Application Report
SPRAAE7B – February 2007

Using Adapters to Run Existing xDAIS Algorithms with Codec Engine

Vincent Wan, Prateek Bansal *Software Development Organization*

ABSTRACT

This application note proposes a novel approach--using adapters--to running xDAIS algorithms with a Codec Engine (CE) application. An adapter fits between the Codec Engine Runtime and the algorithm (usually on the DSP). Its purpose is to adapt the interface exposed by the algorithm so that it better conforms to a given system programming interface (SPI), such as xDM. In addition, an adapter can be used to do pre- and post-processing on the same processor as the algorithm.

A number of scenarios make the adapter approach a good complement to the customization of stubs and skeletons described in the *Codec Engine Algorithm Creator User's Guide* (SPRUED6).

This application report contains project code that can be downloaded from this link: http://www-s.ti.com/sc/techlit/spraae7.zip

Contents

Figures

1 Why Use Adapters?

This application note proposes a novel approach—using adapters—to running xDAIS algorithms with a Codec Engine (CE) application. Before continuing, you may want to review Appendix A for a summary of terminology used to describe CE-based applications.

The Codec Engine is designed to allow users to instantiate and run xDAIS algorithms. (Note that CE documentation also refers to algorithms as “codecs” as its original intent was to focus on running codecs. Hence this application note uses these two terms interchangeably.) It provides an API to facilitate interactions with xDM-compliant xDAIS algorithms. It also provides support for writing custom stubs and skeletons for defining custom system programming interfaces (SPIs [1]), so that CE can be expanded to handle new classes of algorithms.

Given an SPI, independent of whether it is user-defined or xDM-defined, there is sometimes a benefit to writing an extra layer of code—called an adapter—that fits between the Codec Engine Runtime and the algorithm. Its purpose is to adapt the interface exposed by the algorithm so that it better conforms to the SPI. In addition, an adapter can perform pre- and post-processing on the same processor as the algorithm. In fact, it could even allow you to run a pre-compiled legacy C6400 algorithm with the Codec Engine with no change to the binary algorithm itself.[2]

Here are some possible scenarios where an adapter is needed:

1. John, a server integrator, had previously purchased a C6400 binary algorithm from an algorithm provider, and he wants to add this algorithm to his DSP server for the DaVinci. However, the module interface (IMOD) provided by the pre-compiled xDAIS algorithm is ill-suited to allowing remote execution of the algorithm. The algorithm exposes a processing function that does not define the input and output buffer sizes as part of its parameters (this was defined at algorithm-creation time). Yet stubs and skeletons, in general, require buffer size information to be passed along in order to perform correct cache coherency operations.
2. As a server integrator, Mary wants to add pre- and post-processing to the DSP algorithm she obtained from a third party. The pre- and post-processing functions are already available and optimized for the DM642 DSP from a previous project.
3. Tom, a speech algorithm creator, is concerned about the learning curve needed to write stubs and skeletons and would rather implement an existing xDM interface such as ISPHENC for the moment, since his algorithm is not that different from a speech encoder

[1] Note that this has nothing to do with the SPI hardware peripheral on the DaVinci.
[2] Note that this is only possible if the algorithm binary can be run on the C64x+ platform. In particular, it must not use the DMA via ACPY2 APIs, since the DMA structure is different on the C64x+ and requires the use of ACPY3.

as defined by the ISPHENC interface. He wants to rapid-prototype the initial implementation and reuse the built-in stubs and skeletons for ISPHENC as provided by the Codec Engine. His plan is to write custom stubs and skeletons later, and to change the interface of the algorithm to match the new SPI at that point in time. So he wants to decouple the code that implements the xDM interface from the rest of his algorithm.

Scenarios 1 and 3 illustrate cases in which people are trying to use an existing, non-xDM binary algorithm in CE without modification. Scenario 2 is a case where the role of an adapter is expanded to add functionality around an existing algorithm.

These scenarios are just a subset of all possible circumstances where one might want to add code in an adapter layer. In such cases, adding an adapter layer solves the problem.

This application note is accompanied by an example that provides two implementations of an adapter for a xDAIS color rotation algorithm. The algorithm is provided in binary form.

- The first implementation of the adapter implements the IIMGDEC interface. The main sections of this application note provide an overview, walkthrough, and hardware and software setup for this application.
- The second implementation of the adapter implements an SPI that matches a set of custom stubs and skeletons. Appendix C discusses this second implementation and explains why stubs and skeletons are not a necessarily a substitute for adapters and vice versa.

We assume readers have a basic understanding of the elements in a Codec Engine application, including VISA APIs, xDM, servers, and codecs. For details on these, refer to the user guides supplied with the Codec Engine [References 1, 2, 3, 4].

2 Basic Concept of an Adapter

As mentioned previously, an adapter is a software layer that wraps around an xDAIS algorithm in a way that allows it to implement a specific SPI and to run pre- and post-processing functions associated with the algorithm. The following figure shows the adapter in the context of an application that uses the Codec Engine.

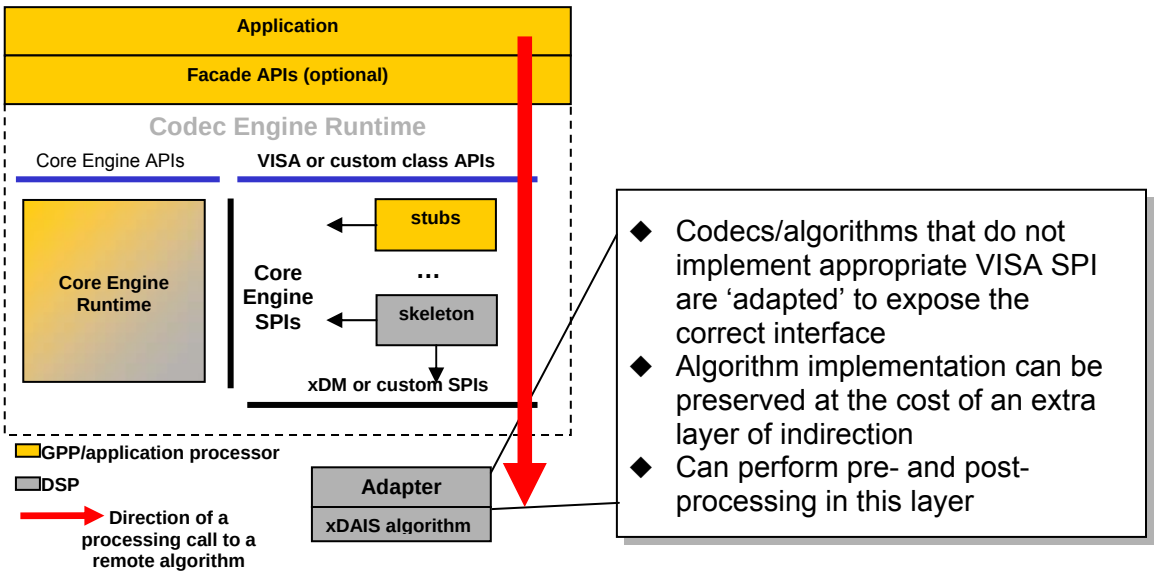


Figure 1. Adapter in Remote Execution of an Algorithm Using the Codec Engine

Note that although we are showing the case of remote execution (that is, the algorithm and the application run on different processors) due to its higher complexity, a similar picture can be drawn for an algorithm running locally (that is, an algorithm running on the application processor). In the local case, the VISA or class APIs directly call into the "adapted algorithm" instead of going through the stubs and skeleton.

An adapter typically wraps around all functions exposed by the function table of a given xDAIS algorithm, and exposes the wrapped functions in its own function table that conforms to the xDAIS standard. Viewed from the outside, a wrapped algorithm is no different from a standard xDAIS algorithm; they both have the same functional behavior, including how they use the IALG interface to request resources from the framework.

3 How to Build an Adapter

In order to illustrate the procedure for building adapter layers, an example based on a color rotation imaging algorithm (ROTATE_TI) is included with this application note. This in-place algorithm conforms to the xDAIS standard and was originally built for the C6416. We took the binary algorithm and packaged it so that it can be used with or without the CE, without modifying or rebuilding the algorithm binary itself.

In our application, we want to apply the algorithm to each incoming video frame from a video source before displaying the result. The structure of this application is shown in the following figure:

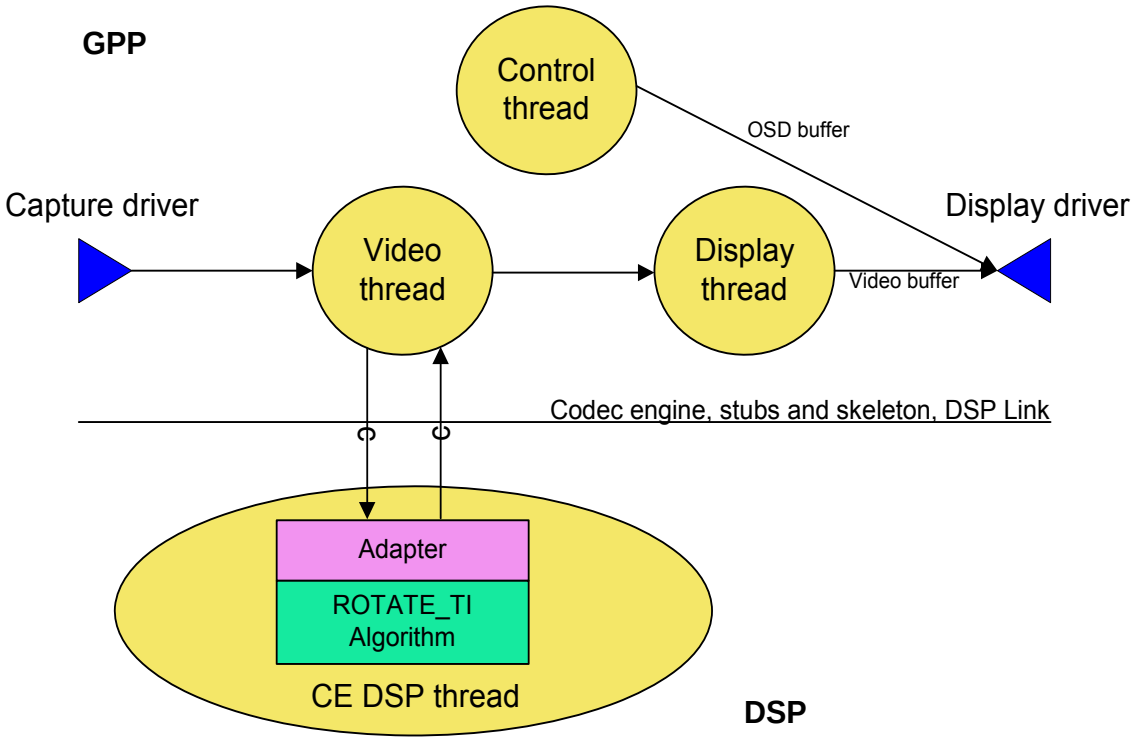


Figure 2. Application Data Flow

We took the encodedecode demo from the DVEVM / DVSDK and modified it to run the ROTATE_TI algorithm instead of the H.264 video encoder and decoder. The ARM side application has three threads: a control thread, a video thread, and a display thread. The control thread manages the user interface, responds to the remote control, computes CPU load information, and displays application data on the on-screen display (OSD). The video thread creates the algorithm and applies the algorithm to each frame it dequeues from the capture driver. Finally, the display thread copies the result to a display buffer that is sent to the display driver.

For more information on how these threads work and how to control the application using the keyboard and the remote, refer to the DVEVM documentation for the encodedecode demo [Reference 5].

Our focus is on the video thread. This is the thread that creates the ROTATE_TI algorithm and calls the Codec Engine to apply the ROTATE_TI algorithm to incoming frames. The API used for algorithm creation and processing goes through the stubs and skeleton and “talks” to the algorithm through an adapter layer. In our example, we use the adapter layer to do the following:

- Make the interface of the algorithm more suitable for remote execution by making the size of the input and output buffers one of the parameters passed by the algorithm’s process call.
- Perform simple pre- and post-processing, which consists of breaking up (demux) the input buffer into Y, CR, and CB component buffers and combining (mux) the Y, CR, and CB output from the algorithm into an output frame.

3.1 Hardware Setup

The following hardware is required to run the example application:

- TMDXEVM6446 Digital Video Evaluation module (DVEVM)
- A video source (for example, the camera that ships with the DVEVM)
- A video display (for example, the LCD that ships with the DVEVM)
- A serial cable hooked up to a PC for terminal access
- An Ethernet connection
- A Linux host machine

To run the example, hook up the DVEVM as shown in the following figure:

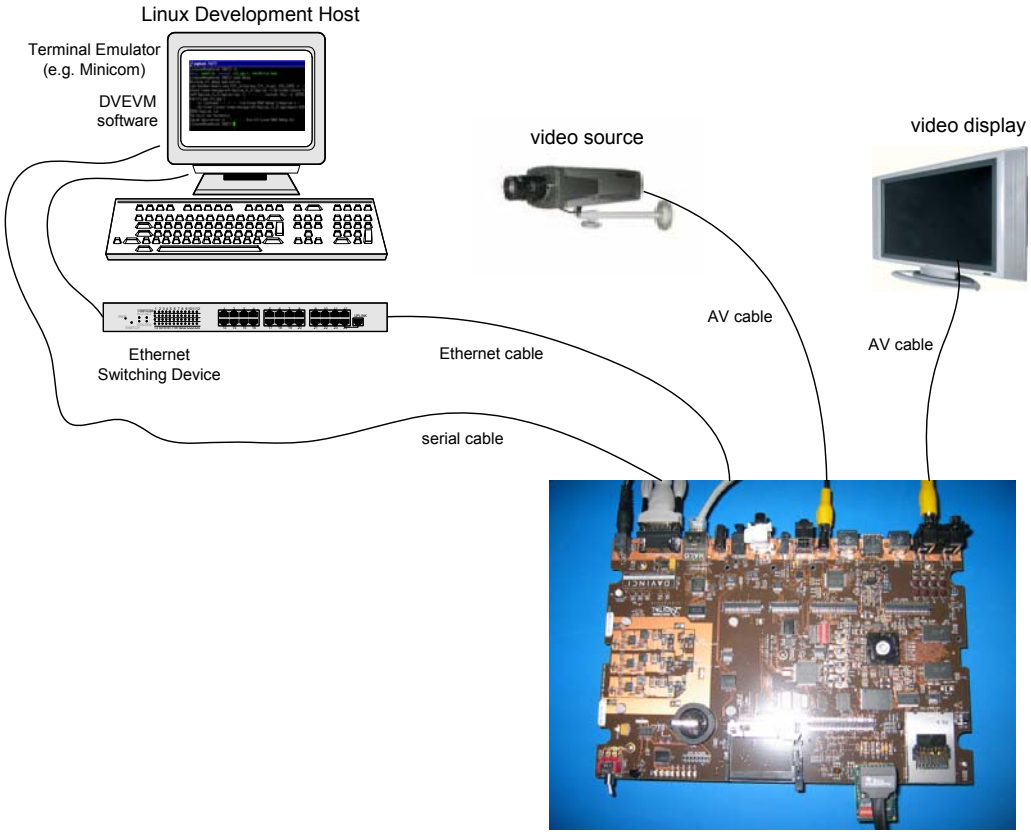


Figure 3. Hardware Setup of the DVEVM for the Rotate Demo

The video source can be the camera that comes with the DVEVM, although a better quality source, such as a DVD player, that can pause the playback is recommended. The demo applies the imaging algorithm to each incoming frame, and the effects of color rotation are best seen on still input frames.

3.2 Software Setup

The following software is required to rebuild and run the example application:

- DVEVM and DVSDK 1.10 software
- Serial terminal software (for example, TeraTerm or HyperTerminal) on a PC connected to the DVEVM via a serial cable

Your DVEVM should be set up with the DVEVM / DVSDK software so that you can boot with the MontaVista Linux kernel (residing in flash or tftp-ed). It should have access to a file system (either on the DVEVM hard drive or NFS mounted). You should be able to run and execute programs on the command line of the DVEVM in the serial terminal software. For more details on DVEVM setup, please see your DVEVM and DVSDK documentation [References 5, 6].

This application report contains project code that can be downloaded from this link: http://www-s.ti.com/sc/techlit/spraae7.zip.

There are two versions of the same application. In the first version, the adapter implements the standard IIMGDEC XDM interface. The application can then run the algorithm by calling the IMGDEC API. The code walkthrough in the following section discusses this implementation in detail.

The second version makes use of a set of customized stubs and skeleton that we wrote to streamline the creation and runtime parameter sets and the message passing that goes on between the ARM and DSP to match the specific needs of the ROTATE_TI algorithm. Details specific to this implementation are further covered in Appendix C.

3.3 To Build and Run

To build under Linux on the development host machine, setup the Codec Engine examples as described in the file CE_INSTALL_DIR/examples/build_instructions.html. From this point on, we assume you have created the directory /home/user/workdir/myexamples for this purpose. (Use the real path for your setup if you copied the CE examples to a different directory.)

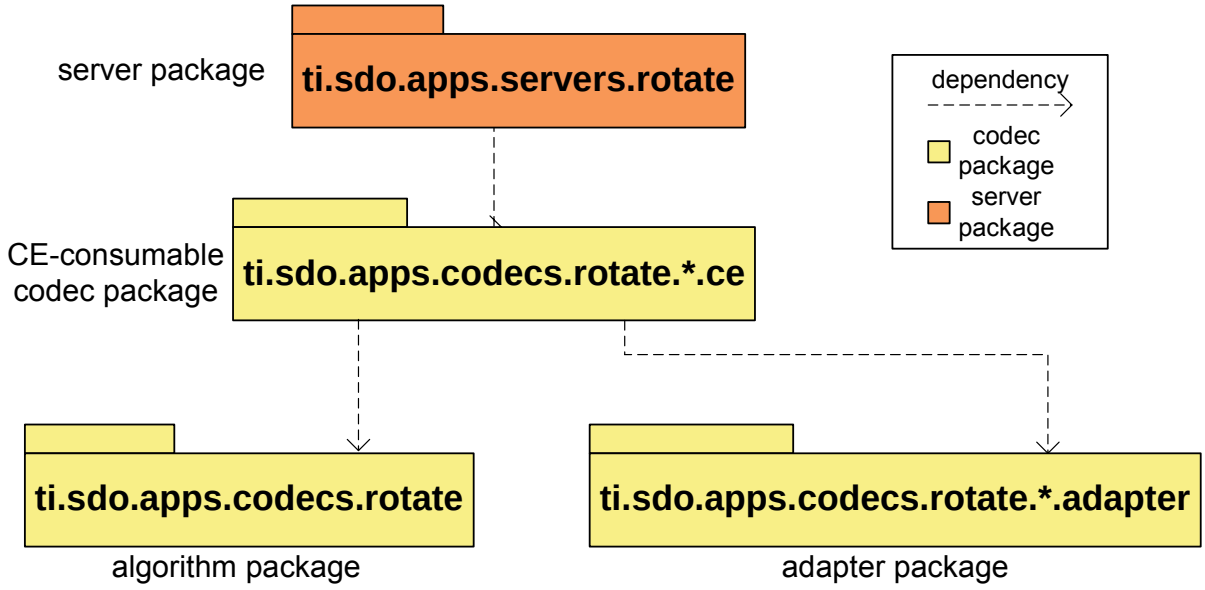
Unpack the contents of the zip file accompanying this application note into the myexamples directory:

```
cd /home/user/workdir/myexamples
unzip spraae7.zip
```

Then follow the instructions of the readme_SPRAAE7.txt file in the myexamples directory to build and run the application.

4 Code Walkthrough

Besides the application code, the code drop consists of different packages for the DSP server and codecs. Figure 4 shows the package view of the packages in the code example:



* denotes either iimgdec or irotate

Package Name	Description
ti.sdo.apps.codecs.rotate	Algorithm package that exports the algorithm library file
ti.sdo.apps.codecs.rotate.*.adapter	Adapter package that builds and exports the adapter library file
ti.sdo.apps.codecs.rotate.*.ce	Codec package that is meant for Codec Engine consumption.
ti.sdo.apps.servers.rotate	Codec Engine server package that uses the CE-consumable codec package

Figure 4. Packages Used by the Rotate Demo Application

The next section discusses CE Consumable codec packages, algorithm packages, and adapter packages (the yellow packages in Figure 4).

Figure 5 shows the directory structure of the code example. You should be able to easily navigate through the directories to find the files of interest in our subsequent discussion.

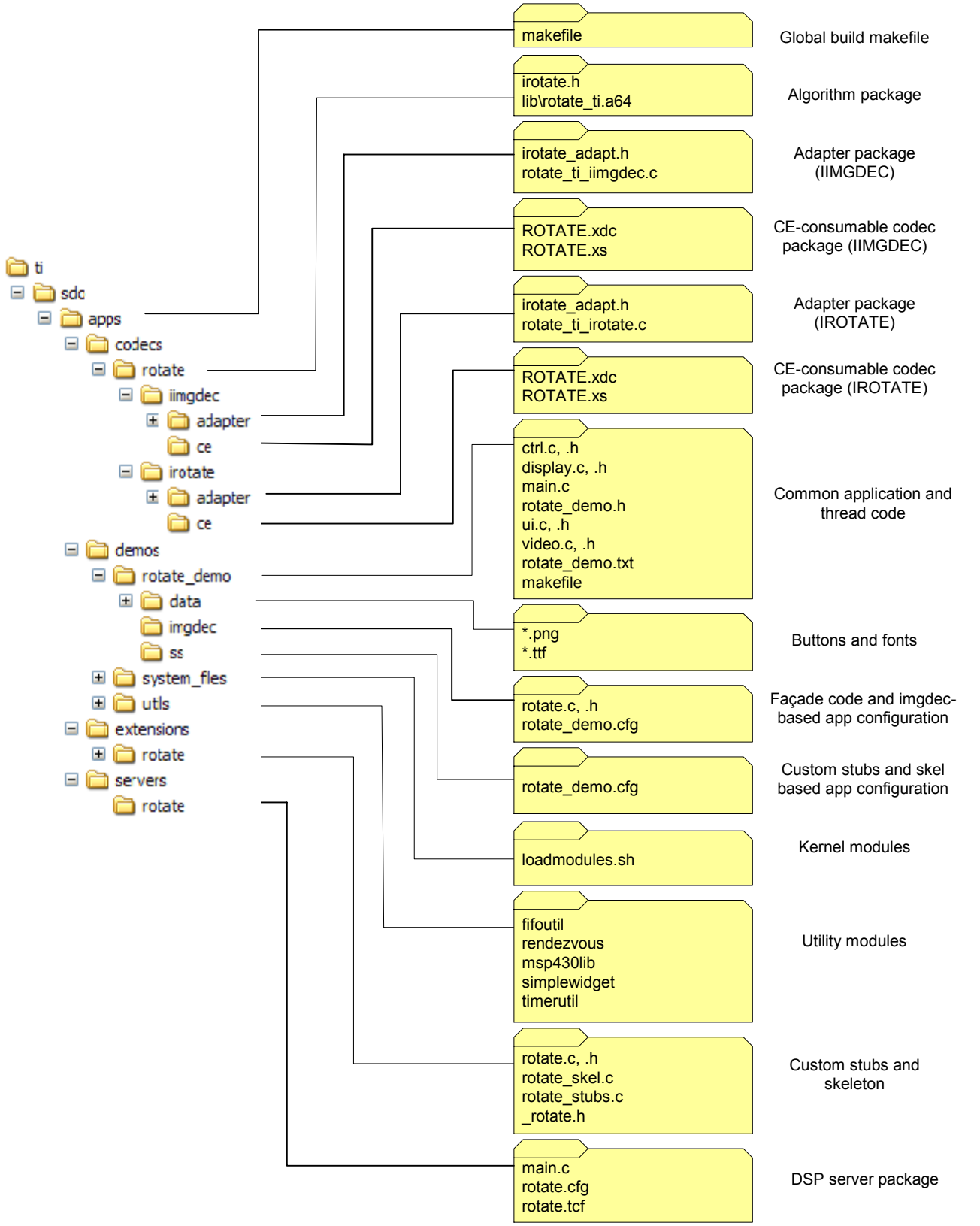


Figure 5. Directory Structure of the Companion Code

4.1 Adapter as Part of a CE-Consumable Codec Package

For the basics on how to make codec packages for the XDC Configuration Kit, refer to the *Codec Engine Algorithm Creator User's Guide* [Reference 1]. As mentioned previously, instead of creating the codec package as a single entity, we have defined three codec-related packages: the xDAIS algorithm package, the adapter package, and the CE-consumable codec package. We prefer this split-up because generic xDAIS algorithm packages should not require the use of the CE as the framework. Users are free to choose the framework of their choice. By decoupling the CE specifics from the core algorithm package, we can promote better reuse in the long run.

4.1.1 How to Create the Algorithm Package with a Pre-Built Library

We started with a pre-built ROTATE_TI algorithm library file (rotate_ti.l64) that was built for the C6400 and wrapped it into a package by following the procedure in Section 2.2 of the *Codec Engine Algorithm Creator User's Guide* [Reference 1]. The resulting package is stored in ti/sdo/apps/codecs/rotate/.

One modification we have made to the procedure is to omit the CE specific files (that is, ROTATE.xdc and ROTATE.xs) from this package. This allows this package to be framework-agnostic. We will show how to package these files separately in a subsequent section.

Since we already had a pre-built library file instead of a set of source files, we did not need to write anything in the package build script package.bld. (We left in a statement to inform the XDC Configuration Kit that all files in the package are to be compressed into a tar file during a release. Please refer to XDC documentation for details on how to release a package as a tar file if you are interested, as this is beyond the scope of this application note.)

In package.xs, we defined the filename of the algorithm library to be exported by our codec package for the C64x+ architecture as follows:

```
    if (prog.build.target.isa == "64P") {
        lib = "lib/rotate_ti.l64";
        print("    will link with " + this.$name + ":" + lib);
    }
    return (lib);
```

This allows us to use the legacy library file. We then placed the library file into the ti/sdo/apps/codecs/rotate/lib/ directory, and the irotate.h header file provided with the algorithm into the ti/sdo/apps/codecs/rotate/ directory (we need this public header file to access the data structures exposed by the algorithm), and voila, we made a codec package.

One important thing to note is that we have chosen our package name to be ti.sdo.apps.codecs.rotate, and placed the package in the corresponding directory ti/sdo/apps/codecs/rotate. It cannot be understated how important it is to have the vendor name (ti.sdo.apps stands for Applications Team at Texas Instruments' Software Development Organization) as part of the package name because it ensures the name's uniqueness and would prevent name collision with other vendors' codec packages. So if another vendor had a different color rotation algorithm package, the XDC Configuration Kit can easily distinguish between the two packages based on their names. This rule applies similarly to server package names.

4.1.2 How to Create the Adapter Package

We deliberately placed the adapter package into a subfolder of the codec package, so that the adapter name becomes ti.sdo.apps.codecs.rotate.iimgdec.adapter for the one that implements the iimgdec interface and ti.sdo.apps.codecs.rotate.irotate.adapter for the one that implements the customized irotate interface. This implies the adapter is meant for this particular codec.

The package has the usual makefile and package.* files to name the package, build it and export the adapter library. It also contains the headers and sources for the adapter.

4.1.3 How to Create the CE-Consumable Codec Package

After creating the codec and adapter packages, we then made an extra package called ti.sdo.apps.codecs.rotate.*.ce (where * is either iimgdec or irotate) that declares static properties relevant to the Codec Engine. In ROTATE.xdc for the iimgdec version, we declare the xDM class we are trying to implement and the algorithm function table:

```
metaonly module ROTATE inherits ti.sdo.ce.image.IIMGDEC
{
    /*!
     *  ======== ialgFxns ========
     *  name of the adapted algorithm's xDAIS alg fxn table
     */
    override readonly config String ialgFxns = "ROTATE_TI_IIMGDEC";
}
```

The key is that we used the symbol name of the function table defined by the adapter and not the one created by the algorithm (named ROTATE_TI_IALG). The function table implemented by the adapter is effectively acting as a substitute for the original algorithm function table.

The ROTATE.xs file provides the getStackSize() function, which is used when building an application for the Codec Engine to declare the combined amount of stack usage for the algorithm and the adapter. One way to determine this stack depth is to run the Perl script ‘call_graph.pl’ supplied in the Code Generation Tools XML Output Perl Utility Scripts, and look up the maximum of the worst-case stack depth of each function in the adapter.

Another key file is package.xdc:

```
requires ti.sdo.apps.codecs.rotate;
requires ti.sdo.apps.codecs.rotate.iimgdec.adapter;

package ti.sdo.apps.codecs.rotate.iimgdec.ce {
    module ROTATE;
}
```

It declares the name of the package and the module it contains. Furthermore, it uses the “requires” statement to specify its dependency on the algorithm and adapter package. Hence when the server package is configured to use this CE-consumable package, it picks up the correct libraries from the packages it requires.

Finally, since this package is not building any code, package.bld is left empty (except for a line to specify that all files in the package are to be compressed into a tar file during a release).

The same discussion applies to the irotate version as well.

4.1.4 How to Use this CE-Consumable Codec Package in a Server Configuration File

In the server configuration file (\ti\sdo\apps\servers\rotate\rotate.cfg), we can use this CE-consumable codec package just as any other codec package by importing it as a module:

```
var ROTATE_MODULE = xdc.useModule('ti.sdo.apps.codecs.rotate.iimgdec.ce.ROTATE');
```

From the configuration file's perspective, the adapter and algorithm are treated as a single entity, that is, as an "adapted algorithm". It does not need to know that an adapter is present.

4.2 A Look at the Adapter Code

Let us start by taking a closer look at the adapter that implements the iimgdec interface. The file rotate_ti_iimgdec.c contains the source for the adapter layer. One of the possible purposes of the adapter layer is to replace an xDAIS algorithm's function table with one better suited for its execution on a CE server. In our case, the adapted algorithm should implement the IIMGDEC interface. We picked the IIMGDEC interface because the color rotation algorithm is an image processing algorithm, and is similar to an image decoder in that they both transform one type of image representation to another. We could have similarly picked the IIMGENC interface.

Implementing the IIMGDEC interface involves supplying functions for the IIMGDEC_fxns function table:

```
typedef struct IIMGDEC_Fxns{
    IALG_Fxns   ialg;             /**< Traditional xDAIS algorithm interface. */

    XDAS_Int32 (*process)(IIMGDEC_Handle handle, XDM_BufDesc *inBufs,
            XDM_BufDesc *outBufs, IIMGDEC_InArgs *inArgs,
            IIMGDEC_OutArgs *outArgs);

    XDAS_Int32 (*control)(IIMGDEC_Handle handle, IIMGDEC_Cmd id,
            IIMGDEC_DynamicParams *params, IIMGDEC_Status *status);
}IIMGDEC_Fxns;
```

If the symbols of the IALG functions (for example, ROTATE_TI_algAlloc) of the algorithm are exposed and are directly callable from outside the algorithm (that is, if the nm6x utility from the CodeGen Tools lists the symbols as global), we can plug these symbols into the function table, so long as we do not need to request memory resources for the internal use of the adapter.

```
/* ======== ROTATE_TI_IIMGDEC ========
 *  Structure defines TI's implementation of IIMGDEC interface for ROTATE_TI module. */
IIMGDEC_Fxns ROTATE_TI_IIMGDEC = {
    { &ROTATE_TI_IIMGDEC,
      NULL
      ROTATE_TI_algAlloc,
      NULL,
      ROTATE_TI_algFree,
      ROTATE_TI_algInit,
      NULL,
      ROTATE_TI_numAlloc
    },
    ROTATE_TI_process,
    ROTATE_TI_control,
};
```

However, xDAIS does not require these symbols to be exposed, so this might not be a viable strategy for some algorithms. In general, the IALG functions of the algorithm can instead be accessed through its exposed function table symbol. So we can simply write wrapper functions that call into these IALG functions, and plug these wrapper functions into our new function table.

```
#define ORIG_IALGFXNS (ROTATE_TI_IROTATE.ialg) /* Fxn table of original algo */

static Void algActivate(IALG_Handle handle)
{
    if (ORIG_IALGFXNS.algActivate != NULL) {
        ORIG_IALGFXNS.algActivate(handle);
    }
    return;
}

static Void algDeactivate(IALG_Handle handle)
{
    if (ORIG_IALGFXNS.algDeactivate != NULL) {
        ORIG_IALGFXNS.algDeactivate(handle);
    }
    return;
}
. . . etc

#define IALGFXNS \
    &ROTATE_TI_IIMGDEC, /* module ID                                */ \
    algActivate,        /* activate                                 */ \
    algAlloc,           /* algAlloc                                 */ \
    algControl,         /* control                                  */ \
    algDeactivate,      /* deactivate                               */ \
    algFree,            /* free                                     */ \
    algInit,            /* init                                     */ \
    NULL,               /* moved                                    */ \
    algNumAlloc         /* numAlloc                                 */

/*  ======== ROTATE_TI_IIMGDEC ========
 *  This structure defines TI's implementation of the IIMGDEC interface
 *  for the ROTATE_TI module.
 */
IIMGDEC_Fxns ROTATE_TI_IIMGDEC = {
    {IALGFXNS},
    ROTATE_TI_process,
    ROTATE_TI_control,
};
```

The only tricky one is algMoved(). This is because xDAIS specification stipulates that when algMoved() is NULL, an algorithm is implicitly declaring that its memory resources cannot be moved. Hence if the original algorithm does not implement algMoved(), the corresponding pointer must be set to NULL in the adapter's function table to make the same declaration. This is the case for the ROTATE_TI algorithm; however, for reference purposes, users can add the following at the top of the file to obtain an implementation in the case where algMoved() has been implemented in ROTATE_TI:

```
#define ALGMOVED_IMPL
```

To determine if algMoved() has been implemented in the case where you do not have access to the source code of the algorithm, you can either refer to the algorithm vendor documentation, or use a debugger to verify the location of the algMoved() function pointer in the algorithm function table at run-time.

The two other functions that are required by the IIMGDEC interface are process() and control(), which are shown in the following snippet as ROTATE_TI_process() and ROTATE_TI_control().

```
/*
 *  ======== ROTATE_TI_process ========
 */
static XDAS_Int32 ROTATE_TI_process(IIMGDEC_Handle h, XDM_BufDesc *inBufs,
    XDM_BufDesc *outBufs, IIMGDEC_InArgs *inArgs, IIMGDEC_OutArgs *outArgs)
{
    IROTATE_ADAPT_InArgs * adapted_inArgs = (IROTATE_ADAPT_InArgs *)inArgs;
    IROTATE_Fxns * irotateFxns = (IROTATE_Fxns *)&ORIG_IALGFXNS;
    ROTATE_TI_Obj_Extension * objExt = (ROTATE_TI_Obj_Extension *)h;
    UChar * y;
    UChar * cr;
    UChar * cb;

    /* Error checking on buffer sizes */
    . . .

    /* Set up intermediate y, cr, cb buffers */
    y = objExt->intBuf;
    cr = y + objExt->ySize;
    cb = cr + objExt->crSize;

    /* Demux input picture into component buffers */
    demux((UChar *)inBufs->bufs[0], y, cr, cb, inBufs->bufSizes[0]);

    /* Do in-place processing on component buffers */
    irotateFxns->apply((IROTATE_Handle)objExt->origHandle, (UChar *)y,
        (UChar *)cr, (UChar *)cb,
        inBufs->bufSizes[0]/2, inBufs->bufSizes[0]/4, adapted_inArgs->cosine,
        adapted_inArgs->sine); /* Assume input is 4:2:0 */

    /* Mux component buffers into output buffer */
    mux(y, cr, cb, (UChar *)outBufs->bufs[0], outBufs->bufSizes[0]);

    return (IIMGDEC_EOK);
}
```

```
/*
 *  ======== ROTATE_TI_control ========
 *  This algorithm does not support any control command.
 */
static XDAS_Int32 ROTATE_TI_control(IIMGDEC_Handle handle, IIMGDEC_Cmd id,
    IIMGDEC_DynamicParams *params, IIMGDEC_Status *status)
{
    XDAS_Int32 retVal;

    . . .

    switch (id) {
        default:
            /* unsupported cmd */
            retVal = IIMGDEC_EFAIL;

            break;
    }

    return (retVal);
}
```

Since ROTATE_TI does not expose any control-type function as part of its function table, control() does not need to do anything other than return failure. Note that this function still needs to be implemented since it is required by the IIMGDEC interface.

In process(), we first cast the input arguments to be of type IROTATE_ADAPT_InArgs, which is defined in irotate_adapt.h.

```
typedef struct IROTATE_ADAPT_InArgs {
    IIMGDEC_InArgs  iimgdecInArgs;   /* Must be first field */
    XDAS_Int16 cosine;  /**< cosine of angle by which to rotate chroma planes */
    XDAS_Int16 sine;    /**< sine of angle by which to rotate chroma planes */
} IROTATE_ADAPT_InArgs;
```

This is an extended version of the standard IIMGDEC_InArgs structure. When extending a structure, we need to include the original structure as the first field of the extended structure. This is a quick way to ensure correct offsets to the parameters of interest (that is, sine and cosine) to our algorithm, which appear at the end of our data structure. Furthermore, the VISA APIs supports extensibility of its data structures by providing a size field as the first field in all its creation and runtime parameter/argument structures. By setting this size field in iimgdecInArgs.size to include the two extra fields, the application code can ensure that IMGDEC APIs and its corresponding stubs and skeleton transfer the correct number of bytes for the extended data structure across from the application to the server or vice-versa.

Similarly, we did the same for the output arguments of process().

```
typedef struct IROTATE_ADAPT_OutArgs {
    IIMGDEC_OutArgs iimgdecOutArgs;  /* Must be first field */
} IROTATE_ADAPT_OutArgs;
```

Even though the color rotation algorithm does not return any output arguments, we decided to rename the structure for consistency. Note that the application needs to call IMGDEC_process() with the size field of the iimgdecOutArgs set to the size of IROTATE_ADAPT_OutArgs.

In the same header file, we also extended the creation parameters for the algorithm using the same principle.

```
typedef struct IROTATE_ADAPT_Params {
    IIMGDEC_Params iimgdecParams;  /**< Must be first field */
    IROTATE_Params irotateParams;  /**< Parameters for the algorithm */
    XDAS_Int32 maxImageSize;       /**< Max size of image buffer to be processed */
} IROTATE_ADAPT_Params;
```

In addition to the creation parameters for the ROTATE_TI algorithm, we have also added a field to specify the maximum image size. This extra information is needed by the adapter as we will see later. Again, the use of an extended structure implies we need to set the size field of iimgdecParams to sizeof(IROTATE_ADAPT_Params) when calling IMGDEC_create() in the application with this extended structure.

In process(), we pre-process the input buffer using demux() to obtain the input in separate component buffers, run the ROTATE_TI algorithm on these buffers, and post-process the buffers with mux() to fill the output buffer in standard interleaved format. To do this, process() needs intermediate buffers to store the Y, CR and CB components. But how can we obtain these intermediate buffers? Putting them on the stack is a bad idea, because the adapter layer is effectively part of an algorithm from the CE framework's perspective, and algorithms are encouraged to use minimal amount of stack space as per Guideline 6 in xDAIS specifications [Reference 7]. Another possibility is to allocate contiguous buffers in the application and pass them to the algorithm as input buffers to process(). In this case, because the buffers are passed back and forth between the DSP and the GPP, maintaining cache coherency could introduce extra delays, and buffers are forced to be in external memory as they are typically allocated through the GPP CMEM module.

A third possibility is to allocate these buffers by requesting them through the IALG interface. This is convenient because the adapter is effectively treated by the framework as being part of the algorithm. In our example, we adopted this approach:

```
#define ADAPTER_MEMRECS 2   /* Number of MEMRECS requested by adapter */

static Int algNumAlloc()
{
    if (ORIG_IALGFXNS.algNumAlloc != NULL) {
        return (ORIG_IALGFXNS.algNumAlloc() + ADAPTER_MEMRECS);
    }
    else {
        return (IALG_DEFMEMRECS + ADAPTER_MEMRECS);
    }
}

static Int algAlloc(const IALG_Params * params, IALG_Fxns **fxns,
                    IALG_MemRec memTab[])
{
    IALG_MemRec * origMemTab = &memTab[ADAPTER_MEMRECS];
    const IROTATE_ADAPT_Params * adaptedParams = (IROTATE_ADAPT_Params *)params;
    Int numBufs = ORIG_IALGFXNS.algAlloc(
        (const IALG_Params *)&adaptedParams->irotateParams, fxns, origMemTab);
```

```
    /* Allocate space for the extension */
    memTab[0].size = sizeof(ROTATE_TI_Obj_Extension);
    memTab[0].alignment = 4;
    memTab[0].space = IALG_ESDATA;
    memTab[0].attrs = IALG_PERSIST;

    /* Allocate intermediate buffers */
    memTab[1].size = adaptedParams->maxImageSize;
    memTab[1].alignment = 4;
    memTab[1].space = IALG_ESDATA;
    memTab[1].attrs = IALG_SCRATCH;

    return (numBufs + ADAPTER_MEMRECS);
}

static Int algFree(IALG_Handle handle, IALG_MemRec memTab[])
{
    ROTATE_TI_Obj_Extension * objExt = (ROTATE_TI_Obj_Extension *)handle;
    IALG_MemRec * origMemTab = &memTab[ADAPTER_MEMRECS];
    Int numBufs = ORIG_IALGFXNS.algFree(handle, origMemTab);

    /* Allocate space for the extension */
    memTab[0].size = sizeof(ROTATE_TI_Obj_Extension);
    memTab[0].alignment = 4;
    memTab[0].space = IALG_ESDATA;
    memTab[0].attrs = IALG_PERSIST;

    /* Allocate intermediate buffers */
    memTab[1].size = objExt->ySize + (2 * objExt->crSize);
    memTab[1].alignment = 4;
    memTab[1].space = IALG_ESDATA;
    memTab[1].attrs = IALG_SCRATCH;

    return (numBufs + ADAPTER_MEMRECS);
}

static Int algInit(IALG_Handle handle, const IALG_MemRec memTab[],
    IALG_Handle p, const IALG_Params * params)
{
    Int status;
    ROTATE_TI_Obj_Extension * objExt = (ROTATE_TI_Obj_Extension *)handle;
    const IROTATE_ADAPT_Params * adaptedParams = (IROTATE_ADAPT_Params *)params;

    objExt->intBuf = memTab[1].base;
    objExt->origHandle = memTab[2].base;
    objExt->ySize = (adaptedParams->maxImageSize)/2;
    objExt->crSize = (adaptedParams->maxImageSize)/4;

    /* Set the function table in original algorithm instance object */
    objExt->origHandle->fxns = (IALG_Fxns *)&ORIG_IALGFXNS;

    status = ORIG_IALGFXNS.algInit(objExt->origHandle, &memTab[ADAPTER_MEMRECS],
        p, (const IALG_Params *)&adaptedParams->irotateParams);

    return (status);
}
```

For those of you who had experience in implementing xDAIS algorithms, this code should look somewhat familiar to you. In algInit(), it is important to remember to initialize the “fxns” field with the original algorithm function, as this is expected as part of the xDAIS specifications. Furthermore, in addition to requesting the intermediate buffers using the maxImageSize creation parameter, we also request an extra memTab entry in algInit() to store an extension of the instance object of the algorithm. However, because the instance object of the algorithm is opaque (meaning its structure is not required to be exposed in the algorithms’ header files), we cannot extend this structure in the same way we extend the creation and runtime parameters. So we request a totally separate piece of memory to store any state information for an adapter instance, in which we store the original algorithm handle:

```
/* Extension of the algorithm instance object */
typedef struct ROTATE_TI_Obj_Extension {
    IALG_Obj ialgObj;         /* IALG Object must be first field */
    IALG_Handle origHandle;   /* Handle to original instance object */
    Int ySize;                /* Size of intermediate buffer for y */
    Int crSize;               /* Size of intermediate buffer for chroma */
    UChar * intBuf;           /* Pointer to intermediate buffer */
} ROTATE_TI_Obj_Extension;
```

Whenever we need to call the original algorithm’s functions, we simply need to pass origHandle to the call.

Important Note: If you extend the instance object, all IALG functions that require an IALG_Handle in its parameter list and all functions in the IDMA3 interface (if the IDMA3 interface is implemented in the original algorithm) *must* be wrapped in the adapter. The wrapper functions must call the original IALG/IDMA3 functions by passing in the original instance object handle. Passing in the wrong handle may result in many hours of tedious debugging.

Other concepts, such as the use of facade layers and stubs and skeletons, can optionally be used to complement the use of adapters. For more information on the facade API layer built for this example to promote readability, please refer to Appendix B.

Given there are significant differences between the interface provided by the ROTATE_TI algorithm and the IIMGDEC interface, the cleanest approach is to build custom stubs and skeleton for the ROTATE_TI algorithm. Appendix C has a discussion of an implementation in which we used custom stubs and skeletons alongside an adapter that performs pre- and post-processing.

5 Procedure Summary

In summary, to use an adapter to adapt an algorithm so that it implements a particular SPI defined by a given set of stubs and skeleton (including the ones for the built-in VISA classes), you follow these steps:

1. Choose the SPI you want the adapter to implement. Determine whether you need to define a new interface by writing custom stubs and skeletons. See Appendix C for details.
2. Extend any creation or run-time parameter/argument data structures with extra fields if necessary.
3. Create the adapter's function table that implements the SPI. This includes these steps:
   a. If the IALG function symbols are not individually exposed by the algorithm library, write IALG wrappers that call into the algorithm function table.
   b. Implement any extra functions (for example, process and control) required by the chosen SPI.
4. Package the adapter and the algorithm into a CE-consumable codec package, as discussed in Section 4.1.
5. Create a facade layer for application use, if desired. See Appendix B for details.

Alternatively, to use an adapter to add pre- and post-processing, follow these steps:

1. Extend any creation or run-time parameter/argument data structures with extra fields if necessary for pre/post processing.
2. Create the adapter's function table that implements the same SPI as the algorithm. This includes the following steps:
   a. Write IALG wrappers that call into the algorithm function table. If pre- and post-processing require extra buffer resources, request these resources through the IALG wrappers. You will also need to request extra space to store an extended algorithm instance object, which you need to define.
   b. Write IDMA3 functions if pre- and post-processing need to use the DMA. See Appendix D if this is the case.
   c. Implement any extra functions (for example, process and control) that are required by the chosen SPI. These functions can call into the pre- and post-processing functions.
3. Package the adapter and the algorithm into a CE-consumable codec package, as discussed in Section 4.1.

6 Conclusion

Adapters can be used to implement both xDM and custom SPIs for an xDAIS algorithm while providing a placeholder for pre- and post-processing. When used alongside custom stubs and skeletons, adapters provide a powerful and flexible mechanism to run any xDAIS algorithm, even when the latter is only available as a pre-built binary.

7 References

1. *Codec Engine Algorithm Creator User's Guide* (SPRUED6)
2. *Codec Engine Server Integrator's Guide* (SPRUED5)
3. *Codec Engine Application Developer User's Guide* (SPRUE67)
4. *xDAIS-DM (Digital Media) User's Guide* (SPRUEC8)
5. *DVEVM Getting Started Guide* (SPRUE66)
6. *DVSDK Getting Started Guide* (SPRUEG8)
7. *TMS320 DSP Algorithm Standard Rules and Guidelines* (SPRU352)
8. *Using DMA with Framework Components for 'C64x+* (SPRAAG1)

Appendix A – The Layout of a Codec Engine Application

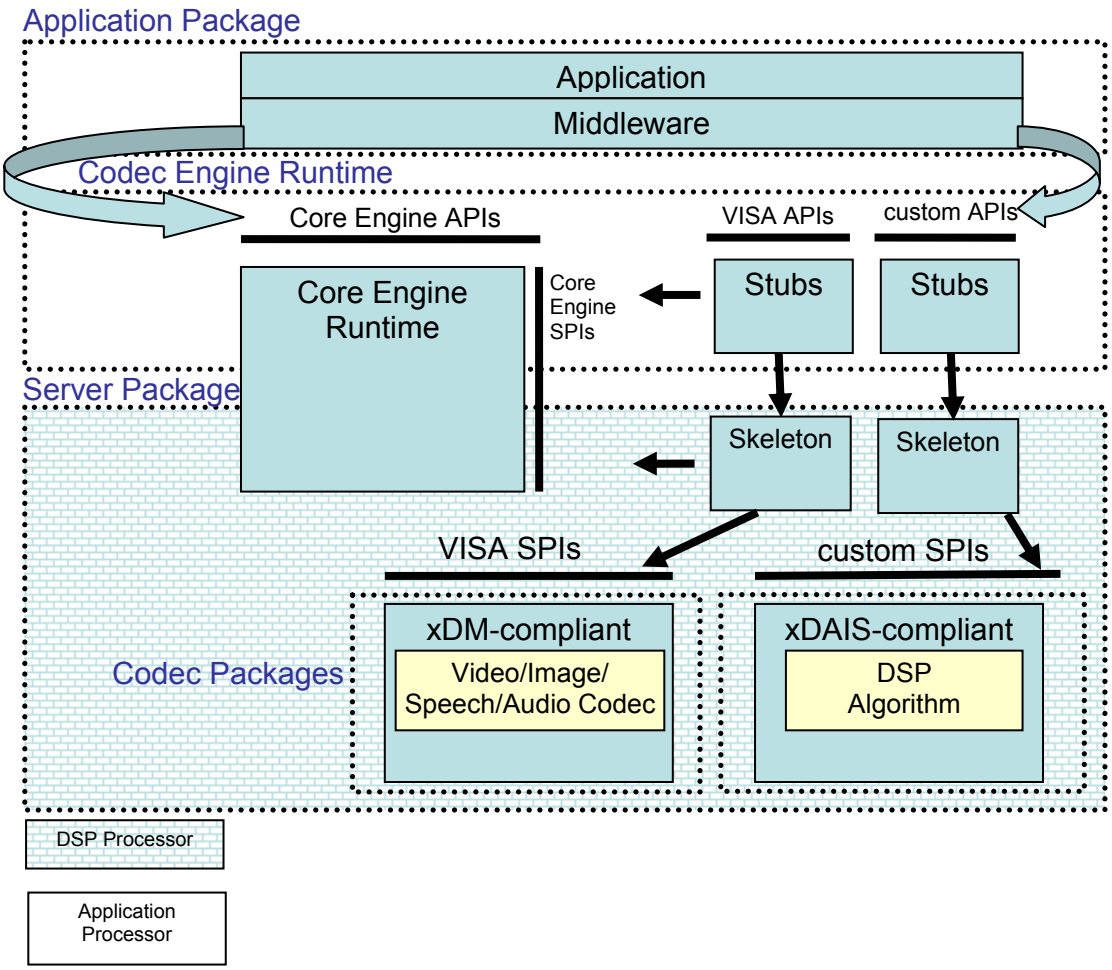


Figure 6. Typical Layout of a CE-Based Application

The preceding figure shows the general architecture of an application that uses Codec Engine. The application (or the middleware) calls the core Engine APIs and the VISA (or custom) APIs. The VISA (or custom) APIs use the stubs to access core engine SPIs (system programming interfaces) and the skeleton. The skeleton then calls into the xDM or custom SPI implemented by the algorithm.

The terms used in the preceding figure are explained briefly in the following list:

- ***xDAIS***: stands for eXpress DSP Algorithm Interoperability Standard. The algorithm interface standard that allows for easy portability, interoperability and measurability of any DSP based algorithms across different vendors.
- ***xDM:*** stands for xDAIS – Digital Media Extension. An extension to xDAIS algorithms that enables plug-and-play architecture for multimedia codecs (Video, Imaging, Speech and Audio also known as VISA) across various classes of algorithms and vendors.

- ***VISA APIs:*** Application interface to talk to Video, Imaging, Speech and Audio Classes of APIs. VISA APIs can be used to call any xDM-compliant algorithm. Custom APIs are needed in order to talk to any xDAIS algorithm that is not XDM-complaint.
- ***Application Processor***: The processor/core that runs the application. It can be a different processor than the one that runs the DSP algorithm (for example, ARM926EJ-S on the TMS320DM6446)
- ***DSP Processor***: The processor/core that runs the DSP algorithm or codecs
- ***Stubs:*** The Codec Engine component that marshals the arguments sent from the application processor (GPP) to the DSP processor. This involves placing all the arguments in an inter CPU message and converting all the GPP virtual memory references to DSP physical memory addresses. It also unmarshals the arguments returned from the DSP processor to the application processor (GPP).
- ***Skeletons***: The CE component on the DSP processor that unmarshals the arguments sent from the stubs and marshals the arguments on the way back from the DSP processor to the application processor.
- ***SPIs***: System programming Interface
- ***APIs:*** Application programming Interface
- ***Codec***: Any xDAIS-compliant algorithm
- ***Server***: The DSP executable that runs the codecs.

If you are interested in additional details on these components, please see references [1,2,3,4].

Appendix B – Facade API Built on Top of IMGDEC

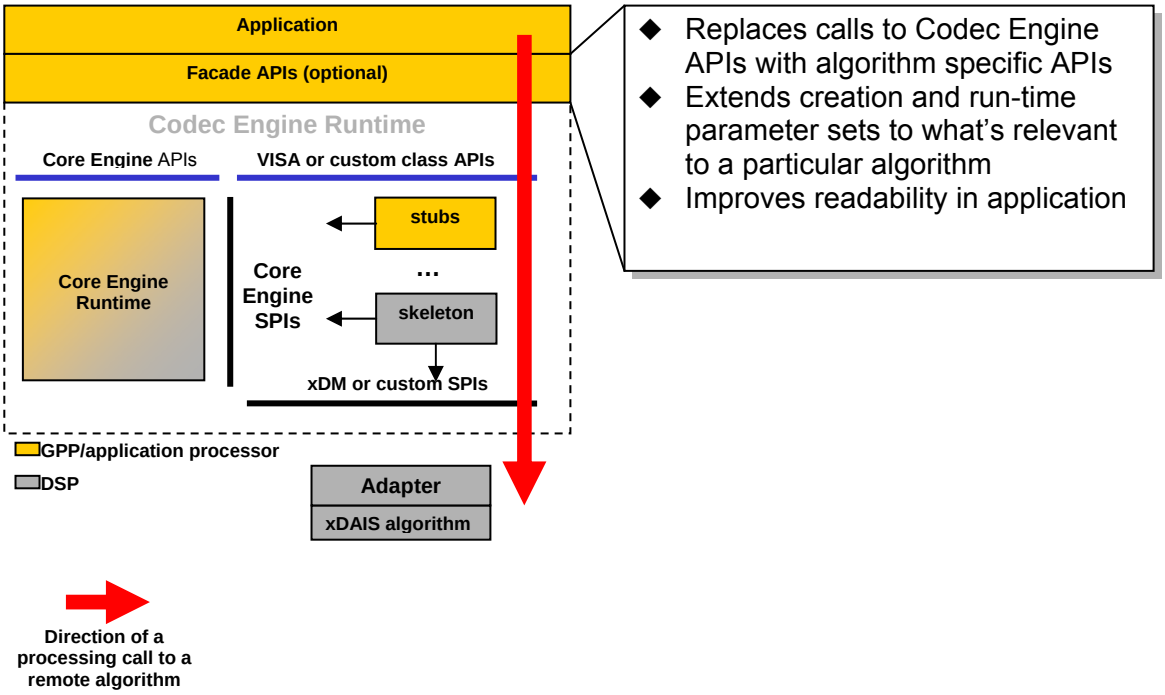


Figure 7. Facade Layer in Remote Execution of an Algorithm Using the Codec Engine

In Figure 7, we show an optional facade API that calls into the Codec Engine runtime APIs. This extra layer can be used to define extended creation and runtime parameter structures of an algorithm in the event that the standard structures provided by the class API are lacking or insufficient for a specific algorithm you are trying to run. It also enhances readability as the layer can define function names that are specific to your algorithm. For example, say if the MP3 standard evolved and an "advanced" MP3 encoder now requires an extra creation parameter named "genre" that specifies the way audio is encoded. This parameter clearly does not appear as a field in the AUDENC_Params structure in the Codec Engine. Consequently, in the facade layer, we can define an extended version of AUDENC_Params named ADVMP3_Params that has all the fields in an AUDENC_Params plus the extra "genre" field. We can also define APIs prefixed with ADVMP3_ such as ADVMP3_create that calls into AUDENC_create to create a more user-friendly API that would use the extended creation parameter structure.

In our example code, to enhance readability of the code at the application level, and also to differentiate the use of the IMGDEC APIs to run a color rotation algorithm versus a standard image decoder defined by xDM, we put together an optional, thin facade layer in the /myexamples/ti/sdo/apps/demos/rotate_demo/imgdec/rotate.* files. This also helps reduce the chance of misuse of the IMGDEC APIs, as the arguments expected by a color rotation algorithm are not 100% identical to those of an image decoder. For example, the API defines a ROTATE_process() function that has the following signature:

```
XDAS_Int32 ROTATE_process(ROTATE_Handle handle, char inBuf[],
    char outBuf[], XDAS_Int32 bufSize, XDAS_Int16 cosine, XDAS_Int16 sine)
```

This API calls into IMGDEC_process() internally. However, it explicitly defines the buffers and input arguments it expects instead of using the XDM_BufDesc and IMGDEC_InArgs structure, hence an application using this API is much more readable. When using this API, there is no question regarding how many buffers need to be passed and which fields in the input argument structure need to be set, etc.

This layer is also used to define default creation parameters, which assist the application writer to set the fields that are not used by the color rotation algorithm. In the case of the ROTATE_TI algorithm, the only field we are reusing is the size field in the IIMGDEC_Params structure. Hence we set it accordingly and set the rest of its unused fields to 0:

```
typedef IROTATE_ADAPT_Params ROTATE_Params;

/* Default creation parameters */
ROTATE_Params ROTATE_PARAMS = {
    {sizeof(ROTATE_Params),  /**< Total size of the extended structure */
    0,                       /**< Maximum height. (unused) */
    0,                       /**< Maximum width. (unused) */
    0,                       /**< Maximum number of scans. (unused) */
    0,                       /**< Endianness of output data. (unused) */
    0},                      /**< Force decode in given Chroma format. (unused) */
    {sizeof(IROTATE_Params), /**< size of original algorithm params structure */
    FALSE},                  /**< Bool reverseImage */
    0                        /**< Max image size to be processed */
};
```

Remember that this layer is optional, and is only useful when the functionality of the algorithm is quite different from the VISA class chosen. If we were truly running an image decoder instead of a color rotation algorithm, this facade layer would probably not be necessary, and the application can directly call the IMGDEC APIs without any loss in readability.

Appendix C – Using Custom Stubs and Skeletons with Adapters

Using the IMGDEC class, though convenient, may require extra data structures to be copied between the application and the server. We can try to simplify the message passing and improve readability of the application by writing our own custom stubs and skeleton.

Immediately, a question comes to mind: instead of writing an adapter, would it be possible to put the equivalent code into the application API and/or skeleton layer? The answer is: not always. Though stubs and skeletons allow customization of the interface between the algorithm and the application, there is ground that cannot be covered solely by stubs and skeletons, and that would require an adapter to be added:

- An adapter is needed if pre/post processing is to be done on the same processor as the algorithm.
  - Pre/post processing code should not go into the skeleton. This is because the process and control calls only go through the skeleton when the algorithm executes remotely. When the algorithm is local, the application API calls directly into the algorithm's function table.
  - Pre/post processing code should not go into the application API layer. The application API stays the same both locally and remotely to ensure an algorithm can be called in the same way in both cases. In a remote call, the skeleton calls the application API a second time on the DSP server. Hence any pre/post processing done in this layer would be done twice.
- An adapter is needed if the interface exposed by the algorithm is not well-suited for remote execution.
  - A given set of stubs and skeleton is designed to support a specific system programming interface (SPI). The algorithm's interface might not necessarily be a good SPI, in the sense that it might not be well-suited to support remote execution. For example, if the algorithm does not require the input buffer size as an argument (perhaps it was set to a fixed value during algorithm creation time), the stubs would not know the size of the buffer to cache invalidate. Therefore, as a general rule of thumb, sizes of input and output buffers must be part of the arguments passed to any run-time processing function.

Now, another group of skeptics may ask: why do we need custom stubs and skeletons? In general, here are the advantages in building custom stubs and skeletons instead of using the VISA APIs when an adapter is in use:

- No excess baggage from unused data structures and features supported by VISA and xDM. This eliminates overhead by reducing the size of the messages exchanged between the application and the server and the amount of data to be cache invalidated and written back in a remote procedure call.
- No need for any facade layer to enhance readability in the application. The class API supported by the stubs and skeleton can expose only what's relevant to the algorithm in question.

Hence custom stubs and skeletons can be a really good complement to adapters, in the sense that they can make the infrastructure much better suited for the adapted algorithm.

Going back to our color rotation code example, we have put in switches in both the ARM side makefile and the DSP server's configuration file to allow easy switching between using the stubs and skeleton for the built-in IMGDEC class and using a custom set of stubs and skeleton. See Section 3.3 for build instructions. By flipping these switches, the following changes occur:

- The DSP server picks up the CE-consumable codec package that implements the custom IROTATE SPI supported by the custom stubs and skeleton in the /myexamples/ti/sdo/apps/extensions folder.
- The application no longer uses the facade layer, and instead calls into the ROTATE class API exported by the custom stubs and skeleton.

The stubs and skeleton are built in such a way that they implement the SPI defined in /myexamples/ti/sdo/apps/codecs/rotate/irotate/adapter/irotate_adapt.h. For more details on building stubs and skeletons, refer to the Codec Engine Algorithm Creator User's Guide [Reference 1]. Note that the data structures no longer carry references to the structures in IIMGDEC, and the process function signature is simplified to match what is necessary for the adapter to run the algorithm.

The adapter code in /myexamples/ti/sdo/apps/codecs/rotate/irotate/adapter/rotate_ti_irotate.c is similar to the one in the iimgdec case. One difference is that we no longer need to expose a control function pointer in the adapter's function table:

```
IROTATE_ADAPT_Fxns ROTATE_TI_IROTATE_ADAPT = {
    {IALGFXNS},
    ROTATE_TI_process,
};
```

Also the code in the IALG wrapper functions and the process() function is simplified as the data structures used to specify the arguments and creation parameters are no longer cluttered with unused fields inherited from the IIMGDEC interface.

In light of this implementation, it is recommended to build custom stubs and skeletons if it is judged that there are too many fields in the creation parameter and runtime argument data structures that are unused, or if the function table exported by xDM is unsuitable for the purposes of the algorithm we are dealing with.

Appendix D – Using DMA in Adapters

All eXpressDSP-compliant algorithms are not allowed to directly access or control hardware or peripherals that include the DMA. The Codec Engine framework uses an algorithm's IDMA3 interface to query that algorithm's DMA resource requirements and to request access. As a result, algorithms are granted controlled access to DMA resources. Since an adapter effectively acts as a part of an algorithm, it should use the IDMA3 interface to request DMA resources if the algorithm needs DMA for pre- or post-processing. For further details on the IDMA3 interface, please see *Using DMA with Framework Components for 'C64x+* [Reference 8].

The IDMA3 interface can be implemented in adapters in the following two scenarios.

a. If the xDAIS algorithm implements an IDMA3 interface, then the adapter needs to write wrappers for it. This implementation scenario would involve the same principles that were used in wrapping the IALG interface of the algorithm. The adapter would implement a substitute IDMA3 function table similar to the code snippet below. Each of the 4 functions in the function table are implemented by calling the corresponding IDMA3 function implementation from the xDAIS algorithm. Any extra resources needed by the adapter can be requested through these wrapper functions.

b. If the xDAIS algorithm does not use the DMA, then the adapter simply implements the four IDMA3 functions to request the DMA resources it needs for pre- and post-processing and exposes them in a function table. An example is provided in the companion code of this application note. As per Section 4.2, the ROTATE_TI_process calls pre-process function demux() to separate Y, Cb and Cr components of input buffer and post-process function mux() to join Y, Cb and Cr component to form interleaved output buffer. Both these functions could benefit from the use of DMA resources as they primarily involve data re-ordering. The **rotate_ti_iimgdec_idma3.c** file in ti/sdo/apps/codecs/rotate/iimgdec/adapter implements a version of the adapter that uses DMA to transfer the input buffer from external memory to internal memory and the output buffer from internal memory to external memory in ping-pong fashion, while the DSP CPU is working on demux() and mux() operations on internal memory data. This file exposes IDMA3 function table ROTATE_TI_IIMGDEC_IDMA3 in addition to IIMGDEC function table ROTATE_TI_IIMGDEC (Section 4.2). The code snippet below shows this function table:

```
/*
 * ==========ROTATE_TI_IIMGDEC_IDMA3==========
 *  This structure defines TI's implementation of the IDMA3 interface
 *  for the ROTATE_TI module.
 */

IDMA3_Fxns ROTATE_TI_IIMGDEC_IDMA3 = {
    &ROTATE_TI_IIMGDEC,
     ROTATE_TI_dmaChangeChannels,
     ROTATE_TI_dmaGetChannelCnt,
     ROTATE_TI,dmaGetChannels,
     ROTATE_TI_dmaInit,
}
```

As you can see from the preceding IDMA3 function table, the first entry is the IIMGDEC function table's address for the ROTATE_TI module, which is used as the implementation ID. The last four functions are to be implemented by the algorithm to allow the Codec Engine framework to negotiate DMA resources.

After requesting the resources, the adapter is then free to use the ACPY3 library to configure and initiate DMA transfers in its processing function(s).

Besides implementing the IDMA3 interface, the following other changes are required in order to use DMA in an adapter:

- After implementing the IDMA3 function table above in the adapter package, we need to make the Codec Engine aware of this function table. How we do it? We modify the ROTATE.xdc file (in the ti/sdo/apps/codecs/rotate/iimgdec/ce directory) in the CE-consumable package of IIMGDEC to declare the IDMA3 function table ROTATE_TI_IIMGDEC_IDMA3 as follows:

.

```
override readonly config String idma3Fxns = "ROTATE_TI_IIMGDEC_IDMA3";
```

  This informs Codec Engine that DMA is being used within the adapter/algorithm.

- As discussed in Section 4.2, the original adapter allocates the intermediate buffers Y, Cr and Cb for the rotate example. This is done in the algAlloc function by specifying a memTab[1] structure. For the DMA version, we have a total of five memTab structures for intermediate buffers for interleaved data and planar component data (Y, Cr and Cb). Also, all the intermediate buffers are allocated in internal memory, and memory alignment is changed to 128 instead of 4. This allows any cache operations (for example, invalidate and writeback) to be performed on cache line boundaries without causing memory corruption. The additional memTab structures are added to the algFree() function as well.

```
static Int algAlloc(const IALG_Params * params, IALG_Fxns **fxns,
            IALG_MemRec memTab[])
{
    ...

   /* Allocate PING YCrCb planar buffers */
    memTab[1].size = (adaptedParams->maxImageSize)/NUM_SLICES;
    memTab[1].alignment = 128;  /* Align to cache line boundary */
    memTab[1].space = IALG_DARAM0;
    memTab[1].attrs = IALG_SCRATCH;

    ...

    /* Allocate PONG interleaved buffers */
    memTab[4].size = (adaptedParams->maxImageSize)/NUM_SLICES;
    memTab[4].alignment = 128;  /* Align to cache line boundary */
    memTab[4].space = IALG_DARAM0;
    memTab[4].attrs = IALG_SCRATCH;

    return (numBufs + ADAPTER_MEMRECS);
}
```

- As discussed in Section 4.2, the adapter extends the instance object via the ROTATE_TI_Obj_Extension structure. Additional fields are needed in this structure to store DMA channel handles and extra intermediate buffer pointers. The code snippet that follows shows four DMA channel handles added to the instance object for DMA version of the adapter.

```
/* Extension of the algorithm instance object */
typedef struct ROTATE_TI_Obj_Extension {
    IALG_Obj ialgObj;           /* IALG Object must be first field */
    IALG_Handle origHandle;     /* Handle to original instance object */
    Int ySize;                  /* Size of intermediate buffer for y */
    Int crSize;                 /* Size of intermediate buffer for chroma */
    UChar * pingPlanarBuf;      /* Pointer to PING planar YCbCr buffer */
    UChar * pongPlanarBuf;      /* Pointer to PONG planar YCbCr buffer */
    UChar * pingIntBuf;         /* Pointer to PING Interleaved buffer */
    UChar * pongIntBuf;         /* Pointer to PONG Interleaved buffer */
    IDMA3_Handle pingdmaInput;  /* Pointer to input DMA Channel for PING buffer*/
    IDMA3_Handle pongdmaInput;  /* Pointer to input DMA Channel for PONG buffer*/
    IDMA3_Handle pingdmaOutput; /* Pointer to output DMA Channel for PING buffer */
    IDMA3_Handle pongdmaOutput; /* Pointer to output DMA Channel for PONG buffer */
} ROTATE_TI_Obj_Extension;
```

The steps to run the DMA version of the application (using rotate_ti_iimgdec_idma3.c instead of rotate_ti_iimgdec.c) are described in the readme_SPRAAE7.txt file.

IMPORTANT NOTICE

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

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

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

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

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

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

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

Products		**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
Low Power Wireless	www.ti.com/lpw	Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

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