Application Report
SPRAAG9A – April 2007

Decode Demo for the DVEVM/DVSDK 1.2

Niclas Anderberg *SDO Applications*

ABSTRACT

The DaVinci Digital Video Evaluation Module (DVEVM) comes with demonstration applications that illustrate the use of its software and hardware components. This document describes the design of the “decode” demo application. The decode demo uses the Codec Engine as well as video, audio, and speech algorithms from Texas Instruments to decode video and sound data from files on the Linux file system, and outputs them to Linux device drivers controlling the video and audio peripherals on the DM6446.

Contents

Figures

1 Overview

The decode demo shows how to decode video and audio or speech using algorithms and the Codec Engine from Texas Instruments on the DaVinci DM6446 DVEVM board. The audio algorithms used are MPEG1L2 and AAC. For speech, G.711 is used. For video, the MPEG2, MPEG4, and H.264 algorithms are used. These algorithms implement the xDM interface (see Section 4 for information references) and are packaged in a Codec Server (decodeCombo.x64P) that is managed by the Codec Engine. The algorithms are executed on the DM6446 DSP core. The encoded video, audio, and speech data is read from separate elementary streams on the Linux file system, and the decoded data is output to peripherals on the DM6446 device.

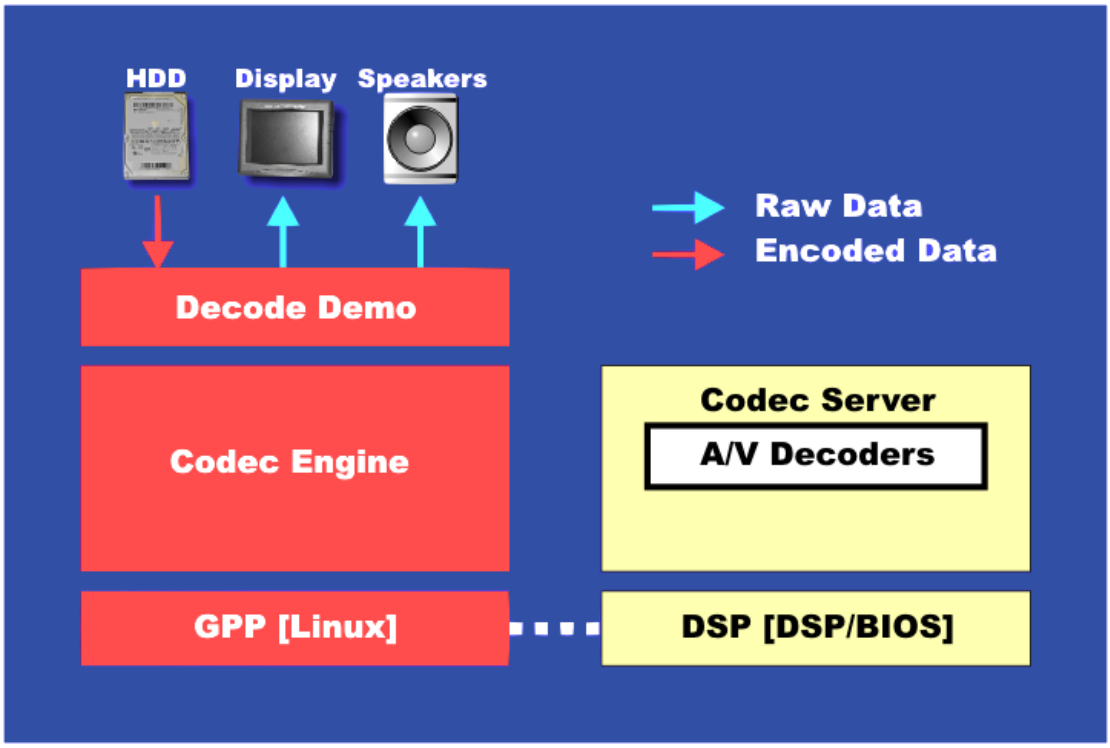

Figure 1. Decode Demo Architecture

The DaVinci ARM core runs the demos on the Linux operating system, and all peripherals are controlled through Linux device drivers. The ARM also displays a user interface on the OSD (On Screen Display) and takes input from a remote control that allows users to send commands through the EVM board's IR interface. The DSP core runs the DSP/BIOS real time operating system and performs algorithm processing.

For information on how to run the decode demo, including documentation on the command-line parameters, see Section 4 on how to find the decode.txt file.

2 Application Design

The application consists of five separate POSIX threads (pthreads): the *main thread* (main.c), which eventually becomes the *control thread* (ctrl.c), the *video thread* (video.c), the *display thread* (display.c), the *audio thread* (audio.c), and the *speech thread* (speech.c). The video, display, audio, and speech threads are spawned from the main thread before the main thread becomes the control thread. The video and display threads are only created if a video file was provided on the command line. The same goes for the speech and audio threads. The user must supply at least one file (speech, audio, or video) for the demo to run, and only one audio or speech file (since audio and speech require the same peripherals). This means at least 2 and at most 4 application threads are running in the demo process.

All threads except the original main/control thread are configured as preemptive and priority-based scheduled (`SCHED_FIFO`). The display thread has the highest priority, followed by the video thread. The speech and audio threads have a lower priority than the video thread, and the control thread has the lowest priority of all. For more on POSIX threads see Section 4.

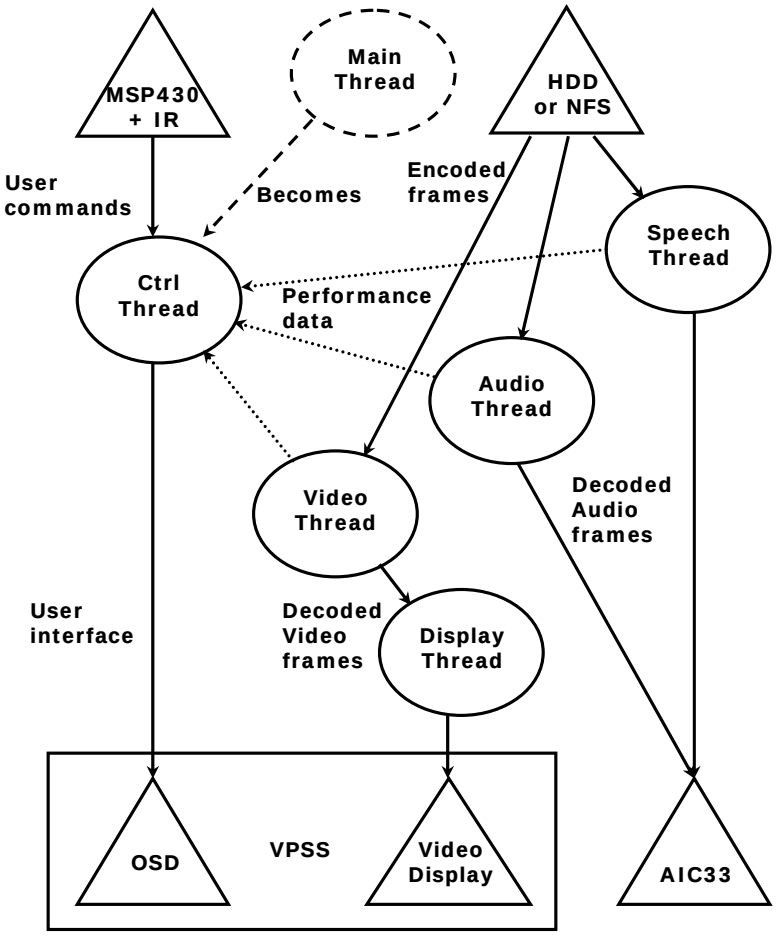

Figure 2. Decode Demo Threads

The initialization and cleanup of the threads are synchronized using the provided Rendezvous utility module, which is initialized early in the main thread. This module uses POSIX conditions to synchronize thread execution. Each thread performs its initialization and signals the Rendezvous

object when completed. When all threads have finished initializing, all threads are unlocked simultaneously and start executing their main loops. The same method is used for thread cleanup. This way buffers which are shared between threads are not freed in one thread while still being used in another.

2.1 Main Thread

The job of the main thread is to perform necessary initialization tasks, to parse the command-line parameters provided by the user when invoking the application, and to spawn the other threads with parameters depending on the value of the command-line parameters.

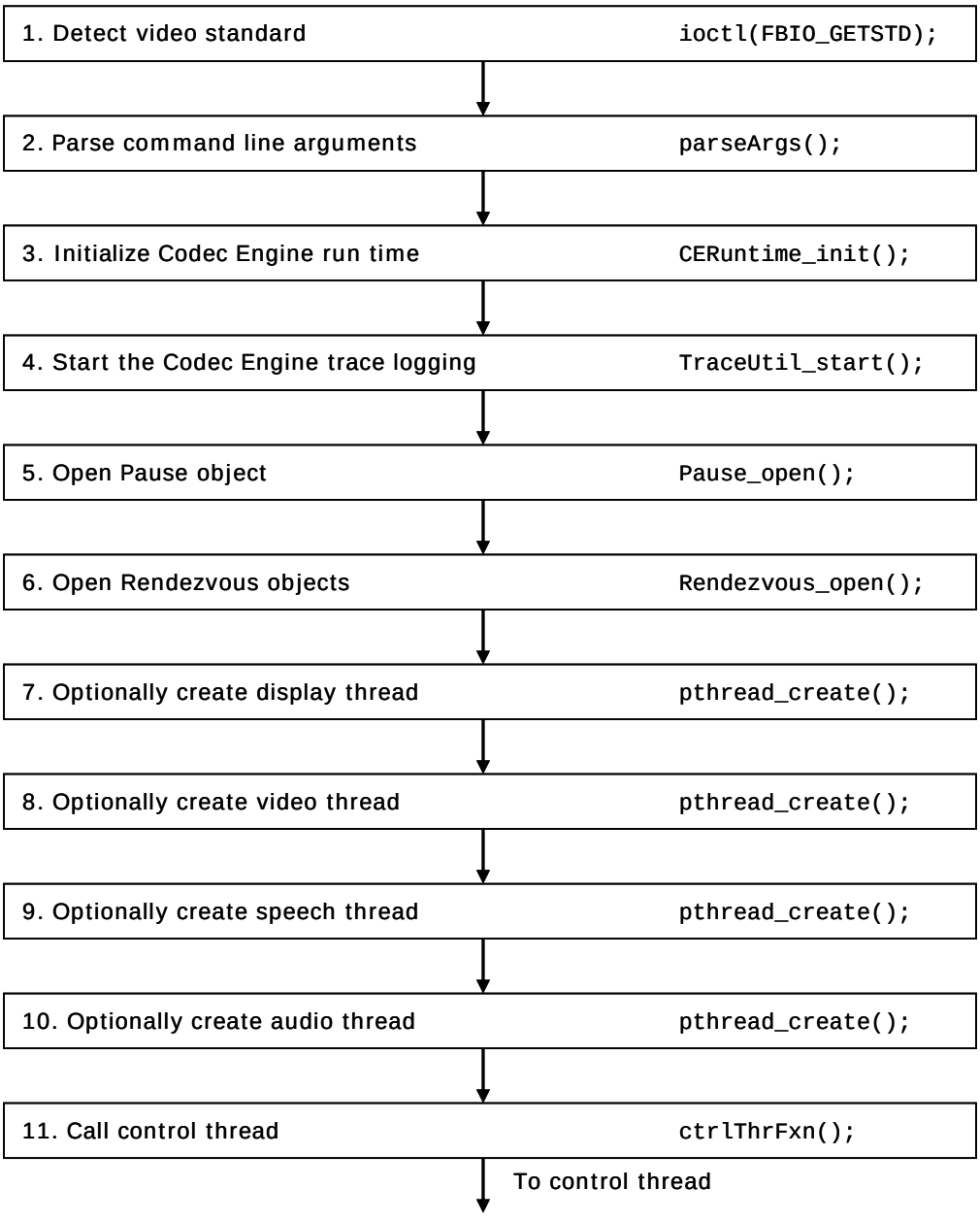

Figure 3. Main Thread Flow

As Figure 3 shows, first the video standard chosen by switch 10 on S3 on the DVEVM board is detected using the `FBIO_GETSTD` ioctl of the FBDev display device driver. The command-line parameters passed are then parsed, and thread environment variables are set accordingly. The Codec Engine and its TraceUtil module are initialized for trace logging (see Section 4 for documents with more details). The Pause object for synchronizing processing pausing and the Rendezvous objects for synchronizing thread initialization and cleanup are opened, and then the processing threads are created depending on the command-line parameters passed to the application. After one or more of these threads have been created, the control thread's main function `ctrlThrFxn()` is called and the main thread becomes the control thread.

2.2 Control Thread

This thread is responsible for the user interface. It uses the utility library msp430lib to poll the msp430 processor, which controls the IR interface on the DaVinci EVM board, for commands. Optionally, if the keyboard interface has been enabled from the command line, stdin is polled to see if a command has been given from the command line in `getKbdCommand()`. Once a new IR command is received or a command-line command is given, the command is identified and the corresponding action is taken in `keyAction()`. Since the msp430 has to be polled for whether a new key has been pressed or not, `usleep()`puts the thread to sleep for a while before checking for another command.

The control thread also draws and updates the text and graphics on the OSD. On the DaVinci platform the OSD window (accessible through `/dev/fb/0`) is in the foreground of the video window (accessible through `/dev/fb/3`). The transparency of the OSD—that is, how much of the video window is seen through the OSD—is set using the attributes window (accessible through `/dev/fb/2`). In the attributes window the transparency of every pixel is represented by a nibble (4 bits) and its value ranges from 0 (completely transparent) to 7 (no transparency). The control thread uses the function `setOsdTransparency()` to set the transparency of the OSD window. The demo defaults to a transparency of 5.

The control thread uses the simplewidget utility library to draw the buttons and render text on the OSD. In addition to initializing the OSD device in `osdInit()` and creating and drawing the static text and buttons on the OSD during initialization using `uiCreate()`, the control thread also updates the dynamic text approximately once per second using `drawDynamicData()`. In this function performance data (such as bit rates) is gathered from other threads and then displayed on the OSD. Since this performance data is accessed from several threads it must be protected using a mutex, and safe access to these variables is wrapped in inline functions in decode.h. The function `getArmCpuLoad()` calculates the ARM-side CPU load in percent, and the Codec Engine call `Engine_getCpuLoad()` determines the DSP-side CPU load. Other dynamically-displayed data are bit rates, video frames processed per second, and time elapsed. The OSD window is double buffered, in that one display buffer is being displayed while data is being rendered into another buffer called the work buffer. After the dynamic data has been rendered into the work buffer, the work buffer is swapped for the display buffer using the `FBIOPAN_DISPLAY` ioctl before the thread waits on the next vertical sync (29.97 Hz on NTSC and 25 Hz on PAL) using the `FBIO_WAITFORVSYNC` ioctl.

2.3 Speech Thread

The speech thread reads encoded speech data from a file on the Linux file system, decodes the data using a speech decoder, and writes the resulting samples to the AIC33 device driver.

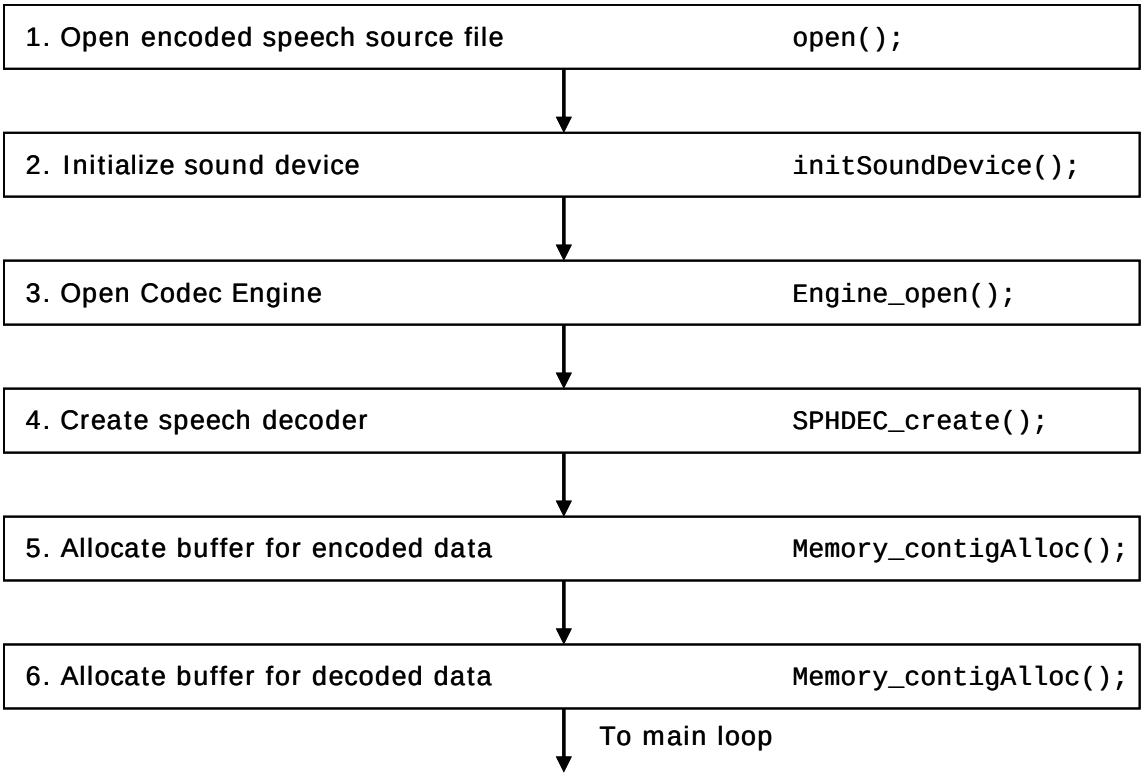

Figure 4. Speech Thread Initialization Flow

As Figure 4 shows, the speech thread initialization is performed as follows:

1. The source file with encoded speech data is opened for reading on the Linux file system.
2. The AIC33 sound device driver is initialized for writing. The device driver is an OSS device driver, see Section 4 for links to documents with more details. First the sound device (`/dev/dsp`) is configured. The AIC33 sound device driver currently supports only 2 channels and 16-bit little endian (`AFMT_S16_LE`) samples. The parameters are set accordingly. Since the speech algorithm supported use an 8 KHz sample rate, the AIC33 is set to this rate. Also, the mixer device (/dev/mixer) is configured with the output gain.
3. A Codec Engine instance is created with `Engine_open()`. This returns a handle to use when instantiating algorithm instances for this engine. All threads using the same engine need a separate handle; access to the engine through this handle is not thread safe.
4. The speech decode algorithm instance is created using `SPHDEC_create()`. Currently the only speech algorithm provided is G.711, and only using alaw. A handle to the algorithm instance is returned that will be used to process (decode) data in the main loop.

5. A contiguous buffer for encoded data of size `READBUFSIZE` is allocated for the encoded data using `Memory_contigAlloc()`. This buffer will be used by the file loader to hold the encoded data, since it is being read from the encoded source file on the Linux file system. See Section 2.6 for details on the loader. Note that a normal buffer allocated with `malloc()` will not work with the Codec Engine remote Codec Servers on the DaVinci platform. Such a buffer would likely be segmented on several pages (one page is 4096 bytes on ARM Linux), and the DSP core requires contiguous memory to work with since it has no MMU.

6. Another contiguous buffer is allocated. This one is for the raw samples after decoding the speech data. Since it will be passed to the DSP side algorithm for processing, it too needs to be contiguous.

When the speech thread has finished initializing, it synchronizes with the other threads using the Rendezvous utility module. Only after the other threads have finished initializing is the main loop of the speech thread executed. The main loop looks like the one in Figure 5:

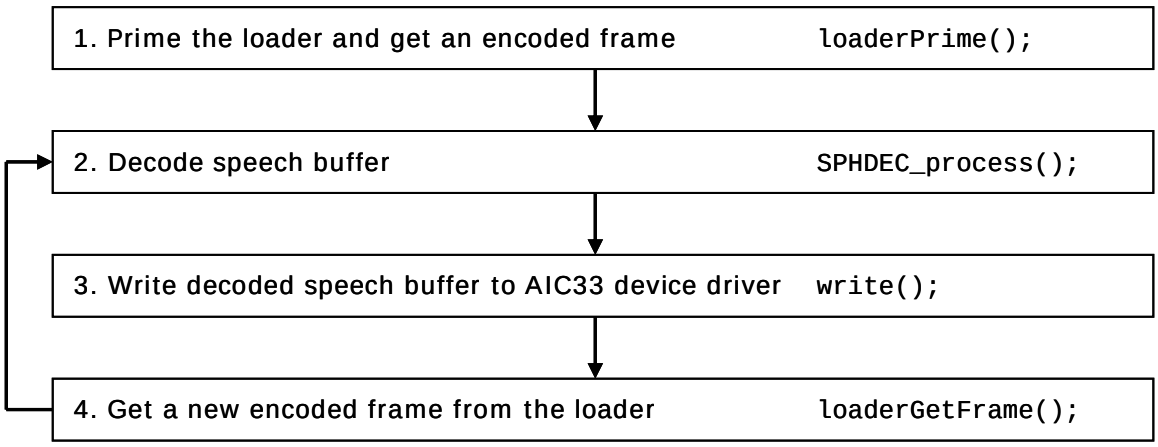

Figure 5. Speech Thread Main Loop Flow

1. Prime the encoded file loader and get an encoded frame. The loader is described in detail in Section 2.6.

2. The encoded frame is decoded using the `SPHDEC_process()` call. This is a Codec Engine procedure call that decodes (DEC) a buffer using a speech (SPH) algorithm. The speech algorithm was configured when it was created. The parameters needed by the process call are the input buffer, the output buffer, and their respective sizes. Because the AIC33 device driver supports only stereo, the decoded mono data is also expanded to stereo samples.

3. Write the decoded stereo sound data to the AIC33 device driver using the standard UNIX `write()` call.

4. Load a new encoded frame from the Linux file system using the file loader. Then return to step 2.

This loop continues until the application is told to quit by the control thread.

2.4 Audio Thread

The audio thread reads encoded audio data from a file on the Linux file system, decodes the data using an audio decoder, and writes the resulting samples to the AIC33 device driver. The audio thread flow is similar to the speech thread flow, except it works with audio data using audio decoders (MPEG1L2 or AAC).

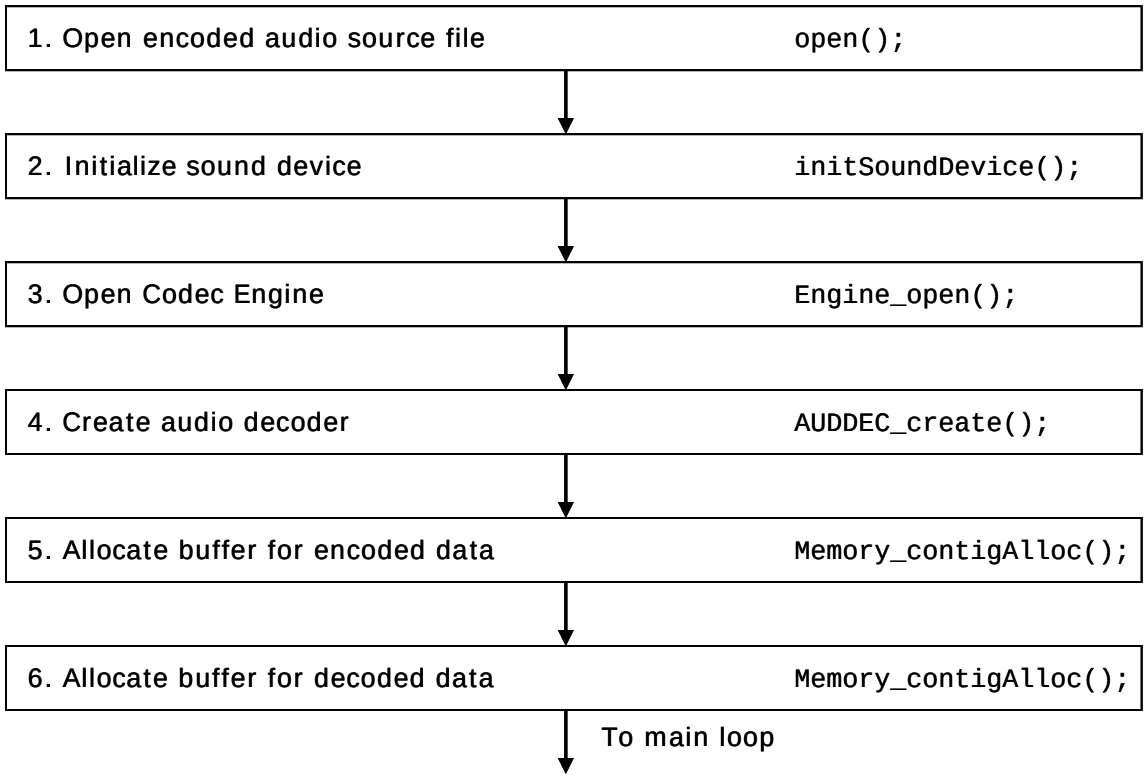

Figure 6. Audio Thread Initialization Flow

As Figure 6 shows, The audio thread initialization is performed as follows:

1. The source file with encoded audio data is opened for reading on the Linux file system.
2. The AIC33 sound device driver is initialized for writing. The device driver is an OSS device driver. See Section 4 for links to documents with more details. First the sound device (`/dev/dsp`) is configured. The AIC33 sound device driver currently supports only 2 channels and 16-bit little endian (`AFMT_S16_LE`) samples. The parameters are set accordingly. Also, the mixer device (/dev/mixer) is configured with the output gain.
3. A Codec Engine instance is created with `Engine_open()`. This returns a handle to use when instantiating algorithm instances for this engine. All threads using the same engine need a separate handle; access to the engine through this handle is not thread safe.
4. The audio decode algorithm instance is created using `AUDDEC_create()`. Currently the supported audio decoders are AAC or MPEG1L2. A handle to the algorithm instance is returned that will be used to process (decode) data in the main loop.

5. A contiguous buffer for encoded data of size `READBUFSIZE` is allocated for the encoded data using `Memory_contigAlloc()`. This buffer will be used by the loader to keep the encoded data, since it is being read from the encoded source file on the Linux file system. See Section 2.6 for details on the loader. Note that a normal buffer allocated with `malloc()` will not work with the Codec Engine remote Codec Servers on the DaVinci platform. Such a buffer would likely be segmented on several pages (one page is 4096 bytes on ARM Linux), and the DSP core requires contiguous memory to work with since it has no MMU.

6. Another contiguous buffer is allocated. This one is for the raw samples after decoding the audio data. Since it will be passed to the DSP side algorithm for processing, it too needs to be contiguous.

When the audio thread is finished initializing, it synchronizes with the other threads using the Rendezvous utility module. Only after the other threads are finished initializing is the main loop of the audio thread executed. The main loop looks like the one in Figure 7:

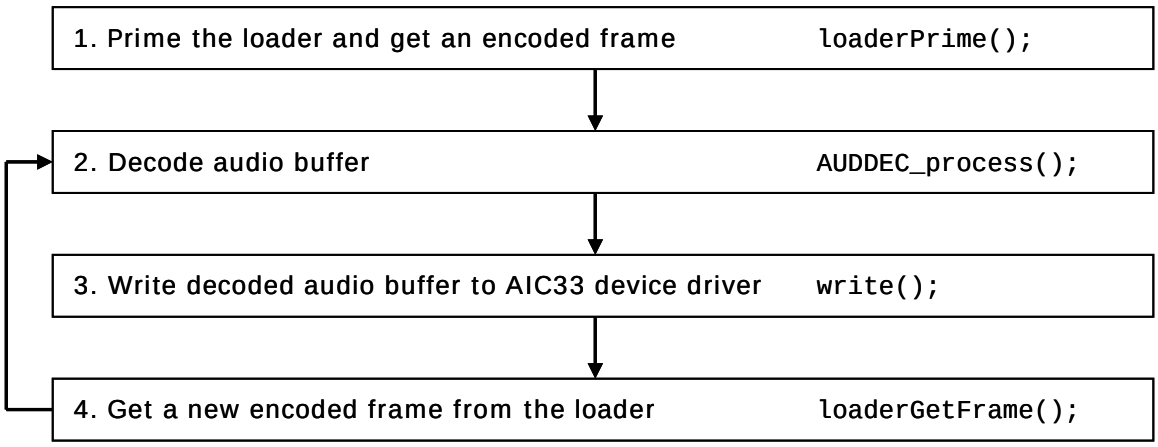

Figure 7. Audio Thread Main Loop Flow

1. Prime the encoded file loader and get an encoded frame. The loader is described in detail in Section 2.6.

2. The encoded frame is decoded using the `AUDDEC_process()` call. This is a Codec Engine procedure call that decodes (DEC) a buffer using an audio (AUD) algorithm. The audio algorithm was configured when it was created. The parameters needed by the process call are the input buffer, the output buffer, and their respective sizes.

3. Write the decoded stereo sound data to the AIC33 device driver using the standard UNIX `write()` call.

4. Load a new encoded frame from the Linux file system using the file loader. Then, return to step 2.

This loop continues until the application is told to quit by the control thread.

2.5 Video Thread

The video thread reads an encoded video frame from the Linux file system and decodes it using a video decoder algorithm.

In order to get more reliable performance, and to avoid dropping frames when one or more frames are demanding to decode, a separate display thread is used to display the frames. If the same thread is used for decoding and displaying the buffer, any frame exceeding its real-time budget (33 ms for NTSC and 40 ms for PAL) will cause a frame to be dropped. By decoupling the decode processing from the display using a number of display buffers (specified by `DISPLAY_BUFFERS`), the video system can handle one or more consecutive frames that exceed their budgets as long as the frames that follow are less expensive to allow the video thread to recover. (An average of 33 ms for NTSC or 40 ms for PAL per frame is required.) The higher the value of `DISPLAY_BUFFERS`, the more consecutive frames can exceed their budgets. However, as a downside, increasing `DISPLAY_BUFFERS` also increases video latency as well as memory requirements. The demo defaults to a `DISPLAY_BUFFERS` setting of 3. This allows for a few consecutive expensive frames while keeping latency low.

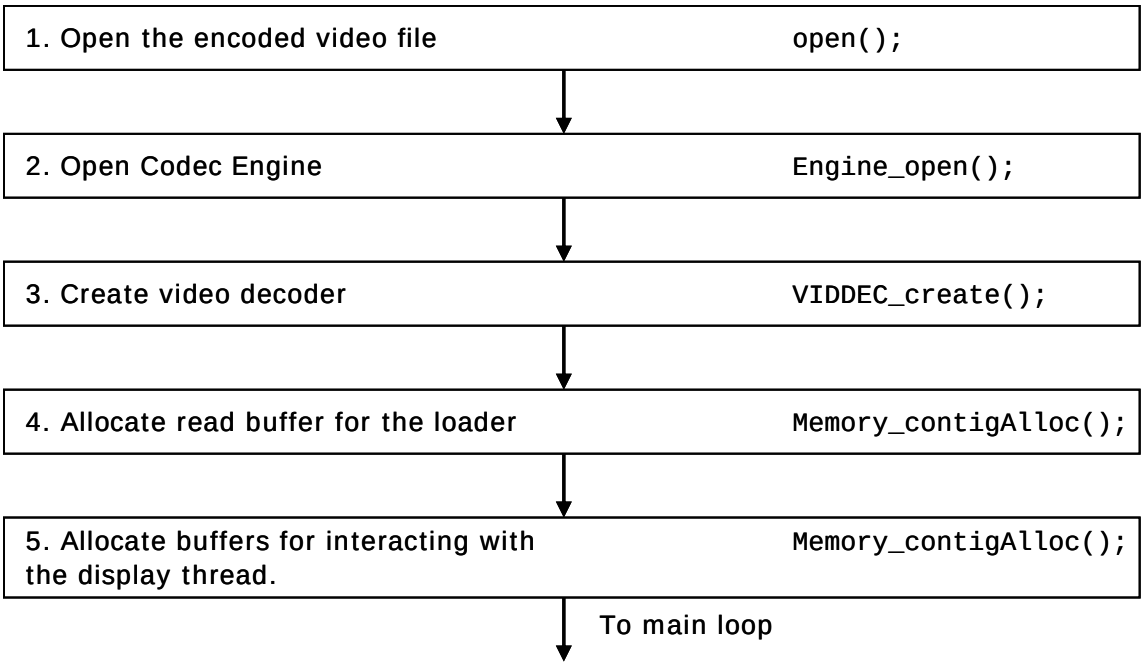

Figure 8. Video Thread Initialization Flow

As Figure 8 shows, the video thread initialization is performed as follows:

1. The video file containing the encoded video clip is opened for reading.
2. A Codec Engine instance is created with `Engine_open()`. This returns a handle to use when instantiating algorithm instances for this engine. All threads using the same engine need a separate handle; access to the engine through this handle is not thread safe.

3. The video decoder is created by `videoDecodeAlgCreate()`. Currently the decode demo supports decoding video using the MPEG2, H.264, or MPEG4 algorithm. A codec instance is created using the static parameters in the `VIDDEC_create()` call. The dynamic video decoder parameters are then set using the `VIDDEC_control()` call with the `XDM_SETPARAMS` command. The video decoder is asked what it's worst case encoded buffer size is using the `VIDDEC_control()` call with the `XDM_GETBUFINFO` command. We will use this value in the loader; see Section 2.6 for details.

4. A contiguous buffer for the encoded data of the size returned by `XDM_GETBUFINFO` above is allocated using `Memory_contigAlloc()`.

5. A number of contiguous display buffers (equal to `DISPLAY_BUFFERS`) are allocated using `Memory_contigAlloc()`. These will be used to exchange buffers with the display thread as described above.

When the video thread is finished initializing, it synchronizes with the other threads using the Rendezvous utility module. Because of this, only after the other threads are finished initializing is the main loop of the video thread executed.

2.5.1 Display Thread

In order to decouple the processing from the displaying of the video frames, a separate display thread is responsible for copying the decoded video buffer into the frame buffer of the FBDev display device driver. (See Section 4 for links to documents with more details on the FBDev interface.) This lets the decoded buffer be copied in parallel with the DSP processing. The thread execution begins by initializing the FBDev display device driver in `initDisplayDevice()`. In this function the display resolution (D1) and bits per pixel (16) are set using the `FBIOPUT_VSCREENINFO` ioctl, before the three (triple buffered display) buffers are made available to the user space process from the Linux device driver using the `mmap()` call. The buffers are initialized to black, since the video resolution might not be full D1 resolution and the background of a smaller frame should be black. Next a Rszcopy job is created. The Rszcopy module uses the VPSS resizer module on the DM6446 to copy an image from source to destination without consuming CPU cycles.

When the display thread is finished initializing, it synchronizes with the other threads using the Rendezvous utility module. Because of this, only after the other threads are finished initializing is the main loop of the display thread executed.

2.5.2 Video Thread Interaction

Figure 9 shows the interaction of the video and display thread main loops (after the threads have been released by the Rendezvous object) while processing a video frame. The descriptions of the interactions in the figure are from the point of view of the video thread.

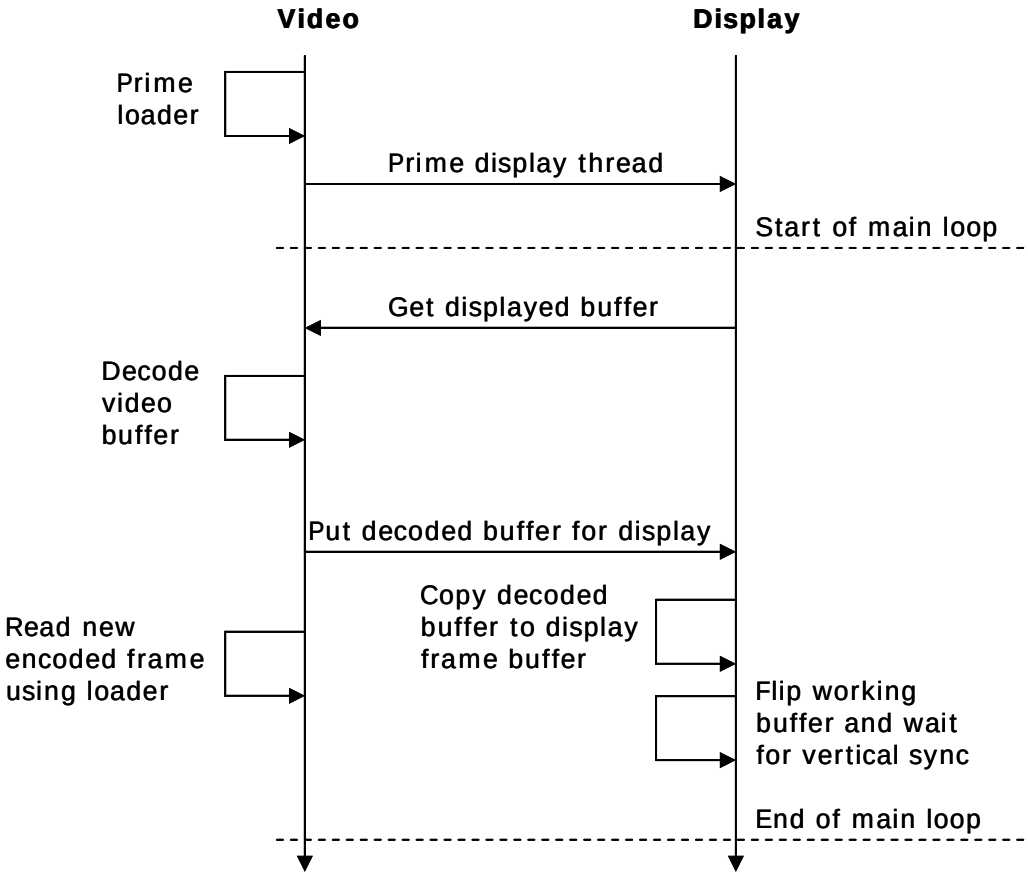

Figure 9. Video Thread Interactions

First the loader is primed (initialized) with data using the `loaderPrime()` function (see Section 2.6 for details), and then the video thread primes the display thread with the display buffers it allocated earlier using `FifoUtil_put()`. After this all `DISPLAY_BUFFERS` number of contiguous display buffers are in the display thread ready to be returned to the video thread when the main loop starts executing.

The video thread main loop starts by asking the display thread for a display buffer using `FifoUtil_get()`. Then the encoded frame obtained by the priming of the loader is decoded on the DSP core into the display buffer using `VIDDEC_process()`. Upon completion, a pointer to the display buffer, now containing a decoded video frame, is sent to the display thread using `FifoUtil_put()`. Now a new encoded buffer is read from the Linux file system using `loaderGetFrame()` before starting the main loop over again.

Meanwhile, the display thread has been waiting for a buffer from the video thread using `FifoUtil_get()`. When a buffer is received, it is copied to the FBDev display device driver's frame buffer using `Rszcopy_execute()`.This uses the VPSS resizer module on the DM6446 to copy the frame to the FBDev frame buffer. After the copy is complete, the display buffer is returned to the video thread using `FifoUtil_put()`. Finally the newly filled FBDev frame buffer is made available for display on the next vertical sync using the `FBIOPAN_DISPLAY` ioctl before waiting on the next vertical sync using the `FBIO_WAITFORVSYNC` ioctl.

2.6 Encoded Data File Loader

The file loader (loader.c) loads data from encoded audio, speech, and video elementary stream files in an efficient manner for the decoders. Note that the loader is only a couple of functions (interface described in loader.h) that are executed in the context of the thread calling it (speech, audio, or video thread), and not a thread of its own. The problems the loader tries to solve are:

1. Because the application is working on elementary streams, there is no way of knowing ahead of time how big an encoded frame is without processing the frame.
2. The decoders expect at least a full frame every time the VISA process function (speech, audio, or video) is called.
3. Since the decoder algorithm determines how big the encoded frame really was, we don't know how big the last frame was until the VISA process call completes. This makes parallelizing the I/O and the DSP processing difficult, and makes it important that as little as possible is read from the file each time.

Note: It is possible to parallelize the DSP processing and the I/O by keeping big I/O buffers constantly filled using a separate I/O thread while just doing the stream positioning in the video, audio, or speech thread. This was not deemed necessary, since none of the decode algorithms stresses the system to a point where the chosen solution becomes a problem.

The picture below describes the file loader algorithm. The variable names in the picture are either local to the loader functions or part of the `LoaderState` structure.

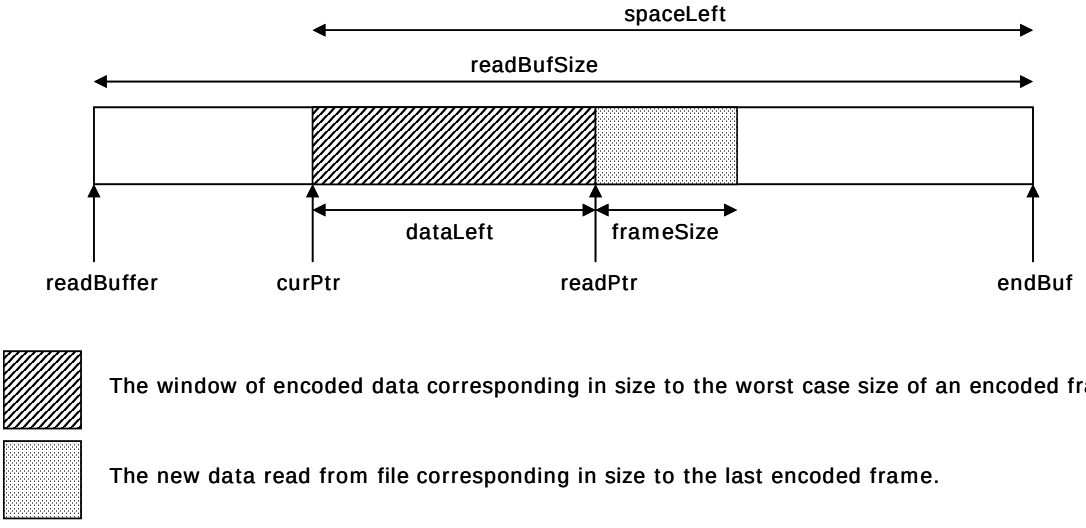

Figure 10. File Loader Algorithm

The loader algorithm works by always making sure a window of encoded data is available to the decode algorithm. In the video case the window size is determined by asking the decoder what the worst case size of an encoded frame is using the XDM_GETBUFINFO control call. For speech and audio an adequate size was chosen for this window, since the size of speech and audio frames are significantly smaller than those of video frames, and don't need to be optimized in the same way.

Every time the VISA process function is called, the file loader makes sure that at least one encoded frame is passed to the algorithm (until there is no more data). In the picture above, this means that the window will slide from left to right during the decoding. Every time the loaderGetFrame() function is called with a frameSize parameter stating how big the last encoded frame really was, the window will move right by frameSize bytes using an incremental read() call and return a new frame pointer pointing to a full window of data that is at least one frame of encoded data.

Eventually, when spaceLeft is smaller than the window size, the loader will have to restart from the beginning of readBuffer. The remainder of the encoded data is copied to the beginning of readBuffer using memcpy() before doing the usual incremental read() call.

The loader maintains a lot of state information but still needs to be reentrant, since it is called by the video, audio, and speech threads. A LoaderState object (defined in loader.h) keeps track of the state:

```
typedef struct LoaderState {
    int inputFd;            // INPUT - The file descriptor of the input file
    int loop;               // INPUT - true if clip is to start over
    char *readBuffer;       // INPUT - A large CMEM allocated buffer
    int readBufSize;        // INPUT - Total size of the readBuffer
    int readSize;           // INPUT - Size of the 'window'
    int doneMask;           // INPUT - VIDEO_DONE or SOUND_DONE
    int firstFrame;         // OUTPUT - True if first frame of a clip
    int endClip;            // OUTPUT - True if time to start clip over
    char *curPtr;           // INTERNAL - Points to current frame
    char *readPtr;          // INTERNAL - Points to the end of current 'window'
} LoaderState;
```

The members listed as INPUT needs to be set before using the loader while the OUTPUT variables are set by the loader for reading after each loaderGetFrame() call. Do not set the variables marked INTERNAL.

The doneMask entry keeps track of which media types have finished playing (video or sound, where sound means speech or audio). This is used to make sure that both media types restarts simultaneously if the loop (-l) command-line parameter has been given to the application.

Before loaderGetFrame() can be called, the loaderPrime() function has to be called (for each LoaderState object). The loaderPrime() function fills the readBuffer and initializes the internal state variables.

The readBuffer should be at least 2*readSize–1 bytes to allow for the worst case memcpy() (while making no assumptions about the implementation of memcpy()). The bigger the readBuffer is, the fewer times the memcpy() has to be performed, but more memory is used.

3 Adapting the Application

The subsections that follow discuss how to adapt the application if you want to remove the control thread and just do video, speech or audio. This creates a single-threaded application that is concerned with only one media type and has no user interface on the OSD. Section 3.5 shows how to replace the decode algorithms with the Codec Engine copy codec example algorithms, a process that applies to more complex algorithms as well.

3.1 Speech Only

This subsection describes how to adapt the application to have just a speech thread. In main.c, remove the speech and video thread creation in `main()` and make sure the speech thread is *called* as opposed to *created* as follows:

```
    /* Become the speech thread if a file name is supplied */
    if (args.speechFile) {
        speechEnv.hRendezvous   = &rendezvous;
        speechEnv.speechFile    = args.speechFile;
        speechEnv.speechEncoder = args.speechEncoder;
        speechEnv.soundInput    = args.soundInput;

        ret = speechThrFxn(&speechEnv);

        if (ret == THREAD_FAILURE) {
            status = EXIT_FAILURE;
        }
    }

cleanup:
```

Make sure `ctrlThrFxn()` is no longer called from `main()` and that `numThreads` is set to 1 when `Rendezvous_open()` is called (since there is only 1 thread to “synchronize” now).

Since we removed the control thread, the speech processing loop never exits. That is, `gblGetQuit()` never returns `TRUE`. You can use Ctrl+C to exit the application now, since the resulting `SIGINT` signal will not be caught by the application and it will close as a result. This is not a clean way to exit an application, since all resources might not be freed up correctly. Section 3.3 describes how to exit the application in a cleaner fashion.

3.2 Audio Only

Creating an audio only application is similar to the speech version described in Section 3.1, but you should call `audioThrFxn()` instead of `audioThrFxn()`.

3.3 Video Only

Creating a video only application is similar to the speech version described in Section 3.1, but you should spawn the display thread as before, but call `videoThrFxn()`instead of `speechThrFxn()`, and you should set `numThreads` to 2 when `Rendezvous_open()` is called, because the video and display threads need synchronization in the video case.

3.4 Exit Cleanly Without Control Thread

One way to exit cleanly without using the control thread is to catch the SIGINT signal generated when the user presses Ctrl+C. This is done by putting the following signal handler somewhere above the speech or video thread function:

```
#include <signal.h>

void quit(int signal)
{
    gblSetQuit();
}
```

This signal handler needs to be registered with the Linux OS. Do this by putting the following line just before the speech or video processing loop:

```
signal(SIGINT, quit);
```

Now the application exits cleanly when a user presses Ctrl+C.

3.5 Replacing the Decode Algorithms with Other Codecs

This section shows how to replace the decoders used by the decode demo (h.264, mpeg2 or mpeg4 for video, aac or mpeg1l2 for audio and g.711 for speech). This example shows how to replace these algorithms with the example copy codecs shipped as examples with the Codec Engine. These copy codecs essentially do a copy of the data and no real processing, but could just as well have been real algorithms. From an application point of view, all codecs of a VISA class are essentially treated the same no matter the complexity of the algorithm.

First the decode.cfg file needs to be edited. This file contains the configuration of the Codec Engine for the decode demo. First the copy codec packages needs to be pulled in and made available using the following statements:

```
var SPHDEC_COPY  = xdc.useModule('codecs.sphdec_copy.SPHDEC_COPY');
var AUDDEC_COPY  = xdc.useModule('codecs.auddec_copy.AUDDEC_COPY');
var VIDDEC_COPY  = xdc.useModule('codecs.viddec_copy.VIDDEC_COPY');
```

The declarations of the G711DEC, AACDEC, MP3DEC, MPEG2DEC, H264DEC and MPEG4DEC variables should be removed, since these algorithms will not be used anymore. Next, we need to describe our codec server (demoEngine):

```
var demoEngine = Engine.create("decode", [
    {name: "sphdec_copy", mod: SPHDEC_COPY, local: false},
    {name: "auddec_copy", mod: AUDDEC_COPY, local: false},
    {name: "viddec_copy", mod: VIDDEC_COPY, local: false},
]);
```

Again, the lines for h264dec, mpeg2dec, aacdec, mp3dec, g711dec and mpeg4dec should be removed from this array, since these algorithms will not be used anymore.

Finally, the Codec Engine needs to be told where to find the file containing this codec server by changing the `demoEngine.server` assignment to:

```
demoEngine.server = "./all.x64P";
```

The codec server file that contains copy codecs for all 8 VISA classes (all.x64P) can be found at codec_engine_1_02/examples/servers/all_codecs, and should be copied to the directory on your target file system where your demos reside (typically /opt/dvevm).

Now the Makefile needs to be edited to add the search path to these copy algorithm packages in order for the configuration tool to find them. Find the line where the XDC_PATH variable is set and append the following to the list of package search paths:

```
$(CE_INSTALL_DIR)/examples
```

This adds the Codec Engine examples (where the copy codecs reside) to the package search path, and the configuration tool can find the copy codec packages when the configuration step is executed.

Since the names of the codecs have changed from "h264dec", “mpeg2dec”, or "mpeg4dec" to "viddec_copy", and from “aacdec” or “mp3dec” to “audio_copy”, and from "g711dec" to "sphdec_copy", the video.c, audio.c and speech.c files of the demo need to be changed to reflect this. In video.c, find the line where the `algName` variable is assigned and make sure it reads as follows:

```
algName = "viddec_copy";
```

Note: Because the video decode copy codec doesn't support the `XDM_SETPARAMS` or `XDM_GETBUFINFO` control calls, these calls needs to be commented out. Instead of dynamically checking the codecs worst size encoded buffer size, manually assign `*readSizePtr` to:

```
*readSizePtr = D1_FRAME_SIZE;
```

In audio.c, find the line where `algName` is assigned and change it to:

```
algName = “auddec_copy”;
```

Note: Since the audio decode copy codec doesn’t support the `XDM_SETPARAMS`, `XDM_RESET`, or `XDM_SETDEFAULT` control calls, these calls needs to be commented out.

In speech.c, find the line where the `SPHDEC_create()` Codec Engine call is made, and modify this line so it reads as follows:

```
hDecode = SPHDEC_create(hEngine, "sphdec_copy", &params);
```

Now recompile the decode demo using "make" and install it to the target file system using "make install" before running this altered decode demo.

Your altered decode demo will play back raw audio and video, since the copy codecs merely do a copy and no decompression. If you need raw clips to play back, you can use the results of the corresponding encode demo copy codec adaptation from the encode demo documentation. Note that the decode demo now “processes” raw data that may overload your file system I/O for video if you use a large image resolution, since the bit rate for uncompressed video is very high.

4 More Information

For more information, see the following documentation:

- *Encode Demo for the DVEVM/DVSDK 1.2* (SPRAA96A)
- *EncodeDecode Demo for the DVEVM/DVSDK 1.2* (SPRAAH0A)
- Decode Demo readme file. $(DVEVM_INSTALL_DIR)\demos\decode\decode.txt. Contains information on how to invoke the demo from the command line.
- Encode Demo readme file. $(DVEVM_INSTALL_DIR)\demos\encode\encode.txt.
- EncodeDecode Demo readme file. $(DVEVM_INSTALL_DIR)\demos\encodedecode\encodedecode.txt.

DVEVM Product

- *DVEVM Getting Started Guide* (SPRUE66). Hardware and software overview, including how to run demos, install software, and build the demos.
- *DaVinci System Level Benchmarking Measurements* (SPRAAF6)

Codec Engine

- *Codec Engine Application Developer's Guide* (SPRUE67A)
- Codec Engine API Reference $(DVEVM_INSTALL_DIR)\codec_engine_1_02\docs\html\index.html

Codec Servers

- Codec Servers Data Sheets: Encode, Decode, and Loopback (Encode/Decode) $(DVEVM_INSTALL_DIR)\codec_servers_1_00\docs\data_sheets

Linux Device Drivers

- *Linux Device Drivers 3rd Edition*, J. Corbet & A. Rubini [ISBN 0-596-00590-3].
- Open Sound System (OSS) website. http://www.opensound.com
- Video for Linux 2 (v4l2) website. http://www.thedirks.org/v4l2
- FBdev website. http://linux-fbdev.sourceforge.net

POSIX Threads

- *Programming with POSIX Threads*, David R. Butenhof [ISBN 0201633922].

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