Codec Engine Application Developer User's Guide

Literature Number: SPRUE67D
September 2007

TEXAS INSTRUMENTS

Preface

About This Book

The Codec Engine is a set of APIs that you use to instantiate and run xDAIS algorithms. A VISA interface is provided as well for interacting with xDM-compliant xDAIS algorithms.

The intended audience for this document is the embedded-OS application developer treating the DSP-side of DaVinci as a black box represented by an API.

Additional Documents and Resources

You can use the following sources to supplement this user’s guide:

❏ *Codec Engine Server Integrator User's Guide* (SPRUED5)

❏ *Codec Engine Algorithm Creator User's Guide* (SPRUED6)

❏ Codec Engine Application (API) Reference Guide. CE_INSTALL_DIR/docs/html/index.html

❏ Configuration Reference Guide. CE_INSTALL_DIR/xdoc/index.html

❏ Example Build and Run Instructions. CE_INSTALL_DIR/examples/build_instructions.html

❏ *xDM API Reference.* XDAIS_INSTALL_DIR/docs/html/index.html

❏ DaVinci EVM Home at Spectrum Digital: http://c6000.spectrumdigital.com/davincievm/revc/

❏ TI Linux Community for DaVinci Processors: http://linux.davincidsp.com

❏ *xDAIS-DM (Digital Media) User Guide* (SPRUEC8)

❏ *TMS320 DSP Algorithm Standard Rules and Guidelines* (SPRU352)

❏ *TMS320 DSP Algorithm Standard API Reference* (SPRU360)

❏ *TMS320 DSP Algorithm Standard Developer’s Guide* (SPRU424)

❏ *TMS320 DSP Algorithm Standard Demonstration Application* (SPRU361)

Notational Conventions

This document uses the following conventions:

- ❑ Program listings, program examples, and interactive displays are shown in a `special typeface`. Examples use a **`bold version`** of the special typeface for emphasis; interactive displays use a **`bold version`** of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).
- ❑ Square brackets ([and]) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets. Unless the square brackets are in a **bold** typeface, do not enter the brackets themselves.
- ❑ This manual uses an icon like the one to the left to identify information that is specific to a particular type of system. For example, this icon identifies information that applies if you are using Codec Engine on a dual-processor GPP+DSP system.

GPP+DSP

Trademarks

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

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

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

Linux is a registered trademark of Linus Torvalds.

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

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

Contents

Chapter 1

Codec Engine Overview

This chapter introduces the Codec Engine.

Topic **Page**

1.1 What is the Codec Engine?

From the application developer’s perspective, the Codec Engine is a set of APIs that you use to instantiate and run xDAIS algorithms. A VISA interface is provided as well for interacting with xDM-compliant xDAIS algorithms.

The API is the same for all of the following situations:

❑ The algorithm may run locally (on the GPP) or remotely (on the DSP).

❑ The system may be a GPP+DSP, DSP-only, or GPP-only system.

❑ All supported GPPs and DSPs have the same API.

❑ All supported operating systems have the same API. For example, Linux, PrOS, VxWorks, DSP/BIOS, and WinCE.

GPP+DSP

This manual uses an icon like the one to the left to identify information that is specific to a particular type of system. For example, this icon identifies information that applies if you are using Codec Engine on a dual-processor GPP+DSP system.

xDM is the eXpressDSP Algorithm Interface Standard for Digital Media. It is sometimes referred to as xDAIS-DM.

Any xDM algorithm is compliant with the eXpressDSP Algorithm Interface Standard (xDAIS). Additionally, it implements the xDAIS-DM (xDM) interface, an extension to the xDAIS standard that provides support for digital media encoders, decoders, and codecs. The xDM specification defines APIs for digital media codecs by class, with extensions defined for video, imaging, speech, and audio codec classes.

The xDM interfaces divide codec algorithms into four classes: Video, Image, Speech, and Audio (VISA). VISA reflects this xDM interface. One set of APIs is provided per codec class. Thus, MP3 can be replaced with WMA without changing the application source code. Only the configuration needs to be changed.

The Codec Engine also supports real-time, non-intrusive visibility into codec execution. It provides APIs for accessing memory and overall CPU usage statistics and execution trace information.

The Codec Engine runtime is supplied in binary form. Thus, application libraries built with same Codec Engine release are always compatible.

1.2 Why Should I Use It?

The Codec Engine is designed to solve some common problems associated with developing system-on-a-chip (SoC) applications. The most significant problems include:

- ❏ Debugging in a heterogeneous processor environment can be painful. There are multiple debuggers and complex bootstrapping.
- ❏ Different implementations of the same algorithm, such as MP3, have different APIs. Changing to a more efficient algorithm involves significant recoding.
- ❏ Portability issues are compounded with two processors. You may want to port to a different board with a newer DSP or a newer GPP.
- ❏ Some algorithms may run on either the GPP or the DSP. To balance system load, “low complexity” algorithms can run on a GPP, but the definition of “low” changes over time. If changing the location where the algorithm runs were easy, you wouldn’t have to weigh performance issues against the difficulty of changing the application.
- ❏ For market success, most applications need to support multiple codecs to handle the same type of media. For example, an application might need to support three or four audio formats.

GPP+DSP

- ❏ Programmers with a GPP (general-purpose processor) view typically don’t want to have to learn to be DSP programmers. And, they don’t want to have to worry about a DSP’s complex memory management and DSP real-time issues.

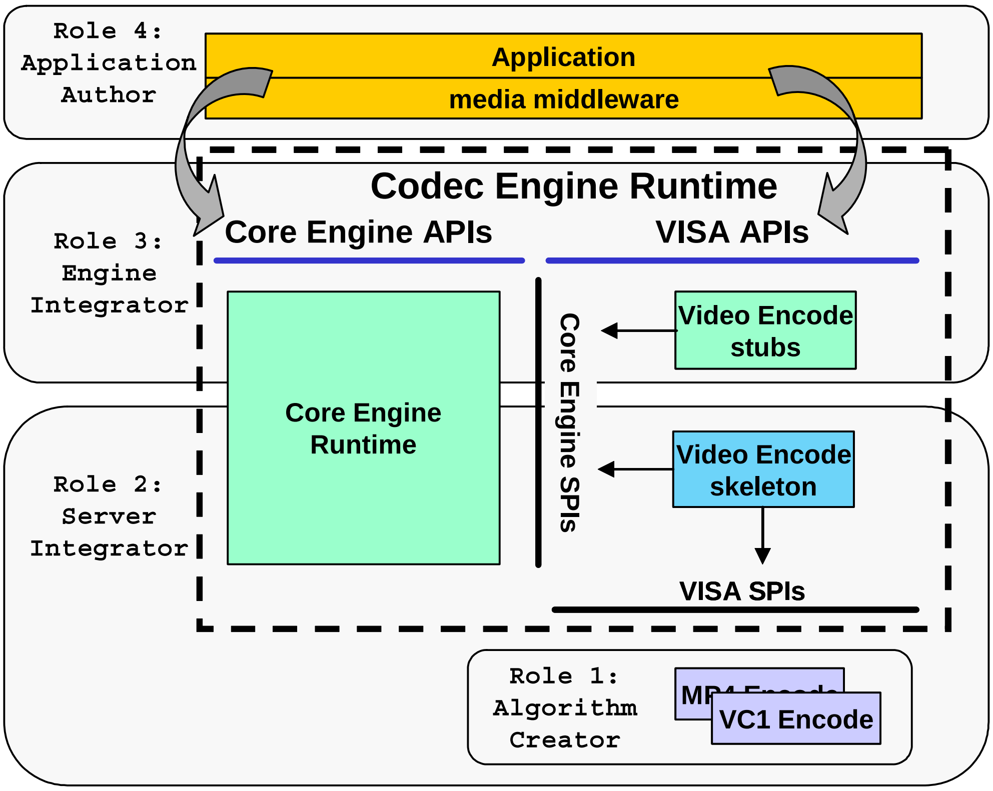
The Codec Engine addresses these problems by providing a standard software architecture and interfaces for algorithm execution. The Codec Engine is:

- ❏ **Easy-to-use.** Application developers specify what algorithm needs to be run, not how or where.
- ❏ **Extensible and configurable.** New algorithms can be added by anyone, using standard tools and techniques.
- ❏ **Portable.** The APIs are target, platform, and even codec independent.

1.3 Where Does the Codec Engine Fit into My Architecture?

The application code (or the middleware it uses) calls the Codec Engine APIs. Within the Codec Engine, the VISA APIs use stubs and skeletons to access the core engine and the actual codecs, which may be local or remote.

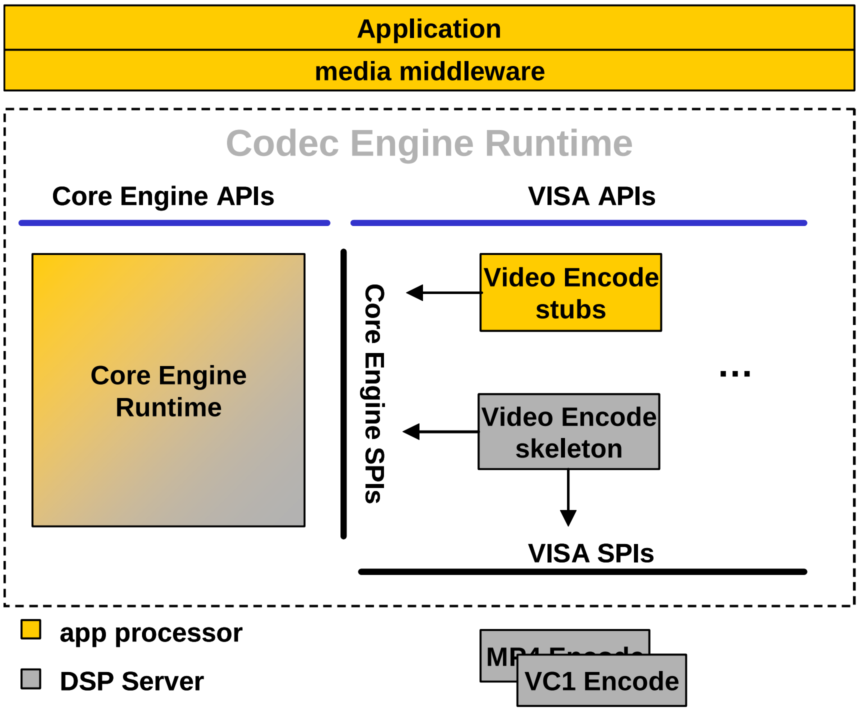
The following figure shows the general architecture of an application that uses the Codec Engine. It also shows the user roles involved in creating various portions of the application. See Section 1.4, *What Are the User Roles?* for more on user roles.



The application (or middleware it uses) calls the core Engine APIs and the VISA APIs. The VISA APIs use stubs to access the core engine SPIs (System Programming Interfaces) and the skeletons. The skeletons access the core engine SPIs and the VISA SPIs. The VISA SPIs access the underlying algorithms.

GPP+DSP

The following figure is a modification of the previous diagram that shows how this architecture is distributed in a GPP+DSP system. In this example, yellow portions run on the GPP, and grey portions run on the DSP. That is, the video encoder skeleton and the video encoder codecs are on the DSP and the application and video encoder stubs are on the GPP.



Since Codec Engine is flexible, alternate diagrams could be shown for GPP-only and DSP-only systems.

1.4 What Are the User Roles?

The Codec Engine has several customer use cases, from application developers to codec authors. In some cases, these roles may be played by a single person. In other development environments, a different developer may be assigned each role. This topic describes the primary roles that Codec Engine users will play.

Because Codec Engine is very portable and configurable and can run in many different environments, the descriptions of these roles are intentionally generalized. When applicable, specific hardware and software environments are described after the general descriptions.

This document describes the APIs available to the Application Author. Other documents are referenced for the other roles.

1.4.1 Algorithm Creator

The Algorithm Creator is responsible for creating an xDAIS algorithm and providing the necessary packaging to enable these algorithms to be consumed and configured by Codec Engine.

If the codec is xDM-compliant, Codec Engine's VISA APIs support remote execution without additional support. However, if the codec is not xDM-compliant and the codecs support remote execution, the Algorithm Creator should supply Codec Engine skeletons and stubs.

The Algorithm Creator uses xDAIS and the XDC Tools. Using these, the Algorithm Creator generates a codec library with the IALG and optional IDMA3 interface symbols exported. This person also creates an XDC module that implements the ti.sdo.ce.ICodec interface.

The Algorithm Creator hands a released Codec package to the Server Integrator. This package includes a module that implements ti.sdo.ce.ICodec, as well as the libraries that contain the algorithm's implementation.

The Algorithm Creator uses the following resources:

❏ *Codec Engine Algorithm Creator User's Guide* (SPRUED6)

❏ *Codec Engine SPI Reference Guide*.
CE_INSTALL_DIR/docs/spi/html/index.html

❏ *xDAIS-DM (Digital Media) User Guide* (SPRUEC8)

❏ *xDM API Reference.* XDAIS_INSTALL_DIR/docs/html/index.html

❏ *TMS320 DSP Algorithm Standard Rules and Guidelines* (SPRU352)

❏ *TMS320 DSP Algorithm Standard API Reference* (SPRU360)

❏ *TMS320 DSP Algorithm Standard Developer’s Guide* (SPRU424)

❏ Example codecs

1.4.2 Server Integrator

GPP+DSP

To support Engines with remote codecs, a Codec Server must be created. The Codec Server integrates and configures the various components necessary to house the codecs (e.g. DSP/BIOS, Framework Components, DSP/BIOS Link drivers, codecs, Codec Engine, etc.) and generates an executable.

There are two configuration steps that the Codec Server Integrator must perform, one to configure DSP/BIOS (through a Tconf script) and one to configure "the rest" of the components (through XDC configuration of Framework Components, DSP/BIOS Link, Codec Engine, etc).

The Server Integrator receives the various Codec packages from Algorithm Creators. This person uses Codec Engine and its dependent packages (DSP/BIOS, DSKT2, etc) along with the XDC Tools to create the following:

❏ A server configuration file (.cfg)

❏ A server DSP/BIOS configuration file (.tcf)

❏ A simple main() routine to do minimal initialization

❏ A DSP executable created by executing the configuration scripts and compiling the output. This executable is a Codec Server.

The Server Integrator hands the DSP executable to the Engine Integrator (preferably as a Codec Server package. The Server Integrator should also provide a list of the codecs in the Codec Server, as well as documentation about how they've been configured (for example, thread priorities and resource configuration).

The Server Integrator uses the following resources:

❏ *Codec Engine Server Integrator's Guide* (SPRUED5)

❏ Configuration Reference Guide.
CE_INSTALL_DIR/xdoc/index.html

❏ Example Codec Servers

On GPP-only and DSP-only platforms the Codec Server is not used, so this role is not required.

1.4.3 Engine Integrator

The Engine Integrator defines various Engine configurations. This includes the names of the Engines, as well as the codecs and their names within each Engine, whether each codec is local or remote relative to the application, which groups each codec should be integrated into (for environments which support resource sharing), the name of the Codec Server image if a particular Engine contains remote codecs, etc. This is done via an XDC configuration script (*.cfg).

This script, when run, generates code and build instructions appropriate for the configuration.

The Engine Integrator receives the name of a Codec Server and a list of the codecs it contains from the Server Integrator. Using these, this person creates an Engine configuration file (.cfg) that may reference a Codec Server.

On DSP-only and/or GPP-only platforms, the Codec Server is not used, and all codecs will be configured to run locally (that is "local:true").

The Engine Integrator hands the Engine configuration file to the Application Author.

The Engine Integrator uses the following resources:

❏ Chapter 5 of this manual, *Integrating an Engine*.

❏ Configuration Reference Guide.
CE_INSTALL_DIR/xdoc/index.html

❏ Example Build and Run Instructions.
CE_INSTALL_DIR/examples/build_instructions.html

❏ Example configuration scripts (*.cfg)

1.4.4 Application Author

The application uses the Codec Engine APIs (Engine_, VISA, and other utility APIs) to create/delete preconfigured Engine instances, create/delete and interact with codecs, acquire buffers appropriate for the codecs, etc.

Since Codec Engine doesn't perform any I/O, the application is responsible for handling I/O. This includes such task as file access (for example, open/read/write/seek/close) and driver interaction (for example, open/close/ioctl and buffer management).

The Application Author is responsible for building the application code and for linking "the appropriate content" into the executable image.

The Application Author receives the following:

- ❏ Various Codec packages from Algorithm Creators
- ❏ A Codec Server DSP executable from the Server Integrator if codec will run on a DSP
- ❏ An Engine config file (.cfg) from the Engine Integrator

The Application Author writes application code, generates output from the Engine configuration file (.c and .xdl output files) using the XDC Tools, and compiles the application code and generated files. This person then links the files, including the generated linker command file (.xdl) into an executable. The end result is the application executable.

The process for generating an application executable is highly dependant on the application's operating system. If the application runs on the DSP using DSP/BIOS, for example, a .tcf file is needed to configure the DSP/BIOS kernel as well. If the application runs on Linux, the application does not need to configure the operating system.

In addition to this manual, the Application Author uses the following resources:

- ❏ Codec Engine API Reference.
  CE_INSTALL_DIR/docs/html/index.html
- ❏ Example Build and Run Instructions.
  CE_INSTALL_DIR/examples/build_instructions.html

1.4.4.1 Editing a DSP/BIOS Configuration Script

DSP-only

For DSP-only applications, the Application Author creates a static DSP/BIOS configuration in a .tcf file, as described in the *DSP/BIOS Tconf User's Guide* (SPRU007) and the DSP/BIOS online help. The syntax used in Tconf configurations is based on JavaScript.

To create a .tcf file for an application, follow these steps:

1) Copy local.tcf from one of the example applications located at CE_INSTALL_DIR/examples/apps. This configuration file, in combination with the app_common.tci and bios.tci files that it imports, statically configures several aspects of the DSP/BIOS kernel, such as:
   - ■ base DSP/BIOS kernel
   - ■ memory section names, sizes, and locations
   - ■ platform-specific attributes such as clock rates
   - ■ task manager and dynamic heap allocation
   - ■ 'C64x+ L1 cache and its corresponding memory segment

You can learn more about these modules and attributes in the DSP/BIOS online help or the *C6000 DSP/BIOS API Reference* (SPRU403).

2) Optionally open this .tcf file with a text editor.

3) Make any changes your application requires and save the file. You can add your own non-Codec Engine configuration items here if you need to add your own functionality for the application.

1.5 Where Can I Get More Information?

The release_notes*.html file at the top of the Codec Engine installation provides general information, information about changes in the latest version, devices supported and validation information, known issues, and links to online documentation provided with the Codec Engine. The online documentation provided with the Codec Engine is as follows:

- **Codec Engine API Reference.**
  CE_INSTALL_DIR/docs/html/index.html
- **Codec Engine SPI Reference Guide.**
  CE_INSTALL_DIR/docs/spi/html/index.html
- **Configuration Reference Guide.**
  CE_INSTALL_DIR/xdoc/index.html
- **Example Build and Run Instructions.**
  CE_INSTALL_DIR/examples/build_instructions.html

For information about xDM, see the *xDAIS-DM (Digital Media) User Guide* (SPRUEC8).

For platform-specific help, see the Getting Started Guide for your platform.

Chapter 2

Installation and Setup

This chapter describes steps you may need to perform for installation and setup.

Topic **Page**

2.1 Installing Codec Engine

The Codec Engine may have already been installed on your system as part of a larger installation. For example, the DVSDK software installation installs the Codec Engine in the codec_engine_#_## subdirectory of the main DVSDK software directory.

If you have downloaded the Codec Engine as a standalone piece of software, follow these instructions:

1) Copy the codec_engine_#_##.tar.gz file to the directory where you want to install the software (where #_## is the version number).
2) Unzip the file with an unzip utility.
3) Open the release_notes*.html file at the top level of the installation.
4) Follow the steps to build and run Codec Engine examples.

2.2 Packages and Repositories

Codec Engine is delivered as a set of "packages". A package corresponds to a directory that contains all the files required for an independent component plus metadata about that component.

Each package has a unique name that reflects its directory name. For example, "ti.sdo.ce.audio" is the name of a package that must be in a directory whose path ends with "/ti/sdo/ce/audio". Packages may be nested within another package. "Package repositories" are directories that contain one or more packages.

A package places its metadata in a sub-directory named "package". This sub-directory contains metadata files that you do not need to be concerned with unless you are creating your own packages. A package also always contains a file named “package.xdc”, which declares the package’s name and an optional compatibility key. This key is used to ensure compatibility between packages.

As an application developer using the Codec Engine, the main reason you should be aware of packages is to understand the #include paths you need to use for header files. These paths are relative to a package repository. The package repositories used by the Codec Engine are part of the "package path", which matches the sequence of –I options you need to pass to the compiler when compiling source files that use a module in the Codec Engine. Since packages all have unique names, even if they are in different repositories, the #include statement tells you which package contains a particular header file.

The Codec Engine distribution contains several package repositories:

- ❏ The core set of Codec Engine packages are in a repository named "packages". This corresponds to the CE_INSTALL_DIR/packages directory.
- ❏ The examples are distributed in a separate repository named "examples". This corresponds to the CE_INSTALL_DIR/examples directory.
- ❏ Some distributions of Codec Engine include a third repository containing a collection of dependant packages for convenience. This repository is named "cetools.packages", which corresponds to the CE_INSTALL_DIR/cetools/packages directory.

The XDC Tools provide an xdcpkg command (in the XDC_INSTALL_DIR/bin directory) that identifies all the packages in a directory. For example:

```
xdcpkg -a -l .
```

2.3 Directory Structure

The top-level directories within the Codec Engine installation are as follows:

- ❏ **cetools.** Contains DSP Link and other TI tools used by the Codec Engine. Some distributions of the Codec Engine contain dependent components in this location. Other distributions expect the dependent components to have been installed separately.
- ❏ **docs.** Contains documentation files.
- ❏ **examples.** Contains a number of example applications.
- ❏ **package.** Contains package-related metadata files. You do not need to use this directory unless you are creating packages.
- ❏ **packages.** The Codec Engine packages. The /ti/sdo/ce subdirectory (that is, the ti.sdo.ce package) contains the VISA APIs and the stubs and skeletons that enable remote invocation of the VISA APIs. The /ti/sdo/ce/osal subdirectory contains the OS abstraction layer.
- ❏ **xdoc.** Contains documentation files for the packages in the Codec Engine distribution.

Chapter 3

Using the Sample Applications

This chapter describes how to test the sample applications provided with the Codec Engine.

Topic **Page**

3.1 Overview

The CE_INSTALL_DIR/examples repository contains a collection of example packages demonstrating the various use cases that Codec Engine users may develop. These example packages fall into the following categories:

- ❏ **Codecs.** Contains the codecs distributed with the Codec Engine. These are in the ti.xdais.dm.examples and ti.sdo.ce.examples.codecs namespace. (Note that the codecs in the ti.xdais.dm.examples namespace are copies of those distributed with xDAIS 5.20+.)
- ❏ **Extensions.** Contains a scale example that extends the VISA API. These are in the ti.sdo.ce.examples.extensions namespace.
- ❏ **Servers.** Contains two pre-configured and pre-linked Codec Servers (see Section 1.4.2, *Server Integrator*). These are in the ti.sdo.ce.examples.servers namespace.
- ❏ **Applications.** Contains example applications. These are in the ti.sdo.ce.examples.apps namespace.

The "copy" encoders/decoders replicate data rather than compressing/decompressing it. This is for simplicity in the examples. The Codec Engine distribution includes copy encoders/decoders for audio, image, speech, and video data for both xDM 0.9 and xDM 1.00.

The two pre-configured Codec Servers are: all_codecs and video_copy. If you are using the DVSDK, you use these pre-configured Codec Servers when evaluating the board and learning to use the Codec Engine. Note that the memory maps that these Codec Servers are configured with matches that of the DVSDK.

See the CE_INSTALL_DIR/examples/build_instructions.html file for a full list of examples and links to details about them.

3.2 Building Applications

To build the example applications provided with the Codec Engine, follow the steps in the CE_INSTALL_DIR/examples/build_instructions.html file. These steps may change in different versions of the Codec Engine, so you should review this document whenever you upgrade to a new version.

In general, you will optionally make a copy of the examples tree, edit the user.bld file to specify the locations of your tools, edit various makefiles, and then build the examples.

3.3 Running Applications

To run the example applications provided with the Codec Engine, follow the steps in the CE_INSTALL_DIR/examples/build_instructions.html file. This document provides platform-specific and version-specific steps for running applications.

Chapter 4

Using the Codec Engine APIs

This chapter describes how to use the Codec Engine APIs in an application.

Topic **Page**

4.1 Overview

The Codec Engine has Core Engine APIs and VISA APIs. The following table shows the Core Engine API modules:

Table 4–1 Codec Engine Modules

Description	Module Abbreviation	Package Name	Header File(s)	See Section
Initialization function	CERuntime_	ti.sdo.ce	CERuntime.h	Section 4.2.1
Codec Engine Runtime	Engine_	ti.sdo.ce	Engine.h	Section 4.2
OS Abstraction Layer for memory	Memory_	ti.sdo.ce.osal	Memory.h	Section 4.5

GPP+DSP

In addition to modules that perform setup and teardown activities, a memory abstraction module provides support for applications that use Codec Engine in a GPP+DSP system.

The following table shows the VISA API modules:

Table 4–2 Codec Engine Modules

Description	Module Abbreviation	Package Name	Header File(s)
Video Encoder Interface	VIDENCx_	ti.sdo.ce.video	videnc.h
Video Decoder Interface	VIDDECx_	ti.sdo.ce.video	viddec.h
Image Encoder Interface	IMGENCx_	ti.sdo.ce.image	imgenc.h
Image Decoder Interface	IMGDECx_	ti.sdo.ce.image	imgdec.h
Speech Encoder Interface	SPHENCx_	ti.sdo.ce.speech	sphenc.h
Speech Decoder Interface	SPHDECx_	ti.sdo.ce.speech	sphdec.h
Audio Encoder Interface	AUDENCx_	ti.sdo.ce.audio	audenc.h
Audio Decoder Interface	AUDDECx_	ti.sdo.ce.audio	auddec.h

The VISA interfaces provided with CE 1.20 include support for both the xDM 0.9 and xDM 1.0 interfaces. The "x" suffixes in Table 4–2 represent a version of the interface. In xDM 0.9, the suffix was omitted; in xDM 1.0, it is "1".

Copy codecs complying with the xDM 1.0 interfaces are provided with the xDAIS 5.21 product. CE 1.20 utilizes those codecs in some of its examples. For example, the video1_copy example utilizes the VIDENC1 and VIDDEC1 VISA interfaces to communicate with those copy codecs.

The package name corresponds to the path your application must use to reference the header file it includes to use a particular module. For example, the speech encoder module, SPHENC*x*, has a package name of ti.sdo.ce.speech(*x*). The #include statement for this module is (where x is a version-based suffix):

```
#include <ti/sdo/ce/speechx/sphencx.h>
```

Your application uses the Engine module to open and close instances of the Codec Engine. You can also use this module to get information about memory use and CPU loading. See Section 4.2, *The Core Engine APIs*.

Once your application has opened an instance of the Codec Engine, you use the modules in the VISA packages (for example, VIDENC for video encoding) to create instances of various algorithms. Using the handle for the algorithm instance you create, you then use the same module to run or otherwise control the algorithm. See Section 4.3, *The VISA Classes: Video, Image, Speech, Audio*.

Reference documentation for the Codec Engine APIs is installed with the Codec Engine software at CE_INSTALL_DIR/docs/html. This chapter provides an overview of the APIs and how you use them. For details about the calling syntax, see the reference documentation.

4.2 The Core Engine APIs

The Codec Engine has a "core" module called "Engine". Your application uses this module to open and close Engine instances. Multi-threaded applications must either serialize access to a shared Engine instance or create a separate Engine instance for each thread.

Note: Be aware that Engine handles are not thread-protected. Each thread that uses an Engine instance should perform its own Engine_open() call and use its own Engine handle. This protects each Engine instance from access by other threads in a multi-threaded environment.

You can also use the Engine module to get information about memory use and CPU loading.

The APIs for the Engine module are:

- ❏ **Engine_open().** Open an Engine.
- ❏ **Engine_close().** Close an Engine.
- ❏ **Engine_getCpuLoad().** Get Server's CPU usage in percent.

- ❑ **Engine_getLastError().** Get the error code of the last failed operation.
- ❑ **Engine_getUsedMem().** Get Engine memory usage.
- ❑ **Engine_getNumAlgs().** Get the number of algorithms in an Engine.
- ❑ **Engine_getAlgInfo().** Get information about an algorithm.

After opening an Engine, you create algorithm instances using the VISA APIs described in Section 4.3.

Reference documentation for the Codec Engine APIs is installed with the Codec Engine software at CE_INSTALL_DIR/docs/html. For details about the calling syntax, see the reference documentation.

4.2.1 Codec Engine Setup Code

An application that will use the Codec Engine should include the following header files, where these directory paths are relative to the package path, which includes CE_INSTALL_DIR and XDC_ROOT.

```
#include <xdc/std.h>
#include <ti/sdo/ce/Engine.h>
#include <ti/sdo/ce/CERuntime.h>
```

In addition, the application must include the header file for any VISA modules it uses. For example:

```
#include <ti/sdo/ce/audio/auddec.h>
```

Notice that the paths to the header files exactly correspond to the package paths shown in Table 4–2, *Codec Engine Modules*.

All applications that use the Codec Engine must run CERuntime_init. Typically, this is run from the application’s main() function.

GPP+DSP

In addition, when building a GPP+DSP application that uses Codec Engine, you must load the dependent modules DSP/BIOS Link and CMEM (a contiguous memory allocator). To see how this is done, examine the examples/apps/system_files/davinci/loadmodules.sh file. See the build_instructions.html file for details.

4.2.2 Opening an Engine

When you open an Engine, you specify the name of the Engine you want to open. For example:

```
static String engineName = "auddec";
Engine_Handle ce;
Engine_Error errorcode;

ce = Engine_open(engineName, NULL, &errorcode);
```

Note: Be aware that Engine handles are not thread-protected. Each thread that uses an Engine instance should perform its own Engine_open() call and use its own Engine handle. This protects each Engine instance from access by other threads in a multi-threaded environment.

Engines are configured by your Engine Integrator, who decides which algorithms to configure and build into each Engine. See Chapter 5, "Integrating an Engine" for information the Engine Integrator needs to create such Engines.

GPP+DSP

For example, on dual CPU systems such as the DM644x device, algorithms may run locally (on the GPP) or "remotely" (on the DSP). For remote algorithms, the Engine transparently uses a "DSP Server" and the DSP Link transport to run the high-MIPS algorithms. Here, the first call to Engine_open() results in the following underlying actions:

- ❏ Power ON the DSP (if support is available and configured via the ti.sdo.ce.osal.Global configuration).
- ❏ Call the Link APIs needed to initialize the DSP and the transport: PROC_Setup(), PROC_Attach(), POOL_Open(), PROC_Load(), PROC_Start(), and MSGQ_TransportOpen().
- ❏ Perform the initial handshakes from the GPP side to the remote dispatcher on the DSP.

The Engine_open() function allows you to pass an Engine_Attrs structure to the Engine. This type is defined in Engine.h, which is included by the application. Currently, this structure allows you to specify a string procID of the processor that runs the Engine’s DSP Server. This is only needed if there is more than one processor that can provide the same server and that is currently being used. The default procID is 0.

```
typedef struct Engine_Attrs {
   String procId;
} Engine_Attrs;
```

If the Engine_Handle returned by Engine_open() is NULL, the Engine could not be opened. If the errorcode parameter is non-NULL, the Engine_Error value is set to one of the following values:

- ❏ Engine_EOK. Success.
- ❏ Engine_EEXIST. Name does not exist.
- ❏ Engine_ENOMEM. Can't allocate memory.
- ❏ Engine_EDSPLOAD. Can't load the DSP.
- ❏ Engine_ENOCOMM. Can't create a communication connection to the DSP.
- ❏ Engine_ENOSERVER. Can't locate the Server on the DSP.
- ❏ Engine_ECOMALLOC. Can't allocate a communication buffer.

Your application can handle this error. For example:

```
ce = Engine_open(engineName, NULL, &errorCode);
if (ce == NULL) {
    printf("Error: could not open engine \"%s\";
            Error code %d.\n", engineName, errorCode);
}
```

4.2.3 Closing an Engine

To close an Engine instance and free memory it uses, your application should call Engine_close(). For example:

```
Engine_close(ce);
```

You should do this only after you have deleted any algorithm instances created for this Engine and freed any buffers or other memory related to the algorithm instances.

GPP+DSP

For example, given the DM644x-based example described in the previous section with remote algorithms performed on the DSP, the last call to Engine_close() results in the following underlying actions:

- ❏ Call the Link APIs needed to "finalize" the DSP and the transport: MSGQ_TransportClose(), PROC_Stop(), POOL_Close(), PROC_Detach(), and PROC_Destroy().
- ❏ Power OFF the DSP (if support is available and configured via the ti.sdo.ce.osal.Global configuration).

4.2.4 Getting Memory and CPU Information from an Engine

GPP+DSP

You can use the Engine_getUsedMem() function to get information about memory used by an Engine instance. The value returned is the total amount of memory (in MAUs) currently allocated from the available heaps. Note that this value may vary between calls, depending upon DSP Server activity. For example, when the first algorithm is instantiated on the DSP Server, data structures in addition to those needed for the individual algorithm instance may be allocated. This extra memory is allocated "when first needed" and remains allocated with its global state retained even after this algorithm is deleted. The memory is not re-allocated on subsequent allocations of this or other algorithms on the DSP Server.

In addition to Engine_getUsedMem(), there are Server APIs for getting information about the memory usage of individual heaps on the DSP. See Section 4.4 for a description of these functions.

You can use the Engine_getCpuLoad() function to get the DSP Server's CPU usage as an integer from 0 to 100. This value indicates the percentage of time the DSP Server is processing measured over a period of approximately 1 second. If the load is unavailable, a negative value is returned.

4.2.5 Getting Information About Algorithms Configured into an Engine

An application may determine the number of algorithms configured into an Engine and their properties, such as the name of the algorithm and whether it is local or remote.

The number of algorithms can be obtained with the following API:

```
Engine_Error Engine_getNumAlgs(String name, Int *numAlgs)
```

The parameter, "name", is the name of the Engine. This function returns the following values:

- ❏ Engine_EOK. Success. In this case, *numAlgs returns the number of algorithms configured into the Engine.
- ❏ Engine_EEXIST. There is no Engine with the given name.

Once the number of algorithms in the Engine is known, the application can iteratively call the function Engine_getAlgInfo() to obtain information about each of the algorithms. The information is put into the Engine_AlgInfo structure, which is defined as follows:

```
typedef struct Engine_AlgInfo {
    Int         algInfoSize;    /* Size of this structure */
    String      name;           /* Name of algorithm */
    String      *typeTab;       /* inheritance hierarchy */
    Bool        isLocal;        /* if TRUE, run locally */
} Engine_AlgInfo;
```

The first field of the Engine_AlgInfo structure, "algInfoSize", must be set by the application to the size of this structure; it will be used to support future enhancements to this structure. The following example shows the usage of these APIs (error checking has been left out for readability):

```
Int                 numAlgs, i;
Engine_AlgInfo      algInfo;
Engine_Error        err;

err = Engine_getNumAlgs("audio_copy", &numAlgs);

for (i = 0; i < numAlgs; i++) {
    err = Engine_getAlgInfo(name, &algInfo, i);
    printf("alg[%d]: name = %s typeTab = %s local = %d\n",
        i, algInfo.name, *(algInfo.typeTab), algInfo.isLo-
cal);
}
```

The output may look like the following:

```
alg[0]: name = auddec_copy typeTab = ti.sdo.ce.audio.IAUDDEC local = 0
alg[1]: name = audenc_copy typeTab = ti.sdo.ce.audio.IAUDENC local = 0
```

The field, typeTab, is actually a NULL-terminated array of strings giving the inheritance hierarchy. In this example, we printed out the first string only.

The return values of Engine_getAlgInfo() are the following:

❑ Engine_EOK. Success.

❑ Engine_EEXIST. There is no Engine with the given name.

❑ Engine_EINVAL. The algInfoSize field of the Engine_AlgInfo object passed to this function does not match the size of the Engine_AlgInfo object in the Codec Engine library.

❑ Engine_ENOTFOUND. The index of the algorithm is out of range.

4.3 The VISA Classes: Video, Image, Speech, Audio

For the encoder and decoder sides of each of the VISA classes, the following APIs are provided, where *MOD* is the module prefix:

❑ *MOD*_create. Create an instance of this type of algorithm.

❑ *MOD*_process. Execute the "process" method in this instance of the algorithm.

❑ *MOD*_control. Execute the "control" method in this instance of the algorithm.

❑ *MOD*_delete. Delete the specified instance of this type of algorithm.

4.3.1 VISA API Setup Code

For each VISA API module your application uses, you should include the appropriate header file. For example, the following statement includes the header file for the audio decoder API module. The directory path is relative to the CE_INSTALL_DIR/packages package repository.

```
#include <ti/sdo/ce/audio/auddec.h>
```

4.3.2 Creating an Algorithm Instance

To create an algorithm instance within an Engine, you use the *_create() function for the appropriate VISA encoder or decoder module. For example:

```
Engine_Handle ce;
AUDDEC_Handle dec;
static String decoderName = "auddec_copy";

/* allocate and initialize audio decoder on the Engine */
dec = AUDDEC_create(ce, decoderName, NULL);
```

In this function, the first argument—ce—is the Engine_Handle returned by the Engine_open() function.

The second argument—decoderName—is a string that identifies the type of algorithm to create. These strings are part of the configuration created by your system integrator.

The third argument allows you to specify parameters to use when instantiating the algorithm. These parameters control aspects of algorithm behavior. The parameter structure is different for each VISA encoder or decoder class. For example, the audio decoder parameter structure is as follows:

```
typedef struct IAUDDEC_Params {
   XDAS_Int32 size;           /* Size of this structure */
   XDAS_Int32 maxSampleRate;  /* Max sampling freq in Hz */
   XDAS_Int32 maxBitrate;  /* Max bit-rate in bits per sec */
   XDAS_Int32 maxNoOfCh;      /* Max number of channels */
   XDAS_Int32 dataEndianness; /* Endianness of input data */
} IAUDDEC_Params;
```

This function returns a handle that other functions use to access the algorithm instance.

4.3.3 Deleting an Algorithm Instance

To delete an algorithm instance and free the memory it uses, your application should call *MOD*_delete(). For example:

```
/* tear down the codec and Engine */
AUDDEC_delete(dec);
```

You should do this only after you have freed any buffers or other memory related to the algorithm instance.

4.3.4 Controlling an Algorithm Instance

You can control and query the capabilities of an algorithm using the module's *_control() function.

For example, the following code uses the AUDDEC_control() function to query a decoder to verify that the decoder accepts one input buffer, returns one output buffer, and uses buffer sizes that can handle NSAMPLES bytes of data.

```
#define NSAMPLES    1024
#define IFRAMESIZE (NSAMPLES * sizeof(Int8)) /* raw (in) */
#define OFRAMESIZE (NSAMPLES * sizeof(Int8)) /* decoded */

static Char inBuf[IFRAMESIZE];
static Char outBuf[OFRAMESIZE];

XDM_BufDesc                 inBufDesc;
XDM_BufDesc                 outBufDesc;
XDAS_Int32                  status;
XDAS_Int32                  bufSizes = NSAMPLES;
IAUDDEC_DynamicParams       decDynParams;
IAUDDEC_Status              decStatus;

/* prepare "global" buffer descriptor settings */
inBufDesc.numBufs = outBufDesc.numBufs = 1;
inBufDesc.bufSizes = outBufDesc.bufSizes = &bufSizes;

/* Query the decoder */
status = AUDDEC_control(dec, XDM_GETSTATUS, &decDynParams,
       &decStatus);
if (status != AUDDEC_EOK) {
   /* failure, report error and exit */
   printf("decode control status = %ld\n", status);
   return;
}

/* Validate decoder codec will meet buffer requirements */
if ((inBufDesc.numBufs > decStatus.bufInfo.maxNumInBufs) ||
   (sizeof(inBuf) > decStatus.bufInfo.maxInBufSize[0]) ||
   (outBufDesc.numBufs > decStatus.bufInfo.maxNumOutBufs) ||
   (sizeof(outBuf) > decStatus.bufInfo.maxOutBufSize[0])) {

   /* failure, report error and exit */
   printf("Error:  decoder codec feature conflict\n");
   return;
}
```

In the AUDDEC_control() function example, the first argument—dec—is the handle to the algorithm returned by the AUDDEC_create() function.

The second argument is a command ID constant from xdm.h. The options are:

❑ XDM_GETSTATUS. Queries the algorithm and fills a structure that contains status information about the capabilities of the algorithm.

❑ XDM_SETPARAMS. Sets the run-time dynamic parameters of the algorithm.

❑ XDM_GETPARAMS. Gets the current settings of the run-time dynamic parameters of the algorithm.

❑ XDM_RESET. Resets the algorithm. This may run an initialization function or a special function to recover after an error or data loss.

❑ XDM_SETDEFAULT. Sets all parameters to their defaults.

❑ XDM_FLUSH. Handles end of stream conditions. Forces algorithm to output data without additional input. The recommended sequence is to call the *_control() API with XDM_FLUSH and then make repeated calls to the *_process() API.

❑ XDM_GETBUFINFO. Queries the algorithm instance regarding the properties of its input and output buffers.

For more about xDM, see the *xDAIS-DM (Digital Media) User Guide* (SPRUEC8).

The third argument is the address of a dynamic parameter structure that is used if you specify the XDM_SETPARAMS or XDM_GETPARAMS command codes. This structure is different for each of the VISA encoder and decoder modules.

The fourth argument is the address of a status structure that is used if you specify the XDM_GETSTATUS command code. This structure is also different for each of the VISA encoder and decoder modules.

4.3.5 Processing Data with an Algorithm Instance

You can run an algorithm using the module’s *_process() function.

For example, the following code continues the example in the previous section. It uses the AUDDEC_process() function to read frames from "in", decode the audio, and write the output to "out".

```
Int                         n;
XDM_BufDesc                 inBufDesc;
XDM_BufDesc                 outBufDesc;
IAUDDEC_InArgs              decInArgs;
IAUDDEC_OutArgs             decOutArgs;

/* prepare "global" buffer descriptor settings */
inBufDesc.numBufs = outBufDesc.numBufs = 1;
inBufDesc.bufSizes = outBufDesc.bufSizes = &bufSizes;

decInArgs.size = sizeof(decInArgs);

...

/* Read complete frames from in, decode and write to out */
for (n = 0; fread(inBuf, sizeof (inBuf), 1, in) == 1; n++) {
    XDAS_Int8 *src = inBuf;
    XDAS_Int8 *dst = outBuf;

    /* prepare "per loop" buffer descriptor settings */
    inBufDesc.bufs = &src;
    outBufDesc.bufs = &dst;
    decInArgs.size = sizeof(decInArgs);
    decInArgs.numBytes = sizeof(inBuf);

    /* decode the frame */
    status = AUDDEC_process(dec, &inBufDesc, &outBufDesc,
          &decInArgs, &decOutArgs);

    if (status != AUDDEC_EOK) {
       printf("frame %d: decode status = %ld\n", n, status);
    }

    /* write to file */
    fwrite(dst, sizeof (outBuf), 1, out);
}
printf("%d frames decoded\n", n);
```

In this AUDDEC_process() function example, the first argument—dec—is the handle to the algorithm returned by the AUDDEC_create() function.

The second and third arguments for the audio decoder module (and for most other VISA modules) provide the address of a buffer descriptor structure of type XDM_BufDesc. This type has the following structure definition:

```
typedef struct XDM_BufDesc {
    XDAS_Int8   **bufs;
    XDAS_Int32   numBufs;
    XDAS_Int32  *bufSizes;
} XDM_BufDesc;
```

The fourth and fifth arguments for the audio decoder module (and for most other VISA modules) provide the address of input and output arguments for the algorithm. This structure is different for each of the VISA encoder and decoder modules.

4.3.6 Overriding a Remote Algorithm's Priority and Memory Requests

4.3.6.1 Overriding the Algorithm's Configured Priority

In some situations, an application developer may want to run multiple instances of a remote codec at different priorities. As an example, suppose you want to run two instances of the sample audio encoder copy codec: one at priority 4, the other at priority 5. The server containing this codec is originally configured with the audio encoder running at priority 4, as shown in the following configuration code (assuming that Server.MINPRI is 1):

```
Server.algs = [
    {name: "audenc_copy", mod: AUDENC_COPY, threadAttrs: {
        stackMemId: 0, priority: Server.MINPRI + 3}
    },
   ...
];
```

It may seem that the solution to this problem is to configure the DSP server by adding another audio encoder with the new priority and a different name to the list of server algorithms, as follows:

```
Server.algs = [
    /* Audio copy encoder configured with priority 4 */
    {name: "audenc_copy", mod: AUDENC_COPY, threadAttrs: {
        stackMemId: 0, priority: Server.MINPRI + 3}
    },

   /* Audio copy encoder configured with priority 5 (WRONG)*/
    {name: "audenc_copy_2", mod: AUDENC_COPY, threadAttrs: {
        stackMemId: 0, priority: Server.MINPRI + 4}
    },
   ...
];
```

However, this generates an error when trying to rebuild the server, since the generated UUIDs for these two codecs, determined by the mod (AUDENC_COPY) configuration parameter, will be identical. Since it is the UUID, and not the codec name, that is passed internally from the ARM application to the DSP server to instantiate the codec, these two codecs would be indistinguishable. Therefore, this method will not work.

The correct way to create a codec with a priority other than the one configured in the DSP server, is through the name parameter passed to the *MOD*_create() API, where *MOD*, is one of the VISA modules. The name will be the codec name with the overriding priority appended to it, separated with a ":". For example, to run the audio encoder shown above at priority 5, pass the name "audenc_copy:5" to AUDENC_create(). The following code fragment creates two audio copy encoders running at different priorities (error checking is left out for readability).

```
Engine_Handle  ce;
AUDENC_Handle  enc;
AUDENC_Handle  enc_high;

ce = Engine_open("audio_copy", NULL, NULL);

/* Create codec at the configured priority */
enc = AUDENC_create(ce, "audenc_copy", NULL);

/* Create second instance of codec, overriding the
 * configured priority with a priority of 5 */
enc_high = AUDENC_create(ce, "audenc_copy:5", NULL) ;
```

4.3.6.2 Overriding an Algorithm's Memory Requests

It is possible to ignore a codec's requests for placement of allocated buffers and force all of the codec's memory requests to be allocated in the external heap mapped to the DSKT2 module's ESDATA configuration parameter.

This is also done by appending to the codec name passed to the *MOD*_create() function. To override memory placement requests, append ":1" to the name after the adjustment for priority. For example, the names below passed to AUDENC_create() have the following meanings:

❑ `"audenc_copy:5:1"`
  Create audenc_copy with priority 5 and with buffers allocated in external memory.

❑ `"audenc_copy::1"`
  Create audenc_copy with its configured priority and with buffers allocated in external memory.

Appending "::0" to the codec name (or ":0" if a new priority is also appended), means the codec memory requests should be respected. For example, passing the following names to AUDENC_create() are equivalent:

❑ `"audenc_copy"`

❑ `"audenc_copy::0"`

4.4 The Server APIs

GPP+DSP

On dual CPU systems, Engines that are configured to run algorithms "remotely" (on the DSP) transparently use a "DSP Server". The DSP Server is an executable that integrates algorithms and their frameworks (for example, DSP/BIOS, Framework Components, codecs, and DSP Link drivers), which will be loaded and started on the DSP when the Engine is opened.

The Server APIs can be used by applications running on the GPP to access information about the DSP Server and to control the DSP Server. More specifically, these APIs allow a GPP application to obtain information about the number of memory heaps configured in the DSP Server, the current usage of an individual memory heap, and to reconfigure the base and size of the DSP Server's algorithm heap.

The APIs related to the Server are:

- ❑ **Engine_getServer().** Get the handle to a Server.
- ❑ **Server_getNumMemSegs().** Get the number of heaps in a Server.
- ❑ **Server_getMemStat().** Get statistics about a Server heap.
- ❑ **Server_redefineHeap().** Set base and size of a Server heap.
- ❑ **Server_restoreHeap().** Reset Server heap to default base and size.

4.4.1 Getting a Server Handle

To access the DSP Server for the Engine, the GPP application must first obtain a Server handle by calling the Engine_getServer() API. For example:

```
static String engineName = "auddec";
Engine_Handle engine;
Server_Handle server;
Engine_Error err;

engine = Engine_open(engineName, NULL, &err);
server = Engine_getServer(engine);
```

Note: As with Engine handles, Server handles are not thread protected. Each thread that uses a Server handle must perform its own Engine_getServer() call (using its own Engine handle) or guarantee synchronized access to a shared Server handle.

If the value returned by Engine_getServer() is NULL, then the Engine has no Server.

4.4.2 Getting Memory Heap Information

The GPP application can obtain the number of memory heaps configured into the DSP Server by calling Server_getNumMemSegs(). For example:

```
Server_Handle server;
Server_Status status;
Int numSegs;

/* Get the server handle from a previously opened Engine */
server = Engine_getServer(engine);
status = Server_getNumMemSegs(server, &numSegs);
```

This API returns the following error codes:

- ❑ Server_EOK - success. In this case, numSegs contains the number of heaps in the DSP Server.
- ❑ Server_ERUNTIME - an internal runtime error occurred.

Once the number of heaps is known, the GPP application can then iterate through this number, to obtain statistics about each heap, using Server_getMemStat(). The memory statistics are returned in a Server_MemStat structure:

```
typedef struct Server_MemStat {
    Char name[Server_MAXSEGNAMELENTH+1];
                              /* Name of heap segment */
    Uint32 base;              /* Base address of heap */
    Uint32 size;              /* Original heap size */
    Uint32 used;              /* DSP MAUs of heap used */
    Uint32 maxBlockLen;       /* Length of largest free block */
} Server_MemStat;
```

The following example code shows the usage of these APIs (error checking is left out for readability).

```
Server_Handle  server;
Int            numSegs, i;
Server_MemStat stat;
Server_Status  status;

status = Server_getNumMemSegs(server, &numSegs);
for (i = 0; i < numSegs; i++) {
    status = Server_getMemStat(server, i, &stat);
    printf("%s: base: 0x%x size: 0x%x used: 0x%x
        max free block: 0x%x",
        stat.name, stat.base, stat.size, stat.used,
        stat.maxBlockLen);
}
```

The values returned by Server_getMemStat() are the following:

- ❏ Server_EOK. Success.
- ❏ Server_ENOTFOUND. The segment number was out of range.
- ❏ Server_ERUNTIME. An internal runtime error occurred.

4.4.3 Reconfiguring the DSP Server's Algorithm Heap

The DSP Server can be configured with a memory segment that is used exclusively for algorithm heaps. In some situations, the DSP Server may be configured with a small algorithm heap, and the GPP application may want to provide, at runtime, a larger contiguous memory block to be used by the DSP Server for the algorithm heap. Then, when the heap is not being used by the DSP, this memory could be reclaimed from the DSP and used by the GPP. The following Server APIs provide the means to reconfigure the DSP algorithm heap:

```
Server_Status Server_redefineHeap(Server_Handle server,
    String name, Uint32 base, Uint32 size);
Server_Status Server_restoreHeap(Server_Handle server,
    String name);
```

The "name" parameter passed to these functions is the name of the heap to be reconfigured; it must not be more than Server_MAXSEGNAMELENGTH characters long. The "base" address passed to Server_redefineHeap() must be a DSP address, and the memory from base to base + size must be contiguous in physical memory. The "size" parameter is given in DSP MADUs (minimum addressable data units). The base address should be 8-byte aligned, but there are no alignment restrictions on size; a value of 0 for size is acceptable.

The DSP algorithm heap can only be reconfigured when no memory is currently allocated in the heap. The Server_restoreHeap() function resets the base address and size of the algorithm heap back to their original values (the values before any calls to Server_redefineHeap() were made). After a successful call to Server_restoreHeap(), the memory previously "redefined" to the heap can be used again by the system.

The return values of Server_redefineHeap() are the following:

- ❏ Server_EOK. Success.
- ❏ Server_EINVAL. Changing to the new base address and size would cause an overlap with another heap.
- ❏ Server_EINUSE. Memory is currently allocated in the algorithm heap.
- ❏ Server_ENOTFOUND. No heap with the given name was found.
- ❏ Server_ERUNTIME. An internal runtime error occurred.

Server_restoreHeap() returns any of the following values:

- ❏ Server_EOK. Success.
- ❏ Server_EINVAL. Changing back to the original base address and size would cause an overlap with another heap.
- ❏ Server_EINUSE. Memory is currently allocated in the algorithm heap.
- ❏ Server_ENOTFOUND. No heap with the given name was found.
- ❏ Server_ERUNTIME. An internal runtime error occurred.

The following code illustrates how these two APIs could be used on the DM644x, a GPP+DSP device. In this example, a contiguous chunk of memory is allocated by the GPP application using Memory_contigAlloc(). However, the address returned by this function is a virtual address on the GPP, so it must be converted to a DSP address before passing it to Server_redefineHeap(). The Memory_getBufferPhysicalAddress() function converts the virtual address to a physical address on the GPP, which, in the case of the DM644x, is the same as the DSP address.

After the algorithm is run, the algorithm heap is reset to its original size and location. Error checking is left out for better readability.

```
Server_Handle  server = NULL;
Server_Status  status;
Engine_Handle  ce = NULL;
XDAS_Int8     *buf;
Uint32         base;

String decoderName  = "auddec_copy";
String encoderName  = "audenc_copy";
String engineName   = "audio_copy";

/* Open the Engine and get Server handle. Note, the
 * Engine_open() call will load and start the DSP. */
ce = Engine_open(engineName, NULL, NULL);
server = Engine_getServer(ce);

/* Allocate block of memory, contiguous in physical memory */
buf = (XDAS_Int8 *)Memory_contigAlloc(BUFSIZE, ALIGNMENT);

/* Convert virtual address to physical address, which on
 * DM644x, happens to be the same as the DSP address. */
base = Memory_getBufferPhysicalAddress(buf, BUFSIZE, NULL);

/* Reconfigure the algorithm heap */
status = Server_redefineHeap(server, "DDRALGHEAP", base,
      BUFSIZE);

'Create and run codecs'
'Delete codecs'

/* Reconfigure algorithm heap back to its original state. */
status = Server_restoreHeap(server, "DDRALGHEAP");

/* Free the buffer */
Memory_contigFree(buf, BUFSIZE);
```

In other scenarios, the application may need to reconfigure the algorithm heap to an address that is not obtained by allocating a buffer on the ARM. For example, suppose there are fixed memory spaces on the DSP that the application will alternate between for the algorithm heap, depending on what algorithms will be run. In this case, the application can pass the DSP address directly to Server_redefineHeap().

4.5 What Happens to DSP Memory Issues?

The VISA APIs to create and delete algorithms provided by the Codec Engine manage all algorithm resources. This includes the CPU, memory, direct memory access (DMA), and more. The VISA creation and deletion APIs hide most of the details of the codecs' memory and resource management.

4.5.1 Buffer Handling and Shared Memory Maps

GPP+DSP

It is the responsibility of the application to handle all I/O and buffering issues. The control APIs use pointers to input and output buffers of type XDM_BufDesc. The structure is as follows:

```
typedef struct XDM_BufDesc {
    XDAS_Int8    **bufs;
    XDAS_Int32   numBufs;
    XDAS_Int32   *bufSizes;
} XDM_BufDesc;
```

You must place the buffers it points to in memory shared by the GPP and DSP. This does away with the prohibitive performance impact of passing large amounts of signal data from the GPP to the DSP and back.

In addition, the buffers you use for shared data must be contiguous and cache-aligned.

For example, the DM644x default memory map is designed with the intent to provide generous amount of space for DSP code and data, plenty of private heap for DSP algorithms, and a large space for shared buffers between GPP and DSP.

Table 4–3 DM644x Default Memory Map

Address (hex)	Address (decimal)	Size	Segment	Comments
0x80000000 .. 0x87800000	0-120MB	120MB	Linux	booted with MEM=120M
0x87800000 .. 0x88000000	120-128MB	8MB	CMEM	shared buffers between GPP and DSP
0x88000000 .. 0x8FA00000	128-250MB	122MB	DDRALGHEAP *	DSP segment used exclusively for algorithm heaps
0x8FA00000 .. 0x8FE00000	250-254MB	4MB	DDR*	DSP segment for code, stack, and static data

Table 4–3 DM644x Default Memory Map

Address (hex)	Address (decimal)	Size	Segment	Comments
0x8FE00000 .. 0x8FF00000	254-255MB	1MB	DSPLINKMEM *	memory for DSPLINK
0x8FF00000 .. 0x8FF00080	255MB-255MB	128B	CTRLRESET *	memory for reset vectors
0x8FF00080 .. 0x8FFFFFFF	255MB-256MB	1MB	-- unnamed --	

(*) are actual DSP linker segments.

4.5.2 Memory Fragmentation

GPP+DSP

For dual CPU applications, the exception to the rule that the Codec Engine hides DSP memory management issues from the GPP application developer is that buffers passed to the DSP must be contiguous in physical memory and cache-aligned.

This differs from buffer handling on the GPP because Linux and similar GPP operating systems handle non-contiguous buffers through a memory management unit (MMU) that holds a table matching virtual addresses to physical addresses. DSPs have no such table.

The Codec Engine verifies that these constraints are met in the required platform for data buffers. Algorithm buffers are managed by the Memory_ module, which uses pools of different sizes to ensure that memory is not fragmented.

However, the storage space for codec instances created by the Codec Engine must also be contiguous and cache-aligned. The creation and deletion of codec instances is non-deterministic. For example, if your application follows steps like those in Figure 4–1, it may be impossible to recreate a codec instance that was created and deleted earlier:

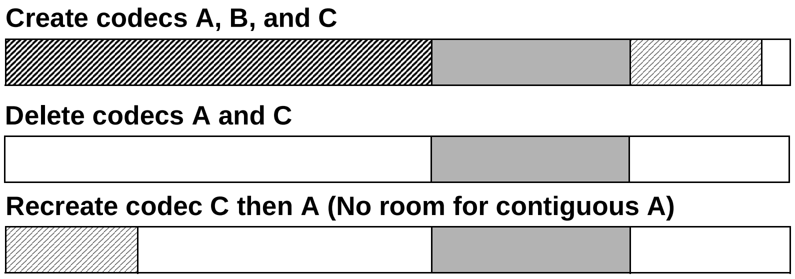

Create codecs A, B, and C
Delete codecs A and C
Recreate codec C then A (No room for contiguous A)

Figure 4–1 Non-Deterministic Buffer Creation and Deletion

Since instance creation occurs “in the background” while other codecs are running at higher priorities, you cannot guarantee the time required to create an instance. You can, however, control the order in which instances are created and deleted.

If a codec or shared buffer is not physically contiguous, when the caller calls Memory_getBufferPhysicalAddress() with a non-NULL pointer for the Boolean *isContiguous arguments, the Codec Engine sets the ptr data to true or false without printing any message. If the pointer is NULL (which is most likely, since this function is called by the codec stubs, which pass NULL for this ptr), the level 7 trace message is:

```
Memory_getBufferPhysicalAddress> ERROR: user buffer at
addr=<hex addr>, size=<size in bytes> is NOT contiguous
```

This message is printed only if level 7 tracing is enabled, which is the default.

4.5.3 Cache Alignment

Devices that utilize a cache (for example, the C64+) also require that I/O buffers be cache-aligned. For example, the DSP L2 cache line size on the C64x+ is 128 bytes. Storage space allocated must start at a cache line boundary, and the size must be a multiple of the cache line length.

If these alignment and size constraints are violated, any data object allocated adjacent to the application buffer will share a cache line with a portion of the application buffer. This line may be corrupted as a result of the Cache Controller writing back the shared cache line.

4.5.4 Cache Coherence

When developing an application for a multiprocessor platform (including those with multiple processing cores, hardware accelerators, and DMA Engines) in which some memory regions are cached, you must perform some memory coherence operations. The Codec Engine framework, when it has enough information, automatically handles some cache coherence operations. However, ultimately, the application developer is responsible for ensuring that certain pre- and post-conditions are met for the buffers the application submits and receives from the Codec Engine.

The subsections that follow summarize the responsibilities of the application developer for different processor environments.

4.5.4.1 GPP + DSP Environments

GPP+DSP

The following are common issues in the DaVinci environment, though they are present in any multi-core system that utilizes a cache.

Note that the Codec Engine Framework enforces some of these rules in the implementation of the VISA APIs. You should be aware of these rules when accessing shared memory outside the use of the VISA APIs.

- **Input Buffers.** This is the case when a GPP application captures/generates a buffer and passes it to the DSP.
  - **GPP side.** Input buffers must be written-back before each process/control call. (Otherwise, DSP CPU/DMA accesses will access incoherent data in external memory, with no ability to writeback the GPP-side cache.) If a buffer is not cached on the GPP side, a writeback is not required.

    When a driver fills a GPP-cached input buffer, before passing it to the Codec Engine, the driver should do the following: 1) Start with a cache invalidated buffer. 2) The driver can use DMA or CPU writes to fill the buffer. 3) If CPU is used to fill the buffer, it must be written-back before passing the buffer to Codec Engine.
  - **DSP side.** Input buffers must be invalidated before each process/control call. (Otherwise the DSP may read stale data from its cache. This is possible if the same buffer was passed in an earlier call.) Note that the default skeletons for the VISA APIs automatically invalidate input buffers prior to invoking the algorithm's process function.
- **Output Buffers.**
  - **GPP side.** Output buffers must be invalidated before accessing them on the GPP side following DSP-side processing. (Otherwise the GPP may access stale data resident in its GPP-

side cache.) If the buffers are not cache-enabled on the GPP side invalidation is not required.

  - **DSP side.** Output buffers must be invalidated before each process/control call. (Otherwise, if DMA is used to fill the buffer, there may be overwrites as cache lines are evicted due to unrelated CPU activity.) Also, output buffers must be written-back after each process/control call. (Otherwise the GPP may read incoherent data from external memory.) Note that the default skeletons for the VISA APIs automatically writeback output buffers following each process/control call.

- **DMA-Related.** If the GPP or DSP uses DMA to access shared buffers, there is more work to ensure coherence. xDAIS provides some DMA rules for frameworks. See *TMS320 DSP Algorithm Standard Rules and Guidelines* (SPRU352) for details.

C6000 algorithms must not issue any CPU read/writes to buffers in external memory that are involved in DMA transfers. This also applies to the input buffers passed to the algorithm through its algorithm interface.

Some common cache-related errors are:

- Doing a cache writeback-invalidate for DSP "input" buffers, instead of just an invalidate "before" a process/control call. In this case, if any of the "current" input buffers has been referenced in a "previous" process/control call, then a stale fragment of that buffer may already be resident in the DSP cache. A writeback will corrupt the "current" input buffer with stale data from the cache.
- Doing a blind "ALL L2 Cache" writeback-invalidate, instead of a writeback or invalidate on only the algorithm's own input/output buffers. This creates potential problems for other algorithm instances, whose input/output buffers will be affected.
- Invalidating all of L2 severely degrades performance for all algorithm instances, due to the resulting cache misses.

4.5.4.2 Single-Processor Environments

The following are common issues in the DM643x environment, though they are present in any single-CPU system that utilizes cached memory.

- **Input Buffers.** This is the case when a DSP application captures/generates a buffer and passes it to the Codec Engine. Depending upon how the input buffers have been captured, the buffers must be either invalidated or written-back and invalidated:
  - If the application (or a driver) modified the contents of the input buffer using CPU read and/or write operations, the buffer must be written-back and invalidated.

  - If the application (or driver) modified the contents of the input buffer using DMA, then the buffer must **not** be written-back, but must still be invalidated.

  In both cases, the application (or driver) should start filling a cache-invalidated buffer.

- **Output Buffers.** Output buffers (those filled by Codec Engine) must be invalidated before being submitted to the Codec Engine to be filled. And, when returned from the Codec Engine, the buffers should be written-back to ensure all data is written out to external memory.

4.6 What Happens to DSP Real-Time Issues?

GPP+DSP

It is the responsibility of the GPP application to handle all multi-threading and real-time issues from a GPP perspective. For example, this may involve scheduling higher-frequency, short-duration processing (such as audio) at a higher priority than long-duration processing (such as video) on Linux-based systems.

The DSP Server used transparently by the Codec Engine for remote algorithms handles multi-threading and reentrancy issues on the DSP. For platforms such as the DM644x that treat the DSP as a black box, threading issues on the DSP are managed by the Codec Server integration.

However, there are still some important considerations.

4.6.1 Transaction Latency

GPP+DSP

It is important to consider transaction latency when running DSP algorithms from the GPP. For example, on the DM644x, the round-trip time required to schedule a DSP algorithm from the GPP limits transactions-per-second to approximately 7000. That is, the application can use the Codec Engine to run or control an algorithm up to 7000 times per second.

This may seem like plenty of headroom when considering typical frame rates of 30 to 50 per second. However, be aware that applications with a high density of channels may run up against this limit.

4.6.2 Multi- vs. Uni-Processor Performance

The VISA APIs wait for the function to return. Thus, your application needs to be multi-threaded if you want other threads to use the processing time while waiting for the DSP to perform its algorithms.

A discussion of managing multiple GPP threads in conjunction with the Codec Engine is beyond the scope of this document. See the documentation for your GPP operating system and/or middleware.

4.6.3 Local Performance

The Codec Engine is also optimized for local algorithm execution. The execution overhead is the same as that of xDAIS algorithms. The creation times are slightly higher.

4.7 What About Codec Engine Debugging?

Codec Engine modules, both on the application and the server side, provide plenty of trace information that can be activated, to reveal what's happening internally.

When any object in your application fails to be created—a codec or an Engine, either locally or on the DSP—follow the instructions in this section to turn on Codec Engine trace in order to do basic debugging. Section 4.8, *What About Software Trace?* provides details about Codec Engine tracing, although it is generally needed when debugging real-time, performance issues.

4.7.1 Codec Engine Debugging from the ARM on ARM+DSP Systems

To turn on the *minimal* level of Codec Engine debugging—catching all warnings and errors on the ARM and the DSP—simply set the environment variable CE_DEBUG=1 on the target prior to running your application. All the application's and server's CE warnings and errors will be printed, in correct order, to standard output.

Setting the CE_DEBUG environment variable causes Codec Engine on the ARM, besides printing its own trace information, to automatically collect any DSP server's trace information upon completing any CE API—whether it failed or was successful. The value of the variable only affects the detail level of the information collected and printed.

For a very detailed trace, set CE_DEBUG=2. This generates plenty of text, so we recommend that you run your application as follows:

```
root@146.252.161.13:~# CE_DEBUG=2 ./app.out [any app args here...] | tee log.txt
```

After the application runs, examine the log.txt file.

To turn on all tracing, set CE_DEBUG=3. You will usually need the help of a CE expert to analyze trace information generated this way.

4.7.2 Codec Engine Debugging on a DSP-only System

DSP-only

On a DSP-only system, assuming you are debugging your application from Code Composer Studio, you turn tracing on from your C code.

To do so, when you are ready to show Codec Engine trace information—which can be as soon as right after a call to CERuntime_init()—add the following lines to your code (assuming you have done #include <stdio.h> and #include <ti/sdo/ce/trace/gt.h>):

```
GT_setprintf( (GT_PrintFxn)printf );
GT_set( "*+67" ); /* turn on trace for warnings and errors */
```

The last line shows how much tracing to turn on. To turn on all tracing, use the following line instead:

```
GT_set( "*+01234567,GT_prefix=1235,GT_time=0" );
```

4.8 What About Software Trace?

GPP+DSP

A utility module you use to assist with software tracing in Codec Engine applications is the TraceUtil module. You can use this module for debugging and/or to collect real-time data.

Additionally, tools like SoC Analyzer can be developed to help display trace data. TraceUtil can be used to simplify the use of such tools.

TraceUtil lets you specify the amount of tracing you want and where you want it collected as follows:

❑ **At design time** by setting configuration file attributes

❑ **At start time** by setting environment variables

❑ **At run-time** by writing command strings to a named UNIX pipe

TraceUtil manages the three kinds of tracing that Codec Engine modules can produce:

❑ **Tracing on the GPP side.** Many Codec Engine and other GPP-side modules drop trace strings describing their state or warning and error messages.

❑ **Tracing on the DSP side.** DSP-side modules may provide trace information that can be collected by TraceUtil on the GPP-side.

❑ **DSP/BIOS logging on the DSP side.** DSP/BIOS provides the TRC and LOG modules to collect information about various DSP/BIOS system events such as task switching. You can use the TraceUtil module to enable such DSP/BIOS tracing remotely. Unlike the other kinds of trace, which are ASCII text, the DSP/BIOS log is a binary file.

As a supplement to TraceUtil, GPP-side code can also use printf(), or you can use the GNU Project Debugger (GDB) on GPP-side code.

4.8.1 Configuring TraceUtil at Design Time

GPP+DSP

To enable the TraceUtil module, your must add this line to your GPP application's configuration (.cfg) script. Any location in the script is fine.

```
var TraceUtil = xdc.useModule('ti.sdo.ce.utils.trace.TraceUtil');
```

The default TraceUtil settings cause the GPP application to:

- ❏ Print all GPP-side errors and warnings to the standard output.
- ❏ Collect DSP-side errors and warnings every 200 ms and print them to standard output.
- ❏ Not enable or capture any DSP/BIOS logging.

Constants are provided to set trace attributes for NO_TRACING, DEFAULT_TRACING, SOCRATES_TRACING, and FULL_TRACING.

Instead of using the default, you can add the following line to your .cfg file to print information in the form the SoC Analyzer can use:

```
TraceUtil.attrs = TraceUtil.SOCRATES_TRACING;
```

The set of attributes configured with the SOCRATES_TRACING option enable SoC Analyzer tracing and DSP/BIOS logging. GPP-side trace information is stored in the /tmp/cearmlog.txt file, DSP-side trace information is placed in /tmp/cedsp0log.txt, and DSP/BIOS logging goes to /tmp/bioslog.dat. Polling is initially disabled.

With this option, the application begins running with tracing disabled. To turn tracing on, you or your program must write a command to turn tracing on to the trace command pipe. See Section 4.8.6, *Controlling Trace at Run-Time Through a Named Pipe* for details.

Another option is to add the following line to your .cfg file to enable all types of tracing possible:

```
TraceUtil.attrs = TraceUtil.FULL_TRACING;
```

The output destinations are the same as for SOCRATES_TRACING, but FULL_TRACING enables all levels of trace for both the GPP and DSP.

You can further control the details of tracing behavior by setting individual TraceUtil.attrs fields in your .cfg file. For details, see the reference documentation for the ti.sdo.ce.utils.trace.TraceUtil module in the Configuration Reference, which is available at CE_INSTALL_DIR/xdoc/index.html

4.8.2 Supporting TraceUtil in Your Application's C Code

GPP+DSP

To collect the trace information that the DSP produces, you must add these lines of C code to your GPP application:

```
#include <ti/sdo/ce/utils/trace/TraceUtil.h>
 ...
/* call TraceUtil_start() after CERuntime_init() */
TraceUtil_start(engineName); /* engineName is a string */
 ...
TraceUtil_stop();            /* call at end of your app */
```

This code spawns a thread that collects all available DSP trace messages and dumps them to a file or standard output. (It also collects and stores DSP/BIOS LOG information if you want it to do so.)

4.8.3 Configuring the DSP Server for DSP/BIOS Logging

GPP+DSP

If you set TraceUtil on the GPP side to use DSP/BIOS logging, you must also have DSP/BIOS logging enabled in your DSP Server image. To do this, add the following line to your DSP Server's configuration script:

```
var LogServer = xdc.useModule('ti.sdo.ce.bioslog.LogServer');
```

If your DSP Server is incapable of DSP/BIOS logging, you will see GPP-side error/warning messages like the following:

```
LogClient_connect> Error: failed to locate server queue,
Check if your DSP image has DSP/BIOS logging enabled
```

```
LogClient_fwriteLogs> Warning: not connected to the DSP/BIOS
log server on the DSP, cannot collect any DSP/BIOS log data.
```

4.8.4 Configuring the DSP Server To Redirect Trace Output

When debugging a DSP Server or single-processor DSP application using Code Composer Studio (CCStudio), you can direct trace information to go directly to CCStudio's output window. To do this, modify the main() routine to make the following call before calling CERuntime_init():

```
GT_setprintf((GT_PrintFxn)printf);
```

This causes each trace call to map to the DSP standard I/O library's printf() function, which sends output to the CCStudio console window.

Note that the argument to the GT_setprintf() function can be any function that takes (char *format, …) arguments. So, for example, you could provide your own function that, for example, sends trace information to a serial port.

4.8.5 Configuring TraceUtil at Application Start Time

GPP+DSP

Before you run your TraceUtil-enabled application, you can set one or more of the following environment variables to override the TraceUtil attributes you specified in your .cfg script:

❏ **CE_TRACE.** Mask for the GPP-side tracing. See Section 4.8.7, *Trace Mask Values* for mask details. For example:

```
CE_TRACE="*=0567;OM-0"
```

❏ **CE_TRACEFILE.** Specify the output file for GPP trace information. This can be a full path (for example, /tmp/local.txt) or a path relative to the executing application. If the file can't be opened (for example, if this points to a directory that doesn't exist), the trace goes to the standard output. For example:

```
CE_TRACE="trace/armtrace.txt";
```

❏ **CE_TRACEFILEFLAGS.** Set file creation flags for all files to be opened. Use the standard fopen() flags—"a" means append; "w" means over-write. For example:

```
CE_TRACEFILEFLAGS="a"
```

❏ **TRACEUTIL_DSP0TRACEFILE.** Specify the output file for DSP trace information. As with CE_TRACEFILE, this can be a full path or a path relative to the executing application. If the file cannot be opened, the trace goes to the standard output.

❏ **TRACEUTIL_DSP0BIOSFILE.** Specify the output binary file for the DSP/BIOS log. This can be a full path or a path relative to the executing application. If the file cannot be opened, the log information is not collected.

❏ **TRACEUTIL_DSP0TRACEMASK.** Mask for DSP-side tracing. See Section 4.8.7, *Trace Mask Values* for mask details. For example:

```
TRACEUTIL_DSP0TRACEMASK="*+01;ti.bios=01234567"
```

❏ **TRACEUTIL_REFRESHPERIOD.** Specify the number of milliseconds to sleep before the GPP-side collects the next set of DSP-side trace information. Your choice should vary depending on the amount of trace generated and the size of the trace logs. Failure to set this low enough may result in data loss.

❏ **TRACEUTIL_CMDPIPE.** The name of a UNIX named pipe (for example, "fifo") to which the TraceUtil module should listen for run-time trace commands.

❏ **TRACEUTIL_VERBOSE.** Set to 1 if you want TraceUtil to print the trace settings (masks and files) it is using and where it got them from. Set to 2 or higher to show more debugging information. In most cases, TRACEUTIL_VERBOSE=1 is recommended.

If you use the bash shell on Linux, it is especially convenient to set environment variables in the same line where you start your application, so they apply to that execution of the application only:

```
CE_TRACE="*+5" CE_TRACEFILE="mylog" TRACEUTIL_VERBOSE=1 ./app.out
```

Note that these environment variables are read only at application startup time. Changing them after the application is running has no effect.

Note: The CE_DSP0TRACE environment variable described in previous versions is ignored if you enable the TraceUtil module.

4.8.6 Controlling Trace at Run-Time Through a Named Pipe

GPP+DSP

If the TRACEUTIL_CMDPIPE environment variable is set to a valid name or if the TraceUtil.attrs.cmdPipeFile configuration option is set, the TraceUtil thread listens for any trace commands that appear in the pipe.

The SOCRATES_TRACING profile uses the command pipe feature. The pipe is /tmp/cecmdpipe by default, but the name can be overridden by setting the TRACEUTIL_CMDPIPE environment variable.

When you start a SoC Analyzer-enabled application, it initially provides no trace other than (potentially) warnings and errors. Ways to override this initial behavior are:

❏ Define the following environment variables before starting the application. See Section 4.8.7, *Trace Mask Values* for mask details.

```
CE_TRACE="*+5"
TRACEUTIL_DSP0TRACEMASK="*+5,ti.bios=3"
```

❏ Issue the following command before running the application:

```
mkfifo /tmp/cecmdpipe; echo socrates=on > /tmp/cecmdpipe
```

The mkfifo command is necessary only for the first run; TraceUtil creates the pipe if it doesn't exist and doesn't delete it at the end.

When a SoC Analyzer-enabled application is running, you can turn tracing on by writing the following string to the /tmp/cecmdpipe file:

```
socrates=on
```

You can turn tracing off by writing the following string to the /tmp/cecmdpipe file:

```
socrates=off
```

The socrates=on and socrates=off pipe commands are aliases for a group of appropriate masks. These aliases are defined in the TraceUtil.xdc file.

The best way to write a string to the pipe is to use an open-write-close sequence (as opposed to keeping the pipe file open for writing throughout the session). The [create_pipe]->open_pipe->write_text->close_pipe sequence can be either done from the command line, from a script (as in the example above), or from a C program

The following list shows the supported trace pipe commands.

❑ tracemask={GPP trace mask value}
Sets the GPP-side trace mask. For example,
`tracemask=*+01234567,OM-1`

❑ dsp0tracemask={DSP0 trace mask value}
Sets the DSP0 trace mask. For example,
`dsp0tracemask=*-1,ti.bios-012`

❑ refreshperiod={number of milliseconds}
Sets the refresh period for DSP0 trace and log collection. If 0, there is no collection until a non-zero refreshperiod is specified. For example, `refreshperiod=10`

❑ resetfiles (no arguments)
Resets all open files for GPP trace, DSP0 trace, and DSP0 log (those that are currently in use) by truncating the files to 0 bytes.

Note that only one command per line should be written to the trace pipe. However, as was done for socrates=on, you can define—in the application's configuration script—command pipe aliases to issue several pipe commands. For example:

```
var TraceUtil =
   xdc.useModule('ti.sdo.ce.utils.trace.TraceUtil');
TraceUtil.attrs.cmdAliases = [
   {
       alias: "mycommands_1",
        cmds:  [
                 "resetfiles",
                 "tracemask=*+5",
                 "dsp0tracemask=*+5,ti.bios+3",
                 "refreshperiod=200",
               ],
   },
   {
       alias: "mycommands_2",
        cmds:  [
                 "tracemask=*-5",
                 "refreshperiod=0",
                 "dsp0tracemask=*-5,ti.bios-3"
               ],
   },
   /* and so on -- no limit on the number of aliases */
];
```

4.8.7 Trace Mask Values

GPP+DSP

Every VISA module can supply real-time trace output. This output can be enabled and disabled on a per-module basis at run-time. Each module can supply up to 8 levels of trace information. Several levels have universal meaning. For example, 0 corresponds to "entry" tracing (each module entry point displays its name and the arguments passed to it).

The NO_TRACING, DEFAULT_TRACING, SOCRATES_TRACING, and FULL_TRACING constants you can use in your application configuration provide easy ways to set commonly desired tracing levels. If you want custom trace levels for various modules, you can do that using the information in this section.

You can set trace masks (in a configuration file, environment variable, or command pipe) to a name/value pair or sequence of pairs. The name indicates the module whose tracing should be set, and the value indicates the trace levels enabled for that module.

For example, the following setting uses * (asterisk) as a wildcard to enable full Codec Engine tracing. This results in a lot of output, but is often useful in identifying what is going on internally.

```
setenv CE_TRACE "*=01234567"
```

You can also set modules to different trace levels in the same environment variable. To configure more than one module, you can separate masks with a semi-colon. Any module settings after the asterisk name/value pair override the wildcard setting.

For example, the following sets all modules to "1567", except "OM" (which you don't want to see), and "CV" (for which you want to see all information):

```
setenv CE_TRACE "*=1567;OM=;CV=01234567"
```

The following table lists the module names you can use in masks. It shows which modules apply to GPP trace (CE_TRACE) and which apply to DSP trace (TRACEUTIL_DSP0TRACEMASK):

Module Abbreviation	**Description**	**Valid for GPP**	**Valid for DSP**
OC	OSAL Communication. Abstracts messaging APIs across operating systems.	Yes	Yes
OP	OSAL Process. Abstracts process APIs across operating systems and loads the server image to the DSP.	Yes	No
OM	OSAL Memory. Abstracts memory APIs across operating systems.	Yes	Yes
OG	OSAL Global. Abstracts generic APIs across operating systems.	Yes	Yes
CE	Codec Engine runtime APIs.	Yes	Yes
CS	Server Runtime APIs	Yes	No
CV	Codec Engine VISA APIs.	Yes	Yes
CR	Codec Engine - RMS. Codec Engine’s server daemon.	No	Yes
CN	Codec Engine - Node. Instantiates codecs and communicates through custom skeletons.	No	Yes
ti.sdo.ce.osal.AlgMem	OSAL Algorithm. Used for creating, deleting, and controlling algorithms.	Yes	Yes

Module Abbreviation	Description	Valid for GPP	Valid for DSP
ti.sdo.ce.osal.power	OSAL Power. Used in heterogeneous configurations to power on/off a server. Not supported in all releases.	Yes	Yes
ti.bios *	Control the DSP/BIOS TRC module.	No	Yes
GT_prefix *	Control what information is included in each trace line prefix.	Yes	Yes
GT_time *	Control the format of timestamps in trace lines.	Yes	Yes

***** The ti.bios, GT_prefix, and GT_time modules are special in that they are not affected by module wildcards in a trace mask. You must name them directly to change their flags. For example:

```
setenv CE_TRACE "*=67;GT_prefix=12"
setenv TRACEUTIL_DSP0TRACEMASK "*=567;ti.bios=012"
```

For the standard modules, the levels 0 through 7 report the following types of messages:

- 7 = fatal errors
- 6 = warnings
- 5 = benchmarks
- 4 through 1 = internal Codec Engine messages
- 0 = function enter/exit reporting

For the special modules (ti.bios, GT_prefix, and GT_time) the levels have special meanings as follows:

Level	ti.bios Module	GT_prefix Module	GT_time Module
none	no DSP/BIOS logging	no prefix	microseconds in hex form (0xa0cf80fe)
0	TRC_LOGCLK	short module name	microseconds, in decimal form (4,021,348us)
1	TRC_LOGPRD	long module name	seconds (0.004s)
2	TRC_LOGSWI	trace line class (level)	delta in microseconds, excluding print time (+0,000,259us)
3	TRC_LOGTSK	thread ID	---
4	TRC_STSHWI	stack address	---

Level	ti.bios Module	GT_prefix Module	GT_time Module
5	TRC_STSPRD with TRC_STSSWI and TRC_STSTSK	time stamp	---
6	TRC_USER0	---	---
7	TRC_USER1	---	---

For GT_prefix, the default levels used are 1, 3, and 5.

For GT_time, DSP time stamps are always in cycles, not in microseconds. Setting GT_time currently makes sense only on the GPP.

Chapter 5

Integrating an Engine

This chapter describes how the Engine Integrator should configure an Engine for use by the application developer.

Topic **Page**

5.1 Overview

As described in Section 1.4.3, *Engine Integrator*, the Application Author gets an Engine configuration from the Engine Integrator. In practice, these roles may be shared by one person.

An Engine configuration is stored in an XDC *.cfg file and processed by the makefile using package.xdc to generate a *.c file and a linker command file (*.xdl) from the *.cfg file. For build instructions, see CE_INSTALL_DIR/examples/build_instructions.html.

The Codec Engine example applications typically support GPP+DSP, GPP-only, and DSP-only usage. For GPP+DSP usage, the Engine configuration is often defined in a file named remote.cfg, designating codecs run remotely on a DSP server. For GPP-only or DSP-only usage the configuration often resides in a file named "local.cfg", designating that the codecs run locally, on the same CPU as the main application.

An Engine configuration can include the names of the Engines, as well as the codecs and their names within each Engine, whether each codec is local or remote relative to the application, which groups each codec should be integrated into (for environments that support resource sharing), the name of the Server image if a particular Engine contains remote codecs, and more.

In its simplest form, however, the Engine configuration script can simply name the DSP server package and the corresponding server executable, and automatically import all the server's codec definitions. This makes the codecs available for use from the application as remote codecs.

For the latter form of configuration to work, it is important that you receive your DSP server in a package—which is the default server delivery method as of CE 2.00—instead of getting just one DSP server binary executable.

5.2 A Reusable Example

The video_copy example uses the following ceapp.cfg configuration file:

```
/*   ======== ceapp.cfg ======== */
/* use the tracing utility module */
var TraceUtil = xdc.useModule('ti.sdo.ce.utils.trace.TraceUtil');
//TraceUtil.attrs = TraceUtil.SOCRATES_TRACING;

/* set up OSAL */
var osalGlobal = xdc.useModule('ti.sdo.ce.osal.Global');
osalGlobal.runtimeEnv = osalGlobal.DSPLINK_LINUX;

/* ======== Engine Configuration ======== */
var Engine = xdc.useModule('ti.sdo.ce.Engine');
var myEngine = Engine.createFromServer(
    "video_copy",        // Engine name (as referred to in the C app)
    "./video_copy.x64P", // path to server exe, relative to its package dir
    "ti.sdo.ce.examples.servers.video_copy.evmDM6446" // server package
);
```

Most of this configuration file can be used as is. For your applications, you should modify the portions shown in bold: the name of your engine (your choice), the name of the server executable, and the name of the server executable's package.

The codecs included in the server will be available to the application under their original names, which for convenience are shown during the application's build—as shown in this excerpt:

```
configuring ceapp.x470MV from package/cfg/ceapp_x470MV.cfg ...
Info: Configuring engine named 'video_copy' from the info file for DSP server
'./video_copy.x64P', located in package
'ti.sdo.ce.examples.servers.video_copy.evmDM6467':
     Target app will look for the DSP server image 'video_copy.x64P' in its current
directory.
      Adding codec 'viddec_copy'
(ti.sdo.ce.examples.codecs.viddec_copy.VIDDEC_COPY), scratch groupId=0
      Adding codec 'videnc_copy'
(ti.sdo.ce.examples.codecs.videnc_copy.VIDENC_COPY), scratch groupId=0

Info: Reading DSP memory map from the info file for DSP server './video_copy.x64P',
located in package 'ti.sdo.ce.examples.servers.video_copy.evmDM6467':
...
```

It is good practice to verify that the information in this build log (server executable/package name, codec names, scratch groups) matches what you expect.

Even though the engine is configured from the information stored in the server package, you still must have all the codecs included in the server in your package path. If you do not have a required codec package, the build will fail. If you have a codec package of a different version than the one used to build the server, you will get a warning.

5.2.1 Advanced Engine Creation

The Engine.createFromServer() method in the previous configuration example is available as of Codec Engine 2.00. It replaces the lower-level Engine.create() method for most common use cases.

The lower-level engine creation method allows you to add non-local codecs, use different names for remote codecs, omit codecs you don't have, and so on. You may need it in some advanced cases. The example that follows uses the lower-level Engine.create() method:

```
/* get various codec modules; i.e., implementation of codecs */
var VIDDEC_COPY =
    xdc.useModule('ti.sdo.ce.examples.codecs.viddec_copy.VIDDEC_COPY');
var VIDENC_COPY =
    xdc.useModule('ti.sdo.ce.examples.codecs.videnc_copy.VIDENC_COPY');

/*  ======== Engine Configuration ========  */
var Engine = xdc.useModule('ti.sdo.ce.Engine');
var myEngine = Engine.create("video_copy", [
    {name: "videnc_copy", mod: VIDENC_COPY, local: false},
    {name: "viddec_copy", mod: VIDDEC_COPY, local: false}
]);

myEngine.server = "./video_copy.x64P";

/*  ======== Server memory map (DSPLINK) configuration ========
 *  This table must match exactly the addresses and sizes of segments in the Server’s
 *  BIOS configuration (.tcf) script. There is exactly one "main", one "link", and
 *  one "reset" segment type, and zero or more of "other" types. */
myEngine.armDspLinkConfig = {
    memTable: [
        ["DDRALGHEAP", {addr: 0x88000000, size: 0x07A00000, type: "other"}],
        ["DDR2",       {addr: 0x8FA00000, size: 0x00400000, type: "main" }],
        ["DSPLINKMEM", {addr: 0x8FE00000, size: 0x00100000, type: "link" }],
        ["RESETCTRL",  {addr: 0x8FF00000, size: 0x00000080, type: "reset"}],
    ],
};
```

Again, characters in bold show what you may need to change in your application.

5.3 Understanding Engine Configuration Syntax

The syntax used in Engine configurations is based on JavaScript, which is also used for the Tconf language used to statically configure DSP/BIOS. (See SPRU007 for details.)

Unlike the JavaScript used in web pages, an object model is provided to meet the needs of Engine configuration. This object model is documented in the Configuration Reference, which is available at CE_INSTALL_DIR/xdoc/index.html.

For example, the following statements cause the Global module in the ti.sdo.ce.osal package to be made available to the configuration script. It then sets the runtimeEnv attribute of the Global module to DSPLINK_LINUX. This indicates that an application that uses this Engine can use the DSP/BIOS Link and Linux operating environments.

```
var osalGlobal = xdc.useModule( 'ti.sdo.ce.osal.Global' );
osalGlobal.runtimeEnv = osalGlobal.DSPLINK_LINUX;
```

To see the other options for the runtimeEnv attribute, follow these steps:

1) Open CE_INSTALL_DIR/xdoc/index.html to see the Configuration Reference. Depending on your browser, you may need to enable active content to view the list of nodes on the left.
2) Click the link to the ti.sdo.ce.osal package.
3) Click the link to the Global module.
4) You see the valid settings for runtimeEnv and other documentation for the Global module.
5) Click "Back" in the upper-right corner of the window. Note that the usual Back button in your browser does not function as expected in this online help system.

After setting the runtime environment, the example ceapp.cfg configuration file gets access to the codec modules it will need. For example:

```
var VIDDEC_COPY =
   xdc.useModule('ti.sdo.ce.examples.codecs.viddec_copy.VIDDEC_COPY');
```

This statement "uses" the VIDDEC_COPY module in the "ti.sdo.ce.examples.codecs.viddec_copy" package, and stores the handle to it in a variable named VIDDEC_COPY. A similar statement gets the corresponding video encoder. You can modify these statements to reference any of the codecs provided with Codec Engine.

The ti.sdo.ce.examples.codecs.viddec_copy package corresponds to CE_INSTALL_DIR/examples/ti/sdo/ce/examples/codecs/viddec_copy and VIDDEC_COPY matches the VIDDEC_COPY.xdc filename in that directory.

The next group of statements declare the contents of an Engine.

```
var Engine = xdc.useModule('ti.sdo.ce.Engine');
var vcr = Engine.create("video_copy", [
    {name: "videnc_copy", mod: VIDENC_COPY, local: false},
    {name: "viddec_copy", mod: VIDDEC_COPY, local: false}
]);
```

First, they make the Engine module in the ti.sdo.ce package available to the script. Then they use the create() method of the Engine module to create an Engine. As with the Global module in the ti.sdo.ce.osal package, you can use the Configuration Reference online help to get details about the Engine module in the ti.sdo.ce package.

Each Engine has a name that will be used by the Application Author in the Engine_open() API they call. In this case, the Engine name is "video_copy".

The create method then expects an array of algorithm descriptions. Each algorithm description contains the following fields:

❑ **name.** This string specifies the "local" name to be used by the Application Author to identify an algorithm to instantiate in the VIDENC_create and VIDDEC_create VISA APIs.

❑ **mod.** This field is a reference that identifies the actual module implementing the algorithm to instantiate. This is the same as the name declared as a variable in the previous statement that called xdc.useModule to get the ti.sdo.ce.examples.codecs.viddec_copy.VIDDEC_COPY module.

❑ **local.** If true, the algorithm is instantiated on the "local" CPU. Otherwise, the Codec Server creates a remote instance of the algorithm identified by mod.

5.3.1 Framework Components Configuration

The example configuration files—remote.cfg (for GPP+DSP) and local.cfg (for GPP-only and DSP-only)—often configure the ti.sdo.fc.dskt2.DSKT2 and ti.sdo.fc.dman3.DMAN3 modules, which are part of Framework Components. DSKT2 is the xDAIS algorithm memory allocation manager, and DMAN3 is the DMA manager. For details on configuring these modules, see the Framework Components documentation in CE_INSTALL_DIR/xdoc/index.html.

Index