Encode 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 “encode” demo application. The encode demo uses the Codec Engine as well as video and speech algorithms from Texas Instruments to encode video and sound data from peripheral device drivers to files on the Linux file system.

Contents

Figures

1 Overview

The encode demo shows how to encode video and/or speech using algorithms and the Codec Engine from Texas Instruments on the DaVinci DM6446 DVEVM board. The speech algorithm used is G.711. For video, the 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 (encodeCombo.x64P) that is managed by the Codec Engine. The algorithms are executed on the DaVinci DSP core. The resulting encoded elementary streams are written to separate files (one for video and one for speech) on the Linux file system.

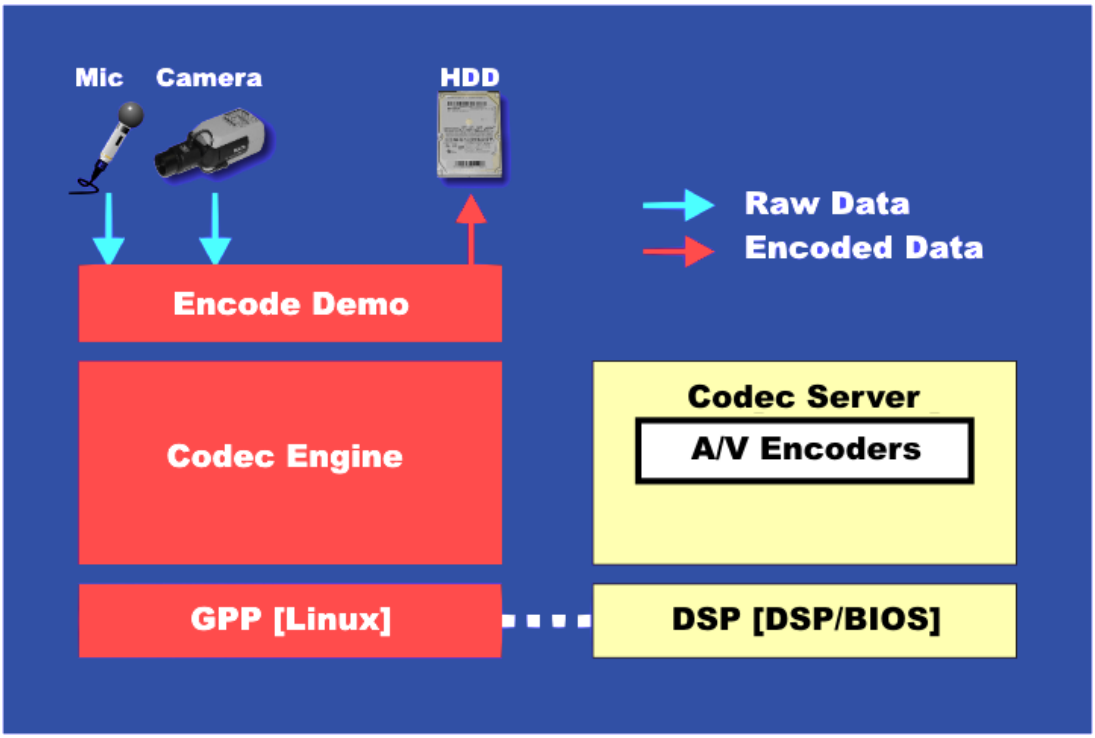

Figure 1. Encode Demo Architecture

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 encode demo, including documentation on the command-line parameters, see Section 4 for how to find the encode.txt file.

2 Application Design

The application consists of six 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 *capture thread* (capture.c), the *writer thread* (writer.c), and the *speech thread* (speech.c). The video, display, capture, writer, and speech threads are spawned from the main thread before the main thread becomes the control thread. The video, display, capture, and writer threads are only created if a video file was provided on the command line. The same goes for the speech thread. The user must supply at least one file (speech and/or video) for the demo to run. This means at least 2 and at most 6 application threads are running in the demo process.

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

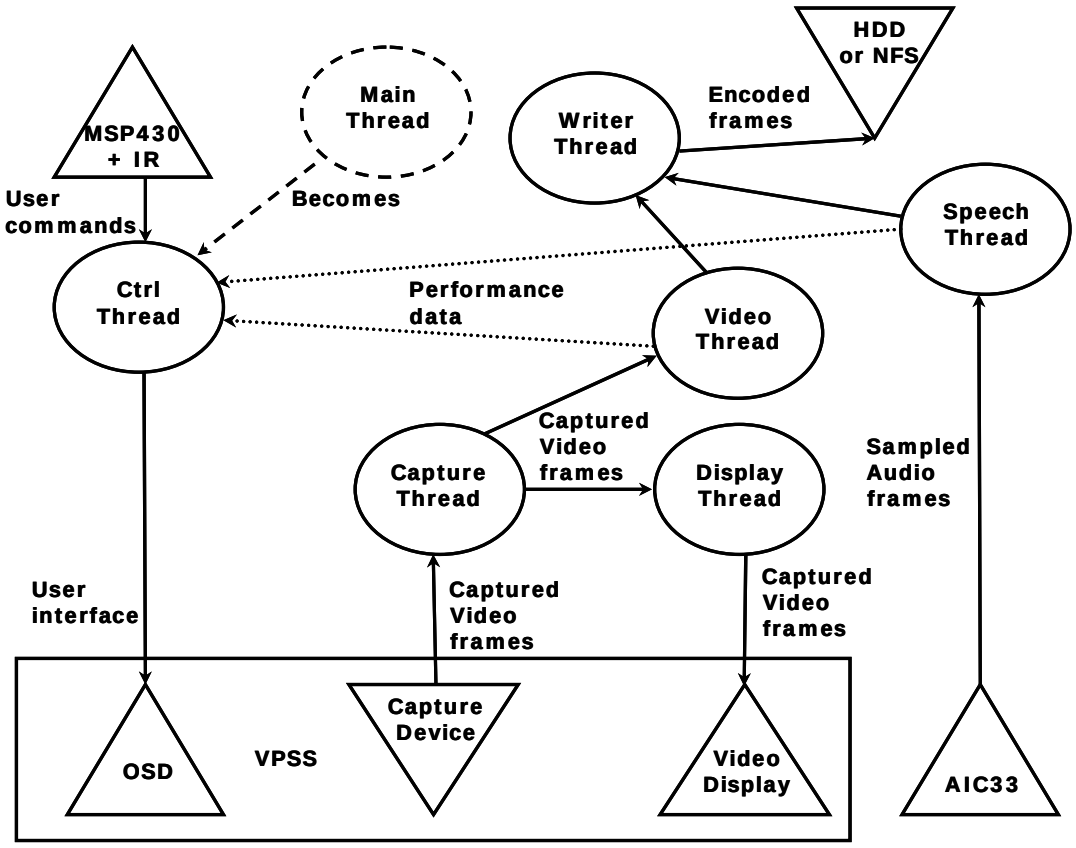

Figure 2. Encode Demo Threads

The initialization and cleanup of the threads are synchronized using the provided Rendezvous utility module. 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 that 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 values of the command-line parameters.

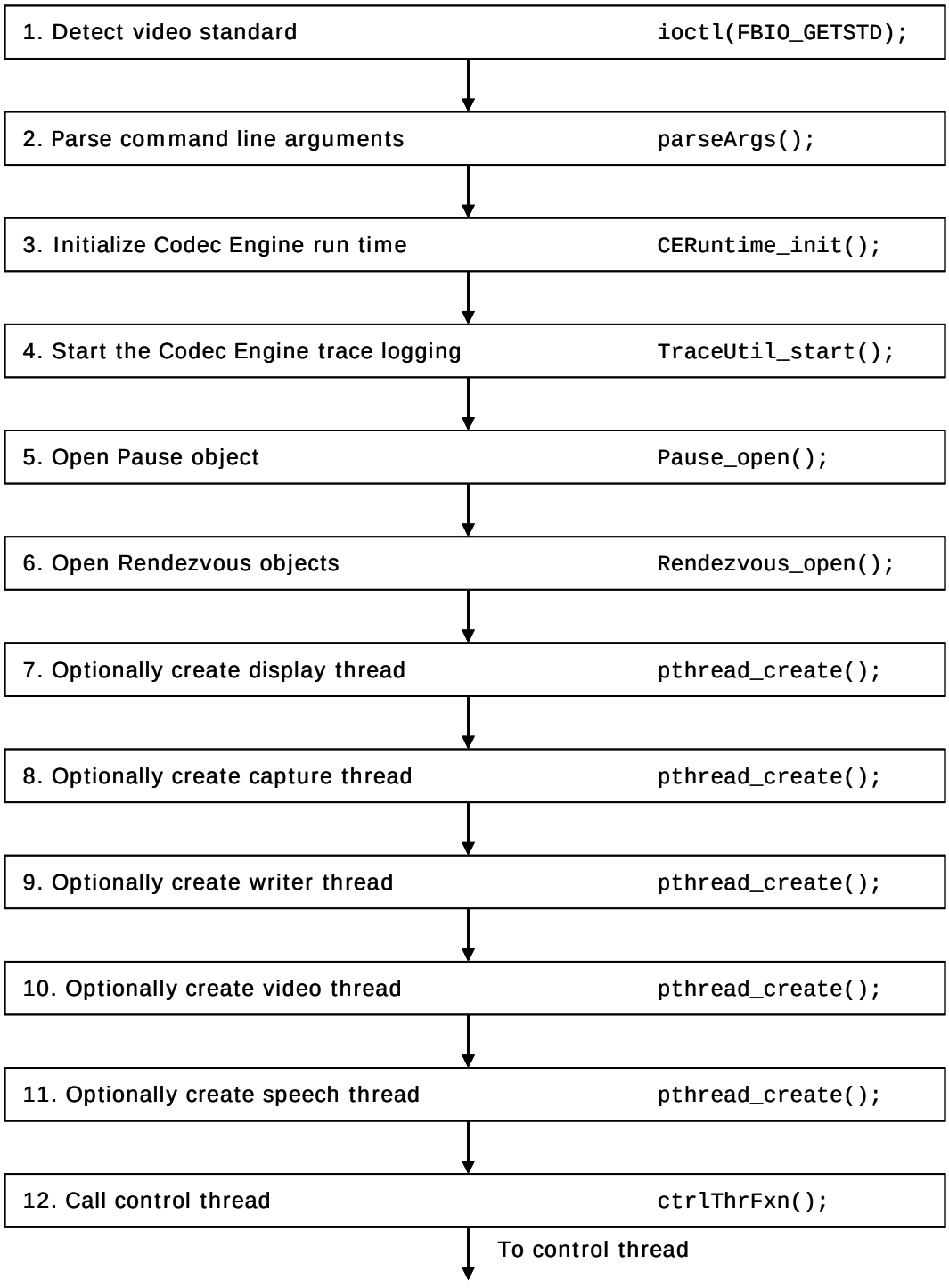

Figure 3. Main Thread Flow

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 both 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 encode.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 samples from the AIC33 device driver, encodes them using a speech encoder, and stores the encoded samples in a file on the Linux file system.

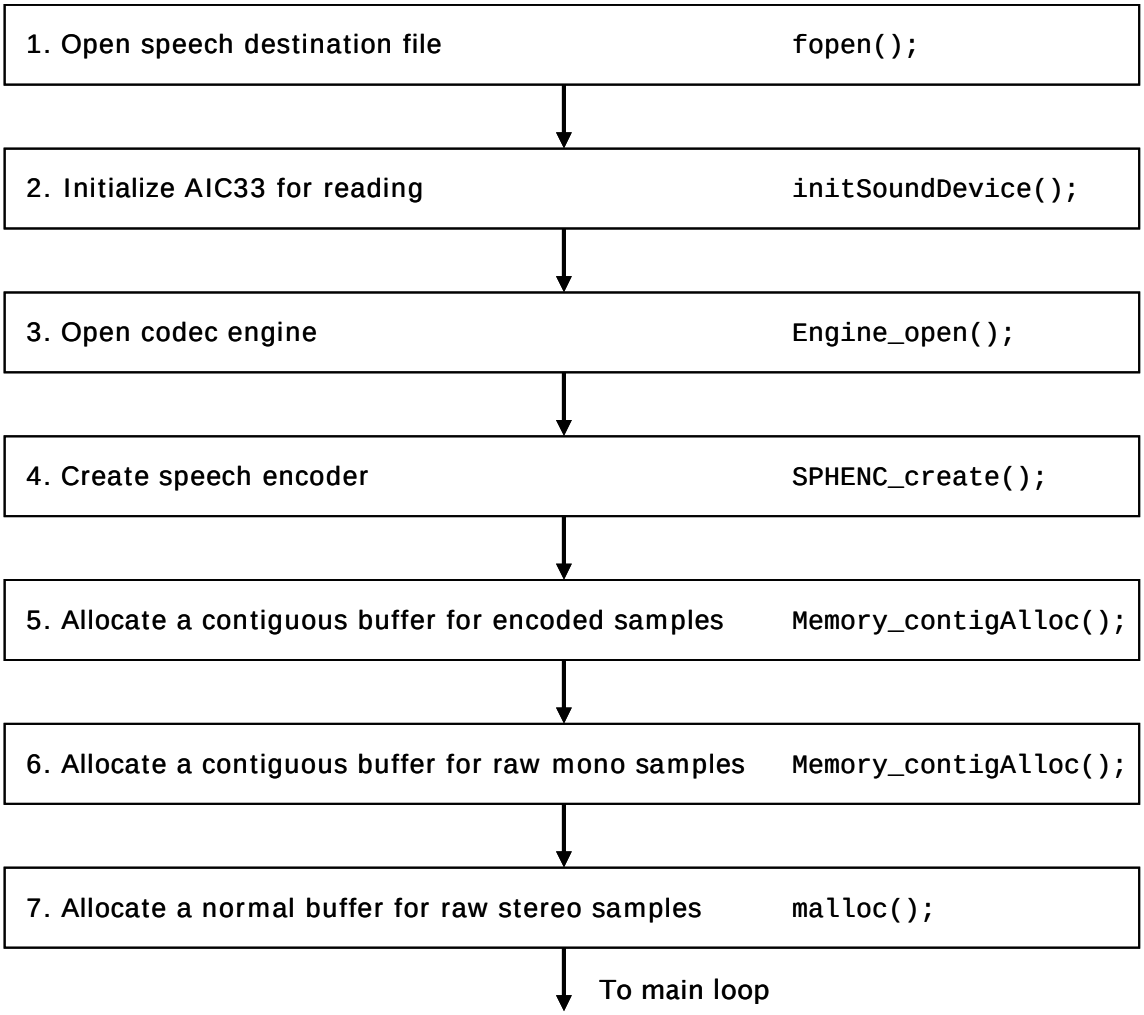

Figure 4. Speech Thread Initialization Flow

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

1. The destination speech file is opened for writing on the Linux file system.
2. The AIC33 sound device driver is initialized for reading. The device driver is an OSS device driver. See Section 4 for links to documents with more details. First the mixer (`/dev/mixer`) is configured for either line in or microphone recording depending on whether the `'-l'` command line option was passed to the application or not. Then 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 algorithms supported use an 8 KHz sample rate, the AIC33 is set to this rate.

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 encode algorithm instance is created using `SPHENC_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 (encode) data in the main loop.

5. A contiguous buffer is allocated for the encoded data using `Memory_contigAlloc()`. This is where the encoded data will be stored by the speech encode algorithm. 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 for the raw mono samples. Since it will be passed to the DSP side algorithm for processing, it too needs to be contiguous.

7. An input buffer is allocated. This is the buffer that will be given to the AIC33 device driver to be filled with stereo samples. Since this buffer will not be passed to the DSP (the stereo-to-mono conversion is done on the ARM), this buffer does not need to be contiguous, and therefore `malloc()` is sufficient as allocation mechanism for this buffer.

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:

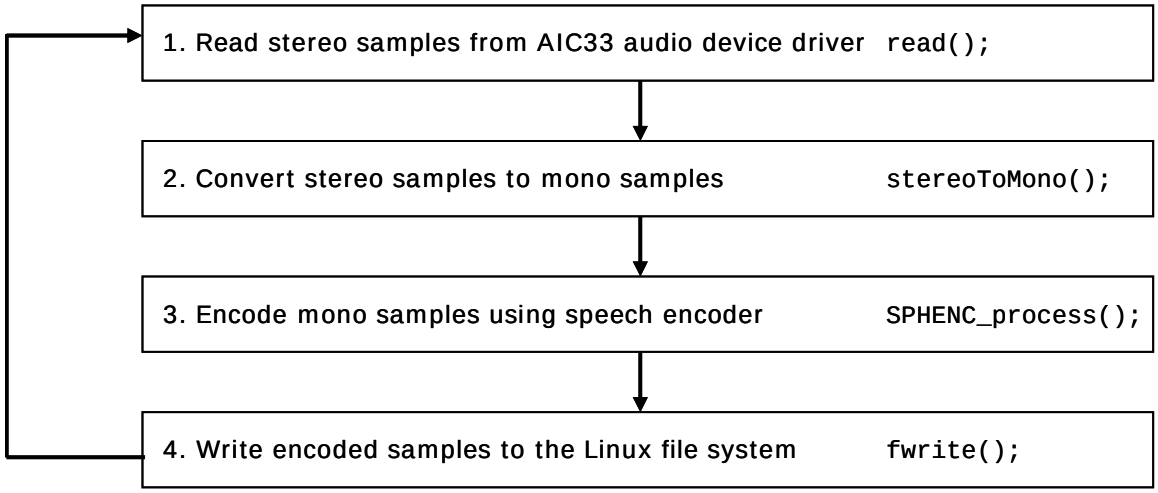

Figure 5. Speech Thread Main Loop Flow

1. Since the AIC33 sound driver supports only stereo, two channels of samples are read into the input buffer.

2. The stereo samples are converted into mono samples in the raw buffer using the `stereoToMono()` function.

3. The raw mono data is encoded using the `SPHENC_process()` call. This is a Codec Engine procedure call that encodes (ENC) 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.
4. The encoded samples are written to the file on the Linux file system using the standard I/O `fwrite()` call.

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

2.4 Video Thread

The video thread receives a buffer from the capture thread and encodes it using a video encoder algorithm. It sends the encoded frames to a separate writer thread to write the encoded frame to the Linux file system. Using dedicated I/O and capture threads maximizes the utilization of the ARM and DSP cores.

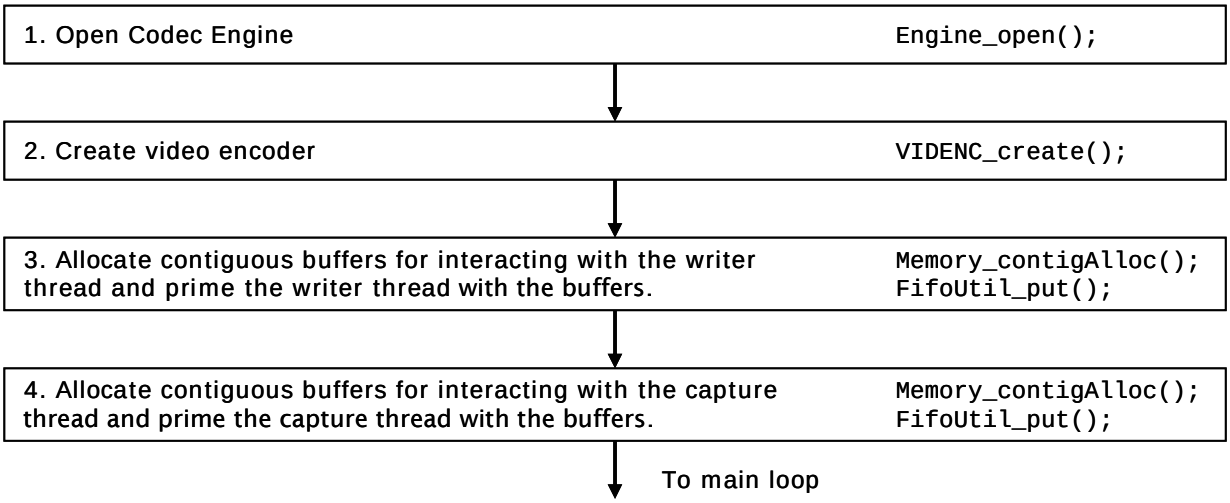

Figure 6. Video Thread Initialization Flow

As Figure 6 shows, the video thread initialization is performed as follows:

1. 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.
2. The video encoder is created by `videoEncodeAlgCreate()`. Currently the encode demo supports encoding video using the H.264 or MPEG4 algorithms. A codec instance is created using the static parameters in the `VIDENC_create()` call. The `targetFrameRate` and `refFrameRate` parameters are set depending on whether the demo is running on a PAL or NTSC system. These settings are important for the algorithm to match the target bit rate given on the command line correctly (the algorithm needs some concept of real world time). If the user supplied a negative value as bit rate on the command line, a variable bit rate is chosen using the parameter `rateControlPreset` value `IVIDEO_NONE` (as opposed to the constant bit rate setting of `IVIDEO_LOW_DELAY`). The dynamic video encoder parameters are then set using the `VIDENC_control()` call with the `XDM_SETPARAMS` command.

3. The writer thread is primed with `IO_BUFFERS` number of buffers (but these buffers are not written to the Linux file system) to allow the encoded buffers to be written while the DSP is encoding. First contiguous buffers are allocated with `Memory_contigAlloc()` and then they are sent to the writer thread using `FifoUtil_put()`. The buffer ID is set to indicate priming and to prevent the writer thread from actually writing the buffers to the Linux file system.

4. The capture thread is primed with `CAP_BUFFERS` number of buffers to be filled by the capture thread. First contiguous buffers are allocated with `Memory_contigAlloc()` and then they are sent to the capture thread using `FifoUtil_put()`. The physical addresses of the buffers are also translated, as the VPSS resizer module uses these and not virtual addresses.

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

2.4.1 Display Thread

In order to show a preview of the frames being encoded while they are being encoded; the captured raw frames from the VPSS front end need to be copied to the frame buffer of the VPSS back end. To allow the copying to be performed in parallel with the DSP processing, it is performed by a separate display thread. 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 has finished initializing, it synchronizes with the other threads using the Rendezvous utility module. Because of this, only after the other threads have finished initializing is the main loop of the display thread executed.

2.4.2 Capture Thread

The demo gives the option of removing interlacing artifacts using the VPSS resizer module of the DM6446 before encoding the data. To parallelize this artifact removal with the DSP processing (both are blocking calls), a separate capture thread takes care of the interlacing artifact removal before the captured buffer is encoded in the video thread.

First, because the Smooth module needs more vertical rows (defined by `EXTRA_ROWS`) than it produces for interpolation purposes, the number of rows to capture is increased. This is not possible if the resulting height is more than 480 on NTSC or 576 on PAL as these are the max heights, but the top rows are normally not visible on a full D1 display (TV).

Then the video capture device is initialized by `initCaptureDevice()`. The video capture device driver is a Video 4 Linux 2 (v4l2) device driver. (See Section 4 for documents with more details.) In this function, the user-selected input connector (composite or s-video) is set using the `VIDIOC_S_INPUT` ioctl, and the capabilities of the capture device are verified using the `VIDIOC_QUERYCAP` ioctl. Next the video standard (NTSC or PAL) is auto-detected from the capture device and verified against the display video standard selected on the Linux kernel command line. The format is set to D1 resolution, and the capture device is told to join the two interlaced fields into a frame (`V4L2_FIELD_INTERLACED`) using the ioctl `VIDIOC_S_FMT`. Then the capture device driver is told to crop the D1 formatted picture to the resolution given by the user on the command line using the `VIDIOC_S_CROP` ioctl. Next three video capture buffers are allocated inside the capture device driver using the `VIDIOC_REQBUFS` ioctl, and these buffers are mapped to the user space application process using `mmap()`. Finally the capturing of frames in the capture device driver is started using the `VIDIOC_STREAMON` ioctl.

Depending on whether the user has chosen to remove interlacing artifacts from the captured frame buffer or not, a Smooth or Rszcopy job is created. The Smooth module removes interlacing artifacts using the VPSS resizer module, while the Rszcopy job merely copies the buffer without any alterations, but using the same peripheral.

2.4.3 Writer Thread

To allow the writing of encoded video frames to the Linux file system to be done in parallel with the DSP processing, the Linux file system I/O is performed by a separate writer thread. First the destination file on the Linux file system is opened using `fopen()`. Then the Rendezvous object is notified that the writer thread's initialization is complete. Note that the speech thread, unlike the video thread, does its I/O in the speech thread itself. This is because speech has lower performance requirements than video.

2.4.4 Video Thread Interaction

Figure 7 shows one iteration of each of the threads involved in processing a video frame once they start executing their main loops, and how these threads interact.

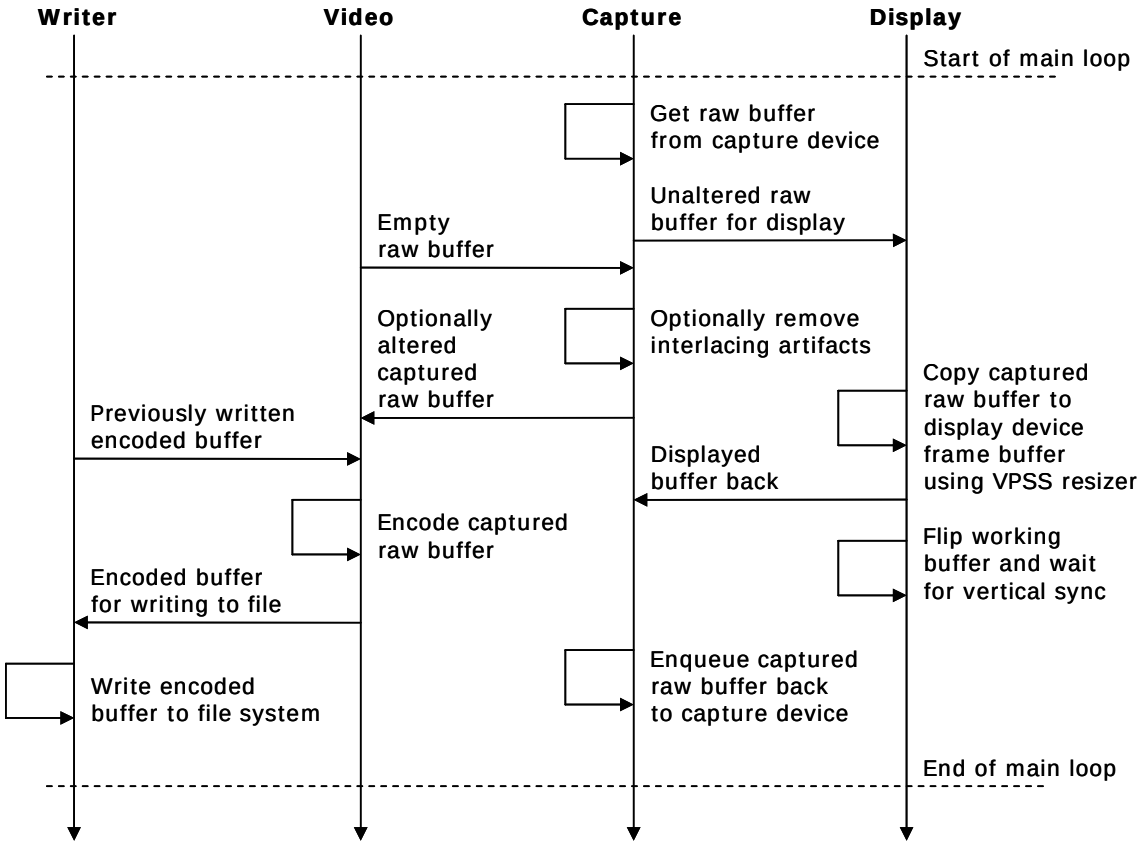

Figure 7. Video Thread Interactions

First the capture thread dequeues a raw captured buffer from the VPSS front end device driver using the `VIDIOC_DQBUF` ioctl. To show a preview of the video frame being encoded, a pointer to this captured buffer is sent to the display thread using `FifoUtil_put()`. The capture thread then fetches an empty raw buffer from the video thread. It uses either the Smooth module to fill it with an altered version where some interlacing artifacts have been removed or the Rszcopy module to copy an unaltered version to the raw buffer. This buffer is then sent to the video thread for encoding.

The video thread receives this captured buffer and then fetches an I/O buffer using `FifoUtil_get()` from the writer thread. The encoded video data will be put in this I/O buffer.

While the display thread copies the captured raw buffer to the FBDev display frame buffer using the VPSS resizer module and the `Rszcopy_execute()` call, the video thread is encoding the same captured buffer into the fetched I/O buffer on the DSP using the `VIDENC_process()` call. Note that the encoder algorithm on the DSP core and the VPSS resizer module might access the captured buffer simultaneously, but only for reading. When the display thread has finished copying the buffer, it makes the new frame buffer to which we just copied our captured frame the

new display buffer on the next vertical sync using the FBIOPAN_DISPLAY ioctl before the thread waits on the next vertical sync using the FBIO_WAITFORVSYNC ioctl.

When the video encoder running on the DSP core is finished encoding the captured buffer into the I/O buffer, the I/O buffer is sent to the writer thread using FifoUtil_put(), where it is written to the Linux file system using the standard I/O call fwrite().The capture raw buffer is sent back to the capture thread to be refilled.

The captured buffer pointer is “collected” in the capture thread from the display thread as a “handshake” that indicates that the display copying of this buffer is finished using FifoUtil_get(), before the captured buffer is reenqueued at the VPSS front end device driver using the VIDIOC_QBUF ioctl.

The writer thread writes the encoded frame to the Linux file system while the capture thread is waiting for the next dequeued buffer from the VPSS front end device driver to be ready. If the writing of the encoded buffer is not complete when the next dequeued buffer is ready and the capture thread is unblocked, there is no wait provided IO_BUFFERS is larger than 1 since another buffer will be available on the FIFO at this time. The encode demo application has IO_BUFFERS set to 2.

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 either video or speech. This creates a single-threaded application that is concerned with only one media type and has no user interface on the OSD. Section 3.4 shows how to replace the encode 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 Video Only

Creating a video only application is very similar to the speech version in Section 3.1, but you should call `videoThrFxn()`instead of `speechThrFxn()`, and you should set `numThreads` to 4 when `Rendezvous_open()` is called because the video, capture, display, and writer threads need synchronization in the video case.

3.3 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.

3.4 Replacing the Encode Algorithms with Other Codecs

This section shows how to replace the encoders used by the encode demo (h.264 or mpeg4 for video 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 encode.cfg file needs to be edited. This file contains the configuration of the Codec Engine for the encode demo. First the copy codec packages needs to be pulled in and made available using the following statements:

```
var SPHENC_COPY = xdc.useModule('codecs.sphenc_copy.SPHENC_COPY');
var VIDENC_COPY = xdc.useModule('codecs.videnc_copy.VIDENC_COPY');
```

The declarations of the G711ENC, H264ENC and MPEG4ENC 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("encode", [
    {name: "sphenc_copy", mod: SPHENC_COPY, local: false},
    {name: "videnc_copy", mod: VIDENC_COPY, local: false},
]);
```

Again, the lines for h264enc, g711enc and mpeg4enc 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 "h264enc" or "mpeg4enc" to "videnc_copy" and from "g711enc" to "sphenc_copy", the video.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 = "videnc_copy";
```

Note: Because the video encode copy codec doesn't support the `XDM_SETPARAMS` control call, this call needs to be commented out.

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

```
hEncode = SPHENC_create(hEngine, "sphenc_copy", &params);
```

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

Your altered encode demo will capture and “process” raw audio and video, since the copy codecs merely do a copy and no compression. Note that the encode demo now captures 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:

- *Decode Demo for the DVEVM/DVSDK 1.2* (SPRAAG9A)
- *EncodeDecode Demo for the DVEVM/DVSDK 1.2* (SPRAAH0A)
- Encode Demo readme file. $(DVEVM_INSTALL_DIR)\demos\encode\encode.txt. Contains information on how to invoke the demo from the command line.
- Decode Demo readme file. $(DVEVM_INSTALL_DIR)\demos\decode\decode.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