Codec Engine Algorithm Creator User's Guide

Literature Number: SPRUED6C
September 2007

Texas Instruments

IMPORTANT NOTICE

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

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

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

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

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

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

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

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

Products		Applications	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
RFID	www.ti-rfid.com	Telephony	www.ti.com/telephony
Low Power Wireless	www.ti.com/lpw	Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

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

Preface

About This Book

The intended audience for this document is the Algorithm Creator, who creates an algorithm/codec for use by Codec Engine server integrators and application developers.

This manual may also be used by Server Integrators who want to package a non-xDM algorithm for use in a Codec Server.

This manual tells what steps the Algorithm Creator should take to package an algorithm for Codec Engine usage.

Additional Documents and Resources

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

- *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
- *Codec Engine Application Developer's Guide* (SPRUE67)
- *Codec Engine Server Integrator's Guide* (SPRUED5)
- *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)
- *XDC User's Guide* and other XDC documents.
  XDC_INSTALL_DIR/doc/index.html

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.

Trademarks

Contents

1 Packaging Overview 1-1
This chapter provides an overview of the packaging model used for Codec Server applications.
1.1 Overview. 1-2
1.2 Getting Started 1-3
1.2.1 XDCPATH Environment Variable. 1-3
1.2.2 Editing user.bld. 1-4
1.3 Building a Package. 1-5
1.4 Creating a Release Package 1-7

2 Developing an xDM Codec 2-1
This chapter describes how to package an xDAIS-DM algorithm for use in Codec Server applications.
2.1 Overview. 2-2
2.1.1 What's in a Codec Package? 2-2
2.2 Creating Package Files 2-3
2.2.1 package.xdc File 2-4
2.2.2 package.xs File 2-5
2.2.3 package.bld File 2-6
2.2.4 <MODULE>.xdc File 2-6
2.2.5 <MODULE>.xs File 2-7

3 Supporting Non-xDM Algorithms 3-1
This chapter describes how to create stubs and skeletons to allow non-xDM algorithms to be used remotely in Codec Server applications.
3.1 Overview. 3-2
3.1.1 Tasks 3-2
3.1.2 Roles and Interactions 3-3
3.2 Core Algorithm Interface Requirements 3-4
3.3 Creating Codec Engine Extensions 3-5
3.4 Designing a New Application Interface. 3-5
3.4.1 Creation and Deletion APIs 3-6
3.4.2 Processing Function APIs 3-7
3.5 Developing Stubs and Skeletons 3-9
3.5.1 Developing Codec Engine Stubs 3-9
3.5.2 Developing Skeletons for CE Extensions. 3-14
3.6 Binding Stubs and Skeletons to Framework Extensions 3-19

3.7 Packaging and Configuring the Core Algorithm 3-20
3.8 Non-xDM Stub and Skeleton Example: SCALE 3-22

Chapter 1

Packaging Overview

This chapter provides an overview of the packaging model used for Codec Server applications.

Topic **Page**

1.1 Overview **1–2**

1.2 Getting Started **1–3**

1.3 Building a Package **1–5**

1.4 Creating a Release Package **1–7**

1.1 Overview

The Codec Engine framework is designed to work with any xDAIS-compliant algorithm that is delivered in a custom RTSC (Real-Time System Component) package. This user guide explains how to package a xDAIS-compliant algorithm, also referred to as a codec, so that it is consumable by the Codec Engine's Configuration Kit.

For those algorithms that implement the xDAIS Digital Media (xDM) interface, the RTSC packaging is a straight-forward procedure, as explained in Chapter 2 of this guide. The Codec Engine framework natively supports an application programming interface—the VISA API—which the application programmer uses to instantiate and use xDM-compliant algorithms.

A xDAIS algorithm that does not implement the xDM interface may still be used within the Codec Engine framework. However, this requires some advanced development work, which involves creating an extension package with custom stub and skeleton functions. Chapter 3 provides detailed instructions along with an example, the SCALE API and SCALE_TI algorithm.

This guide does not describe how to design and implement the actual xDAIS- or xDM-compliant algorithms. For additional information on developing xDAIS or xDM algorithms see the *xDAIS-DM (Digital Media) User Guide* (SPRUEC8).

For details on Codec Engine and Codec Server development, see the following documents:

- *Codec Engine Application Developer's Guide* (SPRUE67)
- *Codec Engine Server Integrator's Guide* (SPRUED5)

1.2 Getting Started

To build custom Codec Engine Configurations, TI provides the Codec Engine Configuration Kit which is based on Texas Instruments' RTSC (Real-Time System Component) technology, utilizing the XDC (eXpress DSP Component) foundational tools. This toolkit is also called the XDC Configuration Kit.

The XDC Configuration Kit enables embedded and DSP developers to automate their target content configuration, optimization, delivery, building, and testing.

The easiest strategy for creating a package for your algorithm or codec is to make a copy of an example package and modify it to meet your needs. This manual uses the files in the ti.sdo.ce.examples.codecs.viddec_copy and ti.sdo.ce.examples.codecs.scale packages, which are provided with the Codec Engine distribution as a starting point.

Each example package directory contains a makefile, which invokes the xdc command to build the package. Additionally, the file xdcpaths.mak is provided in a common folder and referenced by each makefile for global XDC paths and other installation settings.

To use the XDC tools, you must configure the following:

- ❏ XDCPATH environment variable (by editing xdcpaths.mak)
- ❏ build targets (by editing user.bld)
- ❏ codegen tools and setting for each target (by editing user.bld)

Documentation for XDC is provided with the XDC tools. Go to the xdctools_#_##/doc subdirectory and open the index.html file for a list of and links to XDC documentation. For example, the xdc.html file contains man page information about the xdc command line and the environment variables it recognizes.

1.2.1 XDCPATH Environment Variable

The XDCPATH environment variable contains a list of directories that contain packages. This path is used to locate packages that are used by the package being built.

By default, Codec Engine's XDCPATH is defined in CE_INSTALL_DIR/examples/buildutils/xdcrules.mak to use the definition of the XDC_PATH environment variable. XDC_PATH is defined in the CE_INSTALL_DIR/examples/xdcpaths.mak file.

Therefore, to set XDCPATH, you should edit the xdcpaths.mak file. This file contains definitions of where the Codec Engine, DSP/BIOS, XDC tools, and individual Codec Engine packages are located.

The xdcpaths.mak file is included by the individual makefiles for all example codecs, servers, and applications. Open the xdcpaths.mak file in a text editor. Uncomment lines and edit the directory paths to specify where the installations for the following components are located. Use full, absolute paths.

- CE_INSTALL_DIR. Codec Engine location.
- BIOS_INSTALL_DIR. DSP/BIOS location.
- XDC_INSTALL_DIR. XDC tools location.

If your Codec Engine distribution does not include a cetools directory, you may also need to define the following variables:

- XDAIS_INSTALL_DIR. xDAIS location (xDAIS 5.00 or greater).
- DSPLINK_INSTALL_DIR. DSP/BIOS Link location.
- CMEM_INSTALL_DIR. CMEM location.
- FC_INSTALL_DIR. Framework Components location.

Most build problems occur if one of these *_INSTALL_DIR variables is incorrect. Make sure there are no extra spaces, that every individual path (segment separated by the semicolon) is correct, character for character.

Full paths are recommended for use in the XDCPATH environment variable. Relative paths are likely to cause problems. If used, relative paths are treated as relative to the package being built rather than the directory where the xdc command is invoked. Thus, a relative path refers to a different location for each package being built.

1.2.2 Editing user.bld

The config.bld script is run prior to all build scripts. It sets defaults for targets and platforms. It then attempts to find a user.bld script that sets the property rootDir which is used to locate the code generation tools for each target as well as setting other properties to override the default settings in config.bld.

You should edit the CE_INSTALL_DIR/examples/user.bld file as follows:

- Set rootDir to point to your installation of the TI DSP code generation tools for your platform.

- Add new target definitions if necessary. Refer to the target definition format from the CE_INSTALL_DIR/packages/config.bld file. You must also define the rootDir for that target.

  For example, the default file contains the following settings for the C64P platform (where "remarks" evaluates to a number of related compiler options):

```
var C64P = xdc.useModule('ti.targets.C64P');
C64P.platform = "ti.platforms.evmDM6446";
C64P.ccOpts.prefix += " --mem_model:data=far " + remarks;
```

  You can modify these settings, or create settings for a platform other than C64P.

- Add any new targets to the list of available targets by editing the following Build.targets list to include an additional target. Comment out any targets for which you do not want to build.

```
Build.targets = [
    C64P,
    MVArm9,
    Linux86,
];
```

1.3 Building a Package

You deliver a codec that is to be consumable by the Codec Engine as a "package". 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 generated metadata files. 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.

The following five types of files are required to form a package:

- Package definition files:
  - **package.xdc.** Defines the package by specifying its contents (that is, modules), and notes any packages that this package depends upon.
  - **package.xs.** Provides the JavaScript implementation of the methods declared or inherited in package.xdc. (required only when there are methods to implement)
- Module definition files. For each *MODULE* declared in package.xdc there must be a *MODULE*.xdc file, and optionally a *MODULE*.xs file:
  - ***MODULE*.xdc.** Defines the module by specifying its contents: variables and methods.
  - ***MODULE*.xs.** Provides the JavaScript source code for any methods declared in MODULE.xdc (required only when there are methods to implement).
- Package build script for configuring and building the package:
  - **package.bld.** Describes the steps required to build and release the package. Contains JavaScript instructions for building any libraries, object files, or executables that are part of the package, and specifications of what goes into a release package.

Additionally, any source and header files needed to build the artifacts and any pre-built files to be included in a release package must be included in the package directory.

Once the package directory contains the required contents and the required environment variables have been set, you can build the package with the following command:

```
% xdc
```

To simplify the task of creating RTSC packaging for an xDM codec, we recommend making a copy of an example package and modifying it to meet your needs. Each example package directory contains a makefile, which invokes the xdc command to build the package using the xdcpaths.mak for global XDC paths and other installation settings. In this case the xdc command can be indirectly issued via the following command:

```
% gmake
```

1.4 Creating a Release Package

Once a buildable package exists, use the following command to create a .tar file containing the binary distributions of the package.

```
% xdc release
```

The package build script, package.bld, is executed when the package producer builds the package. This script names the releases and specifies the files to be included in each.

The generated tar file contains the binary distribution of the package, which includes all the files required for RTSC tools to recognize the package. It also includes any additional files or directories that are specified in package.bld by setting the optional Pkg.otherFiles attribute. The binary files (libraries and executables) built using the package.bld script are automatically included in any release and do not have to be explicitly listed in Pkg.otherFiles.

The default name of the created tar file is obtained from the package name, replacing the periods (.) with underscores (_).

The source or header files used in the package build process are not included in a release package, unless you explicitly add them using Pkg.otherFiles. For example, by adding the following line to package.bld, all files in the src and include directories and the file readme.txt are released:

```
Pkg.otherFiles = [
  'src',              /* the whole src subdirectory */
  'include',          /* the whole include subdirectory */
  'readme.txt',
];
```

In order to include a library, legacy.lib, which was built outside the package using a legacy build system, you can copy the library into the package directory and include it in Pkg.otherFiles in the package.bld file.

```
Pkg.otherFiles = [
  'legacy.lib',
  'readme.txt',
];
```

Chapter 2

Developing an xDM Codec

This chapter describes how to package an xDAIS-DM algorithm for use in Codec Server applications.

Topic **Page**

2.1 Overview. 2–2

2.2 Creating Package Files 2–3

2.1 Overview

This chapter explains how to package an xDAIS-DM compliant codec to make it consumable by the Codec Engine's Configuration Kit. By following these instructions, an algorithm can be used in a Codec Server.

This manual does not describe how to create an xDAIS-DM compliant codec. For information on xDAIS-DM (also called "xDM"), see the *xDAIS-DM (Digital Media) User Guide* (SPRUEC8).

The easiest strategy for creating a package for your algorithm or codec is to make a copy of an example package and modify it to meet your needs. This chapter uses the files in the ti.sdo.ce.examples.codecs.viddec_copy package provided with the Codec Engine distribution as a starting point.

2.1.1 What's in a Codec Package?

The following figure illustrates the contents of a codec package. In this example, the vendor "VEND" delivers a VISA compliant codec module "MOD" as a XDC package.

The package you distribute contains the compiled library file, two .xs files, and two .xdc files. The .xs and .xdc files provide metadata about the package. The source code for the package is shown in brackets to the right of the library icon. You are not required to distribute the source files with a package, though you can choose to if you wish.

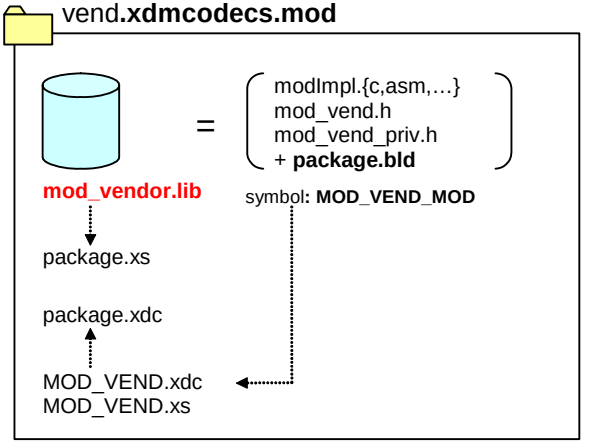

Figure 2–1 Package Contents

2.2 Creating Package Files

As an example, we look at the files in the ti.sdo.ce.examples.codecs.viddec_copy package provided in the Codec Engine's "examples" repository. This codec implements the ti.sdo.ce.video.IVIDDEC interface. That is, it is an xDAIS-DM compliant Video Decoder. Its "decoding" simply passes data through; input buffers are copied to output buffers.

The following files are required in a released package. Filenames in bold are files you edit. Other files are generated when you build the codec.

- **package.xdc.** Package's static properties. For example, name and dependencies. See Section 2.2.1.
- **package.xs.** Package properties that can vary across platforms and configurations. For example, library name. See Section 2.2.2.
- **VIDDEC_COPY.xdc.** Codec's static properties. For example, xDM class, IALG Fxn table, and IDMA Fxn table. The filename will differ for your codecs. See Section 2.2.4.
- **VIDDEC_COPY.xs.** Codec properties that can vary across platforms and configurations. The filename will differ for your codecs. For example, stack size. See Section 2.2.5.
- viddec_copy.a##. The compiled library for the codec for a particular platform.

The following additional files are required in order to build the package. If you want to enable consumers of your codec to rebuild the package, your release should also include these files.

- **package.bld.** The package's build/release script. This defines the contents of the release. See Section 2.2.3.
- **viddec_copy_ti.h** and **viddec_copy_ti.priv.h.** Public and private header files. The filenames will differ for your codecs.
- **viddec_copy.c.** The xDAIS-DM implementation of the codec.
- **makefile.** The file used to build the codec. For the examples, this file includes the xdcpaths.mak file and the xdcrules.mak file.

The following files are generated when you build the codec, and you should not modify them. You should also not modify any files that begin with "." or any files in the "package" subdirectory, since these are all automatically generated.

- .xdcenv.mak. This file is generated during builds. It describes the user's environment when the package was built.
- package.mak. This file is generated from package.bld.
- viddec_copy.a##.mak. The build file for a particular target and release. These files are generated from package.bld.
- .dlls, .executables, .interfaces, .libraries. These files provide information about the build. These files are all generated during builds; you should not modify them with a text editor.
- "package" directory. These files are used by the XDC Configuration Kit. The files in this directory are all generated during builds; you should not modify them with a text editor.

2.2.1 package.xdc File

The package.xdc file defines the package's static properties such as its name and any dependencies.

For example, the package.xdc file for the viddec_copy codec references its dependency on the video portion of the xDM VISA APIs as follows:

```
requires ti.sdo.ce.video;
```

It then uses the following statements to specify that the name of the package is "ti.sdo.ce.examples.codecs.viddec_copy" and that this package contains a single Module called "VIDDEC_COPY".

```
package ti.sdo.ce.examples.codecs.viddec_copy {
    module VIDDEC_COPY;
}
```

2.2.2 package.xs File

The package.xs file specifies package properties that can vary across platforms and configurations. For example, the default package.xs file for the VIDDEC_COPY example sets the library name as follows:

```
function getLibs(prog)
{
  /* "mangle" program build attrs to directory name */
  var name = "lib/viddec_copy.a" + prog.build.target.suffix;

  if (prog.build.target.isa == "64P") {
      /* return the library name: name.a<arch> */
      print("  will link with " + this.$name + ":" + name);
  }
  else {
      name = "";
  }
  return (name);
}
```

The getLibs() function returns the library name that the package exports. The XDC build system calls this special function—whenever an XDC application "uses" a package—to link against the package library matching the program's configuration. The exported library does not have to be built using the housing package's build script. It can be built with any legacy build system and placed into the package.

Note that in the previous example, the getLibs() function returns a library only if you are currently building a DSP server or a DSP standalone Codec Engine application. It does not return a library if you are building an ARM application. This is what the following if() statement in the function does:

```
if (prog.build.target.isa == "64P") {
```

The string "64P" is a code for DSP target binaries. (ARM target binaries have a different code, "MVArm9".) If you are building an ARM application and this codec package gets consumed, the condition above is false and the getLibs() return an empty library name—which is what you want when you don't provide any libraries for the ARM.

If you develop codecs only for the DSP, it is important to have this same if() block in your getLibs() function. Note that the example codecs contain libraries for both the DSP and for the ARM, so they don't have this if() clause.

2.2.3 package.bld File

The package.bld file defines the build-related attributes for the package. For example, it specifies whether to build a "debug" or "release" version.

This file does not need to be shipped with the package. It should be shipped if a release variant of the codec is "rebuildable".

If you are converting the package.bld file from the VIDDEC_COPY example, change the following array to list the source files for your codec:

```
var SRCS = ["viddec_copy.c", "other source file", ...];
```

Also, change the following line to reference your codec library:

```
Pkg.addLibrary("lib/viddec_copy", targ).addObjects(SRCS);
```

See the XDC User's Guide which is installed with the XDC tools at XDC_INSTALL_DIR/doc/XdcUsersGuide.pdf for information about items that may be configured in the package.bld file.

In this file, Pkg.attrs.profile may be any of the following:

- "debug"
- "release"

The targets (for example, x86 Linux, C64P, and MVARM9) come from the config.bld file. You can modify the user.bld file to add your own targets. The user.bld file is in the CE_INSTALL_DIR/examples directory.

2.2.4 <MODULE>.xdc File

This is the module declaration and definition file.

You should rename the VIDDEC_COPY.xdc file to match the module name you used in the package.xdc file. This file declares that this module is a codec that can be incorporated into a Codec Engine server. That is, it specifies the codec's name, type, and any additional properties you want to declare.

By inheriting ti.sdo.ce.video.IVIDDEC in the following line, VIDDEC_COPY declares that it is a VISA video decoder algorithm. This allows the Codec Engine to automatically supply default stubs and skeletons for transparent execution of DSP codecs by the GPP.

```
metaonly module VIDDEC_COPY inherits ti.sdo.ce.video.IVIDDEC
```

In addition to declaring the type of algorithm, this file uses the following statement to declare the external symbol required by xDAIS to identify the algorithm's implementation functions.

```
/*!
 *  ======== ialgFxns ========
 *  name of this algorithm's xDAIS alg fxn table
 */
override readonly config String ialgFxns =
        "VIDDECCOPY_TI_VIDDECCOPY";
```

If the codec uses DMA, you must also specify the idma3Fxns vtable symbol's name in this file.

2.2.5 <MODULE>.xs File

This is the module's implementation.

Rename the VIDDEC_COPY.xs file to match the module name you used in the package.xdc file. This file should contain the implementation of any methods declared in VIDDEC_COPY.xdc. By analogy, the *.xs file is to its corresponding *.xdc file as a *.c file is to a *.h file.

This file defines codec properties that can vary across platforms and configurations.

It implements the functions specified in the ti.sdo.ce.ICodec interface. These functions enable the configuration tool to validate user-supplied configuration parameters. For example, the default VIDEEC_COPY.xs file specifies the stack size for the thread running this codec.

```
function getStackSize(prog)
{
   if (verbose) {
      print("getting stack size for " + this.$name
      + " built for the target " + prog.build.target.$name
      + ", running on platform " +  prog.platformName);
   }

   return (1024);
}
```

Chapter 3

Supporting Non-xDM Algorithms

This chapter describes how to create stubs and skeletons to allow non-xDM algorithms to be used remotely in Codec Server applications.

Topic **Page**

3.1 Overview 3–2
3.2 Core Algorithm Interface Requirements 3–4
3.3 Creating Codec Engine Extensions 3–5
3.4 Designing a New Application Interface 3–5
3.5 Developing Stubs and Skeletons 3–9
3.6 Binding Stubs and Skeletons to Framework Extensions 3–19
3.7 Packaging and Configuring the Core Algorithm 3–20
3.8 Non-xDM Stub and Skeleton Example: SCALE 3–22

3.1 Overview

Codec Engine (CE) provides an easy-to-use application interface—VISA—for multi-media developers to integrate and deploy compliant codecs. All VISA-compliant codecs internally use a xDAIS/xDM-compliant algorithm core. CE uses the core algorithm's generic IALG interface to dynamically create and destroy codec instances. CE uses the codec's xDM interface to call specific processing and control functions.

This chapter shows how to extend CE by adding new VISA-like APIs for remotely accessing xDAIS algorithms that do not implement the xDM interface. It also presents techniques for developing a custom media application layer interface, which seamlessly integrates non-xDM algorithms into the extended CE framework.

An extension example, SCALE, illustrates the concepts in this chapter. This example is provided in the ti.sdo.ce.examples.codecs.scale package, which is found in the CE_INSTALL_DIR/examples repository.

In order to use a CE extension, such as SCALE, a GPP client application uses the existing CE interfaces (Engine APIs) to configure and open a processing engine. It then uses new SCALE APIs to create, delete and drive "scale" algorithms.

3.1.1 Tasks

The tasks for developing a CE extension can be broken down as follows:

- ❏ Create core algorithm interface. See Section 3.2.
  - ■ xDAIS IALG interface required for standard instance creation and deletion.
  - ■ I*MOD* xDAIS extension providing core processing and control functions.
- ❏ Create CE extension layer. See Section 3.3.
  - ■ New *MOD* application framework API and library. See Section 3.4.
  - ■ CE RPC layer for remote processing on a dual-processor target. See Section 3.5.
    - ◗ Client-side: custom *MOD* stub interface. See Section 3.5.1.
    - ◗ Server-side: custom *MOD* skeleton interface. See Section 3.5.2.
- ❏ Perform CE packaging and configuration.
  - ■ Configuring engine, binding skeleton interface. See Section 3.6.
  - ■ *MOD* packaging. See Section 3.7.

3.1.2 Roles and Interactions

Figure 3–1 provides a high-level overview of the interactions between a GPP client application, Codec Engine and a remote (DSP) server running xDAIS-compliant algorithms.

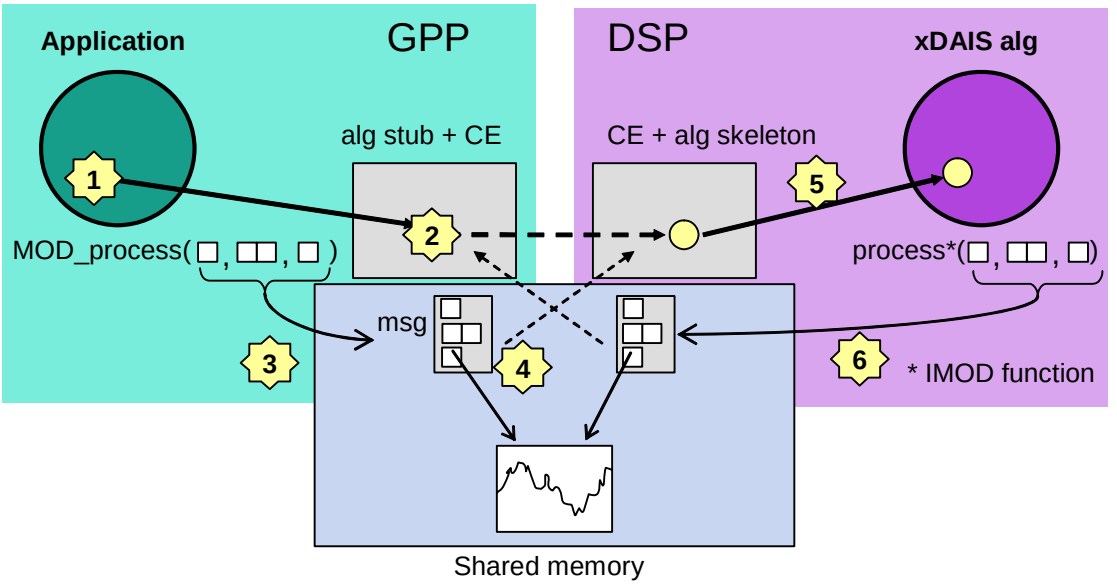

Figure 3–1 Overview of GPP / DSP Interactions

In Figure 3–1, the roles of stubs and skeletons in remote calls to an xDAIS DSP algorithm are shown. The numbers in the figure correspond to the following actions:

1) The GPP-side application makes an algorithm *MOD*_process() call.
2) Codec Engine (CE) forwards this call and its arguments to the GPP-side algorithm stub.
3) The stub places the arguments in a compact inter-CPU message and replaces all GPP-side (virtual address) pointer values with DSP-side (physical address) values. This is called "marshalling" the arguments.
4) CE delivers the message to the DSP-side algorithm skeleton.
5) The skeleton unmarshals the arguments and calls the actual xDAIS algorithm's process() function.
6) On the way back, the skeleton marshals any return arguments, places them in a message, and the stub unmarshals them for the application.

In Codec Engine, memory shared by the GPP and DSP is not cached on the GPP, but is cached on the DSP. The skeleton must manage the cache by invalidating the cache area occupied by the input and output buffers before the xDAIS process() call and by flushing the output buffers' cache area after the xDAIS process() call.

Technically, the process() and control() function prototypes on the GPP side do not need to be the same as those of the xDAIS algorithm. However, if they are not the same, the API layer becomes more complicated, and is beyond the scope of this document.

3.2 Core Algorithm Interface Requirements

The core algorithm interface must be compliant with TI's TMS320 DSP Algorithm Standard (xDAIS). For details, see the following documents:

- *TMS320 DSP Algorithm Standard Rules and Guidelines* (SPRU352)
- *TMS320 DSP Algorithm Standard API Reference* (SPRU360)
- *TMS320 DSP Algorithm Standard Developer's Guide* (SPRU424)

Additionally, to be able to operate in a dual-processor environment, the interface must support remote calls. The xDM interface is an example of a remote callable API. Remote calling within the CE framework currently imposes the following restrictions:

- Creation parameters cannot contain client-side address references.
- Each argument's operational semantics must be specified as either In, Out, or In/Out.
- Arguments passed to algorithm methods—for example, process() and control()—must be serializable.This means:
  - Each address reference passed in an argument must be explicit and must point to a client-provided memory location whose size can be determined by parsing the provided run-time arguments.
  - Each address reference passed in an Out or In/Out argument must be aligned with respect to cache line sizes and must have a size that is a multiple of the cache-line size.
  - The sizes of all arguments must be fixed or computable by parsing the run-time arguments.

Client applications must ensure that all data passed to algorithm processing functions by reference are:

- contiguously allocated
- located within a shared memory segment
- aligned with respect to cache line sizes and have a size that is a multiple of the cache-line size.

3.3 Creating Codec Engine Extensions

The Codec Engine VISA APIs provide uniform synchronous procedure call semantics for client applications. When the server is "local", the calls are true local procedure calls. However, when the server is "remote", Codec Engine provides a classic Remote Procedure Call (RPC) abstraction.

Section 3.4 describes how to develop a VISA-like application interface to support non-xDM algorithms. These extended APIs facilitate VISA-like easy-to-use creation, deletion, and processing operations for both local and remote server configurations.

Then, Section 3.5.1 and Section 3.5.2 describe how to develop the CE stubs and skeletons that implement the RPC abstraction in a remote configuration.

3.4 Designing a New Application Interface

You should create a new API module that can be delivered to the client application as a separate, configurable package that extends (and leverages) the Codec Engine framework. This section uses the SCALE extension example provided in the toolkit to explain the key concepts required to create such a package.

The SCALE example provides a new application interface with contains the following core functions: SCALE_create, SCALE_delete, SCALE_process, and SCALE_control. These functions are used to create and delete local or remote "scale" codec instances and to process data using the algorithm.

The CE extension package, "ti.sdo.ce.examples.extensions.scale", implements the SCALE APIs and provides a configuration interface module, ISCALE, which extends ti.sdo.ce.ICodec to support remote operation and to configure stubs and skeletons. More details on ISCALE and stubs and skeletons are provided in Section 3.5.1 and Section 3.5.2.

3.4.1 Creation and Deletion APIs

The easiest APIs to define and implement are the create and delete APIs, since these are fairly generic and can simply invoke the VISA_create and VISA_delete functions. Follow these guidelines for seamless integration:

- **Algorithm Handle.** All algorithm handles used should be compatible with the VISA_Handle type.
- **Algorithm Creation Parameter.** Creation parameters must extend IALG_Params. The first field contains the size of the structure definition; the rest can be arbitrary non-pointer elements.
- **Engine_Handle.** The Engine_Handle argument to VISA_create must be to an open engine which contains references to the core algorithm package.
- **Algorithm Name.** The name argument to VISA_create must match the string used to configure the algorithm in the engine.
- **Codec Name.** The fully-qualified module name of the codec in the extension package must be passed as the last argument to VISA_create.

The source code for the SCALE_create and SCALE_delete APIs, which follows, shows how to implement them using VISA APIs.

```
/* ======== SCALE_create ======== */
SCALE_Handle SCALE_create(Engine_Handle server,
    String name, SCALE_Params *params)
{
    SCALE_Handle visa;

    if (params == NULL) {
        params = &SCALE_PARAMS;
    }

    visa = VISA_create(server, name, (IALG_Params *)params,
        sizeof (_SCALE_Msg), "extensions.scale.ISCALE");

    return (visa);
}

/* ======== SCALE_delete ======== */
Void SCALE_delete(SCALE_Handle handle)
{
    VISA_delete(handle);
}
```

3.4.2 Processing Function APIs

To create processing function APIs, you create custom functions that use the public core algorithm interface of the xDAIS-compliant (but non-xDM) codec. When implementing these functions, use the following VISA APIs:

- ❏ VISA_getAlgFxns
- ❏ VISA_getAlgHandle
- ❏ VISA_enter, VISA_exit

First, use the VISA_Handle returned by the create call (for example, SCALE_create) to obtain the actual algorithm instance handle and the algorithm's IALG interface function table. Use these to call the core algorithm's processing functions. It is possible, although not recommended, for the framework APIs to have different arguments than those of the core algorithm interface.

The technique recommended here allows the same API implementation to be used for both local and remote invocation. When the application configures the codec as "remote", the VISA layer routes the call to the remote algorithm instance using stubs and skeletons that must be provided as part of the CE extension package. (see Section 3.5) When configured as "local", the processing call is a function call to the actual algorithm instance.

Calls to an algorithm's processing functions must be bracketed with VISA_enter and VISA_exit calls using the instance handle. These calls allow the VISA layer to perform the necessary instance activation and deactivation that may be necessary for xDAIS algorithms.

For example, the SCALE_process function is implemented as follows:

```
/* ======== SCALE_process ======== */
XDAS_Int32 SCALE_process(SCALE_Handle handle,
        XDAS_Int8 *inBuf, XDAS_Int8 *outBuf,
        SCALE_InArgs *inArgs, SCALE_OutArgs *outArgs)
{
   XDAS_Int32 retVal = SCALE_EFAIL;

   if (handle) {
      ISCALE_Fxns *fxns =
       (ISCALE_Fxns *)VISA_getAlgFxns((VISA_Handle)handle);
      ISCALE_Handle alg =
         VISA_getAlgHandle((VISA_Handle)handle);

      if (fxns && (alg != NULL)) {
         VISA_enter((VISA_Handle)handle);
         retVal = fxns->process(alg, inBuf, outBuf,
                  inArgs, outArgs);
         VISA_exit((VISA_Handle)handle);
      }
   }
   return (retVal);
}
```

3.5 Developing Stubs and Skeletons

This section describes how to provide a VISA-like RPC abstraction for remote calls. It assumes you have a xDAIS-compliant algorithm, for which you would like to create a remote-callable API.

When the client GPP application calls the process or control API on a remote server, the call and its arguments must be encoded as a message and sent to the remote server where the actual call is dispatched to the core algorithm instance. The process of delivering the function arguments to the remote dispatcher is typically called "marshalling". A client-side "stub" function acting as a proxy for the remote server performs the marshalling.

On the server side, the Codec Engine's dispatcher, called a "skeleton", unmarshals the arguments and calls the appropriate core algorithm's processing function(s) to serve the request.

The skeleton then marshals the result and output arguments of the actual computation back to the waiting client application stub, which subsequently unmarshals and returns result and output arguments back to the client as if it were an ordinary procedure call.

3.5.1 Developing Codec Engine Stubs

For each process or control call, you must create a separate "stub" function. The signature of each stub function must match the signature of the extended API.

The following outline shows the work the stub function must perform. Details and example code are provided with matching sequence numbers after this list.

1) Define a new VISA "message" type used for marshalling and unmarshalling call arguments used by both stubs and skeletons.
2) Marshal the call and its arguments using the VISA "message":
    a) Create the new "message" using VISA APIs.
    b) Set the function-ID (VISA header command ID).
    c) Marshalling: Translate address references passed as arguments from the client's address space to the server's address space.
    d) Copy all IN, IN/OUT, and mapped address references in OUT arguments to the message.
3) Dispatch the call to the server using VISA_call and block until the server's reply message arrives.

4) Unmarshal the returned arguments:
   a) Ensure or translate all returned output address references back to the client's address space from the server's address space.
   b) Copy OUT arguments from MSG.
5) Free any acquired VISA messages.
6) Return the results of remote computation.

The details of stub processing requirements are as follows:

1) Define a new VISA "message" type used for marshalling and unmarshalling call arguments used by both stubs and skeletons.

The new message type will be used in VISA_call() communication by CE. The first member of the struct type must be a VISA_MsgHeader type. The remaining fields should explicitly list each argument to all possible callable functions.

The VISA_MsgHeader structure includes a field for defining the command-id. This command-id is used to specify the purpose of the remote procedure call. You should create unique command-ids for each remote callable function in the extended framework interface.

The following code from the SCALE example illustrates the types used:

```
#include <ti/sdo/ce/visa.h>
#include <scale.h>

#define _SCALE_CPROCESS        0
#define _SCALE_CCONTROL        1

/* MSGQ message for marshalling/unmarshalling arguments */
typedef struct {
    VISA_MsgHeader  visa;
    union {
        struct {
            XDAS_Int8           *inBuf;
            XDAS_Int8           *outBuf;
            SCALE_InArgs        inArgs;
            SCALE_OutArgs       outArgs;
        } process;
        struct {
            SCALE_Cmd           id;
            SCALE_DynamicParams params;
        } control;
    } cmd;
} _SCALE_Msg;
```

2(a) Marshalling: Create the "message" using VISA APIs.

Use VISA_allocMsg() to acquire a communication message that CE has associated with the remotely created algorithm instance. For example:

```
VISA_Handle visa = (VISA_Handle)h;
_SCALE_Msg *msg;

/* get a message appropriate for this algorithm */
if ((msg = (_SCALE_Msg *)VISA_allocMsg(visa)) == NULL) {
    return (SCALE_ERUNTIME);
}
```

2(b) Marshalling: Set the function-ID (VISA header command ID).

Use the VISA header of the IPC message to indicate the type of the remote function that is being marshalled. For example:

```
/* Specify the processing command the skeleton should do */
msg->visa.cmd = _SCALE_CPROCESS;
```

2(c) Marshalling: Translate the address references passed as arguments from the client's address space to the server's address space.

Use the Codec Engine Memory_getBufferPhysicalAddress() API to translate each address reference. After the calls, check for valid mapped addresses.

For example:

```
/* inBuf is a pointer, so we have to convert it */
msg->cmd.process.inBuf =
  (XDAS_Int8 *)Memory_getBufferPhysicalAddress(inBuf,
    inArgs->inBufSize, NULL);

if (msg->cmd.process.inBuf == NULL) {
    retVal = SCALE_ERUNTIME;
}
```

2(d) Marshalling: Copy all IN, IN/OUT, and mapped address references in OUT arguments to the message.

Some things to remember:

All arguments must be passed by value.

All address references must be mapped to the server's address space and stored in the message.

Data referenced by any pointer argument must not contain any "unmapped" address references.

The same message structure (format) must be used by both the skeleton and the stub.

3) Dispatch the call to the server using VISA_call() and block until the server's reply message arrives.

Use the Codec Engine VISA_call() API to synchronously send the message containing marshalled arguments to the skeleton and wait for its completion.

VISA_call() returns the error or success code returned by the skeleton dispatcher's call function.

When VISA_call() returns successfully, the VISA message header’s status field contains the actual result returned by the core algorithm function.

For example, after the following call, msg.status contains the result returned by the algorithm. If the call was successful, this same value is returned as retVal.

```
/* send message to the skeleton and wait for completion */
retVal = VISA_call(visa, (VISA_Msg *)&msg);
```

Note that the "message" returned by the msg argument in VISA_call is not necessarily the same msg reference that was passed to the call. Therefore, any address references relative to the message base that was obtained prior to VISA_call are invalid after VISA_call returns, and must be re-calculated before accessing the message contents.

4(a) Unmarshal: Ensure or translate all returned output address references back to the client's address space from the server's address space.

Use the Codec Engine Memory_getBufferVirtualAddress() API to translate address references from the remote processor's physical address space to the client's virtual address space. After the calls, you should check for valid mapped addresses.

The SCALE example does not return any address references to the client, so the following example is from the actual xDM video decoder stubs.

```
/* pointers in outArgs.displayBufs are physical; convert */
for (i = 0; i < outArgs->displayBufs.numBufs; i++) {
    outArgs->displayBufs.bufs[i] =
        Memory_getBufferVirtualAddress(
        (UInt32)(outArgs->displayBufs.bufs[i]),
        outArgs->displayBufs.bufSizes[i]);
}
```

4(b) Unmarshal: Copy OUT arguments from the message.

Remember:

All arguments must be passed by value.

All address references in IN/OUT or OUT arguments must be mapped to the client's address space.

Data referenced by an IN/OUT or OUT pointer argument must not contain any address references that are not "unmapped" to the client's address space.

The same message structure (format) must be used by both the skeleton and the stub.

5) Free any acquired VISA messages.

Use the VISA_freeMsg() API to release the communication message resources that CE associated with the remotely created VISA object. For example:

```
VISA_freeMsg(visa, (VISA_Msg)msg);
```

6) Return the results of remote computation.

When VISA_call() executes successfully, the return value from the VISA_call is the actual result returned by the core algorithm function.

For example:

```
/* send message to the skeleton and wait for completion */
retVal = VISA_call(visa, (VISA_Msg *)&msg);

...

return(retVal);
```

3.5.2 Developing Skeletons for CE Extensions

Codec Engine provides a basic mechanism to deliver the stub's marshalled arguments to the remote algorithm instance's skeleton. The "skeleton" is the remote side (DSP-side) of the RPC. The skeleton defines its interface via a SKEL_Fxns struct with three function pointers. This structure is defined in skel.h as follows:

```
/*
 * ======== SKEL_Fxns ========
 */
typedef struct SKEL_Fxns {
    SKEL_CALLFXN call;
    SKEL_CREATEFXN apiCreate;
    SKEL_DESTROYFXN apiDestroy;
} SKEL_Fxns;
```

When you implement a skeleton, the extension package must provide a global SKEL_Fxns symbol containing pointers to these three functions: call, apiCreate and apiDestroy.

The "call" function is executed only on the DSP side; it is ultimately the DSP-side handler for the GPP-side VISA_call() API, and handles processing functions. The "apiCreate" and "apiDestroy" functions, however, should be implemented independently of whether they run on the GPP or DSP (because they'll run on both). This enables CE to easily

support "local" codecs—that is, codecs that run on the same processor as the application (in the DM644x case, this is the GPP)—as well as remote codecs.

Here is an outline of the work the skeleton's "call" function must perform. Details and example code are provided with matching sequence numbers after this list.

1) Select the dispatch function by parsing the command-id in the message header.
2) Unmarshal "processing" function arguments.
    a) Cache invalidate all buffers, including the IN, IN/OUT, and OUT buffers.
    b) Construct an actual argument list for the target function by parsing the dispatch message.
3) Call the core xDAIS algorithm interface function.
4) Marshal return arguments into the reply message.
    a) Cache writeback the OUT and IN/OUT buffers.
    b) Copy OUT arguments back to the reply message.
5) Send the reply message back to the caller.

Note: This section includes descriptions of cache operations for IN/OUT buffers for a CE extension. It is important to note that the xDM specification provides only for IN and OUT buffers. IN/OUT buffers are not allowed for xDM-compliant algorithms, but you may use them for proprietary interfaces.

The details of skeleton processing requirements are as follows:

1) Select the dispatch function by parsing the command-id in the message header.

The signature of the skeleton function is defined in the ti.sdo.ce.skel interface. For example:

```
#include <ti/sdo/ce/skel.h>
...
/*  ======== call ======== */
static VISA_Status call(VISA_Handle visaHandle,
                        VISA_Msg visaMsg)
{
    _SCALE_Msg *msg  = (_SCALE_Msg *)visaMsg;
    SCALE_Handle handle = (SCALE_Handle)visaHandle;
    . . .

    /* call the requested SCALE func by parsing message */
    switch (msg->visa.cmd) {
       case _SCALE_CPROCESS: {
          . . .
       }
       case _SCALE_CCONTROL: {
          . . .
       }
    }
```

2(a) Unmarshal: Cache invalidate IN, IN/OUT, and OUT buffers.

Marshalled address references have already been mapped to the server's address map, so no additional address mapping is needed.

Compute the "size" of each data region passed as an address-reference using the current arguments.

When client-side data references are mapped to server side "cached" memory addresses, any memory block that is being referenced in an IN, IN/OUT, or OUT buffer must be invalidated using the ce.osal.Memory package function: Memory_cacheInv(). This is necessary to prevent reading potentially stale data from server-side cache.

For example:

```
/* unmarshal inBuf and outBuf */
inBuf  = msg->cmd.process.inBuf;
outBuf = msg->cmd.process.outBuf;

/* invalidate cache for input and output buffers */
Memory_cacheInv(inBuf, msg->cmd.process.inArgs.inBufSize);
Memory_cacheInv(outBuf, msg->cmd.process.inArgs.outBufSize);
```

2(b) Unmarshal: Construct an actual argument list for the target function by parsing the dispatch message.

Remember:

All arguments must be passed by value.

All address references have already been mapped to the servers's address space by the stub.

The same message structure (format) must be used by both the skeleton and the stub.

3) Call the core xDAIS algorithm interface function.

Use the dispatched VISA handle and extended APIs to call the process or control functions. For example:

```
static VISA_Status call(VISA_Handle visaHandle,
                        VISA_Msg visaMsg)
{
   _SCALE_Msg *msg  = (_SCALE_Msg *)visaMsg;
   SCALE_Handle handle = (SCALE_Handle)visaHandle;

   . . .
   /* make the process call */
   msg->visa.status = SCALE_process(handle, inBuf, outBuf,
          &(msg->cmd.process.inArgs),
          &(msg->cmd.process.outArgs));
```

4(a) Marshal return arguments: Cache writeback OUT and IN/OUT buffers.

Compute the "size" of each data region passed as address-reference using the current arguments.

When client-side data references are mapped to server-side "cached" memory addresses, any memory block that is being referenced in an OUT buffer or an IN/OUT argument must be written back using the

ce.osal.Memory package function Memory_cacheWb(). This is necessary to update the contents of shared external memory from modified server cache lines.

For example:

```
/* writeback output buffer */
Memory_cacheWb(outBuf, msg->cmd.process.inArgs.outBufSize);
```

4(b) Marshal return arguments: Copy OUT arguments back to the reply message.

The return value of the process or control API must be stored in the VISA message header's "status" field.

Also, any IN/OUT and/or OUT arguments must be copied to the VISA message by value.

For example:

```
/* make the process call */
msg->visa.status = SCALE_process(handle, inBuf, outBuf,
     &(msg->cmd.process.inArgs),
     &(msg->cmd.process.outArgs));
```

5) Reply message returned back to the caller.

The algorithm's status and OUT arguments stored in the message (step 4b above) are passed back to the GPP-side stub automatically by the CE framework. The following sample code shows the GPP-side stub using the return code from VISA_call() and de-referencing outArgs from the returned message:

```
/* send message to the skeleton and wait for completion */
retVal = VISA_call(visa, (VISA_Msg *)&msg);

/* copy out the outArgs */
*outArgs = msg->cmd.process.outArgs;
. . .
return (retVal);
```

3.6 Binding Stubs and Skeletons to Framework Extensions

The codec extension configuration must include a meta-only interface that is derived from the ti.sdo.ce.ICodec interface.

Configure the skeleton functions by defining a global symbol of type SKEL_Fxns. This contains the extension skeleton function pointers and overrides the ICodec configuration interface's serverFxns attribute with the name of the SKEL_Fxns symbol.

Similarly, you should export the stub functions using a global symbol containing the core algorithm interface function table whose module (that is, non-IALG) functions are set to the addresses of the stub functions. This symbol's name is used to override the ICodec configuration interface's stubFxns attribute.

The SCALE example defines one such configuration interface, ISCALE, which follows:

```
/*!
 *  ======== ISCALE ========
 *  Scale Codec Configuration interface
 */
metaonly interface ISCALE inherits ti.sdo.ce.ICodec
{
  override readonly config String serverFxns = "SCALE_SKEL";
   override readonly config String stubFxns = "SCALE_STUBS";

   override readonly config Int rpcProtocolVersion = 1;
}
```

The SCALE_SKEL and SCALE_STUBS structures point to the stub and skeleton functions:

```
/* ======== SKEL_Fxns ======== */
typedef struct SKEL_Fxns {
    SKEL_CALLFXN call;
    SKEL_CREATEFXN apiCreate;
    SKEL_DESTROYFXN apiDestroy;
} SKEL_Fxns;

SCALE_Fxns SCALE_STUBS = {
    {&SCALE_STUBS, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
     NULL}, process, control,
};
```

3.7 Packaging and Configuring the Core Algorithm

Figure 3–2 shows the roles of components (also called "packages") and their files in creating custom stubs and skeletons.

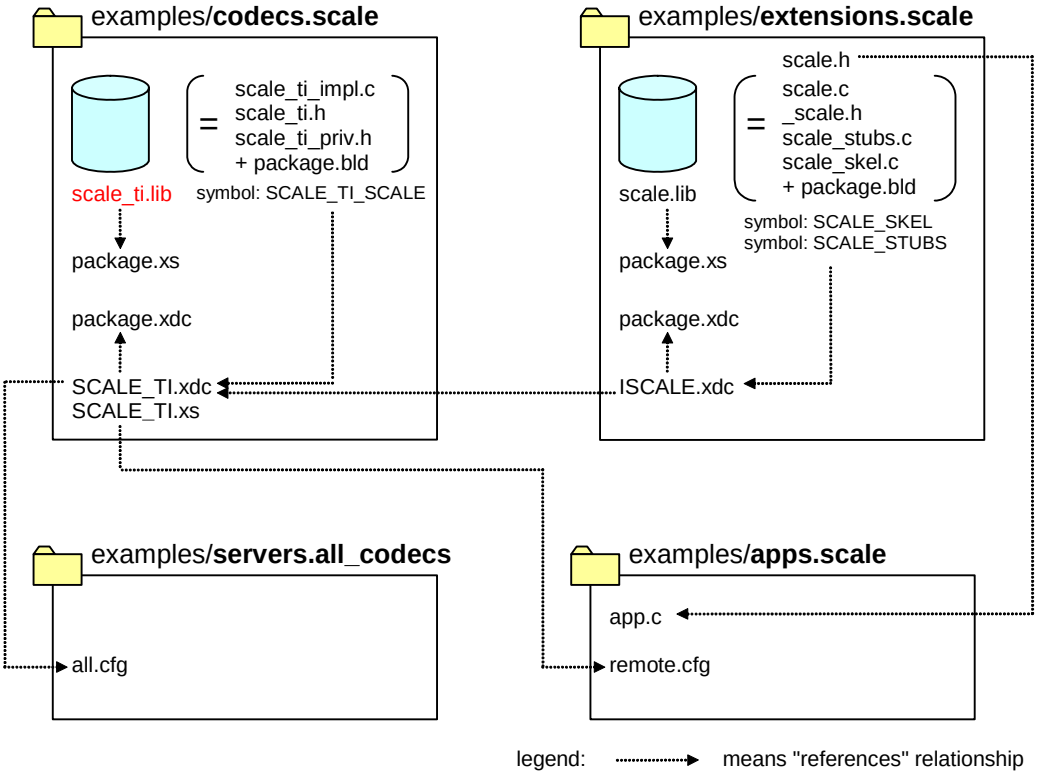

Figure 3–2 Components Used in Stub and Skeleton Creation

We start with the original xDAIS SCALE algorithm library, scale_ti.lib, and create an XDC package, called ti.sdo.ce.examples.codecs.scale, around it. (We build the library from the .c/.h files and the build script, package.bld, but we could also add a pre-built scale_ti.lib to the package.) The package.xs file implements the getLibs() method, which the XDC tools will query and utilize when generating a list of libraries to link into the final executable. The package.xdc file defines the name of the package (ti.sdo.ce.examples.codecs.scale), as well as the Module which the package contains (SCALE_TI). This Module is implemented in SCALE_TI.xdc. This Module implements the ti.sdo.ce.examples.extensions.scale.ISCALE interface and names the xDAIS function table (SCALE_TI_SCALE).

The stubs and skeletons package—ti.sdo.ce.examples.extensions.scale—contains the stubs/skeletons library. This is a separate package to allow other scale algorithms to use the same stubs and skeletons. This package contains the implementation of 1) the processor-independent API, 2) the GPP stubs, and 3) the DSP-side skeletons. (We build the library from .c/.h files and package.bld, but the library could be pre-built.) The package.xs file again implements getLibs() so that the XDC tools can obtain the appropriate library to link into the executable. The package.xdc file defines the name of the package (ti.sdo.ce.examples.extensions.scale), and the Modules which the package implements (ISCALE). The Module is implemented in ISCALE.xdc, which configures the stub and skeleton function tables (SCALE_STUBS and SCALE_SKEL).

On the DSP side, the ti.sdo.ce.examples.servers.all_codecs example simply names the ti.sdo.ce.examples.codecs.scale.SCALE_TI module in its configuration script and the codec is automatically added and configured.

On the GPP side, the app.c file in the ti.sdo.ce.examples.apps.scale package, includes ti/sdo/ce/examples/extensions/scale/scale.h so that it can call the SCALE_create(), SCALE_process(), and SCALE_control() functions. The remote.cfg file, which is the configuration file for the GPP application, names the ti.sdo.ce.examples.codecs.scale.SCALE_TI module.

3.8 Non-xDM Stub and Skeleton Example: SCALE

An extension example, SCALE, illustrates the concepts in this chapter. This example is included in the standard CE distribution. Figure 3–3 shows the role of stub and skeleton files in the remote algorithm process() call.for the SCALE example.

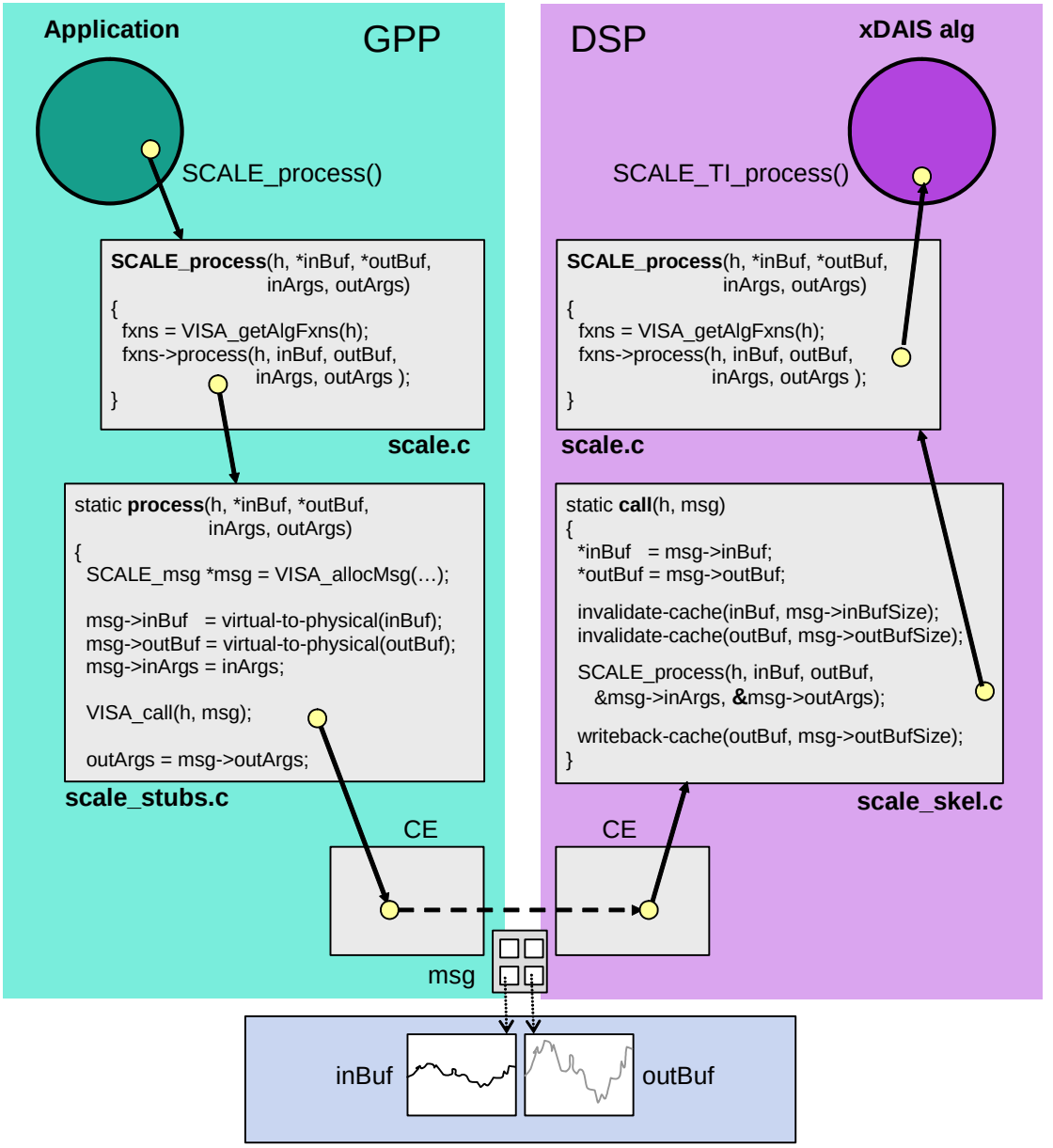

Figure 3–3 Stub and Skeleton Files in SCALE Example

The GPP-side application calls the SCALE_process() function, which is defined in the header file scale.h. The file scale.c implements this function, which calls, through a function pointer, the function table configured into the application. In the case of local codecs, this function pointer is the algorithm's IALG function table. In the case of remote codecs, this function pointer is the interface's stub function table. The stubs are implemented in scale_stubs.c.

The stub allocates a message of type SCALE_msg (defined in file _scale.h) and marshals the arguments—it copies scalar values and stores virtual-to-physical translations of the inBuf and outBuf pointers to the message. It then calls Codec Engine's VISA_call() function to deliver the message to the SCALE skeleton on the DSP.

The skeleton, in its call() function, invalidates the DSP cache for the areas occupied by the input and output buffer, then makes the SCALE_process() call. This call, which is implemented in the same scale.c file that on the GPP initiates the message marshalling and transfer, this time resolves to calling the xDAIS algorithm's SCALE_TI_process() function. Any output arguments from the algorithm are written to the original message used to communicate across the CPU boundary.

Upon returning from the function, the SCALE skeleton writes back the cache for the outBuf area (only), and then returns. Codec Engine, which called the skeleton, sends the response back to the GPP. The response is the same message that was received, except that the "output" part of its contents may have been changed by the algorithm and/or the skeleton. The SCALE stub, when VISA_call returns back with the message, unmarshals the output arguments and returns to the caller.

Index

A

addLibrary method 2-6
address
  GPP vs. DSP 3-4
  translating 3-12
allocate message 3-11
API interface 3-5
apps.scale package 3-21
arguments
  passing 3-12
  skeletons 3-15
  stub 3-9
ARM target 2-5

B

BIOS_INSTALL_DIR environment variable 1-4
.bld files 2-3
Build.targets list 1-5

C

cache 3-4
  alignment and size 3-4
  invalidate 3-16
CE_INSTALL_DIR environment variable 1-4
CMEM directory 1-4
CMEM_INSTALL_DIR environment variable 1-4
codec
  VISA vs. extensions 3-2
Codec Engine
  directory 1-4
  interactions 3-3
codecs.scale package 3-20
command-id 3-10
config.bld file 1-4
copy codecs 2-3
create APIs 3-6

D

debug version 2-6
delete APIs 3-6
directories
  environment variables 1-4
  package 2-4
DSP target 2-5
DSP/BIOS directory 1-4
DSP/BIOS Link directory 1-4
DSPLINK_INSTALL_DIR environment variable 1-4

E

eXpress Dsp Components (XDC) 1-3
extensions.scale package 3-5, 3-21

F

FC_INSTALL_DIR environment variable 1-4
Framework Components directory 1-4

G

generated files 2-4
getLibs() function 2-5

H

header files 2-3

I

ialgFxns table 2-7
ICodec interface 2-7
idma3Fxns table 2-7
In arguments 3-4
In/Out arguments 3-4

L

library 2-3, 3-21
Link directory 1-4

M

makefile 1-3, 2-3
marshalling 3-3, 3-9
Memory_cacheInv function 3-16
Memory_cacheWb function 3-18
Memory_getBufferPhysicalAddress function 3-12
Memory_getBufferVirtualAddress function 3-13
message
  allocating 3-11
  freeing 3-14
message type 3-10
metadata 1-5
MODULE.xdc file 2-6
MODULE.xs file 2-7

O

Out arguments 3-4

P

package 1-5
  building 1-3
  dependencies 2-4
  directory 2-4
  name 2-4
  roles 3-20
package.bld file 2-3, 2-6
package.mak file 2-4
package.xdc file 2-3, 2-4
package.xs file 2-3
packages
  apps.scale 3-21
  codecs.scale 3-20
  extensions.scale 3-5, 3-21
  servers.all_codecs 3-21
paths 1-4
  relative vs. absolute 1-4

R

relative vs. absolute paths 1-4
release version 2-6
remote calls 3-4
Remote Procedure Call (RPC) 3-5
requires statement 2-4
return value 3-18
RTSC 1-2, 1-3

S

SCALE example 3-2, 3-22
serializable 3-4
servers.all_codecs package 3-21
skeletons 3-1, 3-9
  actions performed 3-15
  role 3-3
  signatures 3-16
stack size 2-7
status field 3-18
stubs 3-1, 3-9
  actions performed 3-9
  role 3-3

T

target, selecting 2-6

U

user.bld file 1-4

V

version, selecting 2-6
VISA APIs 2-4, 3-5
VISA_allocMsg function 3-11
VISA_call function 3-10, 3-12
  return value 3-18
VISA_create function 3-6
VISA_delete function 3-6
VISA_freeMsg function 3-14
VISA_Handle type 3-6
VISA_MsgHeader type 3-10

X

xDAIS
  directory 1-4
  documents 3-4
XDAIS_INSTALL_DIR environment variable 1-4
xDAIS-DM 2-2
xdc command 1-3
XDC Configuration Kit 1-3
.xdc files 2-3

XDC tools 1-3
directory 1-4
XDC User's Guide 2-6
XDC_INSTALL_DIR environment variable 1-4
.xdcenv.mak file 2-4
XDCPATH environment variable 1-3
xdcpaths.mak file 1-3
xdcrules.mak file 1-3
xDM 2-2
adding interface 3-2
.xs files 2-3