Creating a TMS320DM6446 Audio Encode Example Using XDC Tools

Zhengting He

ABSTRACT

This application report describes how to create an eXpressDSP™ Algorithm for Digital Media (XDM) compatible audio encode example on the TMS320DM6446 processor using the eXpressDSP Components (XDC) tool. XDM is an extension of TI eXpress DSP Algorithm Interface Standard (xDAIS). The example consists of:

- The Linux ®application code running on the ARM side of DM6446
- The dummy audio encoder on the DSP side of DM6446
- The DSP server, which is a binary including the dummy audio encoder running on the DSP side of DM6446.

The application reads a block of raw audio data from the input file, sends the data to the sound device for playing, and also hands the data to the codec engine, which delivers the data with a request to DSP via DSP link driver. The codec server on DSP accepts the request and delivers the data to the dummy audio encoder. The dummy audio encoder generates the output by making a copy of the input data. The output data are delivered back to application in the reverse sequence: dummy codec → codec server → codec engine → application.

The focus of this application report is to explain how to use XDC to build 1) DSP-side dummy codec, 2) DSP server and 3) ARM-side Linux application.

This application report contains project code that can be downloaded from this link: http://www-s.ti.com/sc/techlit/sprc344.gz. The provided source code only works with DM6446 DVEVM 1.10. To make it work with the latest DM6446 DVEVM 1.20, please make the appropriate changes by following the methodology described in this document.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Dataflow

Figure 1 shows the dataflow of the dummy audio encode example.

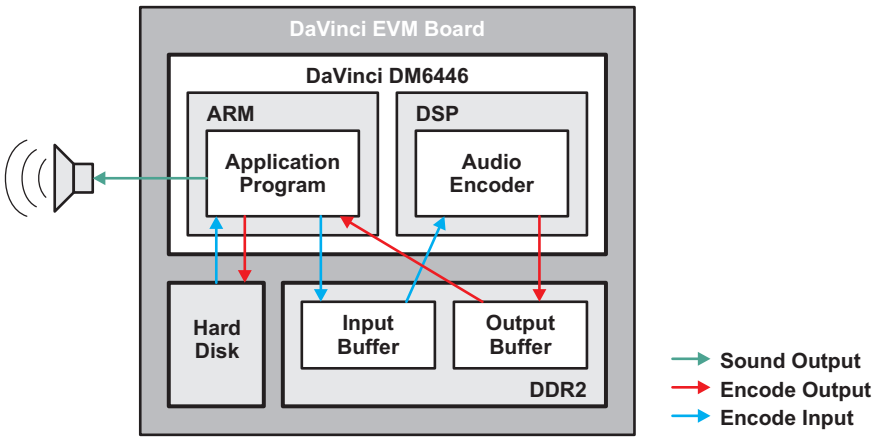

Figure 1. Dataflow

The application program on ARM reads a block of data from the input file on the hard disk, saves the data to the input buffer on external DDR2 memory, calls the sound device driver to play the data, and notifies the dummy audio encoder. In this example, the block size is 4096 bytes. The audio encoder on DSP gets the input data, makes a copy, stores it to the output buffer on DDR2 memory, and notifies the application program which writes the output to file. The application repeats until the end of the file is reached.

1.2 Software Architecture

Figure 2 shows the software architecture of the dummy audio encode example.

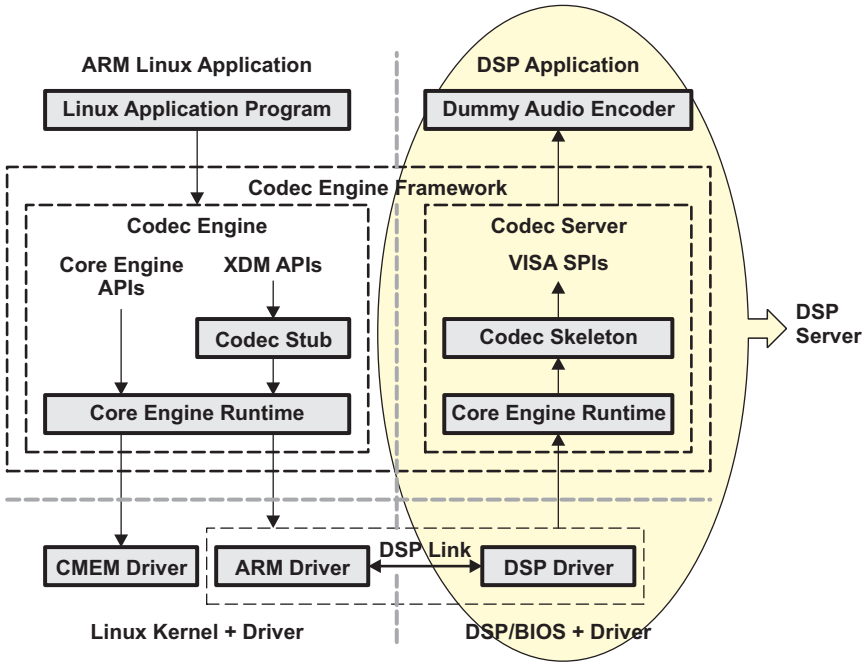

Figure 2. Software Architecture

The application program on ARM is in the Linux application space. From ARM application point of view, asking DSP to encode a frame is a simple XDM API call. Algorithm compliant to xDAIS requests memory usage through xDAIS interface so that different algorithms can be easily integrated together without worrying about corrupting each other's memory. An XDM algorithm is compliant with xDAIS. Additionally, it implements two additional APIs to support multimedia codecs: process and control. The process API triggers the codec algorithm to process the input data. The control API has a broad range of usage, such as inquiring the codec status and dynamically configuring the codec.

The XDM interfaces divide codec algorithms into four classes: video, image, speech, and audio (VISA). For each class, one set of APIs and data structures (struct in C language) are provided. The class specific data structures typically define the parameters that are common for this class of codec at the beginning of the structure. XDM defines an extended parameter in almost each structure to allow users to add their own special parameters there. XDM also allows users to create a codec that does not belong to a VISA class. When any extended parameters are added in data structure, or a completely new codec class is created, the developer needs to use the same methodology to create his/her own codec stub and skeleton to allow the application program to access the codec.

The ARM Linux application calls the core engine APIs for other functionality, such as initializing codec engine runtime environment, allocating buffers for processing usage, realizing communication between ARM and DSP.

Driver CMEM manages the shared memory between ARM and DSP. In fact, ARM Linux application calls codec engine, which in turn, calls CMEM driver in Linux kernel to allocate input and output buffer. Typically, DSP codec requires the input and output buffer reside in physically contiguous memory. The ARM side application is in Linux user space and typical malloc() function call cannot ensure that. CMEM is a Linux driver residing in Linux kernel space. It is able to work with Linux kernel to allocate physically contiguous buffer.

The communication between ARM and DSP is implemented by the DSP link driver. In this example, the ARM Linux application calls the codec engine, which in turn, calls the DSP link to realize communication. The DSP link has an ARM side driver in Linux kernel space and a DSP/BIOS driver on DSP.

Corresponding to the codec engine on ARM, there is a codec server on DSP, which typically translates the request such as encoding a frame received from the ARM Linux application to the VISA SPI calls recognized by the DSP codec.

The dummy audio encoder on DSP is an XDM-compliant codec implementing the XDM audio encoder interface (IAUDENC).

When ARM Linux application issues a XDM process call to ask the DSP codec to process a block of data, the following things happen under the API.

- Pointers to any shared buffer between ARM and DSP need to be translated so that the DSP program can recognize the shared buffer. API arguments do not reside in the shared memory between ARM and DSP so they need to be marshaled before being copied to DSP memory. These tasks are handled by the codec engine. Specifically for this dummy audio encoder example, the audio class stub handles these tasks.
- The DSP link driver handles data copying between ARM and DSP. It has a Linux driver as the ARM portion and the DSP/BIOS driver as the DSP portion.
- Once the API arguments and buffer pointers reach DSP, appropriate codec API needs to be called or a DSP/BIOS task needs to be activated. To encode a new frame, cache also needs to be invalidated, which forces the DSP core to read the new input data from external memory. These tasks are handled by the codec server, specifically the audio class codec skeleton.

For details on the Codec Engine, see [3] *Codec Engine Application Developer User's Guide* (SPRUE67).

The Codec Engine framework consists of codec engine on the ARM side and codec server on the DSP side. For clarification, the Codec Engine framework is referred to as CE in the remainder of the document. *codec engine* means the ARM side of CE, and *codec server* means the DSP side of CE.

The DSP executable image, being called *DSP server* in the remainder of the document, consists of the encoder codec, codec server and the DSP side DSP link driver. When running the example application, the first step for ARM application is to call codec engine core engine API, which in turn, calls DSP link driver to load DSP server to DSP and initialize the DSP.

To develop a multimedia application on DM6446, typically the ARM Linux application and the DSP codec need to be implemented by users. CE, DSP link driver, and CMEM driver are existing libraries provided by TI and can be reused by users. The eXpress DSP component tool is introduced here to help build the ARM Linux application executable and DSP server executable.

To build the ARM Linux application executable, XDC does the following.

- Generates the interface code, which glues the application with the codec engine.
- Compiles the generated interface code with the application code.
- Links the compiled application code and interface code with appropriate codec engine libraries to generate the ARM Linux executable.

To build the DSP server, XDC does the following.

- Generates the interface code, which glues the user developed codec with the codec server. The codec must be XDM compliant. In this example, it is dummy audio encoder implementing the IAUDENC interface.
- Compiles the generated interface code with the codec code.
- Links the compiled codec code and interface code with appropriate codec server libraries, DSP side DSP link library, and DSP/BIOS library to generate the DSP server.

2 Package Contents

The example code is compressed as *sprc344.gz* file. Unzipping the package creates a folder called *audcp*. There are three sub-folders: *audcp_codec*, *audco_server*, and *audcp_arm_linux_app*.

audcp_codec contains the source code of dummy audio encoder on DSP and other files necessary for building the encoder. The files in *audcp_codec* are listed in Table 1.

Table 1. Contents in audcp/audcp_codec

File(s)	Description
audcp.c, audcp_ti.h, audcp_ti_priv.h	Those are the source files of dummy audio encoder. *audcp.c* implements the XDM audio encoder interface "IAUDENC_Fxns" which is a function table. The implemented functions are: 1. AUDCP_TI_alloc that allocates memory before the encoder starts. 2. AUDCP_TI_free that frees memory for encoder before the encoder terminates. 3. AUDCP_TI_initObj that is the initialization function of the encoder. 4. AUDCP_TI_process that is the processing (encoding) function. 5. AUDCP_TI_control that is the control function to enquire encoder status The 1st three functions are XDAIS functions.
makefile, AUDCP.xdc, AUDCP.xs, package.bld, package.mak, package.xdc, package.xs	Those files are used for building the dummy audio encoder using XDC.
lib/	This lib directory contains the generated libraries after building the code. *audcp_codec.a470MV* is an ARM library and is linked with the ARM Linux application to generate the application executable. *audcp_codec.a64P* is the DSP dummy audio encoder library that is linked with the codec server to generate the DSP server.

audcp_server contains the source code and other files necessary for building the DSP server. The files in *audcp_server* are listed in Table 2.

Table 2. Contents in audcp/audcp_server

File(s)	Description
main.c	This is the source code for the DSP server. It contains only a simple main function initializing the codec server. Other source code that glues it with the audio encoder library is generated by XDC.
audpServer.tcf	This is the TConf script used to configure DSP/BIOS.
link.cmd	This file does not contain anything for this example, but it can be used to input additional information to configure memory usage.
Makefile, pacage.bld, package.mak, package.xdc, audcpServer.cfg	These files are used for building the DSP server using XDC.
audcpServer.x64P	This is the generated DSP server executable after building the code.

audcp_arm_linux_app contains the source code and other files necessary for building the ARM Linux application executable. The files in *audcp_arm_linux_app* are listed in Table 3.

Table 3. Contents in audcp/audcp_arm_linux_app

File(s)	Description
app.c, ceapp.c	These are the source files for the ARM Linux application. The behavior has been summarized in Section 1.1.
Makefile, ceapp.cfg	These files are used for building the ARM Linux application executable using XDC.
davinciEffects.raw	This is the default input data file for the application, but the user is allowed to supply his/her own input file.
app.out	This is the generated ARM Linux application executable after building the code.

There are also six files in *audcp* as shown below:

- *xdcpaths.mak*, which defines all the path environment variables required by XDC to build the example code.
- *xdccfg_linuxarm.mak*, which is the actual makefile used to build the ARM Linux application executable.
- *xdcrules.mak*, which is used to build DSP server and dummy audio encoder using XDC.

- *dsplinkk.ko*, which is the ARM part of the DSP link driver. It is required by the ARM Linux application executable to communicate with DSP.
- *cmemk.ko*, which is the memory driver to allocate physically contiguous memory in Linux kernel for ARM Linux application.
- *loadmodules.sh*, which is the script to load dsplinkk.ko and cmem.ko to Linux kernel before running the example code.

3 Installation

3.1 Necessary Hardware and Software

- DM6446 DVEVM board
- Windows PC with terminal program, i.e., hyperterminal installed
- Serial cable connecting Windows PC and DM6446 board
- Linux development host with DM6446 DVEVM and DVSDK packages pre-installed and configured. For details on how to install and configure DVEVM package, see [1] *DVEVM Getting Started Guide* (SPRUE66). For details on how to install and configure DVSDK package, see [2] *DVSDK Getting Started Guide* (SPRUEG8).

3.2 Notations for Names of Directories on Linux Development Host

The following notations are used in the document for referring various DVEVM/DVSDK package directories after they are installed.

- $(DVEVM_INSTALL_DIR) is the directory where DVEVM and DVSDK packages are installed. Typically it is /home/your_user_name/dvevm_x_xx.
- $(BIOS_INSTALL_DIR) is the directory where the DSP/BIOS package is installed. Typically it is in $(DVEVM_INSTALL_DIR)/bios_5_xx.
- $(CG_INSTALL_DIR) is the directory where the code generation tools for TI DSP are installed. Typically it is in $(DVEVM_INSTALL_DIR)/cg6x_6_x_xx.
- $(CMEM_INSTALL_DIR) is the directory where the CMEM driver package is installed. Typically it is in $(DVEVM_INSTALL_DIR)/cmem_x_xx.
- $(CE_INSTALL_DIR) is the directory where the CE package is installed. Typically it is in $(DVEVM_INSTALL_DIR)/codec_engine_x_xx.
- $(CS_INSTALL_DIR) is the directory where the codec server package is installed. Typically it is in $(DVEVM_INSTALL_DIR)/codec_server_x_xx.
- $(DSPLINK_INSTALL_DIR) is the directory where the DSP link driver package is installed. Typically it is in $(DVEVM_INSTALL_DIR)/dsplink_x_xx_xx_xx.
- $(FMWK_INSTALL_DIR) is the directory where the framework component package is installed. Typically it is in $(DVEVM_INSTALL_DIR)/framework_component_x_xx_xx.
- $(XDAIS_INSTALL_DIR) is the directory where the XDAIS package is installed. Typically it is in $(DVEVM_INSTALL_DIR)/xdais_x_xx.
- $(XDC_INSTALL_DIR) is the directory where the XDC package is installed. Typically it is in $(DVEVM_INSTALL_DIR)/xdctools_x_xx.

The following notations are used for referring the directories of the provided example.

- $(AUDCP_ROOTDIR) is the root directory contains the example code. It is $(DVEVM_INSTALL_DIR)/audcp.
- $(AUDCP_CODEC) is the directory contains the dummy audio encoder package. It is $(AUDCP_ROOTDIR)/audcp_codec.
- $(AUDCP_SERVER) is the directory contains the DSP server package. It is $(AUDCP_ROOTDIR)/audcp_server.
- $(AUDCP_APP) is the directory contains the ARM Linux application package. It is $(AUDCP_ROOTDIR)/audcp_arm_linux_app.

3.3 Installation Steps

The following are steps to install the dummy audio encoder example package.

1. Download the example package *audcp.tar.gz* to $(DVEVM_INSTALL_DIR) on your Linux host.
2. Switch to directory $(DVEVM_INSTALL_DIR) on the Linux host and type the following command. The $(AUDCP_ROOTDIR) will be created. It this case, it is $(DVEVM_INSTALL_DIR)/audcp.

```
tar -xzf sprc344.gz
```

4 How to Run

The following are steps to run the dummy audio encoder example on DM6446 EVM board.

1. Create a directory */opt/audcp* on your DM6446 EVM board.
2. Transfer *loadmodules.sh*, *dsplinkk.ko* and *cmemk.ko* in $(AUDCP_ROOTDIR) of your Linux host to */opt/audcp* on your DM6446 EVM board.
3. Transfer *app.out* and *davinciEffects.raw* in $(AUDCP_APP) of your Linux host to */opt/audcp*on your DM6446 EVM board.
4. Transfer *audcpServer.x64P* in $(AUDCP_SERVER) of your Linux host to */opt/audcp* on your DM6446 EVM board.
5. Switch to */opt/audcp* on your board and type *./app.out* to run the program. The correct output is shown below.

```
pp → Application started.
[DSP] @0x000002c3:[T:0x00000000] servers.video_copy - main > Welcome to DSP serve
r's main().
CEapp → Allocating contiguous buffer for 'input data' of size 4096…
CEapp → Contiguous buffer allocated OK (phys. addr=0x87a8d000)
CEapp → Allocating contiguous buffer for 'encoded data' of size 4096…
CEapp → Contiguous buffer allocated OK (phys. addr=0x87a90000)
App → Finished encoding 1000 frames
App → Application finished successfully.
```

By default, program app.out reads the input file davinciEffects.raw and generates the output file davinciEffects.copy. To supply user's own input and output file, type the following.

```
./app.out [input_file_name] [output_file_name].
```

5 Re-Compile the Dummy Audio Encoder Example

To re-compile the ARM Linux application, switch to $(AUDCP_APP) on your Linux host and type *make*. The generated ARM Linux executable is *app.out*

To re-compile the DSP server, switch to $(AUDCP_SERVER) on your Linux host and type *make*. The generated DSP executable is *audcpServer.x64P*.

To re-compile the dummy audio encoder on DSP, switch to $(AUDCP_CODEC) on your Linux host and type *make*. The generated ARM Linux library is $(AUDCP_CODEC)/lib/audcp_codec.x470MV. The generated DSP library is $(AUDCP_CODEC)/lib/audcp_codec.a64P.

If anything is updated for the dummy audio encoder, it is necessary to rebuild the encoder library as well as the ARM Linux application and DSP server because they both link to the encoder code.

The steps to re-compile the example package are simple, but what was done under the *make* command is complicated. In the rest of the section, the compilation process for each package is explained along with the involved files.

5.1 Re-Compile the Dummy Audio Encoder

5.1.1 makefile

- The following step defines the root directory $(AUDCP_ROOTDIR) for this example.

```
...
# [CE] define AUDCP_ROOTDIR to point to root of example directory
AUDCP_ROOTDIR := $(CURDIR)/..
...
```

- The following step includes file xdcpaths.mak in $(AUDCP_ROOTDIR), which defines all the paths related to XDC packages and tools.

```
...
# [CE] include the file that defines paths to XDC packages and XDC tools
include $(AUDCP_ROOTDIR)/xdcpaths.mak
...
```

- The following step adds directory $(AUDCP_ROOTDIR) to XDC related paths.

```
...
# [CE] add the examples directory itself to the list of paths to packages
XDC_PATH := $(AUDCP_ROOTDIR);$(XDC_PATH)
...
```

- This step tells the XDC to build libraries for this codec.

```
...
include $(AUDCP_ROOTDIR)/xdcrules.mak
...
```

xdcrules.mak invokes XDC to build the code using information specified in appropriate files in $(AUDCP_CODEC). These files are explained in the rest of the section.

```
$(XDC_INSTALL_DIR)/xdc XDCPATH="$(XDC_PATH)" \
                XDCOPTIONS=$(XDCOPTIONS) $@
```

5.1.2 package.bld

```
for (var i = 0; i < Build.targets.length; i++) {
    var targ = Build.targets[i];
    print("building for target " + targ.name + " ...");

    /*
     * Add a library to this package and add the files described in
     * SRCS to the library.
     */
    Pkg.addLibrary("lib/audcp_codec", targ).addObjects(SRCS);
}
```

The code in *package.bld* tells XDC to build a library for each target. The targets are specified at the end of file $(CE_INSTALL_DIR)\examples\user.bld. For our case, the interested targets are C64P (DSP on DM6446) and MVArm9 (ARM on DM6446). The library is built to directory $(AUDCP_CODEC)\lib. For DSP library, it is named as audcp_codec.a64P. For ARM library, it is named as audcp_codec.a470MV.

```
Build.targets = [
    //Linux86,
    C64P,
    MVArm9,

    // Note that uclibc support is disabled by default.  To enable it,
    // ensure the UCArm9.rootDir setting above is appropriate for your
    // environment and uncomment the following line.
//    UCArm9,
];
```

5.1.3 package.xs

When building the Linux ARM application code, XDC needs to link the application with the generated ARM library in this codec package. When building the DSP server, XDC needs to link the server code with the generated codec library in this package.

This file tells XDC where to find the necessary library for linking. Because, in file *package.bld*, the name of the generated libraries are specified as *audcp_codec.a64P* (for DSP) and *audcp_codec.a470MV* (for ARM) and they reside in $(AUDCP_CODEC)\lib, the code below gives the exact path and name for these libraries so that XDC can find them.

For this case, *audcp_codec.a470MV* needs to be linked with application code and *audcp_codec.a64P* needs to be linked with codec server.

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

    /* return the library name: name.a<arch> */
    print("    will link with " + this.$name + ":" + name);

    return (name);
}
```

5.1.4 package.xdc

- The following line specifies that this package belongs to the XDM audio class.

```
requires ti.sdo.ce.audio;
```

- The following lines declare the package name as *AUDCP*. The package location is declared as *audcp_codec*, which has to be the same as the name of the directory where the package resides. In other words, package location has to be declared as *package $(AUDCP_CODEC) {...*.

```
// must match directory name
package audcp_codec {
    module AUDCP;
}
```

5.1.5 AUDCP.xdc

This file specifies the information necessary to inherit from codec engine. The name of this file (excluding the file extension) must be the same as the package name declared in *package.xdc* so that XDC tools knows where to find this file after looking at *package.xdc*. In this case the package name is *AUDCP*.

The following code says that the codec inherits from the XDM audio encoder interface. The name of the exposed XDM function table interface is *AUDCP_TI_XDMINTF*.

```
metaonly module AUDCP inherits ti.sdo.ce.audio.IAUDENC
{
    /*!
     *  ======== ialgFxns ========
     *  name of this algorithm's xDAIS alg fxn table
     */
    override readonly config String ialgFxns = "AUDCP_TI_XDMINTF";
}
```

The implementation of *AUDCP_TI_XDMINTF* is in *audcp.c*. The XDM is an extension of XDAIS. In the beginning of AUDCP_TI_XDMINTF, it is the XDAIS function pointers interface. The last two function pointers are process and control, which are extension of XDAIS.

```
#define IALGFXNS                                                \
    &AUDCP_TI_IALG,     /* module ID */                         \
    NULL,               /* activate */                          \
    AUDCP_TI_alloc,     /* alloc */                             \
    NULL,               /* control (NULL => no control ops) */  \
    NULL,               /* deactivate */                        \
    AUDCP_TI_free,      /* free */                              \
    AUDCP_TI_initObj,   /* init */                              \
    NULL,               /* moved */                             \
    NULL                /* numAlloc (NULL => IALG_MAXMEMRECS) */

/*
 *  ======== AUDCP_TI_XDMINTF ========
 *  This structure defines TI's implementation of the IAUDENC interface
 *  for the AUDCP_TI module.
 */
IAUDENC_Fxns AUDCP_TI_XDMINTF = {    /* module_vendor_interface */
    {IALGFXNS},
    AUDCP_TI_process,
    AUDCP_TI_control,
};
```

5.1.6 Make Sequence

1. XDC generates package.mak before compiling any code.
2. XDC tools make configurations for code generation. It reads $(CE_SERVER_INSTALL_DIR)/packages/user.bld and knows to build for target C64P (DSP) and MVArm9 (ARM).
3. Generate $(AUDCP_CODEC)\package\package.c and compile it to $(AUDCP_CODEC)\package\lib\lib\audcp_codec\package.o470MV. *package.c* contains the CE interface code.
4. Compile audcp.c to package/lib/lib/audcp_codec/audcp.o470MV.
5. Archive package/lib/lib/audcp_codec/package.o470MV and package/lib/lib/audcp_codec/audcp.o470MV into lib/audcp_codec.a470MV
6. Compile $(AUDCP_CODEC)\package\package.c to $(AUDCP_CODEC)\package\lib\lib\audcp_codec\package.o64P.
7. Compile audcp.c to package/lib/lib/audcp_codec/audcp.o64P.
8. Archiving package/lib/lib/audcp_codec/package.o64P and package/lib/lib/audcp_codec/audcp.o64P into lib/audcp_codec.a64P

5.2 Compiling DSP Server

5.2.1 makefile

This makefile is the same as the makefile for the dummy audio encoder in $(AUDCP_CODEC).

5.2.2 package.xdc

The following lines declare the package location as *audcp_server*, which has to be the same as the name of the directory where the package resides. In other words, package location has to be declared as *package $(AUDCP_SERVER) {...*.

```
// must match directory name
package audcp_server
{
}
```

5.2.3 package.bld

```
var serverName = "audcpServer";
```

This line sets the DSP server name to be *audcpServer*. The name is important for the following reasons:

- If being built successfully, the DSP server (combo) is called *audcpServer.x64P*. When building the ARM Linux application, file *ceapp.cfg* in $(AUDCP_APP) has to tell XDC to generate code to load the DSP server (combo) with the same name.
- When building the DSP server, XDC requires .cfg and .tcf with the same name. In this case, XDC needs audcpServer.cfg and audcpServer.tcf to build audcp.x64P.

The following code tells XDC to build the server only for DSP, but not for ARM (Linux). When building the DSP server, it needs *audcpServer.tcf* and *link.cmd*.

```
for (var i = 0; i < Build.targets.length; i++) {
    var targ = Build.targets[i];

    if (targ.os == "Linux") {
        /* Linux doesn't host remote codecs (yet) */
        continue;
    }
    else {
        /* presume we're building a full server executable */
        print("building for target " + targ.name + " ...");

        Pkg.addExecutable(serverName, targ, targ.platform,
            {
                cfgScript: serverName + ".tcf",
                lopts: "-l link.cmd",
            }).
            addObjects( [
                "main.c",
            ] );
    }
}
```

5.2.4 audcpServer.cfg

The following line tells XDC tools that the server uses codec package named *AUDCP* in directory $(AUDCP_ROOTDIR)/audcp_codec, which is where the dummy audio encoder exists.

```
/* get various codec modules; i.e., implementation of codecs */
var audioCopy = xdc.useModule('audcp_codec.AUDCP');
```

Note: The module name and directory declared here must match exactly with the names declared in $(AUDCP_CODEC)\package.xdc (shown in Section 5.1.4) so that XDC knows where to find the required library to link with.

The following line tells XDC tools to use the *Server* package in $(CE_INSTALL_DIR)\ti\sdo\ce to build the DSP server.

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

5.2.5 audcpServer.tcf

This is the script configuring DSP/BIOS when building the DSP server.

5.2.6 Make Sequence

1. XDC generates file package.mak before building the DSP server.
2. XDC tools make configurations for code generation. It reads $(CE_SERVER_INSTALL_DIR)/packages/user.bld and knows to build for target C64P (DSP) and MVArm9 (ARM). When it finds package.bld, it only builds for C64P.
3. Generate interface for audcp_server.

4. Configure audcpServer.x64P from file package/cfg/audcpServer_x64P.cfg. This step generates package/cfg/audcpServer_x64P.xdl by linking the following libraries.
   a. *audcp_codec.a64P* which is the dummy audio encoder library in $(AUDCP_CODEC)/lib.
   b. *audio_debug.a64P* in $(CE_INSTALL_DIR)\packages\ti\sdo\ce\audio\lib.
   c. *ce_debug.a64P* in $(CE_INSTALL_DIR)\packages\ti\sdo\ce\lib.
   d. *node_debug.a64P* in $(CE_INSTALL_DIR)\packages\ti\sdo\ce\node\lib.
   e. *bioslog.a64P* in $(CE_INSTALL_DIR)\packages\ti\sdo\ce\bioslog\lib.
   f. *acpy3.a64P* in $(FMWK_INSTALL_DIR)\packages\ti\sdo\fc\acpy3\.
   g. *dman3.a64P* in $(FMWK_INSTALL_DIR)\packages\ti\sdo\fc\dman3\.
   h. *dskt2.a64P* in $(FMWK_INSTALL_DIR)\packages\ti\sdo\fc\dskt2.
   i. *osal_dsplink_bios.a64P* in $(CE_INSTALL_DIR)\packages\ti\sdo\ce\osal\lib.
   j. *utils.a64P* in $(CE_INSTALL_DIR)\packages\ti\\bios\utils\lib.
   k. *dsplinkmsg.lib* and *dsplink.lib* in $(DSPLINK_INSTALL_DIR)/packages/dsplink/package/ti/dsplink/dsp/DspBios/Davinci/DEBUG/
   l. *alg.a64P* in $(CE_INSTALL_DIR)\packages\ti\sdo\ce\osal\alg\lib
   m. *gt.a64P* in n $(CE_INSTALL_DIR)\packages\ti\sdo\ce\trace\lib
5. Compile packeg/cfg/audcpServer_x64Pcfg.s62 to package/cfg/audcpServer_x64Pcfg.o64P.
6. Compile packeg/cfg/audcpServer_x64P.c to package/cfg/audcpServer_x64P.o64P.
7. Compile main.c to main.o64P.
8. Compile packeg/cfg/audcpServer_x64Pcfg_c.c to package/cfg/audcpServer_x64Pcfg_c.o64P.
9. Link package/cfg/audcpServer_x64Pcfg.o64P, package/cfg/audcpServer_x64P.o64P, package/cfg/audcpServer/main.o64P, package/cfg/audcpServer_x64Pcfg_c.o64P, package/cfg/audcpServer_x64P.xdl, and /home/zhe/dvevm_1_10/cg6x_6_0_3/lib/rts64plus.lib to generate audcpServer.x64P and package/cfg/audcpServer.x64P.map.

5.3 Compiling ARM Linux Application

5.3.1 makefile

- The following step defines the root directory $(AUDCP_ROOTDIR) for this example.

```
# [CE] define AUDCP_ROOTDIR to point to root of example directory
AUDCP_ROOTDIR := $(CURDIR)/..
```

- The following step includes file xdcpaths.mak in $(AUDCP_ROOTDIR), which defines all of the paths related to XDC packages and tools.

```
# [CE] include the file that defines paths to XDC packages and XDC tools
include $(AUDCP_ROOTDIR)/xdcpaths.mak
```

- The following step adds directory $(AUDCP_ROOTDIR) to XDC related paths.

```
# [CE] add the examples directory itself to the list of paths to packages
XDC_PATH := $(AUDCP_ROOTDIR);$(XDC_PATH)
```

- This step includes file xdccfg_linuxarm.mak in $(AUDCP_ROOTDIR) to configure the XDC tools before building the application code.

```
# File "xdccfg_linuxarm.mak" is included here to run XDC configuration step.
XDC_CFGFILE = ./ceapp.cfg
include       $(AUDCP_ROOTDIR)/xdccfg_linuxarm.mak
```

As being explained in comments, it reads the input file ceapp.cfg in current directory $(AUDCP_APPDIR) and generates the following output.

- XDC_FLAGS: Additional compiler flags that must be added to existing CFLAGS or CPPFLAGS in makefile.
- XDC_CFILE: Name of the XDC-generated C file. The makefile compiles the generated C file along with the application code written by the user.
- XDC_OFILE: Name of .o files by compiling XDC-generated C file. They are linked with our application.
- XDC_LFILE: List of Codec Engine libraries that also needs to be linked to generate the .out file.

By default, XDC tools generate all its output files to directory $(DIR_FOR_CFGFILE)/$(CFGFILE_NAME)_package, where CFGFILE_NAME is the name of the input configuration file for XDC tool; DIR_FOR_CFGFILE is the directory where the configuration exists. For this case, since the cfg file name is ceapp.cfg and it is in $(AUDCP_APPDIR), the output is generated to directory $(AUDCP_APP)/ceapp_package.

- This step adds XDC_FLAGS to the existing CPPFLAGS.

```
# [CE] Augment the standard $(CPPFLAGS) variable, adding the
# $(XDC_FLAGS) variable, defined by the file above, to it.
CPPFLAGS += $(XDC_FLAGS)
```

- This step tells GNU maker to compile each .c file into the corresponding .o file. *app.c* is the main file, which defines the function not related to codec engine such as initializing the sound device, calling the sound device driver to play the data. *appcfg.c* is the file, which defines functions calling codec engine, such as asking DSP to encode (copy) the input data and inquiring the encoder status.

```
# "normal" makefile settings and rules follow, with some additions for CE
# This app consists of the main, codec-engine unrelated app.c file, and
# the codec-engine-using appcfg.c file.

%.o : %.c
        $(CC) -c $(CPPFLAGS) $(CFLAGS) -o $@ $<
```

- The following lines tell maker where to get the compiler and supply the basic compiling options.

```
# compiler (do we need -MD for dependencies?)
CC=/opt/mv_pro_4.0/montavista/pro/devkit/arm/v5t_le/armv5tl-montavista-linuxeabi/bin/gcc \
    -g -Wall -Os
```

- The following lines define the link step. *app.o* and *ceapp.o* are generated by compiling user's application code *app.c* and *ceapp.c*. $(XDC_OFILE) is the list of object files generated by compiling the XDC generated .C files. $(XDC_LFILE) is the list of pre-existing codec engine libraries. All of them are linked to generate output executable file app.out.

```
# link all the object files
# [CE] app.out, in addition to its standard stuff, includes a compiled
# XDC-generated $(XDC_CFILE) and link list file $(XDC_LFILE)
app.out: app.o ceapp.o $(XDC_OFILE)
        $(CC) -g -o $@ $^ `cat $(XDC_LFILE)` -lpthread

all: app.out
```

5.3.2 ceapp.cfg

- The following lines define a variable *audioCopy*, which is used by XDC tools to create the specific codec engine for this application. It uses module (codec) named *AUDCP*, which resides in directory *audcp_codec*. *audcp_codec* (=$(AUDCP_CODEC)) is the directory where our dummy audio copy DSP codec resides.

Note: The module name and directory declared here must match the names declared in $(AUDCP_CODEC)\package.xdc so that XDC knows where to find the required library to link with.

```
/* The following line tells that the program will get codec
 * named AUDCP from directory $AUDCP_ROOTDIR/audcp_codec
 * "audioCopy" is the name of the module variable. The
 * variable will be used by XDC tools to create and
 * configure my own codec engine.
 */
var audioCopy = xdc.useModule('audcp_codec.AUDCP');
```

- This step tells the XDC tools to create the codec engine for application using variable *audioCopy*. *audcp_engine* is the name of the engine to create. The linux application opens it using the same name string. *audcp* is the name of the codec server. The linux application has to open it using the same name string after opening codec engine.

```
var myEngine = Engine.create("audcp_engine", [
    {name: "audcp", mod: audioCopy, local: false},
]);

See line “ceapp.c” line 29 and 30.
/* define names of codecs to use */
static String encoderName  = "audcp";
static String engineName   = "audcp_engine";
```

- The following lines tell where the generated DSP server exists at run time. Its name is *audcpServer.x64P* as defined in the file package.bld in $(AUDCP_SERVER). In this case, its location is defined as the same directory where the Linux ARM application program (app.out) resides. When running the application, *audcp.Serverx64P* and *app.out* must be in the same directory.

```
/* The following lines tells that when the Linux application
 * runs, it will load DSP combo audcpServer.x64P from the same
 * directory that the application resides.
 */
myEngine.server = "./audcpServer.x64P";
```

5.3.3 Make Sequence

For this application, the make sequence can be divided to the following steps by looking at the printout from make.

1. Compile app.c to app.o.
2. Compile ceapp.c to ceapp.o.
3. XDC tools generate pkg_x470MV.c in $(AUDCP_APP)/ceapp_package/package/cfg/.
4. XDC tools generate other files related to pkg_x470MV.c for configuration and compiling.
5. XDC tools generate library *pkg_x470MV.xdl* which includes all the codec engine data to $(AUDCP_APP)/ceapp_package/package/cfg/. For this application, *pkg_x470MV.xdl* is generated by linking the following libraries. Other than *audcp.a470MV*, which is generated when building the dummy audio encoder. All others are pre-existing libraries.
   a. *audcp.a470MV* in directory $(AUDCP_CODEC)/lib
   b. *audio_debug.a470MV* in folder $(CE_INSTALL_DIR)/packages/ti/sdo/ce/audio/lib/
   c. *audio_debug.a470MV* in folder $(CE_INSTALL_DIR)/packages/ti/sdo/ce/audio/lib/
   d. *node_debug.a470MV* in folder $(CE_INSTALL_DIR)/packages/ti/sdo/ce/node/lib/
   e. *osal_dsplink_linux.a470MV* in folder $(CE_INSTALL_DIR)/packages/ti/sdo/ce/osal/lib/
   f. *alg.a470MV* in folder $(CE_INSTALL_DIR)/packages/ti/sdo/ce/osal/alg/lib/
   g. *dsplink.lib* in folder $(DSPLINK_INSTALL_DIR)/packages/dsplink/package/ti/dsplink/gpp/Linux/Davinci/DEBUG/
   h. *gt.a470MV* in folder $(CE_INSTALL_DIR)/packages/ti/sdo/ce/trace/lib/
   i. *cmemd.a* in folder (CMEM_INSTALL_DIR)/packages/ti/sdo/linuxutils/lib
6. Compile pkg_x470MV.c to pkg_x470MV.o which is in $(AUDCP_APP)/ceapp_package/package/cfg/pkg_x470MV.o.
7. Generate app.out by linking app.o, ceapp.o, pkg_x470MV.o, pkg_x470MV.xdl and other Linux libraries such as Linux pthread library.

6 References

1. *DVEVM Getting Started Guide* (SPRUE66).
2. *DVSDK Getting Started Guide* (SPRUEG8).
3. *Codec Engine Application Developer User's Guide* (SPRUE67).
4. XDC spec 1.1 ($(XDC_INSTALL_DIR)\doc\xdcSpec.pdf)
5. XDC User's Guide ($(XDC_INSTALL_DIR)\doc\xdcUsersGuide.pdf)
6. XDC script ($(XDC_INSTALL_DIR)\doc\xdcSpec.pdf)

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
Clocks and Timers	www.ti.com/clocks	Digital Control	www.ti.com/digitalcontrol
Interface	interface.ti.com	Medical	www.ti.com/medical
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
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

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