55	ti.targets.C64
ti.targets.Arm7	ti.targets.C55_huge	ti.targets.C64_big_endian
ti.targets.Arm7_big_endian	ti.targets.C55_large	ti.targets.C64P
ti.targets.Arm9		ti.targets.C64P_big_endian
ti.targets.Arm9t		ti.targets.C67
		ti.targets.C67_big_endian
		ti.targets.C67P
		ti.targets.C67P_big_endian

Table 2–3 Non-TI Targets

<code>gnu.targets.Linux86</code>	Native target for Linux on PC
<code>gnu.targets.Mingw</code>	Native target for Windows using Mingw compiler
<code>gnu.targets.MVArm9</code>	Embedded target for Linux on Arm9
<code>gnu.targets.Sparc</code>	Native target for Solaris on Sparc workstations
<code>gnu.targets.UCArm9</code>	gcc/uClibc target for Linux on Arm
<code>microsoft.targets.Net32</code>	Microsoft .Net 32-bit native target
<code>microsoft.targets.VC98</code>	Windows 32-bit using Visual C/C++ 6.x compiler.
<code>microsoft.targets.Win32</code>	Windows 32-bit using Visual C/C++ 6,7,8 compilers.

If your target is not listed, it is possible to create your own target, but you may need to contact technical support.

2.4.2 Choosing a Platform

The "platform" is a string that describes the specific board. It specifies a particular device and memory map on which an application will run, and is used for linking.

Platforms are provided for many common TI development boards.

The full list of platforms available is visible in the CDOC online documentation (see Section 1.6.1) in the list of packages under `ti.platforms`. Examples include `ti.platforms.sim6xxx`, `ti.platforms.sim64Pxx`, `ti.platforms.dsk5510`, `ti.platforms.dsk6416`, and `ti.platforms.evmDM6437`.

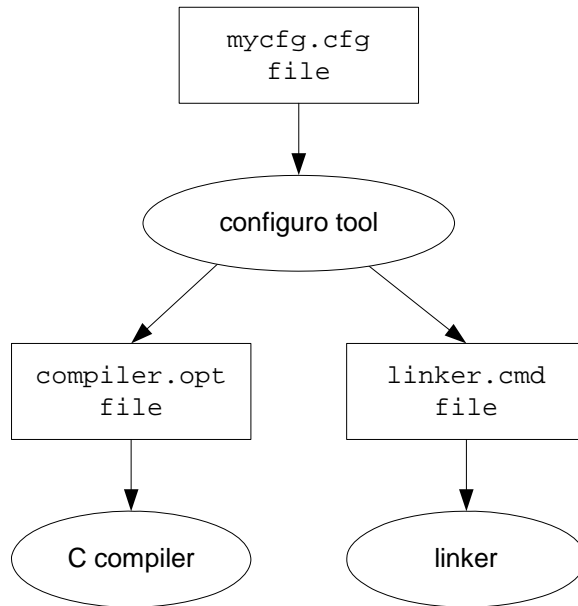
If you are developing for a platform that is not listed in the online documentation, you may be able to use the `ti.platforms.generic` platform. For more information about how to define a platform instance using this platform package, see the CDOC online documentation for the `ti.platforms.generic` package.

2.4.3 Running Configuro

XDC provides a tool called "configuro" that processes XDC configuration files.

Running configuro generates a subdirectory with the same name as your CFG file (for example, `C:/myprojects/hello/mycfg` for the `mycfg.cfg` file). This subdirectory is an XDC "package".

From a consumer perspective, the important thing about this package is that it contains files generated from your configuration to be used when compiling and linking your application. These files include **compiler.opt** and **linker.cmd**, along with header files and libraries.



Generate files from the configuration you created using the following command line:

```
xs xdc.tools.configuro -t <target> -p <platform>  
-c <compiler_location> mycfg.cfg
```

For example:

```
xs xdc.tools.configuro -t ti.targets.C64  
-p ti.platforms.sim6xxx  
-c c:/CCStudio_v3.3/C6000/cgtools mycfg.cfg
```

- ☐ The -t option specifies the target.
- ☐ The -p option specifies the platform.
- ☐ The -c option specifies the location of the compiler you want to use.

You can view further command-line options for the configuro tool with this command:

```
xs xdc.tools.configuro --help
```

When you run `configuro`, you see a message similar to the following:

```
making package.mak (because of package.bld) ...
generating interfaces for package mycfg (because ...)
configuring mycfg.x64 from package/cfg/mycfg_x64.cfg ...
cl64 package/cfg/mycfg_x64.c ...
```

Re-run `configuro` on your CFG file whenever you do any of the following:

- ☐ Change the CFG file
- ☐ Change the target or platform
- ☐ Update a package used by your application

The `configuro` command uses specially generated makefiles to avoid running any unnecessary steps when you make the changes above. So, if you are not sure whether to run `configuro`, go ahead and run it; if nothing needs to be done, `configuro` quickly detects this and does nothing.

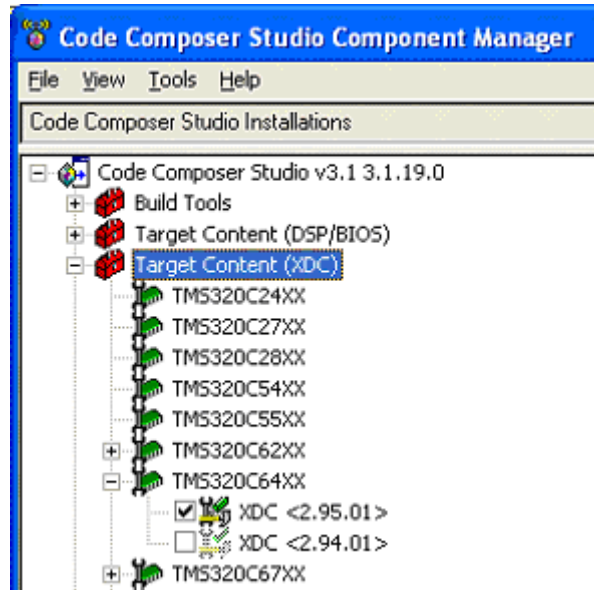
You can integrate the step of running `configuro` into your makefile or Code Composer Studio project file as described in the subsections that follow.

For details about using `configuro`, see Section 3.3 and the CDOC online reference for the `xdc.tools.configuro` package.

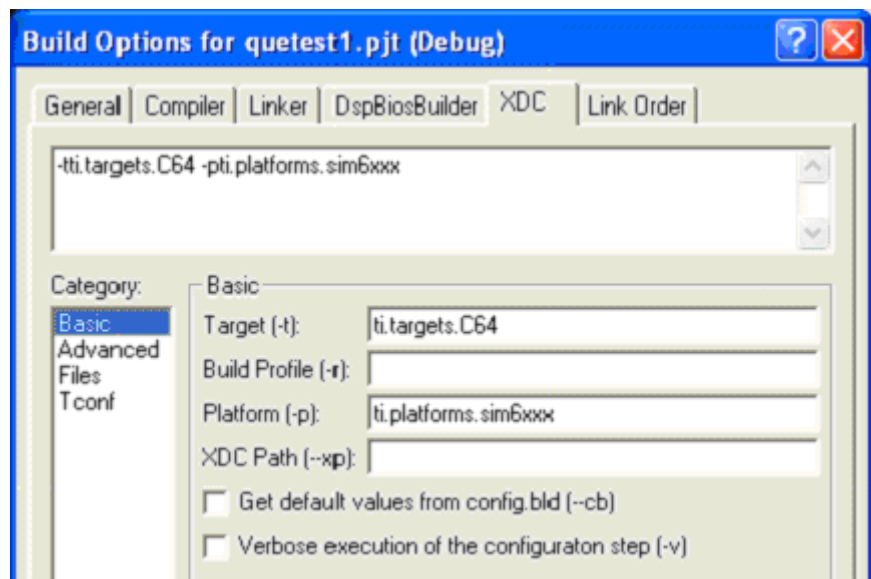
2.4.4 Running Configuro from a CCStudio Project

On Windows systems, if you have installed both XDC and Code Composer Studio, you can use an XDC tab in the Build Options dialog in Code Composer Studio.

- 1) First, use the Component Manager (part of CCStudio) to make sure the latest version of XDC is enabled. To run the Component Manager, use your Windows Start menu to choose **All Programs->Texas Instruments->Code Composer Studio->Component Manager**. Within Component Manager, expand Target Content (XDC). Then expand your target family. Make sure the version of XDC you want to use has a checkmark. When you close Component Manager, save your changes.



- 2) Next, add the CFG file you created to your CCStudio project.
- 3) Then, open the **Project->Build Options** dialog and move to the XDC tab:



- 4) With the Basic category selected, specify the target and platform packages you want to use. Optionally, you can also specify the -r (build profile) and --xp (XDCPATH) command-line options for configuro here.

2.5 Compiling and Linking

The particular compile and link syntax you use is compiler-dependent. See Section A.1 for details about compiler options for various targets. Additional examples are provided with XDC in the examples/configuro directory.

NOTE: The compiler's standard C runtime library must come after the linker.cmd file in the link order in a makefile or CCStudio project. For example, this is an RTS library for TI compilers, libc.a for GCC, and msvcrt.lib for Microsoft Visual Studio.

2.5.1 Compiling and Linking with CCStudio

Add the compiler.opt file to the project in the **Project->Build Options** dialog. In the Compiler tab, select the Files category and add mycfg/compiler.opt in the Options File field.

Add the linker.cmd file to your project using the Add Files to Project command.

If you haven't already added a RTS library to your CCStudio project, use the Add Files to Project command. For example, your RTS library might be \$(CGTOOLS)/lib/rt6400.lib.

2.5.2 Using the Command Line on Windows for TI Targets with CCStudio

- 1) In a command window, run the DosRun.bat file in the CCStudio installation directory.
- 2) Compile your application using a command similar to the following, where *<mycfg>* is the name of your configuration package directory.

```
cl6x -@<mycfg>/compiler.opt -c hello.c
```

- 3) Link your application using a command similar to the following, where *<mycfg>* is the name of your configuration package directory.

```
cl6x -q -z -c hello.obj <mycfg>/linker.cmd
C:/CCStudio_v3.3/C6000/cgtools/lib/rt6400.lib -o hello.out
```

2.5.3 Building with the TI Code Generation Tools from a Makefile

This section and the two sections that follow show how to run the configuro tool and use its output with the compilers and linkers for TI, GNU, and Microsoft. They use the compiler.opt and linker.cmd files generated by configuro.

The example makefiles in this section and the following sections all use GNU make, which is delivered with XDC (XDC_INSTALL_DIR/gmake).

Important Note: In GNU makefiles, always use a tab character to indent, not spaces. Also, all commands should be on a single line. Variable definitions that contain spaces (for example, CGTOOLS often does on Windows) must be surrounded by quotes. For more information about the syntax of makefiles and the operation of GNU make see the online GNU make manual at <http://www.gnu.org/software/make/manual/make.html>.

The following example makefile runs the TI Code Generation Tools compiler and linker.

```
CGTOOLS = C:/CCStudio_v3.3/C6000/cgtools

CC = $(CGTOOLS)/bin/cl6x
LNK = $(CGTOOLS)/bin/lnk6x
RTS = $(CGTOOLS)/lib/rts6400.lib

CONFIG = mycfg
XDCTARGET = ti.targets.C64
XDCPLATFORM = ti.platforms.sim6xxx

%/linker.cmd %/compiler.opt : %.cfg
    xs xdc.tools.configuro -c $(CGTOOLS) -t $(XDCTARGET) -p $(XDCPLATFORM) $<

%.obj : %.c $(CONFIG)/compiler.opt
    $(CC) -@$(CONFIG)/compiler.opt -c $<

hello.out : hello.obj $(CONFIG)/linker.cmd
    $(LNK) -o hello.out -c hello.obj $(CONFIG)/linker.cmd $(RTS)
```

2.5.4 Building with Microsoft Visual Studio from a Makefile

The following example makefile runs `configuro` and uses the `compiler.opt` and `linker.cmd` files generated by `configuro` in the command lines for the Microsoft Visual Studio compiler and linker.

```
CGTOOLS = "C:/Program Files/Microsoft Visual Studio 8"

CC = $(CGTOOLS)/vc/bin/cl
LNK = $(CGTOOLS)/vc/bin/link
RTS = -nodefaultlib -libpath:$(CGTOOLS)/vc/lib kernel32.lib ms-
      vcrtd.lib

CONFIG = mycfg
XDCTARGET = microsoft.targets.Win32

%/linker.cmd %/compiler.opt : %.cfg
    xs xdc.tools.configuro -c $(CGTOOLS) -t $(XDCTARGET) $<

%.obj : %.c $(CONFIG)/compiler.opt
    $(CC) @$$(CONFIG)/compiler.opt -c $<

hello.exe : hello.obj $(CONFIG)/linker.cmd
    $(LNK) -out:$@ hello.obj @$$(CONFIG)/linker.cmd $(RTS)
```

No platform specification is needed because the default platform for each target in `microsoft.targets` is the one required for native execution on an x86 PC (`host.platforms.PC`).

2.5.5 Building with GCC from a Makefile

The following example makefile runs configuro and uses the compiler.opt and linker.cmd files generated by configuro in the command lines for the GCC compiler and linker.

```
CGTOOLS = $(TOOLS)/vendors/gnu/gcc/3.2.1/Linux

CC = $(CGTOOLS)/bin/gcc
LNK = $(CGTOOLS)/bin/gcc
RTS = -lstdc++

CONFIG = mycfg
XDCTARGET = gnu.targets.Linux86

%/linker.cmd %/compiler.opt : %.cfg
    xs xdc.tools.configuro -c $(CGTOOLS) -t $(XDCTARGET) $<

%.o : %.c # forget built-in rule
%.o : %.c $(CONFIG)/compiler.opt
    $(CC) $(shell cat $(CONFIG)/compiler.opt) -c $<

hello : hello.o $(CONFIG)/linker.cmd
    $(LNK) -o $@ hello.o $(CONFIG)/linker.cmd $(RTS)
```

No platform specification is needed because the default platform for the gnu.targets.Linux86 target is the one required for native execution on an x86 PC (host.platforms.PC).

The GCC compiler does not directly support an option to include other command line options from a file. However, this example accomplishes this by using GNU make's "\$\$(shell ...)" function and the Linux cat command to copy the options to the compiler's command line before calling GCC.

XDC Tools

This chapter provides command-line syntax and examples for the tools provided with XDC.

Topic	Page
3.1 Overview of the Tools	3-2
3.2 The cdoc Tool	3-3
3.3 The configuro Tool	3-5
3.4 The path Tool	3-6
3.5 The repoman Tool	3-8

3.1 Overview of the Tools

The following XDC tools are useful when creating applications that use XDC-based content. See the CDOC online documentation for information about each of these tools. (See Section 1.6.1 for information about using CDOC.)

Table 3–1 XDC Tools

Tool Package	Description
xdc.tools.cdoci.sg	Online documentation viewer for XDC-based packages
xdc.tools.cdoci	Command-line based documentation viewer
xdc.tools.configuro	Processes an XDC configuration file to generate compiler.opt and linker.cmd files
xdc.tools.path	Displays a list of packages on the XDCPATH
xdc.tools.path.sg	Displays a list of packages on the XDCPATH in a graphical window
xdc.tools.repoman	Command-line based repository manager
xdc.tools.repoman.sg	Graphical display repository manager

These tools are run using the xs command. The xs executable runs command scripts found using the package path. In most cases, these tools include both command-line (textual) output tools and graphical tools. If a graphical version of a tool exists, its package name is the same as the command-line tool with the additional suffix ".sg".

3.2 The cdoc Tool

The cdoc tool displays help information about XDC packages. The interface is graphical. This tool is in the `xdc.tools.cdoc.sg` package.

To open the CDOC reference help system, run the following command from the command prompt—for example by opening the Microsoft Windows Start menu and choosing **Run**.

```
xs xdc.tools.cdoc.sg
```

You see the "cdoc" window with a tree view of the XDC packages available to you. All of these packages provide documentation that can be displayed by the CDOC viewer.

Click "+" next to a repository to expand its list of packages. Click "+" next to a package name to see the list of modules it provides. You can further expand the tree to see a list of the functions provided by a module. Double-click on a package or module to see its reference information.

Notice the icons in the upper-right corner of the window. The following icons are useful:



View C interface view of documentation.



View XDC interface view of documentation.



Close all page tabs.

Many modules have a dual existence—they have configuration parameters that are used during the configuration process and they have C APIs that are called at runtime. Chose the "XDC interface view" to view a module's configuration interface and the "C interface view" for a its C runtime interface.

For each topic you view, there is a tab across the top of the page area. You can use these to quickly return to other pages you have viewed. You can also use the arrows next to "Views" to move backward and forward in your history of page views.

To close a page and remove its tab, click the X on the tab.

Optionally, you can specify a repository on the command line. If you do, only the packages in that repository are listed. If you do not specify a repository, all packages found using the package path are listed.

3.2.1 Generating Static Help Documentation

If you prefer to use a web browser to view package Reference documentation, you can run a command-line version of the CDOC tool to generate static HTML files for any collection of packages. After generating static pages, you can leverage popular search tools such as Google Desktop to locate relevant documentation.

Suppose you have packages installed in the repositories `c:/bios_6_00/packages` and `c:/codec_engine_1_20/packages`. The following command generates configuration documentation for all of these packages into the output directory `ref_html`.

```
xs xdc.tools.cdoc -l XDC -od:ref_html -PR c:/bios_6_00/packages  
c:/codec_engine_1_20/packages
```

For more information see the CDOC documentation of the `xdc.tools.cdoc` package.

3.3 The configuro Tool

The XDC configuro tool processes XDC configuration files. The interface is via a command line. This tool is in the `xdc.tools.configuro` package. For step-by-step instructions for using configuro, see Section 2.4.

Running the configuro command generates a subdirectory with the same name as your CFG file (for example, `mycfg` for the `mycfg.cfg` file). This subdirectory is an XDC "package".

The configuro command allows you to import XDC content, in the form of reusable modules built using the XDC tools, into your embedded application. This tool is the recommended method for integrating XDC content into traditional command-line based build environments.

Configuro lets you identify and customize the XDC content you want to use. It computes a set of libraries, command-line flags, and other artifacts to include in your application build. By changing values of configuration settings, you can trade off the functionality, memory footprint, and even performance of the XDC content to best meet the needs of your application.

To generate files from a configuration, use this command line syntax:

```
xs xdc.tools.configuro
    [-v|--help] [-b config_bld | -c codegen_dir]
    [-t target] [-p platform[:instance]] [-r profile]
    [-x regex] [-w] [-o outdir] infile.cfg
```

For example:

```
xs xdc.tools.configuro -t ti.targets.C64
    -p ti.platforms.sim6xxx
    -c c:/CCStudio_v3.3/C6000/cgtools mycfg.cfg
```

- ❑ **-b** : Specifies the name of the `config.bld` file.
- ❑ **-c** : Specifies root directory of the code generation tools or compiler.
- ❑ **-o** : Specifies the name of the output directory.
- ❑ **-oc** : Set name of compiler options file to create. Default is `compiler.opt`.
- ❑ **-ol** : Set name of the linker command file to create. Default is `linker.cmd`.
- ❑ **-p** : Specifies the platform to use.
- ❑ **-r** : Specifies the build profile to use.
- ❑ **-t** : Specifies the target to use.
- ❑ **-v** : Show details during build.
- ❑ **-w** : Treat incompatibilities only as warnings.
- ❑ **-x** : Exclude specified packages from compatibility checking.

3.4 The path Tool

The path tool displays the packages located along the specified package path. Both textual and graphical versions of this tool are provided. The textual tool is `xdc.tools.path` and the graphical tool is `xdc.tools.path.sg`.

By default, the path tool displays a list of all packages found in the repositories referenced by the package path.

For example, output for the `xdc.tools.cdoci.sg` package is similar to this:

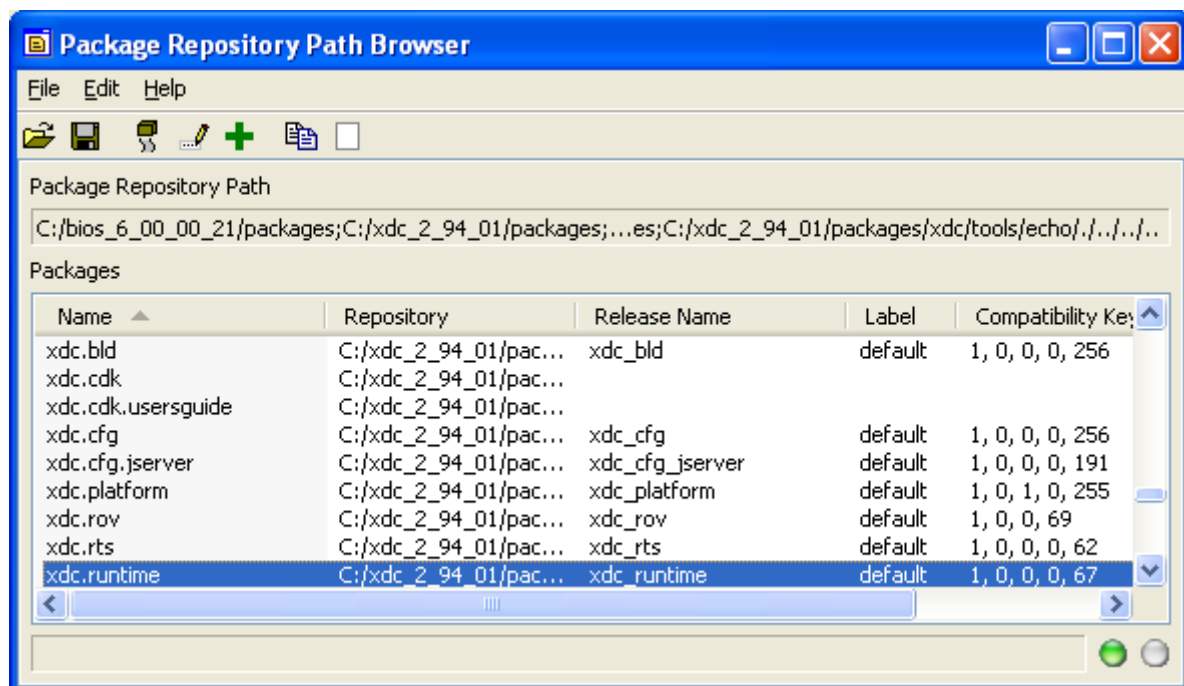
```
Package Name:      xdc.tools.cdoci.sg
Repository:       C:\xdc_2_94_01\packages
Release Name:     xdc_tools_cdoci_sg
Label:           default
Compatibility Key: 1, 0, 0, 62
Date:            04/25/2007  19:07
```

The following command-line options are available:

- ☐ `-n` : Display only the names of the packages.
- ☐ `-p` : Echo only the current package path.
- ☐ `-d` : Display information about the packages in the specified repository only. For example:

```
xs xdc.tools.path -d C:/CCStudio_v3.3/bios_5_31/packages
```

When you run the xdc.tools.path.sg graphical utility, the window looks similar to the following:



This tool allows you to modify the package path and rescan the repositories for packages. You can also save the list of packages to a file.

3.5 The repoman Tool

The repoman tool creates and manages package repositories. It displays information about archived packages, extracts archived packages, extracts repositories from bundles and deletes packages from repositories. Both textual and graphical versions of this tool are provided. The textual tool is `xdc.tools.repoman` and the graphical tool is `xdc.tools.repoman.sg`.

Important Note: The repoman tool, especially its current command-line option set, is preliminary and subject to change in a future release. See the online documentation for the `xdc.tools.repoman` and `xdc.tools.repoman.sg` packages to get the most recent information about this tool.

3.5.1 Command Line Repoman Tool

For the command-line tool, the main options are:

- ☐ `-t` : Display package info for an archive directory or tar file.
- ☐ `-n` : Display package names only.
- ☐ `-x` : Extract packages from an archive directory or tar file.
- ☐ `-d` : Delete packages from a repository.
- ☐ `-c` : Copy packages to a repository.
- ☐ `-o` : Save repository information to a file.
- ☐ `--rrestore` : Restore one or more repositories from information in a file.
- ☐ `--rcreate` : Create one or more repositories from information in a file.

There are a number of other options that can be used in conjunction with these options, but one of these options is required on the command line to tell repoman what action it is supposed to take. This section describes the main functions first and gives the syntax for each. Later, it lists the various extra options you can use in combination with the main options.

The `-t` and `-n` options display information. The `-t` option displays the file, release name, label, version, category, buildable, required packages, references, and description. The `-n` option displays only the name. The syntax for these informational functions is:

```
xs xdc.tools.repoman -t [-v] [-p path ]
                        [ packagename... | packagefile... ]

xs xdc.tools.repoman -n [-v] [-p path ]
                        [ packagename... | packagefile... ]
```


The -x option can extract packages from an archived directory or tar file to a destination repository. If any of the packages in the specified package list are not compatible, the operation is not performed. The syntax is:

```
xs xdc.tools.repoman -x [-v] [-D] [-R] [-f] [-r destrepo ]
                        [-p path ][-b [ bundlerepository... ]]
                        [ packagename... | packagefile... ]
```

The following example extracts the package ti.platforms.sim6xxx from an archive in /db/trees/app-a01 to c:/xdc/packages:

```
xs xdc.tools.repoman -x -r c:/xdc/packages -p /db/trees/app-a01
ti.platforms.sim6xxx
```

The following example extracts the examples repository from c:/xdc-Win32.tar.gz to c:/myexamples:

```
xs xdc.tools.repoman -x -r c:/myexamples
-b examples c:/xdc-Win32.tar.gz
```

The -d option deletes a package from a repository. If the package has any subdirectories that contain packages, they will be left intact. After the package is deleted, if the package has no subdirectories with packages, empty directories that are part of the fully qualified package are deleted, too. The syntax is:

```
xs xdc.tools.repoman -d [-v] [-r destrepo ] packagename...
```

The following example deletes the package ti.platforms.sim6xxx from c:/xdc/packages:

```
xs xdc.tools.repoman -d -r c:/xdc/packages ti.platforms.sim6xxx
```

The -c option copies packages to a destination repository. The syntax is:

```
xs xdc.tools.repoman -c [-v] [-r destrepo] [-p path]
packagename...
```

The following example copies the package ti.catalog.c6000 from c:\srcdir\packages to c:\destdir\packages:

```
xs xdc.tools.repoman -c -r c:\destdir\packages
-p c:\srcdir\packages ti.catalog.c6000
```

The -o option saves repository information to the "outfile" file to be used in a subsequent restore or create operation. The syntax is:

```
xs xdc.tools.repoman -o outfile [-p searchpath] repository ...
```

The following example saves package information for repository c:/myrepo to the c:/myrepo.rmn file. It also saves the c:/pkgarchives;c:/testpkgs search path:

```
xs xdc.tools.repoman -o c:/myrepo.rmn  
-p "c:/pkgarchives;c:/testpkgs" c:/myrepo
```

The --rrestore option restores one or more repositories from information in the "infile" file. The syntax is:

```
xs xdc.tools.repoman --rrestore -i infile [-p searchpath]  
[infile_repo_indicies]
```

The following example restores all the packages in c:/myrepo from information that was saved to the file c:/myrepo.rmn. It uses the search path c:/pkgarchives;c:/testpkgs to find and copy the packages. The example creates c:/myrepo if it doesn't exist.

```
xs xdc.tools.repoman -rrestore -i c:/myrepo.rmn  
-p "c:/pkgarchives;c:/testpkgs"
```

The --rcreate option creates one or more repositories from information in the "infile" file. The syntax is:

```
xs xdc.tools.repoman --rcreate -i infile [-p searchpath]  
[infile_repo_indicies] dest_repo0 ...
```

The following example creates the new repository c:/mynewrepo from information that was saved to the file c:/myrepo.rmn for the repository c:/myrepo. It uses the search path c:/pkgarchives;c:/testpkgs to find and copy the packages. When created, c:/mynewrepo will be identical to c:/myrepo when its package information was saved with the -o option.

```
xs xdc.tools.repoman -rcreate -i c:/myrepo.rmn  
-p "c:/pkgarchives;c:/testpkgs" c:/mynewrepo
```

3.5.2 Additional Command-Line Options

The other options that can be used on the repoman command line with the main options described in the previous section are as follows:

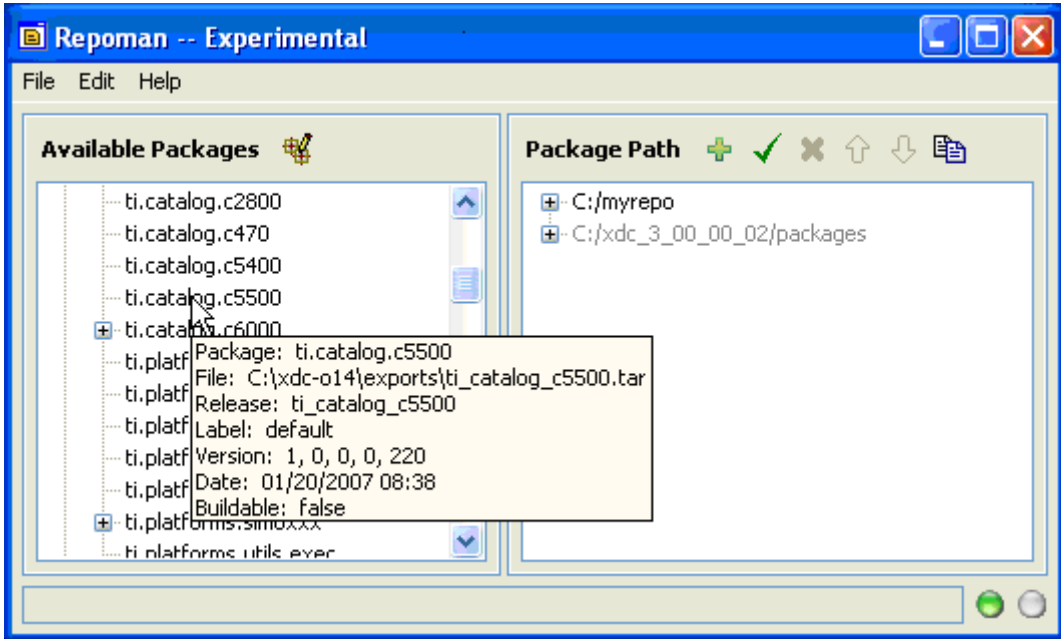
- ☐ **-b** : The repositories to be extracted when **-x** is used to extract repositories from a bundle. In this case, the last argument specifies the bundle file to extract from. If omitted, use "packages". Use with: **-x**.
- ☐ **-D** : Also extracts dependent packages. Use with: **-x**.
- ☐ **-f** : Force all packages to be extracted, regardless of incompatibilities. Use with: **-x**.
- ☐ **-p path** : Search the specified directory. If the **-p** option is omitted, the current directory is searched. Use with: **-t**, **-n**, **-x**, **-c**.
- ☐ **-r destrepo** : Use specified destination repository. If omitted, use the repository for the current package. If there is no current package, use the current working directory. Use with: **-x**, **-d**, **-c**.
- ☐ **-R** : Also recursively extracts dependent packages. Use with: **-x**.
- ☐ **-v** : Print informative messages during execution. Use with: **all**.
- ☐ **package name** : A list of packages to use. If none are specified, all packages found in the search directory are used. May contain wildcards. For example: `ti.platforms.sim6xxx` or `ti.catalog.c??00` or `ti.platforms.*`. Use with: **all**.

A version/key can also be attached to a package name as follows: `ti.platforms.c6000[1,0,0,0,313835]`. A prefix of the full version/key may also be used as follows: `ti.platforms.c6000[1,0]`. If a package name has multiple matches either with or without a version/key specified, the most recent is selected.


- ☐ **packagefile** : A list of files to use. If none are specified, all packages in the search directory are used. May contain wildcards. Use with: **-t**, **-n**, **-x**.

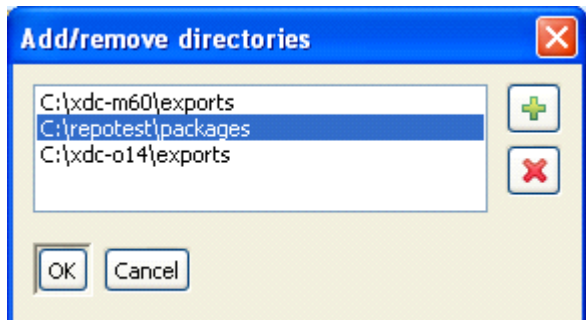
3.5.3 Graphical Repoman Tool

The Repoman graphical tool is used to manage package repositories and check the validity of package paths. The left-hand pane displays archived packages and packages found in repositories. The right-hand pane shows packages in repositories that are treated as a package path. Package details can be viewed by hovering the mouse cursor over the package name, as shown in the following figures.



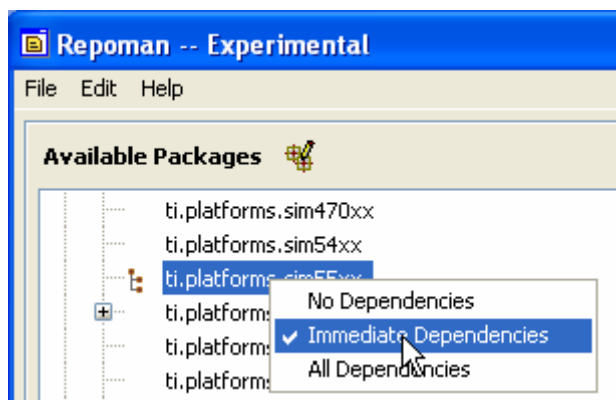
Duplicate packages are displayed under the package name on a separate branch, by version and date.


You can populate the Available Packages pane by clicking the  button on the pane's toolbar and adding or removing directories via the dialog box. Directories are first searched for archived packages. If none are found, they are then recursively searched for package directories.



You can copy or extract any package in the Available Packages pane by selecting it and dragging it to the "Package Path" pane and dropping onto a repository in that pane.


To have an archived package's dependencies extracted along with it, right-click on the package and select a dependency level prior to using drag-and-drop. "Immediate Dependencies" extracts all of the package's dependencies. "All Dependencies" extracts all of the package's dependencies and their recursive dependencies.

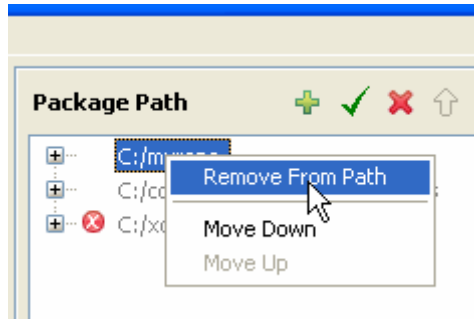


The "Package path repositories pane" is initially populated with the contents of XDCPATH environment variable. You can add additional repositories by clicking the  button on the pane's toolbar and specifying a repository in the dialog box that comes up. The dialog box also allows you to create new repositories (directories). **Only repositories created in this manner can have packages added and deleted.**


A common use for this tool is to copy or extract existing packages to create a "working" repository. You might then modify some of the packages or delete some packages. Then, you can view the "working" repository's internal compatibility and its compatibility in relation to other repositories.


Your XDCPATH is not modified by adding or removing repositories using Repoman. However, you can copy the resulting path to the clipboard and update your XDCPATH manually.




You can remove a repository from the path by selecting it and clicking the  icon, or by right-clicking on the repository and selecting the "Remove From Path" menu item.



You can move a repository up or down in the order by right-clicking it and selecting the "Move up" or "Move down" menu item.

To delete a package from a repository, select the package and click the  icon, or right-click on the package and select "Delete". All of the packages' files will be deleted.

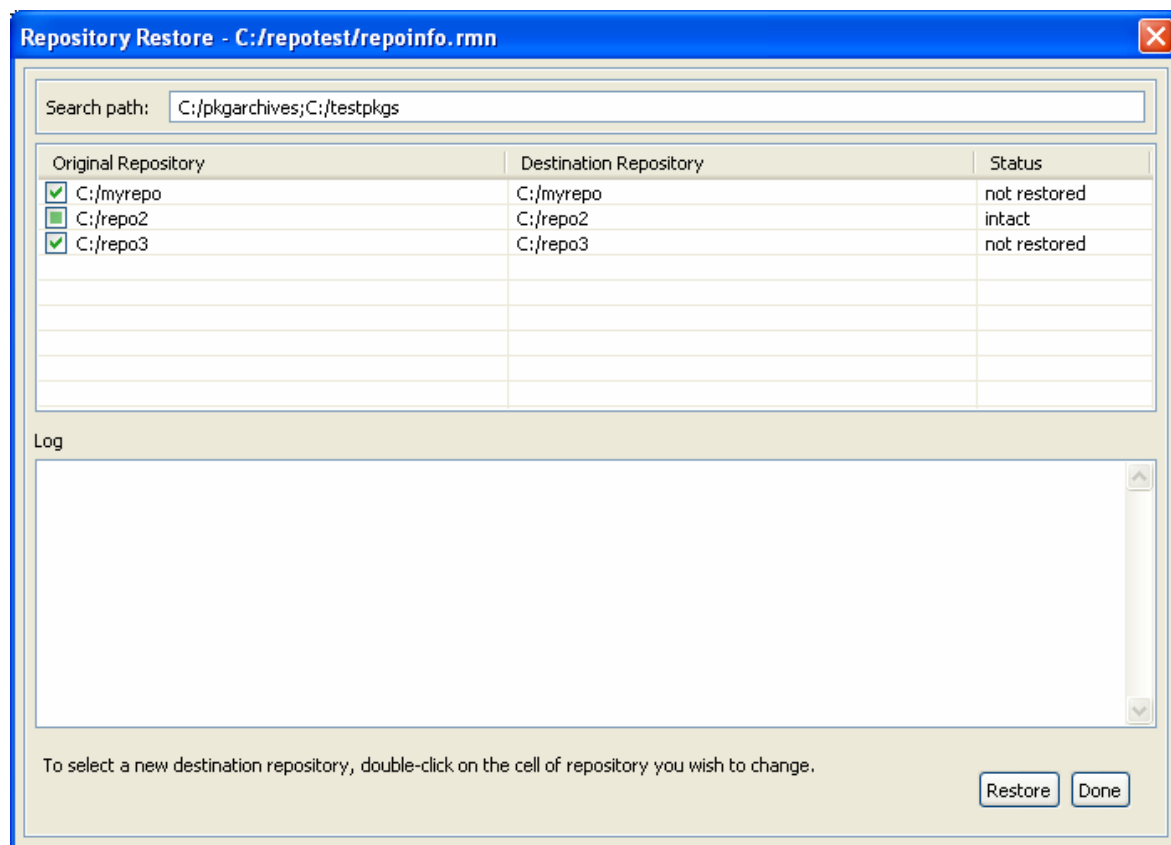
You can copy the pane's repository list to the clipboard by clicking the  icon. The list is delimited by semicolons and provided in the order they are shown in the pane.

You can check the package path for errors by clicking the  icon. If any errors or warnings are generated, an error  or warning  icon is displayed next to the repository containing the package that generated the error and also next to the package. To see a description of the error, hover the mouse over the package's icon.



You can save repository information to a file by choosing **File->Save** from the menu bar. The repository, the name and version of all packages in the Package pane, and the directories used to display packages in the Available Packages pane will be saved.

You can open repository information files that were created with File->Save by choosing **File->Open** from the menu bar. Opening a previously saved file displays a dialog showing the repositories that package information was saved for and their state.



If all of the repositories about which information was saved are intact, you can click **Done** to see the usual Repoman view. If any of the repositories have been modified or deleted, you can start a restore operation for them. You may specify a different name for any repository being restored. If the repository cannot be restored completely to its original state, the operation does not create, modify or copy any directories or files. You may repeatedly retry the restore operation with different search paths.

XDC Runtime Modules

This chapter provides introductory information about the XDC runtime modules.

Topic	Page
4.1 Overview of the Runtime Modules	4-2
4.2 XDC Boot Sequence and Control Points	4-2
4.3 System Module	4-6
4.4 Memory Segments and Sections	4-10
4.5 Memory Allocation and Heaps	4-13
4.6 Timestamp Module	4-17
4.7 Gate Interface and Implementations	4-19
4.8 Diagnostics and Logs	4-21
4.9 Types Module	4-33

4.1 Overview of the Runtime Modules

The modules and interfaces in the `xdc.runtime` package form the "core" XDC target runtime support library. This package is provided with complete sources. It implements basic system services required by virtually all applications.

4.2 XDC Boot Sequence and Control Points

This section describes the sequence of events that occurs immediately after a device is reset until execution reaches the application's `main()` function. It also identifies the specific points during this boot sequence at which user-provided functions ("boot hooks") are called.

All targets follow the same set of boot sequence steps. In outline the target-independent boot sequence steps are:

- 1) Immediately after CPU reset, perform target-specific CPU initialization.
- 2) Run user-supplied reset function (see `Startup.resetFxn`).
- 3) Perform additional target-specific initialization.
- 4) Run user-supplied "first functions" (See `Startup.firstFxn`s).
- 5) Run all module initialization functions.
- 6) Run user-supplied last functions (See `Startup.lastFxn`s).
- 7) Run `main()`.

The reset function (in step 2 above) is run as early as possible in the boot sequence. In fact, for some targets it is called before the C environment is fully initialized. So, the execution context of this function is target-specific and may require that it not use static data, for example.

Although the functions in steps 4-6 are C functions, they are run during normal C++ static object initialization. This has two consequences:

- ❑ Since the C++ standard does not provide a means to control the order of execution of static constructors, the initialization of application modules may not run before a C++ constructor runs. If you need to use a module (for memory allocation, for example), your constructor must explicitly call `Startup_exec()` before using any other module in the system. Since the `Startup_exec()` function is idempotent (multiple calls yield the same result), it is safe to call this function more than once. Thus you may call it from each C++ constructor that needs a module's services.
- ❑ If you are using a target that does not support C++, you must explicitly call `Startup_exec()` prior to using any module's services. You can simply call `Startup_exec()` as the first step in `main()`, for example.

4.2.1 Initialize Registers

Execution on TI targets always starts at `_c_int00`, which is plugged as the reset vector. The boot sequence begins with the initialization of a few critical registers, including the system stack pointer. This part of the sequence is target-specific.

For a detailed view of the initial register setup, the actual boot code is available in the XDC installation. On all targets, the relevant portion of the boot code (in terms of the initial register setup) occurs between `_c_int00` (where execution starts) and the call to the `xdc_runtime_Startup_reset()` function. Boot code for any other targets should be available in a similar location, inside the "rts" folder for that target.

The following sections show what register initialization occurs for each target and where the actual boot code is located:

'C6000

- ☐ Clear the IER and IFR.
- ☐ Set the system stack pointer (B15) to point to the end of the stack section.
- ☐ Set the global page pointer (B14) to point to the beginning of the .bss section.
- ☐ On the 'C6700, also set the floating point registers.

The c6000 boot code is available in:

`<xdc_install_dir>\packages\ti\targets\rts6000\boot.c`

The `auto_init()` function is defined in `autoinit.c` in the same directory.

'C2000

Set up the stack, initialize any status bits not initialized by the reset, and set the C28x modes.

The c2000 boot code is available in:

`<xdc_install_dir>\packages\ti\targets\rts2800\boot_cg.asm`

'C470

Set user mode, change to 16-bit state, and initialize and align the user mode stack.

The c470 boot code is available in:

`<xdc_install_dir>\packages\ti\targets\boot.asm`

ARM7

Set user mode, change to 16-bit state, and initialize and align the user mode stack.

The arm7 boot code is available in:

```
<xdc_install_dir>\packages\ti\targets\arm\rtsarm7\boot.asm
```

ARM9

Set user mode, change to 16-bit state, and initialize and align the user mode stack.

ARM7

The arm9 boot code is available in:

```
<xdc_install_dir>\packages\ti\targets\arm\rtsarm9\boot.asm
```

4.2.2 Perform User Reset Function

The reset function is the first control point in the boot sequence. It occurs almost immediately after the target has been reset. The user-specified reset function is called after the initial register initialization, but before even the cinit records have been processed. Any code that needs to be executed immediately after a reset, such as configuring or disabling a watchdog timer, should occur here.

The Startup module has a module-wide configuration parameter called "resetFxn", which specifies the function to run at this point.

Configuration example: The following example XDC configuration statements demonstrate the Startup module's use:

```
var Startup = xdc.useModule('xdc.runtime.Startup');  
Startup.resetFxn = "&myResetFxn";
```

Note that the "&" isn't actually being used as an operator in these statements; it's just part of a string. Internally, XDC knows to interpret the characters after the & as the name of an external user-supplied function.

The resetFxn is called before the cinit records have been processed, so no data structures or variables have been loaded with their initial values at this point.

4.2.3 Process .cinit Records

After the reset function has been called, the C initialization records are processed.

4.2.4 Start Modules

At this point, the boot sequence calls the `xdc_runtime_Startup_exec()` function. This function then calls all the module startup functions.

The Startup module has two module-wide configuration parameters: "firstFxn" and "lastFxn", which are arrays of user-specified functions to be called before and after the module startup functions, respectively.

Configuration example: The following example XDC configuration statements specify functions to be called with the firstFxn and lastFxn parameters.

```
var Startup = xdc.useModule('xdc.runtime.Startup');
```

```
var len = Startup.firstFxn.length;  
Startup.firstFxn.length++;  
Startup.firstFxn[len] = '&myFirst';
```

```
var len = Startup.lastFxn.length;  
Startup.lastFxn.length++;  
Startup.lastFxn[len] = '&myLast';
```

Functions added to "firstFxn" typically initialize values needed by module startup functions—such as initialization of global data used by modules, which is a function of platform-specific registers. Functions added to the "lastFxn" array might be used by middleware or other modules to perform any necessary initialization before `main()`.

4.2.5 Process .pinit Records

After the module startup functions have run, the pinit records are processed. The .pinit table consists of pointers to initialization functions. For C++ programs, class constructors of global objects execute during .pinit processing.

Note that the module initialization in the previous section may occur in the middle of .pinit processing. So, C++ constructors that require the services of a module must explicitly call `Startup_exec()` prior to using any module.

4.2.6 Main() Function

Finally, execution reaches the `main()` function.

4.3 System Module

The System module provides a façade for two key system level areas: printing and program termination. Since both the printing and termination processes can be very different depending on the product and stage of development, you can configure an application to use different implementations of the underlying mechanisms while maintaining the same code base. This is accomplished by setting the System Support Proxy.

4.3.1 System Support Proxy

The Support Proxy modules inherit from `xdc.runtime.ISystemSupport`. This interface specifies five functions: `abort`, `exit`, `flush`, `putch` and `ready`. The System module calls into the configured Support Proxy functions as needed. There are two implementations in the `xdc.runtime` package:

- ❑ `xdc.runtime.SysStd`: Based on the standard C runtime library. See Section 4.3.6.
- ❑ `xdc.runtime.SysMin` (default): Used for footprint- or performance-constrained applications. See Section 4.3.7.

You set the Support Proxy via the `SupportProxy` configuration parameter of the System module. The following statements set the Support Proxy to `xdc.runtime.SysStd`:

```
var System = xdc.useModule('xdc.runtime.System');  
System.SupportProxy = xdc.useModule('xdc.runtime.SysStd');
```

If an application wants to minimize the memory footprint, it should use the `SysMin` module as the Support Proxy by either omitting the lines above or substituting `SysMin` for `SysStd` above.

4.3.2 System Gate

The System module is a gated module (see Section 4.7). The gate associated with the System module is used for two purposes:

- ❑ Used to manage critical regions within the System Module.
- ❑ Used by any code to protect its small critical regions via the `Gate_enterSystem()` and `Gate_leaveSystem()` APIs.

The System Gate must be restrictive enough to insure that critical regions protected by System's gate are safe. For example, if these regions can be accessed by both interrupt service routines (ISRs) and non-ISR code, the

Gate must disable interrupts on entering and restore interrupts on leaving. The following is example configuration code to set the System Gate to an instance of the GateHwi module in the ti.sysbios.gates package.

```
GateHwi = xdc.useModule('ti.sysbios.gates.GateHwi');
System.common$.gate = GateHwi.create();
```

4.3.3 atexit Handling

An atexit function is a function that is called during an exit. The System module allows you to configure the number of atexit functions that are bound at runtime via the maxAtexitHandlers configuration parameter (default is 8). The following statements set the maximum number of System atexit functions that can be added after the application starts running.

```
var System = xdc.useModule('xdc.runtime.System');
System.maxAtexitHandlers = 4;
```

The System atexit functions have the following typedef:

```
typedef Void (*AtexitHandler)(Int status);
```

The System atexit functions are called when a System_exit() is called. The status that is passed into System_exit() is passed into each of the System atexit functions. The order of the System atexit functions is last in first out (LIFO). The System_atexit() function is used to add a System atexit function.

```
if (System_atexit(myExitFxn) == FALSE) {
    /* failed to add myExitFxn atexit function */
}
```

The System atexit functions are independent of the standard C runtime atexit functions. During startup, the System module registers the System_rtsExit function with the atexit handler. During the execution of the standard C runtime atexit functions, this function (System_rtsExit) executes all System atexit functions.

4.3.4 Termination

There are two types of termination functions:

- ☐ System_exit()
- ☐ System_abort()

The System_exit() function is intended to be used for a graceful termination. The System_abort() is intended for catastrophic termination. The main difference is that the atexit functions (both System and standard C runtime) are executed on a System_exit() and not on an System_abort().

4.3.4.1 System_exit

When called, the `System_exit()` function performs the following operations:

- 1) Calls the standard C runtime `exit()` function.
- 2) Via `System_rtsExit` (the standard C runtime `atexit` function that System registered):
 - a) Enters the System gate (see Section 4.3.2).
 - b) Calls the System `atexit` functions (see Section 4.3.3).
 - c) Calls the Support Proxy's exit function (see Section 4.3.1).

If the standard C runtime exit function is called instead of the `System_exit()`, the exit status passed into `exit()` is not passed to the System `atexit` functions. The constant `System_STATUS_UNKNOWN` (equal to `0xCAFE`) is used instead.

4.3.4.2 System_abort

When called, the `System_abort()` function performs the following operations

- 1) Enters the System gate (see Section 4.3.2)
- 2) Calls the Support Proxy's abort function (see Section 4.3.1).
- 3) Calls the standard C runtime `abort()` function.

The System `atexit` (or standard C runtime `atexit`) functions are not executed.

If the standard C runtime abort function is called instead of the `System_abort()`, the first two steps are not performed.

4.3.5 Printing

There are several printing APIs provided by the System module: `System_putchar()`, `System_printf()`, `System_sprintf()`, `System_vprintf()` and `System_vsprintf()`. Although the prototypes for these APIs are same as their standard C runtime counterparts, not all standard `printf` format strings are supported. The System module manages `printf` format string parsing, while the System Proxy does the actual character output. See `SysMin` and `SysStd` for implementation details.

The `System_flush()` function calls into the Support Proxy's flush function. See `SysMin` and `SysStd` for implementation details.

4.3.6 SysStd Module

The SysStd module inherits the `xdc.runtime.ISystemSupport` interface and is used to provide the underlying services required by the System module via System's `SupportProxy` configuration parameter (see Section 4.3.1).

SysStd directly calls the standard C runtime functions. It does not perform any additional functionality. The standard C runtime `putchar` and `fflush` functions are used. Both SysStd's `abort` and `exit` functions flush the standard output via the standard C runtime `fflush` function.

4.3.7 SysMin Module

The SysMin module inherits the `xdc.runtime.ISystemSupport` interface and is used to provide the underlying services required by the System module via System's `SupportProxy` configuration parameter (see Section 4.3.1).

SysMin is System's default Support Proxy; it is intended to be flexible but minimal. When SysMin is the Support Proxy, all output—such as `System_printf()` and `System_putchar()`—goes into an internally managed buffer. There are several configuration parameters that let you change the features as needed.

- ☐ **bufSize.** The size in MAUs of the internal buffer used for output.
- ☐ **sectionName.** The section in which the internal buffer should be created.
- ☐ **flushAtExit.** If true, the internal buffer is flushed via `SysMin_flush()` during a `System_exit()` or `System_abort()`.

If an application wants to minimize the memory footprint, it should use the SysMin module as the Support Proxy. If the application needs minimal printing capabilities, the `bufSize` configuration parameter can be set to 0, and `flushAtExit` can be set to false to even further reduce the footprint.

4.4 Memory Segments and Sections

XDC manages a device's memory by dividing it first into *memory segments* and then into *memory sections* that lie within those segments.

Memory segments are named ranges of memory with fixed addresses; they generally correspond to the different memory units on a device. Memory segments are configurable in an application's config.bld or configuration file.

Memory sections are smaller allocations of a memory segment, such as the ".text" section, which holds code, and the ".far" section, which holds data. Each memory section is placed in a particular memory segment. The default mapping of sections to segments is based on two factors. First, the XDC target (see Section 2.4.1) defines all the default sections (such as ".text", ".const", and ".cinit") and maps them to either "code", "data", or "stack" memory. Then, the XDC platform (see Section 2.4.2) maps "code", "data", and "stack" to actual memory segments.

The target and platform define the default mapping of sections to segments, but these mappings can be overridden by explicitly placing the sections in the client configuration script using 'Program.sectMap'. For example, to place a section named ".mySection" into the memory segment named "IRAM", the configuration script would contain the following line:

```
Program.sectMap[".mySection"] = "IRAM";
```

This same line can be used to either relocate an existing section, or to create a new section. In this way, the memory managed by a statically created heap can be explicitly placed by specifying a section for the heap, and placing that section into the desired memory segment using 'Program.sectMap'.

Configuration example: The following XDC configuration example places a heap in memory explicitly by placing its section.

```
var HeapMin = xdc.useModule('xdc.runtime.HeapMin');

/* Create a HeapMin and place it in new section ".myHeapSect" */
var myHeap = HeapMin.create();
myHeap.size = 1024;
myHeap.sectionName = ".myHeapSect";

/* Place the heap's section in the IRAM segment */
Program.sectMap["myHeap.sectionName"] = "IRAM";
```

The section map can also be used to accomplish more specific placement, including specifying the load and run addresses for the segment. See the CDOC online reference for more details.

Configuration example: The following XDC configuration statements place the .text section in a different load and run segment. They also place the .myHeapSect section at a specific address.

```
Program.sectMap[".text"] = {loadSegment: "SDRAM",
                           runSegment: "IRAM"};
Program.sectMap[".myHeapSect"] = {loadAddress: 0x80000000};
```

Finally, the current section map can be viewed programmatically in a configuration script using the Program.getSectMap() API. This API returns the current section map based on the Target and Platform section maps as well as the Program.sectMap. See the CDOC online reference for details.

Configuration example: The following XDC configuration statements print all of the sections in the section map, as well as any defined properties of the section.

```
var sectMapView = Program.getSectMap();

/* For each section in the sectMap */
for (var sectName in sectMapView) {
    print(sectName);
    var sectSpec = sectMapView[sectName];
    /* print the section's defined properties */
    for (var p in sectSpec) {
        if (sectSpec[p] != undefined) {
            print("    " + p + " = " + sectSpec[p]);
        }
    }
}
```

Sample output (printed to command line during configuration)

```
.text
    loadSegment = IRAM
.myHeapSect
    loadSegment = IRAM
.switch
    loadSegment = SDRAM
.vecs
    loadSegment = IRAM
.args
    loadSegment = SDRAM
.stack
    loadSegment = SDRAM
.system
    loadSegment = SDRAM
.far
    loadSegment = SDRAM
.data
    loadSegment = SDRAM
.cinit
    loadSegment = SDRAM
.bss
    loadSegment = SDRAM
.const
    loadSegment = SDRAM
.pinit
    loadSegment = SDRAM
.cio
    loadSegment = SDRAM
```

4.5 Memory Allocation and Heaps

The `xdc.runtime.Memory` module provides the interface for all memory allocations, both static and dynamic. This module is a common interface for all the different `IHeap` implementations.

The actual memory management is performed by heaps. Heaps are created via heap-specific create functions. The heap instances are then used in the `Memory` module calls. Internally, the `Memory` module calls the heap's interface functions.

`Memory_alloc()` is used at runtime to dynamically allocate memory.

The `Memory` module is simply a common interface for all memory operations. The actual memory management is performed by a heap instance, such as `HeapMin` or `HeapStd`. All of the `Memory` APIs take a heap instance as one of their parameters. Internally, the `Memory` module calls into the heap's interface functions.

Different heap implementations have different memory management algorithms, but all are accessed through `xdc.runtime.Memory`.

The `Memory` module supports a default heap, which can be used by passing "NULL" as the `IHeap_Handle` to the `Memory` APIs. The default heap can be specified by configuring `Memory.defaultHeapInstance`. Otherwise, a default heap of type `xdc_runtime_HeapStd` is created with size `Memory.defaultHeapSize`.

Memory allocations sizes are measured in "Minimum Addressable Units" (MAUs) of memory. The size of an MAU in bits is determined by the target.

4.5.1 Heap Handles

The memory APIs that require a handle to a heap require a handle of type `IHeap_Handle`. This requires an explicit cast from the specific Heap handle type to the general `IHeap_Handle`. You can perform this explicit cast using a utility function, which always has the form

```
<specific heap type>_Handle_to_xdc_runtime_IHeap(myHeap);
```

For example, to convert a `HeapStd_Handle` to an `IHeap_Handle`, use the following:

```
IHeap_Handle iheap = HeapStd_Handle_to_xdc_runtime_IHeap(myHeap);
```

Of course, you can simply cast the variable `myHeap` to be an `IHeap_Handle`. But, by using the utility function above, your code remains type safe. For example, if `myHeap` is ever changed to an incompatible type, the compiler correctly flags an error, whereas with the cast the compiler silently proceeds.

4.5.2 Allocating Memory Dynamically

The `Memory_alloc()` function allocates memory. Its prototype is as follows:

```
Ptr Memory_alloc(IHeap_Handle heap, SizeT size, SizeT align,
                Error_Block* eb)
```

The "heap" parameter is the heap instance to allocate memory from. The "size" and "align" parameters specify the length of the memory block to allocate and the alignment of the starting address, respectively. The error block is used for error reporting.

Run-time example: The following C code example allocates a task stack from the default heap. The task stack will have a size of 512 MAUs and no particular alignment.

```
Ptr tsk1_stack = Memory_alloc(NULL, 512, 0, NULL);
```

Run-time example: The following C code example allocates space for a structure from a heap.

```
extern HeapStd_Handle myHeap;

typedef struct Foo {
    int bar;
} Foo;

Foo *myFoo;
IHeap_Handle iheap = HeapStd_Handle_to_xdc_runtime_IHeap(myHeap);
myFoo = (Foo *) Memory_alloc(iheap, sizeof(Foo), 0, NULL);
myFoo->bar = 10;
```

The Memory module also supports the standard `calloc` and `valloc` APIs, which initialize the allocated memory to zero or to a specified value, respectively.

```
Ptr Memory_calloc (IHeap_Handle heap, SizeT size, SizeT align,
                  Error_Block* eb);

Ptr Memory_valloc (IHeap_Handle heap, SizeT size, SizeT align,
                  Char value, Error_Block* eb);
```

4.5.3 Freeing Memory

The `Memory_free()` function frees memory. Its prototype is as follows:

```
Void Memory_free(IHeap_Handle heap, Ptr block, SizeT size)
```

Memory can only be freed if the underlying Heap supports it.

A call to `Memory_free()` takes the handle to the heap instance to free memory to, a pointer to the block of memory to be freed, and the size of the block that was allocated.

By requiring the size of the block to be passed to the `Memory_free()` method, the underlying memory allocator can manage heaps with no extra space from the heap itself. This can be particularly important when managing small on-chip memory regions with large alignment requirements; even one word taken from a 32K block of memory wastes significant memory when allocations must be aligned on 8K boundaries, for example.

Run-time example: The following C code example allocates a block, then frees it back to a heap. `Foo` is a structure, and the `doSomething()` function operates on a block of memory. The "myHeap" handle is for a Heap instance.

```
Ptr block;
extern HeapStd_Handle myHeap;

IHeap_Handle iheap = HeapStd_Handle_to_xdc_runtime_IHeap(myHeap);
block = Memory_alloc(iheap, sizeof(Foo), 0, NULL);

doSomething(block);

Memory_free(iheap, block, sizeof(Foo));
```

4.5.4 The `xdc.runtime.HeapMin` Implementation

`HeapMin` is a simple "growth-only" heap, meaning that it does not support `Memory_free()`. It is intended as a minimal footprint heap implementation. `HeapMin` services any size allocation request. Attempting to free memory back to a `HeapMin` instance results in an error.

Configuration example: The following XDC configuration example statically creates a `HeapMem` instance named `myHeap`:

```
var HeapMin = xdc.useModule('xdc.runtime.HeapMin');

/* Create heap as global variable so it can be used in C code */
Program.global.myHeap = HeapMin.create();
Program.global.myHeap.size = 1024;
```

Run-time example: The following C code example dynamically creates a HeapMem instance named myHeap:

```
HeapMin_Params prms;  
char *buf[1024];  
HeapMin_Handle myHeap;  
  
HeapMin_Params_init(&prms);  
prms.size = 1024;  
prms.buf = (Ptr)buf;  
myHeap = HeapMin_create(&prms, NULL);
```

4.5.5 The xdc.runtime.HeapStd Implementation

HeapStd is a wrapper for the "C" standard C runtime malloc() and free() functions. As a consequence, allocating from a HeapStd instance ignores the "align" parameter to Memory_alloc().

Configuration example: The following XDC configuration example statically creates a HeapStd heap instance named myHeap:

```
var HeapStd = xdc.useModule('xdc.runtime.HeapStd');  
  
/* Create heap as global variable so it can be used in C code */  
Program.global.myHeap = HeapStd.create();  
Program.global.myHeap.size = 1024;
```

Run-time example: The following C code example dynamically creates a HeapStd instance named myHeap:

```
HeapStd_Params prms;  
HeapStd_Handle myHeap;  
  
HeapStd_Params_init(&prms);  
prms.size = 1024;  
myHeap = HeapStd_create(&prms, NULL);
```


4.6 Timestamp Module

The `xdc.runtime.Timestamp` module, as the name suggests, provides timestamping services. The Timestamp module can be used for benchmarking code and adding timestamps to Log events (see Section 4.8.4).

Timestamp provides two APIs for timestamping: `Timestamp_get32()` and `Timestamp_get64()`. `Timestamp_get64()` is only truly supported on targets that have a dedicated 64-bit counter. On other targets, the upper part of `Types.Timerstamp` is simply set to zero.

This module also provides a `Timestamp_getFreq()` API that returns the frequency of the timestamp counter in Hz. This API can be used to convert the numbers returned by `Timestamp_get32()` and `Timestamp_get64()` to actual time units (milliseconds).

The Timestamp module has a configuration parameter of type `ITimestampProvider`. Depending on the Timestamp provider used, the Timestamp counter may be a timer or a 64-bit dedicated counter in the core or a 32-bit counter that is part of a timer peripheral.

During debugging, you may plug `xdc.runtime.Timestamp.SupportProxy` with `xdc.runtime.TimestampStd`, which calls the standard C runtime function `clock()`. For example:

```
var Timestamp = xdc.useModule("xdc.runtime.Timestamp");
Timestamp.SupportProxy = xdc.useModule("xdc.runtime.TimestampStd");
```

There is also an `xdc.runtime.TimestampNull` module that can be used when timestamp services are not needed by the application. The `TimestampNull` module always returns a timestamp of `~0`. However, other modules like `xdc.runtime.LoggerBuf` that use `xdc.runtime.Timestamp` will tag events with this unchanging timestamp.

Run-time example: This C example calculates a time delta based on timestamp values:

```
#include <xdc/runtime/Timestamp.h>

time1 = Timestamp_get32();
CallLongFunction();
time2 = Timestamp_get32();
delta = time2 - time1;
```

Run-time example: This C example uses `Timestamp_getFreq()` and prints a message every second.

```
#include <xdc/runtime/Types.h>
#include <xdc/runtime/Timestamp.h>
main ()
{
    ...
    Types.FreqHz freq;
    UInt32 start;

    Timestamp_getFreq(&freq);

    /* works only for timestamps with 32-bit frequencies */
    assert(freq.hi == 0);

    for (;;) {
        start = Timestamp_get32();
        while ((Timestamp_get32() - start) < freq.lo) {
            ; /* do nothing */
        }
        System_print("A second has elapsed");
    }
    ...
}
```

Configuration example: This XDC example showing how to plug proxy

```
var Timestamp = xdc.useModule('xdc.runtime.Timestamp');

Timestamp.SupportProxy =
    xdc.useModule('xdc.runtime.TimestampStd');
```

4.7 Gate Interface and Implementations

A "Gate" is a module that implements the `IGateProvider` interface. Gates prevent concurrent accesses to critical data structures—that is, data structures that must be updated atomically to preserve system integrity. The various Gate implementations differ in how they lock the critical regions.

Threads can be preempted by other threads of higher priority, and some sections of code need to be completed by one thread before they can be executed by another thread. Code that modifies a linked list is a common example of a critical region that needs to be protected by a Gate.

Some modules use Gates internally to protect critical regions of code, such as modifications to the module state, which are shared across all instances of the module. A module that does this is referred to as a "Gated Module". Gated modules are marked with the "@Gated" command in their CDOC documentation. The documentation for such modules discusses how the Gate is used by that module.

Gated modules can be configured to use a specific Gate type, which is set using the common defaults for that module.

Gates generally work by either disabling some level of preemption such as disabling task switching or even hardware interrupts, or by acquiring a binary semaphore prior to any access to the critical data.

Gates have two key APIs: "enter" and "leave". A thread calls `Gate_enter()` to lock a critical region and prevent other threads from passing that point in the code. When the thread has finished executing the critical region, it calls `Gate_leave()` to allow other threads to execute that critical region.

All Gate implementations support nesting through the use of a "key". A call to `Gate_enter()` returns a key that must then be passed back to `Gate_leave()`.

Run-time example: The following C code protects a critical region with a Gate. This example uses a module named GateHwi, a Gate implementation that disables and enables interrupts as its locking mechanism.

```
UInt gateKey;
GateHwi_Handle gateHwi;
GateHwi_Params prms;
GateHwi_Params_init(&prms);

gateHwi = GateHwi_create(&prms, NULL);

/* Simultaneous operations on a Queue by multiple threads could
 * corrupt the Queue structure, so modifications to the Queue
 * are protected with a Gate. */
gateKey = GateHwi_enter(gateHwi);
Queue_dequeue(myQ);
GateHwi_leave(gateHwi, gateKey);
```

4.7.1 The xdc.runtime.GateNull Implementation

GateNull provides no protection. Calling GateNull_enter() and GateNull_leave() has no effect. For applications without preemptive threads or when an application never uses a gated module from more than one thread, GateNull is provided so that a gated module may be configured to not use any real Gate.

4.7.2 Plugging Gates

Gated modules can be configured to use a specific Gate type, which is set using the common defaults for that module.

Configuration example: The following XDC example creates a Gate.

```
ModA.common$.gate = MyGate.create();
```

In addition to gated modules, there is a system-wide "system gate" that can be used anywhere. The system gate is configured to provide the highest level of protection available or required by the system. A real-time operating system, for example, would configure the system gate to disable all preemption down to the hardware level.

The system gate is configured using the common\$.gate property of the System module:

```
Var System = xdc.useModule('xdc.runtime.System');
System.common$.gate = MyGate.create();
```

4.8 Diagnostics and Logs

XDC provides for diagnostics with a set of modules that operate together to configure and implement diagnostics.

These modules can be partitioned into three groups: modules that generate events, a module that allows precise control over when (or if) various events are generated, and modules that manage the output or display of the events.

Assert, Error, and Log provide methods that are added to source code and generate events. The Diags module provides both configuration and runtime methods to selectively enable or disable different type of events on a per module basis. Finally, LoggerBuf and LoggerSys are simple alternative implementations of an event output interface, ILogger. You can provide more sophisticated or platform-specific implementations of ILogger without making any changes to code that uses Assert, Log, Error, or Diags.

The modules described in this section are

- ☐ **Assert.** Add integrity checks to the code. See Section 4.8.1.
- ☐ **Diags.** Manage a module's diagnostics mask. See Section 4.8.2.
- ☐ **Error.** Raise error events. See Section 4.8.3.
- ☐ **Log.** Generate log events in real-time. See Section 4.8.4.
- ☐ **LoggerBuf.** A logger using a buffer for log events. See Section 4.8.5.
- ☐ **LoggerSys.** A logger using printf for log events. See Section 4.8.6.
- ☐ **Types.** Define diagnostics configuration parameters. See Section 4.9.

4.8.1 Assert Module

The Assert module provides configurable diagnostics to the program. Similar to the standard C `assert()` macro, Assert methods are interspersed with other code to add diagnostics to a program. Unlike the standard C assert support, the Assert module provides greater flexibility in managing the messages displayed, the message string space overhead, and the runtime handling of failures. In addition, because the Assert methods build atop the Diags module, you can precisely control which asserts remain in the final application, if any.

The Assert module works in conjunction with the Diags module. Assert statements are added to the code using the `Assert_isTrue()` function. Execution of the assert statements is controlled by the `Diags_ASSERT` bit in the module's diagnostics mask (see Section 4.8.2).

Two types of asserts are supported: public asserts and internal asserts:

- ❑ Public asserts must have an assert ID and are, by default, controlled by the Diags.ASSERT bit.
- ❑ Internal asserts cannot have an assert ID (other than NULL) and are active only when both the Diags.ASSERT and Diags_INTERNAL bits of the module's diagnostics mask are set.

Assert IDs are small integer values that index into a table of assertion descriptors. These descriptors hold an error message and a diagnostics mask that is used to enable and disable the assertion at runtime.

You can remap individual public asserts to different bits in the diagnostics mask, or even disable the asserts altogether. This is done by setting the mask property of the assert ID. Setting the mask to 0 disables the assert. In all other cases, the Diags.ASSERT bit is ORed together with the mask to define the controlling bits. For example, the module's diagnostics mask must have the Diags.ASSERT bit set and any other bit specified in the mask property of the assert ID in order to activate the assert.

Example 1: The following C code adds an assert to application code that is that is not in a module. This assert does not have an assert identifier (the second argument is NULL); This makes it an internal assert.

```
/* file.c */
#include <xdc/runtime/Assert.h>

Assert_isTrue(count > 0, NULL);
```

The following XDC configuration statements set both the ASSERT and INTERNAL bits in the diagnostics mask to enable the internal assert created in the previous C code. Since the C code is not in a module, you must set the bits in the diagnostics mask of the xdc.runtime.Main module. The Main module is used to control all Log and Assert statements that are not part of the implementation of a module; for example, top-level application code or any existing sources that simply call the Log or Assert methods.

```
/* program.cfg */
var Assert = xdc.useModule('xdc.runtime.Assert');
var Diags = xdc.useModule('xdc.runtime.Diags');
var Main = xdc.useModule('xdc.runtime.Main');

Main.common$.diags_ASSERT = Diags.ALWAYS_ON;
Main.common$.diags_INTERNAL = Diags.ALWAYS_ON;
```

Example 2: The following example shows how to use and configure an assert ID that is declared by a module. It adds that assert to the application's C source code, and configures the application to execute the assert.

This is part of the XDC file for the module that declares an Assert Id:

```
/* Mod.xdc */
import xdc.runtime.Assert;
import xdc.runtime.Diags;

config Assert.Id A_nonZero = {
    msg: "A_nonZero: value must be non-zero"
};
```

This is part of the C code for the application:

```
#include <xdc/runtime/Assert.h>

Assert_isTrue(x != 0, Mod_A_nonZero);
```

This is part of the XDC configuration file for the application:

```
var Diags = xdc.useModule('xdc.runtime.Diags');
var Mod = xdc.useModule('my.pkg.Mod');
Mod.common$.diags_ASSERT = Diags.ALWAYS_ON;
```

4.8.2 Diags Module

Every XDC module has a "diagnostics mask" that allows clients to enable or disable diagnostics on a per module basis both at configuration time and at runtime. The Diags module manages a module's diagnostics mask.

You use the Diags module to set and clear bits in a module's diagnostics mask for the purpose of controlling diagnostics within that module. Each bit corresponds to a "type" of diagnostic that can be individually enabled or disabled.

A module's diagnostics mask controls both Assert and Log statements within that module. A module's diagnostics mask may also be used to conditionally execute blocks of code using the Diags_query() runtime function.

A module's diagnostics mask can be set at configuration time or at run-time. The implementation of diagnostics is such that when they are permanently turned off at configuration time, an optimizer can completely eliminate the diagnostics code from the program. Similarly, if diagnostics are permanently turned on at configuration time, the optimizer can eliminate all run-time conditional checking and simply invoke the diagnostics code directly. Run-time checking of the diagnostics mask is performed only when the diagnostics are configured to be run-time modifiable.

Each bit of the diagnostics mask is controlled by one of the following constants.

Constant	Meaning
ENTRY	Function entry
EXIT	Function exit
LIFECYCLE	Object lifecycle
INTERNAL	Internal diagnostics
ASSERT	Assert checking
USER1	User defined diagnostics
USER2	User defined diagnostics
USER3	User defined diagnostics
USER4	User defined diagnostics
USER5	User defined diagnostics
USER6	User defined diagnostics
USER7	User defined diagnostics
USER8	User defined diagnostics

These constants can be used from either JavaScript configuration scripts or C code and, therefore, have two "names". For example, to reference the ENTRY constant from JavaScript you must use "Diag.ENTRY" whereas from C you would use "Diagnostics_ENTRY".

The ENTRY and EXIT bits control Log statements at the entry and exit points, respectively, to each function within a module. This is useful for tracking the execution flow of your program.

The LIFECYCLE bit controls Log statements at the create/construct and delete/destruct points to each instance object for the module. This is useful for tracking the lifecycle of a module instance object.

The ASSERT bit controls Assert statements in a module. There are two classes of asserts:

- ❑ Public asserts, which have an Assert_Id and are documented in the module's reference pages. These asserts are on by default and are meant to help developers validate code that invokes a module's functions.

- Internal asserts, which have no `Assert_Id`. These asserts are off by default and are typically used only when developing code. That is, like the standard C `assert()` mechanism, these asserts are not part of a deployed application.

When a module has the ASSERT bit set (which is set by default) in its diagnostics mask, the module executes all of its public assert statements. To enable internal asserts, you must set both the ASSERT and INTERNAL bits.

The INTERNAL bit is used to classify diagnostic code as being internal. That is to say, this class of diagnostics is undocumented and typically not used by client software.

The USER1-7 bits are available to each module writer to use as they wish.

Configuration example: The following XDC configuration statements set a module's diagnostics mask in a configuration script. (In this case, the Task module of the `ti.sysbios.knl` package is used.) In this example, the ENTER bit is turned on and the EXIT bit is turned off. Both bits are configured to be run-time modifiable.

```
var Diags = xdc.useModule('xdc.runtime.Diags');
var Task = xdc.useModule('ti.sysbios.knl.Task');

Task.common$.diags_ENTER = Diags.RUNTIME_ON;
Task.common$.diags_EXIT = Diags.RUNTIME_OFF;
```

Run-time example: The following C code shows how to disable and re-enable ENTER diagnostics at runtime for the `ti.sysbios.knl.Task` module configured in the previous example. The first call to `Diag_setMask()` enables entry diagnostics ("E") for just the `ti.sysbios.knl.Task` module; any call to any Task method in the application causes an "entry" Log event to be generated. The second call disables ("E") the generation of these events. See the `xdc.runtime.Diags` module's reference documentation for a complete description of the strings accepted by `Diags_setMask()`.

```
#include <xdc/runtime/Diags.h>
:
Diags_setMask("ti.sysbios.knl.Task+E");
:
Diags_setMask("ti.sysbios.knl.Task-E");
```

Configuration example: The following XDC configuration statements turn on asserts in the entire program.

```
var Diags = xdc.useModule('xdc.runtime.Diags');
var Defaults = xdc.useModule('xdc.runtime.Defaults');
Defaults.diags_ASSERT = Diags.ALWAYS_ON;
```

Configuration example: The following XDC configuration statements turn on asserts in all of the modules whose name begins with "ti.sysbios." using the `Diags.setMaskMeta()` function. In this case, no change to the application code is necessary to enable these events. It is important to note that, while the configuration step must be rerun and the application must be re-linked, no application sources need to be recompiled.

```
var Diags = xdc.useModule('xdc.runtime.Diags');
Diags.setMaskMeta("ti.sysbios.%", Diags.ASSERT, Diags.ALWAYS_ON);
```

4.8.3 Error Module

The `xdc.runtime.Error` module is the run-time error manager.

This module provides mechanisms for raising, checking, and handling errors in a program. You can configure it via the `Error.policy` and `Error.raiseHook` configuration parameters.

Modules may define specific error types and reference these when raising an error. Each error type has a custom error message and can be parameterized with up to `NUMARGS` arguments. A generic error type is provided for raising errors when not in a module.

Use the `Error_check()` function to determine if an error has been raised. It is important to understand that it is the caller's responsibility to check the error block after calling a function that takes an error block as an argument. Otherwise, a raised error may go undetected, which could compromise the integrity of the system. For example:

```
Task_create(..., &eb);
if (Error_check(&eb)) {
    ...an error has been raised...
}
```

The `Error.raiseHook` allows a configured function to be invoked when any error is raised. This function is passed a pointer to the error's error block and makes it easy to manage all errors from a common point. For example, you can trap any error (fatal or not) by simply setting a breakpoint in this function. You can use the following functions to extract information from an error block.

- ☐ `Error_getData()`
- ☐ `Error_getCode()`
- ☐ `Error_getId()`
- ☐ `Error_getMsg()`
- ☐ `Error_getSite()`

Example: The following example shows how a module, named ModA, defines a custom error type and shows how this error is raised by the module. The module defines an Id of E_notEven in its module specification file (in this case, ModA.xdc). The error's message string takes only one argument. The module also defines a mayFail() function that takes an error block. In the module's C source file, the function checks for the error condition and raises the error if needed.

This is part of ModA's XDC specification file for the module:

```
config xdc.runtime.Error.Id E_notEven = {
    msg: "expected an even number (%d)"
};

Void mayFail(Int x, xdc.runtime.Error.Block *eb);
```

This is part of the C code for the module:

```
Void ModA_mayFail(Int x, Error_Block *eb)
{
    if ((x % 2) != 0) {
        Error_raise(eb, ModA_E_notEven, x, 0);
        ...add error handling code here...
        return;
    }
    ...
}
```

Run-time Example: The following C code supplies an error block to a function that requires one and tests the error block to see if the function raised an error. Note that an error block must be initialized before it can be used and same error block may be passed to several functions.

```
#include <xdc/runtime/Error.h>
#include <ti/sysbios/knl/Task.h>
Error_Block eb;
Task_Handle tsk;

Error_init(&eb);
tsk = Task_create(..., &eb);

if (Error_check(&eb)) {
    ...an error has been raised...
}
```

Run-time Example: The following C code shows that you may pass NULL to a function requiring an error block. In this case, if the function raises an error, the program is aborted (via `xdc_runtime_System_abort()`), thus execution control will never return to the caller.

```
#include <xdc/runtime/Error.h>
#include <ti/sysbios/knl/Task.h>

tsk = Task_create(..., NULL);
...will never get here if an error was raised in Task_create...
```

Run-time Example: The following C code shows how to write a function that is not part of a module and that takes an error block and raises the generic error type provided by the Error module. Note, if the caller passes NULL for the error block or if the error policy is `Error_TERMINATE`, then the call to `Error_raise()` will call `xdc_runtime_System_abort()` and never return.

```
#include <xdc/runtime/Error.h>

Void myFunc(..., Error_Block *eb)
{
    ...
    if (...error condition detected...) {
        String myErrorMsg = "my custom error message";
        Error_raise(eb, Error_E_generic, myErrorMsg, 0);
        ...add error handling code here...
        return;
    }
}
```

4.8.4 Log Module

The Log module is used to generate log events in real-time while the target program executes. Log events are routed to a "logger" instance for output or storage; for example, an instance of `xdc.runtime.LoggerBuf` or `xdc.runtime.LoggerSys`.

Log event routing is managed at the module level, where each module can specify a logger instance. A logger instance can accept log events from many modules, but a module can route log events to only one logger instance. There can be one or many logger instances in a system. Each logger instance represents a separate log. All Log calls that are not in a module are controlled by the module `xdc.runtime.Main`. The Main module is used to control all Log and Assert statements that are not part of the implementation of a module—for example, top-level application code or any existing sources that simply call the Log or Assert methods.

The generation of a log event is controlled by a module's diagnostics mask. Each log event is associated with a mask. Log events are generated only when a particular bit is set in both the log event mask and the module's diagnostics mask. For example, a log event mask with the USER1 bit set is generated only when the USER1 bit is also set in the module's diagnostics mask.

There are two ways to generate log events:

- **LOG_write()**, which is tailored for module writers and takes full advantage of the XDC configuration model—for example, the message string associated with the Log event need not be part of the final application, significantly reducing the "footprint overhead" of embedding diagnostics in deployed systems. The Log_write[0-8]() functions allow up to 8 values to be passed to the logger. They expect the logger to handle any formatting. A log event type allows you to specify the type of event.
- **LOG_print()**, which is designed for arbitrary C code. The Log_print[0-6]() functions allow up to 6 values to be passed along with a printf-like format string to the logger. They handle printf-style formatting.

Both functions are controlled by the module's diagnostics mask. Their storage or output is defined by the logger that is assigned to the module that calls the Log methods or to the xdc.runtime.Main module if the caller is not part of a module.

The log function call sites are implemented in such a way that an optimizer can completely eliminate log code from the program if the log function has been permanently disabled at configuration time. If the log functions are permanently turned on at configuration time, then the optimizer can eliminate all run-time conditional checking and simply invoke the log function directly. Run-time checking is performed only when the log functions are configured to be run-time modifiable.

The Log module provides log event generation, but in order to store or display a log event, a logger instance must be created and assigned to the module. This is done in the program configuration script. See Section 4.8.5 or Section 4.8.6 for details.

Example 1: The following example defines a log event, uses that log event in a module, and configures the program to generate the log event. In this example, both USER1 and USER2 bits are set in the event mask. This means that if either bit is set in the module's diagnostics mask, then the log event will be generated. Note, in order for the event to be realized, you must also create a logger instance and assign it to the module. See LoggerBuf or LoggerSys for details.

This is part of the XDC specification file for the Mod module (Mod.xdc):

```
import xdc.runtime.Diags;
import xdc.runtime.Log;

config Log.Event L_someEvent = {
    mask: Diags.USER1 | Diags.USER2,
    msg: "my log event message, arg1: 0x%x, arg2: 0x%x"
};
```

This is part of the C code implementation of the Mod module:

```
#include <xdc/runtime/Log.h>
UInt x, y;

Log_write2(Mod_L_someEvent, (IArg)x, (IArg)y);
```

Finally, the following is an example of how the application might control the Log statements embedded in the Mod module at configuration time. In this case, the configuration script arranges for the Log statements within the my.pkg.Mod module (shown above) to always generate events. Without these configuration statements, no Log events would be generated by this module. This is part of the XDC configuration file for the application:

```
var Diags = xdc.useModule('xdc.runtime.Diags');
var Mod = xdc.useModule('my.pkg.Mod');
Mod.common$.diags_USER1 = Diags.ALWAYS_ON;
```

Example 2: The following XDC configuration statements turn on enter and exit logging at configuration time for a module.

```
var Diags = xdc.useModule('xdc.runtime.Diags');
var MyMod = xdc.useModule('my.pkg.MyMod');

MyMod.common$.diags_ENTER = Diags.ALWAYS_ON;
MyMod.common$.diags_EXIT = Diags.ALWAYS_ON;
```

Example 3: The following example configures a module to support enter and exit logging, but defers the actual activation and deactivation of the logging until run-time. See the Diags_setMask() function for details on specifying the control string.

This is part of the XDC configuration file for the application:

```
var Diags = xdc.useModule('xdc.runtime.Diags');
var MyMod = xdc.useModule('my.pkg.MyMod');

MyMod.common$.diags_ENTER = Diags.RUNTIME_OFF;
MyMod.common$.diags_EXIT = Diags.RUNTIME_OFF;
```

This is part of the C code for the application:

```
/* turn on enter and exit logging in the module */
Diags_setMask( "my.pkg.MyMod+EX" );

/* turn off enter and exit logging in the module */
Diags_setMask( "my.pkg.MyMod-EX" );
```

4.8.5 LoggerBuf Module

The `xdc.runtime.LoggerBuf` module provides a logger that stores run-time log events in a buffer.

This logger captures log events in real-time to a buffer. The log events stored in the buffer are unformatted; the log event formatting is deferred. Use `LoggerBuf_flush()` to process the log events stored in the buffer and stream the formatted output to `stdout`.

The implementation of this logger is fast with minimal stack usage making it appropriate for a real-time application. If this logger is used in a preemptive environment, then an appropriate gate must be assigned to the module. For example, if you wish to generate log events from an interrupt context, then a gate that disables interrupts must be used.

```
var LoggerBuf = xdc.useModule('xdc.runtime.LoggerBuf');
LoggerBuf.common$.gate = ...some gate instance...
```

If the buffer type is circular, the log buffer of size `LoggerBuf_numEntries` contains the last `numEntries` log events. If the `bufType` is fixed, the log buffer contains the first `numEntries` log events.

Configuration example: The following XDC configuration statements create a logger instance, assign it as the default logger for all modules, and enable USER1 logging in all modules of the package my.pkg. See the `Diags.setMaskMeta()` function in the CDOC system for details on specifying the module names.

```
var LoggerBuf = xdc.useModule('xdc.runtime.LoggerBuf');
LoggerBuf.enableFlush = true;

var LoggerBufParams = new LoggerBuf.Params();
LoggerBufParams.exitFlush = true;

var Defaults = xdc.useModule('xdc.runtime.Defaults');
Defaults.common$.logger = LoggerBuf.create(LoggerBufParams);

var Diags = xdc.useModule('xdc.runtime.Diags');
Diags.setMaskMeta("my.pkg.%", Diags.USER1,
    Diags.RUNTIME_ON);
```

4.8.6 LoggerSys Module

The `xdc.runtime.LoggerSys` module routes events to the `System_printf()` function.

This logger processes log events as they are generated and routes them through the `System_printf()` function. The final disposition of the log event is dependent on which system provider has been assigned to `System.SupportProxy`.

Note that the log events are processed within the run-time context of the `Log_write()` or `Log_print()` function that generated the event. It is important to account for the processing overhead and stack usage imposed on the run-time context. The cost of this processing is defined by the implementation of the system provider.

Configuration example: The following XDC configuration statements create a logger instance, assign it as the default logger for all modules, and enable USER1 logging in all modules of the package my.pkg. See `Diags.setMaskMeta` for details on specifying the module names.

```
var LoggerSys = xdc.useModule('xdc.runtime.LoggerSys');
var LoggerSysParams = new LoggerSys.Params();

var Defaults = xdc.useModule('xdc.runtime.Defaults');
Defaults.common$.logger = LoggerSys.create(LoggerSysParams);

var Diags = xdc.useModule('xdc.runtime.Diags');
Diags.setMaskMeta("my.pkg.%", Diags.USER1,
    Diags.RUNTIME_ON);
```

4.9 Types Module

The `xdc.runtime.Types` module defines basic constants and types used throughout the `xdc.runtime` package and, in some cases, to every XDC module.

The `Common$` structure defined by the `Types` module is available for (or common to) all XDC modules. Every field of the `Common$` structure is a configuration parameter that may be set within a configuration script for any XDC module (not just the `xdc.runtime` modules). The fields of this structure are typically read by the modules in the `xdc.runtime` package at configuration time to control the generation of data structures that are embedded in the application and referenced by these modules at runtime.

Every XDC module has a configuration parameter named `"common$"` that is of type `Common$`. This allows the user of any module to control the module's diagnostics, where its instances are allocated, how they are allocated, and (for gated modules) what gate it should use to protect critical sections.

Configuration example: The following configuration script specifies that the instance objects managed by the `Task` module in the `ti.sysbios.knl` package should be placed in the `".fast"` memory section and that only `ENTRY` diagnostics should be available at runtime.

```
var Task = xdc.useModule('ti.sysbios.knl.Task');
Task.common$.instanceSection = ".fast";
Task.common$.diags_ENTRY = Diags.RUNTIME_OFF
```

Note that by setting `Task.common$.diags_ENTRY` to `Diags.RUNTIME_OFF` we are both enabling `ENTRY` events and specifying that they are initially disabled; they must be explicitly enable at runtime. See Section 4.8 for additional information.

The Common\$ structure is defined as follows:

```
metaonly struct Common$ {
    Diags.Mode diags_ASSERT ;
    Diags.Mode diags_ENTRY ;
    Diags.Mode diags_EXIT ;
    Diags.Mode diags_INTERNAL ;
    Diags.Mode diags_LIFECYCLE ;
    Diags.Mode diags_USER1 ;
    Diags.Mode diags_USER2 ;
    Diags.Mode diags_USER3 ;
    Diags.Mode diags_USER4 ;
    Diags.Mode diags_USER5 ;
    Diags.Mode diags_USER6 ;
    Diags.Mode diags_USER7 ;
    Diags.Mode diags_USER8 ;
    IGateProvider.Handle gate ;
    Ptr gateParams ;
    IHeap.Handle instanceHeap ;
    String instanceSection ;
    ILogger.Handle logger ;
    CreatePolicy memoryPolicy ;
    Bool namedInstance ;
    Bool namedModule ;
}
```

The following fields in the Common\$ structure are used by the Diags and Log modules:

Field	Description
diags_ASSERT	The Diags.ASSERT bit of a module's diagnostics mask.
diags_ENTRY	The Diags.ENTRY bit of a module's diagnostics mask. (default=Diags.ALWAYS_OFF)
diags_EXIT	The Diags.EXIT bit of a module's diagnostics mask. (default=Diags.ALWAYS_OFF)
diags_INTERNAL	The Diags.INTERNAL bit of a module's diagnostics mask. (default=Diags.ALWAYS_OFF)
diags_LIFECYCLE	The Diags.LIFECYCLE bit of a module's diagnostics mask. (default=Diags.ALWAYS_OFF)
diags_USER1	The Diags.USER1 bit of a module's diagnostics mask. (default=Diags.ALWAYS_OFF)

Field	Description
diags_USER2	The Diags.USER2 bit of a module's diagnostics mask. (default=Diags.ALWAYS_OFF)
diags_USER3	The Diags.USER3 bit of a module's diagnostics mask. (default=Diags.ALWAYS_OFF)
diags_USER4	The Diags.USER4 bit of a module's diagnostics mask. (default=Diags.ALWAYS_OFF)
diags_USER5	The Diags.USER5 bit of a module's diagnostics mask. (default=Diags.ALWAYS_OFF)
diags_USER6	The Diags.USER6 bit of a module's diagnostics mask. (default=Diags.ALWAYS_OFF)
diags_USER7	The Diags.USER7 bit of a module's diagnostics mask. (default=Diags.ALWAYS_OFF)
diags_USER8	The Diags.USER8 bit of a module's diagnostics mask. (default=Diags.ALWAYS_OFF)
logger	The handle of the logger instance used by the module. All log events generated by the module are routed to this logger instance. See LoggerBuf and LoggerSys for two examples of logger instances provided by the xdc.runtime package.
namedInstance	If set to true, each instance object is given an additional field to hold a string name that is used when displaying information about an instance. Setting this to true increases the size of the module's instance objects by a single word but improves the usability of tools that display instance objects. (default=FALSE)
namedModule	If set to true, this module's string name is retained on the target so that it can be displayed as part of Log and Error events. Setting this to false saves data space in the application at the expense of readability of Log and error messages associated with this module. (default=TRUE)

The following fields in the Common\$ structure are used by the Memory module:

Field	Description
instanceSection	Identifies the section in which instances created by this module should be placed.
instanceHeap	Identifies the heap from which this module should allocate memory.
memoryPolicy	Specifies whether this module should allow static object creation only (STATIC_POLICY), dynamic object creation but not deletion (CREATE_POLICY), or both dynamic object creation and deletion (DELETE_POLICY).

The following fields in the Common\$ structure are used by the Gate module:

Field	Description
gate	A handle to the Gate implementation to be used by this module.
gateParams	Gate parameters for use by this module.

XDC Configuration

This chapter provides more information about XDC configuration syntax.

Topic	Page
5.1 More About the XDC Script Language	5-2
5.2 More About the xs Command	5-5
5.3 JavaScript Language	5-13
5.4 XDC Script Methods	5-15

5.1 More About the XDC Script Language

XDC scripts use a general-purpose programming language based upon the ECMA-262 Edition 3 standard—popularly known as JavaScript 1.5.

The `xs` command runs a JavaScript interpreter based on the Mozilla Rhino package. See the Rhino documentation to learn more about the particular features of this implementation of JavaScript. In particular, look for material on LiveConnect, which enables JavaScript programs to transparently invoke Java APIs. This powerful feature provides a mechanism to tap into the vast amount of functionality in the Java ecosystem.

5.1.1 Benefits of Static Configuration

XDC supports both static and dynamic creation of objects. Creating objects at run-time (dynamic) is easier, but extra code is required to support the object creation and deletion. Design-time configuration (static) provides the following benefits over run-time configuration:

- ☐ Improves run-time performance by reducing the time your program spends performing system setup.
- ☐ Can reduce program size by eliminating run-time code required to dynamically create and configure objects. For typical modules, functions to create and delete objects make up 50% of the code in the module.
- ☐ Optimizes internal data structures.
- ☐ Detects errors earlier by validating properties before compilation.
- ☐ Automatically sets a variety of properties that are dependent on other properties. This helps ensure that your configuration is valid.

5.1.2 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 store different data types at different times. 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.

5.1.3 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 XDC.
- ❑ Other object models can be defined for use with JavaScript. Instead of the DOM, XDC provides its own objects via modules contained by packages.
- ❑ JavaScript is not a part of Java. These are two different languages that have similar names for historical marketing reasons. However, XDC scripts can call Java functions to provide file services. JavaScript itself does not provide file services for security reasons on web browsers.
- ❑ XDC runs JavaScript only on the host PC or Linux machine. JavaScript code is never run on the target DSP.

5.1.4 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-international.org/publications/standards/Ecma-262.htm>
- Core JavaScript 1.5 Reference:
http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference
- 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>

5.2 More About the xs Command

Chapter 3 describes a number of XDC tools that you can run using the xs command. For example:

```
xs xdc.tools.cdcc.sg
```

The xs command is actually more than a central command for running various tools. It is actually a program that executes XDC scripts. The tools described in Chapter 3 are implemented as XDC scripts.

Since the xs command is an XDC script execution program, you can also use it to test XDC scripts or script commands. The xs command accepts the following command syntaxes:

- `xs [options] xdc-package [script-args ...]`
Script code is part of an XDC package (default: no option required) Section 5.2.1
- `xs [options] -c xdc-capsule [script-args ...]`
Script code is contained in an XDC capsule. Section 5.2.2
- `xs [options] -f xs-script-file [script-args ...]`
Script file is located anywhere on the filesystem. Section 5.2.3
- `xs [options] -m xdc-module [script-args ...]`
Script code is part of an XDC module. Section 5.2.4
- `xs [options] -i`
Specify a script interactively. Section 5.2.5
- `xs [options] -g`
Debug a script with a graphical tool. Section 5.2.6

The basic model for using the xs command is to specify a script to execute along with optional arguments to be passed to the script. The variations shown here are due to how the “script” is located by xs.

Important: Running a script in one of these modes is different from running it with the configuro utility. When a configuration script is run, it is run in a context (called an “object model” or “domain”) in which the global object “Program” is initialized based on the user-specified target and platform parameters. Several distinct domains are defined by XDC; each defines a top-level object that provides a context for the execution of scripts. Currently, there is a “cfg” configuration domain, a “bld” build domain, and a “rov” runtime domain.

The options you can use on the command line beyond the basic -c, -f, -m, -i, and -g mode-selection options are described in Section 5.2.7.

5.2.1 Running a Script Defined by a Package

An XDC package that contains a `main()` method in a module named `Main` can be invoked directly as shown below.

```
xs sample.stuff world
```

The package is "sample.stuff" and is located along the `XDCPATH` package path. The script must be contained in the file `Main.xs`. The argument "world" is sent to the `main()` function. Note that this is equivalent to using the following invocation using the `-m` module option:

```
xs -m sample.stuff.Main world
```

If none of the options `-m`, `-f`, or `-c` is specified, the argument given to `xs` after [options] is assumed to be a package name. This package must have a module named `Main` that defines a `main()` function.

Arguments are passed as function parameters to the `main()` function that is required in the script file.

5.2.2 Running a Script Defined by a Capsule

An XDC capsule is simply a file that contains an XDC script that can be loaded and used via the `xdc.loadCapsule()` method. The capsule that contains a `main()` method can be run using `xs` by specifying the capsule with the `-c` option. For example:

```
xs -c sample/stuff/hello.xs world
```

In this example, the script `hello.xs` is contained in a package called `sample.stuff` and is located along the package path. Note that instead of using the package name directly, the "." separator is replaced with a forward slash "/" (on both Linux and Windows).

There are no requirements on the naming of the script file (different file extensions can be used) or the actual location of the file within a package (subdirectories can be used).

Arguments are passed as a single string array parameter to the `main()` function that is required in the script file. Each argument passed on the `xs` command line is a separate string in the array passed to `main()`.

5.2.3 Running a Script Defined by a File

As with the previous examples, the following command runs the code in the file `hello.xs` contained in the `sample.stuff` package.

```
xs -f sample/stuff/hello.xs world
```

For example, if `hello.cfg` contains the line, the text “hello world” is echoed to the standard output of the command shell:

```
print("hello " + arguments[0]);
```

The only difference between the capsule and the file techniques is that a capsule defines a `main()` function that is run, whereas a file runs when it is loaded. The capsule is a better way to encapsulate scripts because the capsule can be loaded from other scripts that need finer control over when the script is run.

In both the file and capsule techniques, it is possible to supply an absolute path to the capsule or file on the `xs` command line. In this case, the package path is not used.

The `-f` option specifies that the next argument is the name of a script to execute. Any remaining arguments are passed to the script.

If you start a script using the `-f` option, arguments supplied on the command line can be retrieved using the standard JavaScript “arguments” array. This is an array-like object containing the arguments passed to the function, indexed beginning at 0. The “length” property of the object can be used to determine the number of arguments actually passed. This is true even for scripts that do not have a function defined, but rather just begin execution with the first line in the file.

5.2.4 Running a Script Defined by a Module

A meta-only module that contains a method named “main” can be run using `xs` by specifying the module with the `-m` option. For example:

```
xs -m sample.stuff.Hello world
```

The package in this example is `sample.stuff` and is located along the package path. The module `Hello` must contain a “main” function, which must be specified in the corresponding `Hello.xdc` and implemented in the `Hello.xs` file.

The `-m` option specifies that the next argument is the name of a module, and that the main function of that module should be run. Any remaining arguments on the command line are passed to the script.

Arguments are passed as a single string array parameter to the `main()` function that is required in the script file.

5.2.5 Running an Interactive JavaScript Session

If you use the `-i` option on the command line, you enter the interactive JavaScript shell, which executes statements you type at the `js>` prompt. This mode echoes the results of print statements and expressions to your terminal window. This can be useful for testing XDC script statements.

Thus, the following command starts the XDC script interpreter and issues a prompt for an interactive statement.

```
xs -i
```

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

```
% xs -i
js> xdc.includeFile("xdc/services/io/File.xs")
```

The results of this statement are similar to those if you executed the script from the command line.

Note: Be aware that running a script in interactive mode is different from running it with the `configuro` utility, because when a configuration script is run, it is run in a context in which the global object `Program` is initialized based on the user-specified target and platform parameters.

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

After script execution ends, you are still in the JavaScript shell and are free to execute any methods or use any variables defined by the script.

Alternatively, instead of including a script, you can type commands directly. 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:

```
% xs -i
js> textvar = "hello world";
js> print(textvar);
```

To exit from the interactive shell, type `quit`. The `quit` command cannot be executed in a XDC script; it is only available in the interactive shell. The keywords `quit` and `exit` are reserved for future use in XDC scripts.

5.2.6 The Rhino Script Debugger

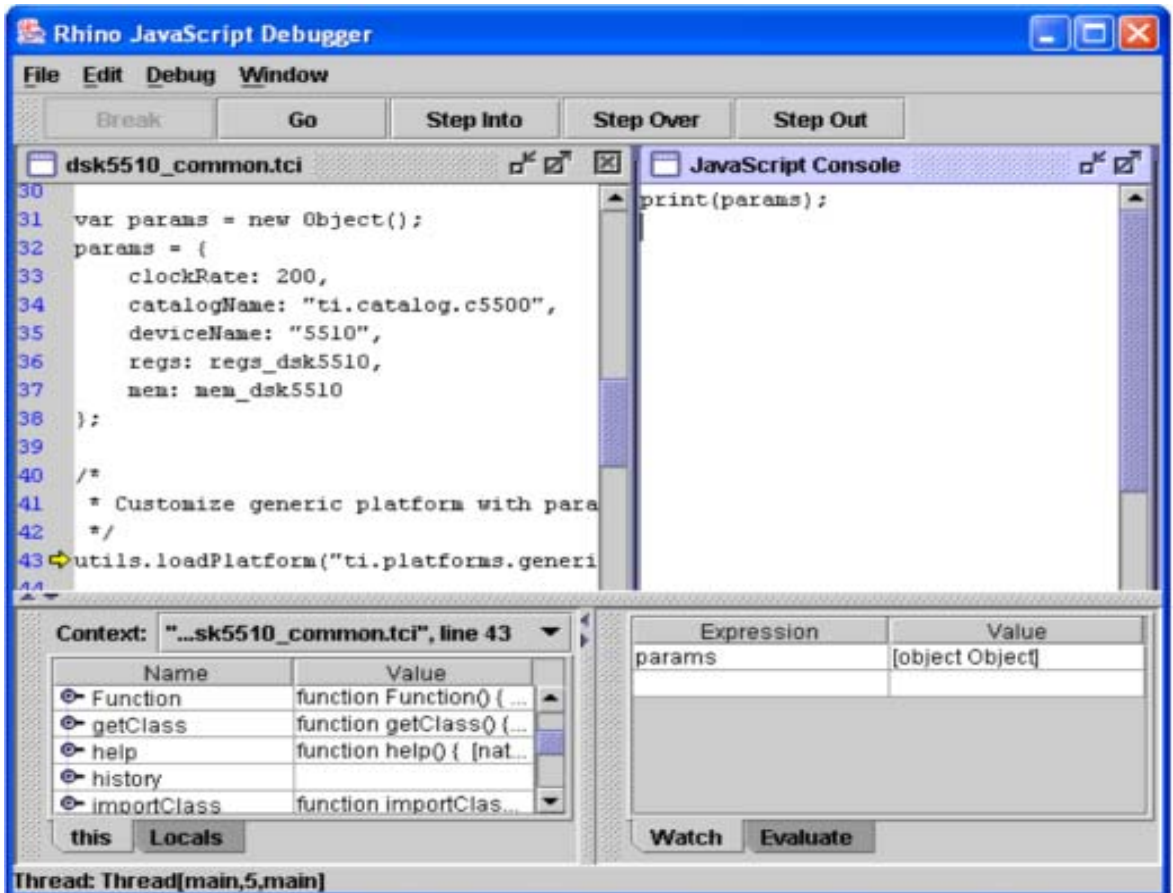
You can use the Rhino GUI debugger to test a script. Rhino is an open-source implementation of JavaScript. It is written entirely in Java (<http://www.mozilla.org/rhino>).

The command-line syntax for the GUI debugger is the same as for the rest of the xs command line, except that you add the -g option:

```
xs -g[=i] myscript.cfg
```

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 5–1 Rhino GUI Debugger Window



Here are some important hints for using the Rhino debugger:

- ☐ The **Debug** menu contains three check boxes: Break on Exception, Break on Function Enter, and Break on Function Return. If the -g option is used on the xs command line, Break on Exception and Break on Function Enter are enabled within the debugger. Only Break on Exception is enabled if you use the -g=i option.
- ☐ When Break on Exception is enabled, non-fatal errors are displayed in exception dialog boxes as they occur.
- ☐ Use **Windows->Tile** or **Windows->Cascade** to open windows for the main script, all its included files, and internal XDC script files.
- ☐ Break on Function Enter and Break on Function Return cause the debugger to stop at entry and exit of each JavaScript function. You may want to deselect these options if you just want to run to a specific breakpoint you have set.
- ☐ You can set a breakpoint by clicking in the gray column next to the line number of the script. You can only set breakpoints on lines that contain executable statements.
- ☐ Choose **Windows->Console** to open the Console window, which receives standard out and standard error.
- ☐ We recommend that you set a breakpoint at the last statement in your script file to see any displayed messages in the Console window before the debugger finishes running the script.
- ☐ The Rhino debugger allows you to browse and view XDC script objects, however the list is not always clear or complete. You can add print() statements to the script. The results of print() statements are displayed in the Console window.

5.2.7 Further Command-Line Options

In addition to the basic -c, -f, -m, -i, and -g mode-selection options for the xs command, you can use a number of additional options and command-line syntaxes. These are:

- ☐ The --xdcpath option to specify the package path. Section 5.2.7.1
- ☐ The -D option to specify environment variables. Section 5.2.7.2
- ☐ Passing arguments to a script. Section 5.2.7.3
- ☐ @ to take options and arguments from a file. Section 5.2.7.4

5.2.7.1 Specifying the Package Path

Normally, you specify the list of locations where XDC should search for packages using the `XDCPATH` environment variable.

Alternatively, you can specify these locations using the `--xdcpath` option of the `xs` command.

All subdirectories in the locations referenced are searched for packages.

For example, the following command adds auto-generated documentation for your own packages to the beginning of the CDOC online help display.

```
xs --xdcpath=c:/mypackages xdc.tools.cdoci.sg
```

If you want to specify more than one directory with the `--xdcpath` option, use semicolons to separate the items in the list.

If you use a UNIX-like shell, you will have to quote the semicolons to prevent the shell from interpreting these characters as command separators. For example:

```
xs --xdcpath="/usr/home/me/packages;/tools/packages" ...
```

5.2.7.2 Setting Environment Variables

It can be useful to set, what is in effect, a “global” variable whose value is specified on the command line when running an XDC script. This can be done using the `-D` option to `xs`. Your script can then retrieve that value from the global “environment” object in the script being run without having to save and parse the command line arguments array. The environment is a table indexed by the name of the environment variable.

For example, this command makes an environment variable called “mode” available to the script. Your script would then need to handle this variable appropriately:

```
xs -Dmode=debug xdc.tools.configuro <options> mycfg.cfg
```

Note that these global values are not the same as the command shell’s environment variables.

As another example, the following command sets an environment variable called “junk” to the value “car”.

```
xs -Djunk=car -f envvar.xs
```

The environment could be accessed by the script and printed as shown below:

```
print("Env var: " + environment["junk"]);
```

This example shows how to set and retrieve an environment variable. This approach works for any of the methods for finding and running a script from the xs command line.

5.2.7.3 Passing Arguments

If you start a script using the -f option, arguments supplied on the command line can be retrieved using the standard JavaScript “arguments” array. This is an array-like object containing the arguments passed to the function, indexed beginning at 0. The “length” property of the object can be used to determine the number of arguments actually passed. This is true even for scripts that do not have a function defined, but rather just begin execution with the first line in the file. For example:

```
for (var i = 0; i < arguments.length; i++ ) {  
    print("argument " + i + " = " + arguments[i]);  
}
```

If you start a script using the default method, -c, or -m, arguments are passed as function parameters to the main() function that is required in the script file. For example,

```
function main(args)  
{  
    for (var i = 0; i < args.length; i++ ) {  
        print("argument " + i + " = " + args[i]);  
    }  
}
```

5.2.7.4 Taking Options and Arguments from a File

The @ option enables you to reuse common command-line options and arguments by saving them in a file. This option reads the specified file and include its contents as arguments to xs.

One common scenario is specifying a user-specific portion of the XDC package path. If several repositories (directories containing packages) are being used, the argument to the --xdcpath option can get quite long and tedious to re-enter each time you use the xs command.

Alternately, you can create a file that specifies arguments, one argument per line. Suppose, for example, you create a file named opts-file that contains the following lines:

```
--xdcpath  
dir-path1;dir-path2;dir-path3
```


Then, run the `xs` command as follows:

```
xs @opts-file ...
```

As a result, the `xs` command effectively is invoked as follows:

```
xs --xdcpath "dir-path1;dir-path2;dir-path3" ...
```

In general, each line of the file supplied with the `@` option is added to the argument list of the `xs` command. The process is recursive so that if there is a line with an `@` option, that file is opened and each line is added to the argument list, and so on.

5.3 JavaScript Language

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

This document does not provide details on the syntax of the JavaScript language. However, several concepts are important when using JavaScript for XDC configuration. This section provides an overview of such concepts. Also see Section 5.1.4 for JavaScript reference sources.

5.3.1 Object and Property Naming and Referencing

JavaScript is object-oriented. The object model is separate from the JavaScript language, but object handling syntax is part of the language. Modules are treated as object classes in XDC scripts. Certain modules can have instances created.

Modules 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 `xdc.runtime.Memory.defaultHeapSize` to refer to the `defaultHeapSize` property of the `Memory` module in the `xdc.runtime` package.

5.3.2 Error Handling and Exceptions

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

Scripts can throw exceptions. Exceptions should be used for confirmed errors where an action has failed. Exceptions thrown by a script can be caught in a script. Uncaught exceptions cause a script to terminate execution and return with a non-zero status. Exceptions are always written to the stderr location, unless they are caught by a script and handled.

In interactive XDC script mode, 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.

To throw an exception, scripts use the "throw" keyword. This example throws an exception if the defaultHeapInstance property of the xdc.runtime.Memory module is null.

```
if (xdc.runtime.Memory.defaultHeapInstance = NULL) {  
    throw new Error("A default heap is required!");  
}
```

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, 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 + ".cfg";  
    xdc.includeFile(fileName);  
}  
catch (e) {  
    print(e + "\nNo " + fileName + " file.");  
}
```

5.4 XDC Script Methods

The XDC script environment provides a number of utility methods available to any script. The following sections describe these methods and point to further documentation.

5.4.1 The print() Method

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

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

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

5.4.2 Methods from Java Packages

For security reasons, JavaScript does not provide file services. In a web browser, the lack of file services prevents most forms of file access on your computer. In XDC scripts, access to Java methods is provided through the Rhino JavaScript interpreter via LiveConnect. The implementation provides unrestricted use of any Java package including, for example, the java.io package.

Calls to Java's java.io package 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>.

5.4.3 Methods and Properties from the xdc Package

The xdc package itself provides a number of methods for use in XDC scripts. This section lists these methods.

- **xdc.csd()** returns the directory of the script file that is currently being loaded. There is always exactly one script file loading. This file is not necessarily the same as the file containing the function currently executing. During the load of a script, the script can refer to its location and possibly load/read/write files relative to this location. When a function defined by the script runs, the directory of the currently loading script may not be the directory of the script where the function is defined.
- **xdc.exec(command, attrs, status).** Executes an arbitrary shell command and optionally puts output into a file. Both stderr and stdout are combined into a single output stream. Returns the command's exit status.
 - "command" is the command string to execute.
 - "attrs" specifies an optional set of attributes used to control the environment of the command:
 - ◆ attrs.envs may be an array of environment variable settings.
 - ◆ attrs.cwd may be the command's current working directory.
 - ◆ attrs.outName may be the name of the output file to create. If the file exists, output is appended to the end of the file.
 - ◆ attrs.filter may be a regular expression to filter command output.
 - "status" is an optional output parameter. If it is non-NULL, the following fields are set:
 - ◆ status.output is a string containing all output from the command.
 - ◆ status.exitStatus is the exit status of the command.
- **xdc.findFile(filename).** Finds the file specified by filename. This method searches for filename according to the following algorithm. It returns the path to the file. The path returned may be absolute or relative. If the file cannot be found in these directories, NULL is returned.
 - a) If filename is an absolute path or begins with "./", xdc.findFile() returns filename if it exists. Otherwise xdc.findFile() returns NULL.
 - b) If filename is a relative path that does not start with "./": the following directories are searched in order:
 - i) the current script directory (see xdc.csd())
 - ii) the directories specified by the XDCPATH
 - filename is a string specifying a file to locate.

- **xdc.getPackageBase(package_name).** Returns the absolute path to the specified package's base directory. This method returns the full path string to the package. If the package cannot be found, this method throws an exception.
 - package_name is a '.' separated package name. For example, xdc.runtime.
- **xdc.getPackageRepository(package_name).** Returns the absolute path to the specified package's repository directory. This method returns the full path string to the repository. If the repository cannot be found, this method throws an exception.
 - package_name is a '.' separated package name. For example, xdc.runtime.
- **xdc.includeFile().** An XDC script can load another script file using the xdc.includeFile() method. For example, this command includes the generic.cfg file

```
xdc.includeFile("generic.cfg");
```

This method normally searches the package path (XDCPATH) for the file specified. However, if you specify a path to the file beginning with "./", XDC looks for the file relative to the current script's location. For example:

```
xdc.includeFile("../project/includes/file.cfg");
```

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

If you do not specify a file extension, this function looks for the specified file with an extension of .cfg.

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

- **xdc.loadPackage(package_name).** Loads the specified package and returns a package object. This method finds a package's schema file (package/<package_name>.xds), loads it, initializes packageBase and packageRepository, and calls the package's initialization function. If the package cannot be found, this method throws an exception.
 - package_name is a '.' separated package name. For example, xdc.runtime.

- **xdc.module(module_name).** Loads the specified module for use in a script.

The only difference between the `xdc.useModule()` and `xdc.module()` methods is that `useModule()` sets the returned module object's `$used` flag to a non-zero value. During configuration, only modules that have `$used` set to non-zero are linked into the final application. So, it is possible to get a module's object and use its meta-only methods via `xdc.module()` without the module's target methods becoming part of the application.

- **xdc.useModule(module_name).** Enables a module contained in a package for use in the script.
 - `module_name` is a '.' separated module name. For example, `xdc.runtime.System`.

Target Compiler Options

This appendix lists the compiler options used by default for the various targets you can select.

Topic	Page
A.1 Compiler Options	A-2

A.1 Compiler Options

The target you specify when you use configuro determines the compiler options added to the compiler command line. The available targets have the following default compiler options:

Target	Compiler Options and Notes
ti.targets.arm.Arm7	"-me -mv4 --abi=ti_arm9_abi"
ti.targets.arm.Arm7_big_endian	"-mv4 --abi=ti_arm9_abi"
ti.targets.arm.Arm9	"-me -mv5e --abi=ti_arm9_abi" Uses little endian
ti.targets.arm.Arm9t	"-me -mt -mv5e --abi=ti_arm9_abi" Uses little endian
ti.targets.C28	"-v28 -mo"
ti.targets.C28_large	"-v28 -DLARGE_MODEL=1 -ml"
ti.targets.C55	""
ti.targets.C55_huge	"--memory_model=huge"
ti.targets.C55_large	"-ml"
ti.targets.C62	""
ti.targets.C62_big_endian	"-me"
ti.targets.C64	"-mv6400"
ti.targets.C64_big_endian	"-me -mv6400"
ti.targets.C64P	"-mv64p"
ti.targets.C64P_big_endian	"-me -mv64p"
ti.targets.C67	"-mv6700"

Target	Compiler Options and Notes
ti.targets.C67_big_endian	"-me -mv6700"
ti.targets.C67P	"-mv67p"
ti.targets.C67P_big_endian	"-me -mv67p"
ti.targets.MSP430	"-vmSP"
gnu.targets.Linux86 (See Note 1)	"-nostdinc -B\$(rootDir)/lib/gcc-lib/\$(GCCTARG)/\$(GCCVERS)/" Native Linux target using GCC compiler
gnu.targets.Mingw (See Note 1)	"-nostdinc -B\$(rootDir)/lib/gcc-lib/\$(GCCTARG)/\$(GCCVERS)/" Native Windows target using Mingw GCC compiler
gnu.targets.MVArm9	"" Embedded Linux target using MontaVista GCC compiler
gnu.targets.Sparc (See Note 1)	"-nostdinc -B\$(rootDir)/lib/gcc-lib/\$(GCCTARG)/\$(GCCVERS)/" Native Solaris target using GCC compiler
gnu.targets.UCArm9	"" ARM target with uClibc binary compatibility and GCC compiler for Linux
microsoft.targets.Net32	"-nologo -c" Microsoft .Net 32-bit native target using Microsoft VC++ 8
microsoft.targets.VC98	"-W3 -Zp1 -DWIN32 -D_DLL -D_AFXDLL -DEXPORT=" Microsoft Windows 32-bit target using Visual C/C++ 6.x compiler
microsoft.targets.Win32	"-W3 -Zp1 -DWIN32 -D_DLL -D_AFXDLL -DEXPORT=" Microsoft Windows 32-bit target using Visual C/C++ 6, 7, or 8 compiler

- 1) Sequences of the form "\$(...)" refer to properties of the target:
- n \$(rootDir) is the value of the target's rootDir property. That is, it is the installation directory of the compiler as specified by the user.
 - n \$(GCCTARG) is the GSSTARG property of a GNU target. That is, it is the output from running "gcc -dumpmachine".
 - n \$(GCCVERS) is the GCCVERS property of a GNU target. That is, it is the output from running "gcc -dumpversion".

Index

C

catch keyword 5-14
catching exceptions 5-14

D

data types 5-2
debugging
 GUI debugger 5-9
design-time configuration 5-2
directory path 5-17
Document Object Model (DOM) 5-3
documentation, other 5-4
dot (.) notation 5-13
dynamic objects 5-2

E

ECMA-262 5-3
error handling 5-14
exceptions 5-14
 catching 5-14
 throwing 5-14
exit keyword 5-8
exit status 5-14
exiting from tconf 5-8

F

file services 5-3, 5-15

G

GUI debugger 5-9
 command line 5-9

I

interactive tconf
 command line 5-8

J

Java 5-3
 documentation 5-4
 Rhino written in 5-9
JavaScript
 documentation 5-4
 language issues 5-13
 misconceptions 5-3
 overview 5-2
 Rhino interpreter 5-9

L

LiveConnect 5-15
loosely-typed language 5-2

N

names
 variables 5-3

O

object-orientation 5-13

P

path
 separators 5-17
pointers 5-3
print() method 5-15
 Rhino GUI 5-9

properties 5-13

Q

quit command 5-8

R

reserved keywords 5-8

Rhino 5-9

S

stdout location 5-15

T

throw keyword 5-14

throwing exceptions 5-14

try keyword 5-14

V

variable names 5-3

variable types 5-2