Application Report
SPRAAH0A – April 2007

EncodeDecode 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 “EncodeDecode” demo application. The EncodeDecode demo uses the Codec Engine as well as video algorithms from Texas Instruments to first encode video data captured using the VPSS front end, and then decode this video data to be displayed on the VPSS back end peripheral of the DM6446.

Contents

1 Overview 2
2 Application Design 3
2.1 Main Thread 4
2.2 Control Thread 5
2.3 Video Thread 6
2.3.1 Display Thread 7
2.3.2 Capture Thread 8
2.3.3 Video Thread Interaction 9
3 Replacing the Encode and Decode Algorithms with Other Codecs 10
4 More Information 12

Figures

Figure 1. EncodeDecode Demo Architecture 2
Figure 2. EncodeDecode Demo Threads 3
Figure 3. Main Thread Flow 4
Figure 4. Video Thread Initialization Flow 6
Figure 5. Video Thread Interactions 9

1 Overview

The encodedecode demo shows how to encode and decode video using algorithms and the Codec Engine from Texas Instruments on the DaVinci DM6446 DVEVM board. The video algorithm used is H.264 and is implemented using the xDM interface (see Section 4 for information reference). The H.264 encoder and decoder algorithms are packaged in a Codec Server (loopbackCombo.x64P) managed by the Codec Engine and executed on the DaVinci DSP core.

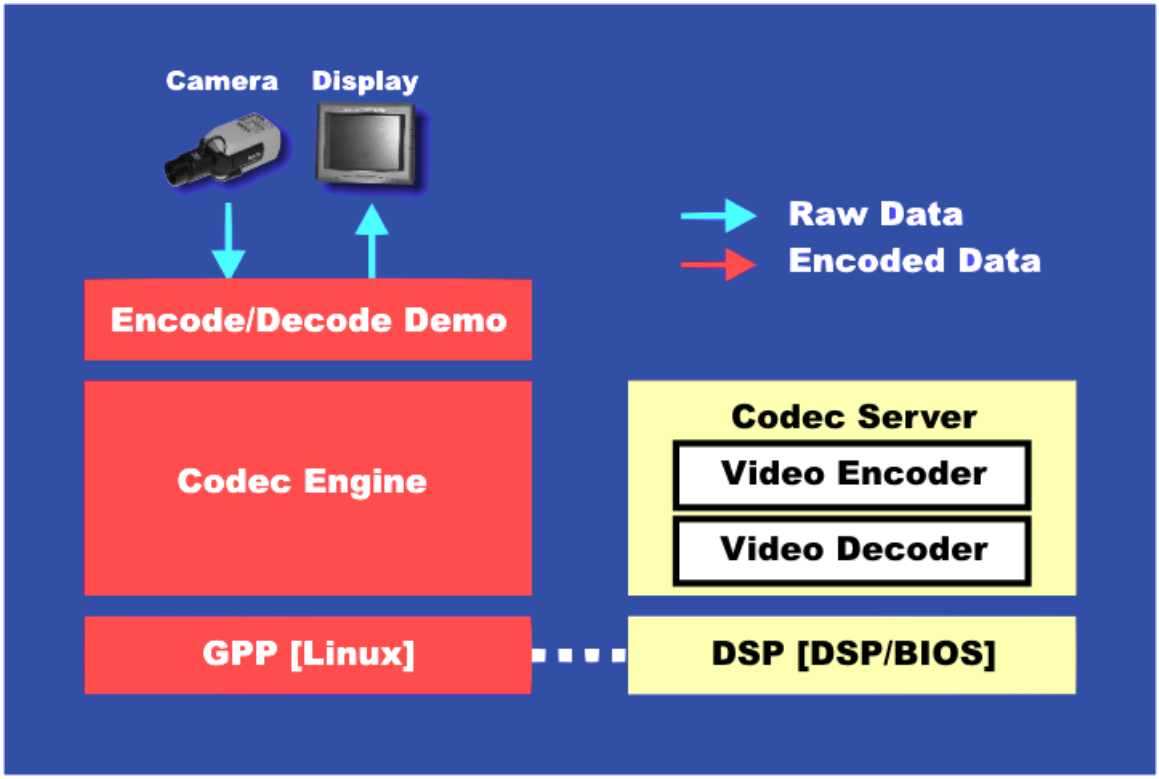

Figure 1. EncodeDecode Demo Architecture

The DaVinci ARM core runs the demos on the Linux operating system, and all peripherals are controlled through Linux device drivers. The ARM 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 encodedecode demo including documentation on the command-line parameters, see Section 4 on how to find the encodedecode.txt file.

2 Application Design

The application consists of four separate POSIX threads (pthreads): the *main thread* (main.c), which eventually becomes the *control thread* (ctrl.c), the *video thread* (video.c), the *capture thread*, and the *display thread* (display.c). The video, display, and capture threads are spawned from the main thread before the main thread becomes the control thread. This means that 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 video, capture, and display threads share the highest priority, while the control thread has the lowest priority. For more on POSIX threads see Section 4.

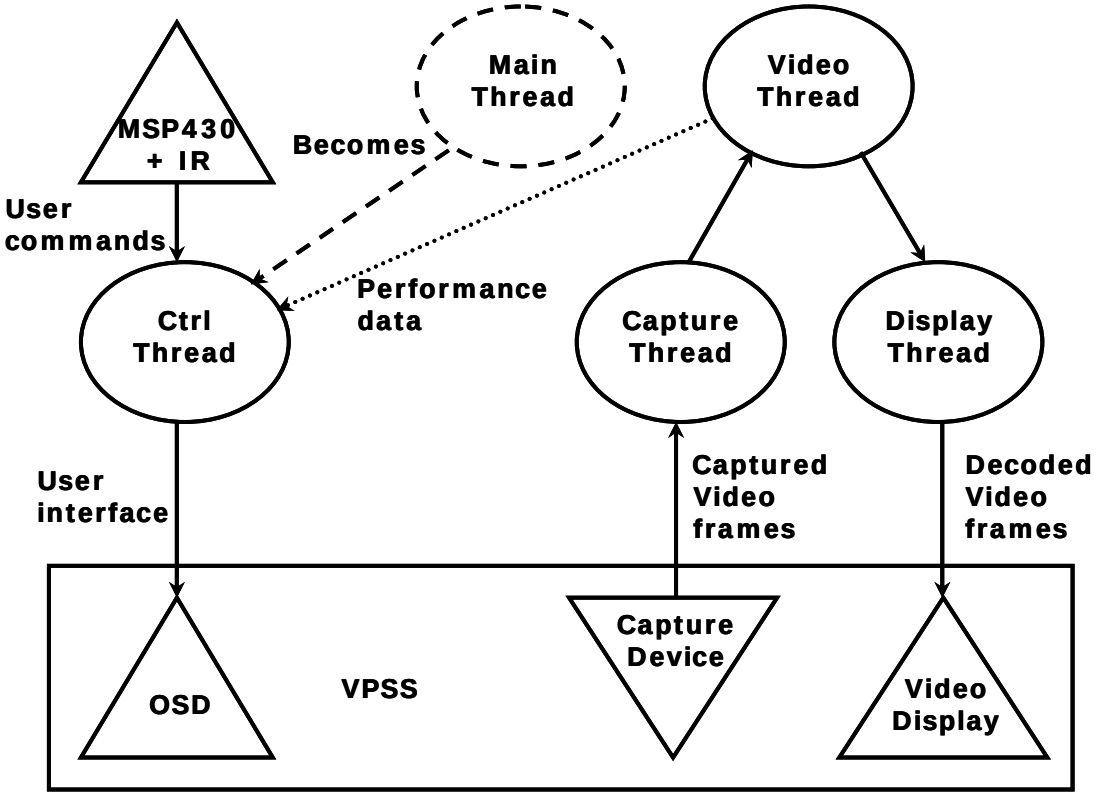

Figure 2. EncodeDecode Demo Threads

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 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 these command-line parameters.

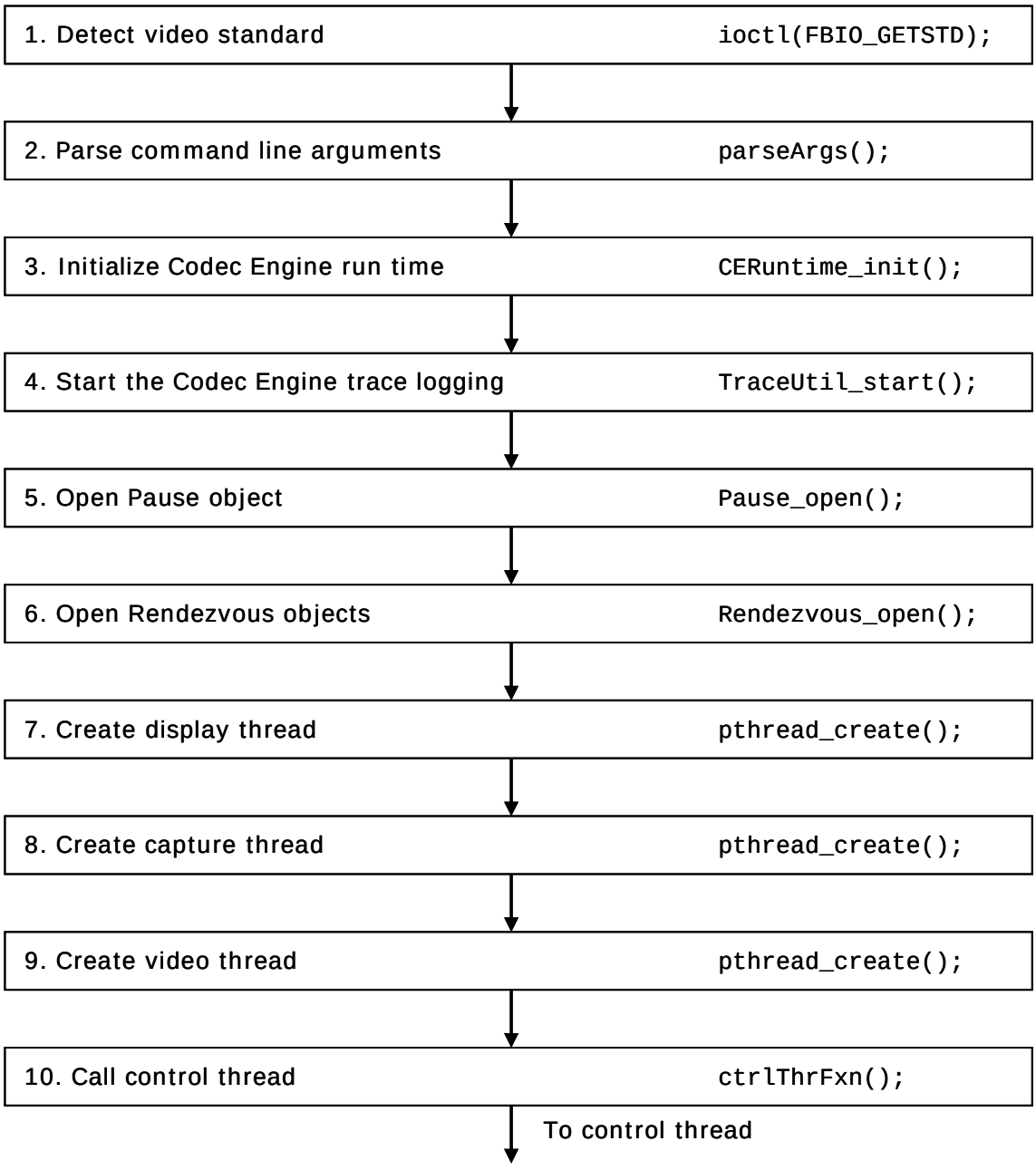

Figure 3. Main Thread Flow

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 object for synchronizing thread initialization and cleanup are opened, and the display, capture, and video threads are created. Finally, 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 encodedecode.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 Video Thread

The video thread receives frame buffers from the capture thread and encodes them using a video encoder algorithm. It then decodes the video buffers again before sending the buffers to the display thread to be displayed on the display device (VPSS back end).

In order to get more reliable performance, and to avoid dropping frames when one or more frames are demanding to encode and decode, a separate display thread is used to display the frames. If the same thread is used for encoding, decoding and displaying the buffer, any frame exceeding its real-time budget (33 ms for NTSC and 40 ms for PAL) is dropped. By decoupling the encoding and decoding 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.

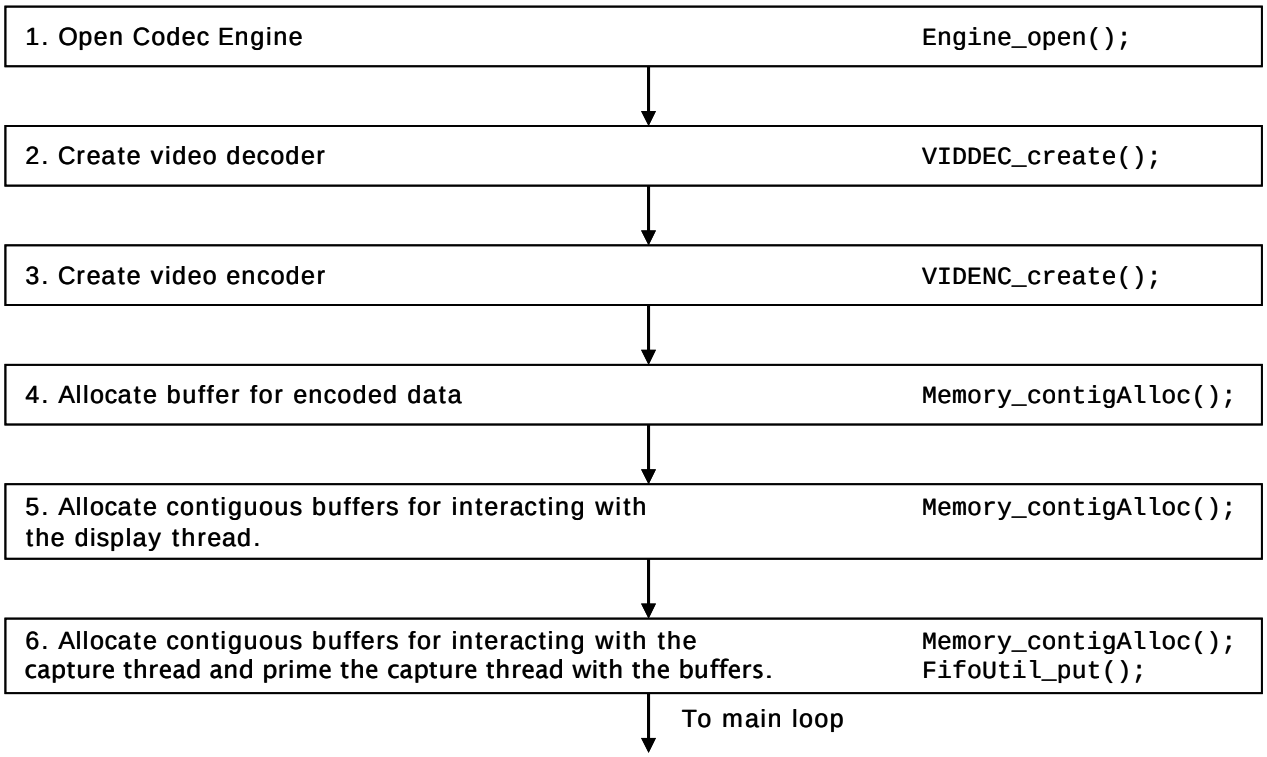

Figure 4. Video Thread Initialization Flow

As Figure 4 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 decoder is created by `videoDecodeAlgCreate()`. The encodedecode demo supports decoding video using the H.264 algorithm. A codec instance is created using the static parameters in the `VIDDEC_create()` call. **Important!** Currently due to a bug in the H.264 encoder and decoder initialization, one cannot call `VIDDEC_control()` or `VIDENC_control()` between creating the encoder and decoder. The `ALGO_INIT_WORKAROUND` pre-processor variable is set, which hardcodes the worst-case encoded buffer size. This value is normally obtained by asking the decoder what its worst-case encoded buffer size is using the `VIDDEC_control()` call with the `XDM_GETBUFINFO` command (code for this approach is shown but commented out using `#ifdef`). Obtaining this value dynamically was sacrificed to allow the setting of a bit rate for the encoder (see step 6 below) until the codec issue is resolved.

3. The video encoder is created by `videoEncodeAlgCreate()`. Currently the encode demo supports encoding video using the H.264 algorithm. 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, 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.

4. A contiguous buffer for the encoded data of the size returned by `XDM_GETBUFINFO` above (but currently hardcoded as a workaround) is allocated using `Memory_contigAlloc()`.

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

6. 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.

2.3.1 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 has 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.

2.3.2 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 and decoded 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 selected 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.

2.3.3 Video Thread Interaction

Figure 5 shows the interaction of the video, capture, and display thread main loops (after the threads have been released by the Rendezvous object) while processing a video frame.

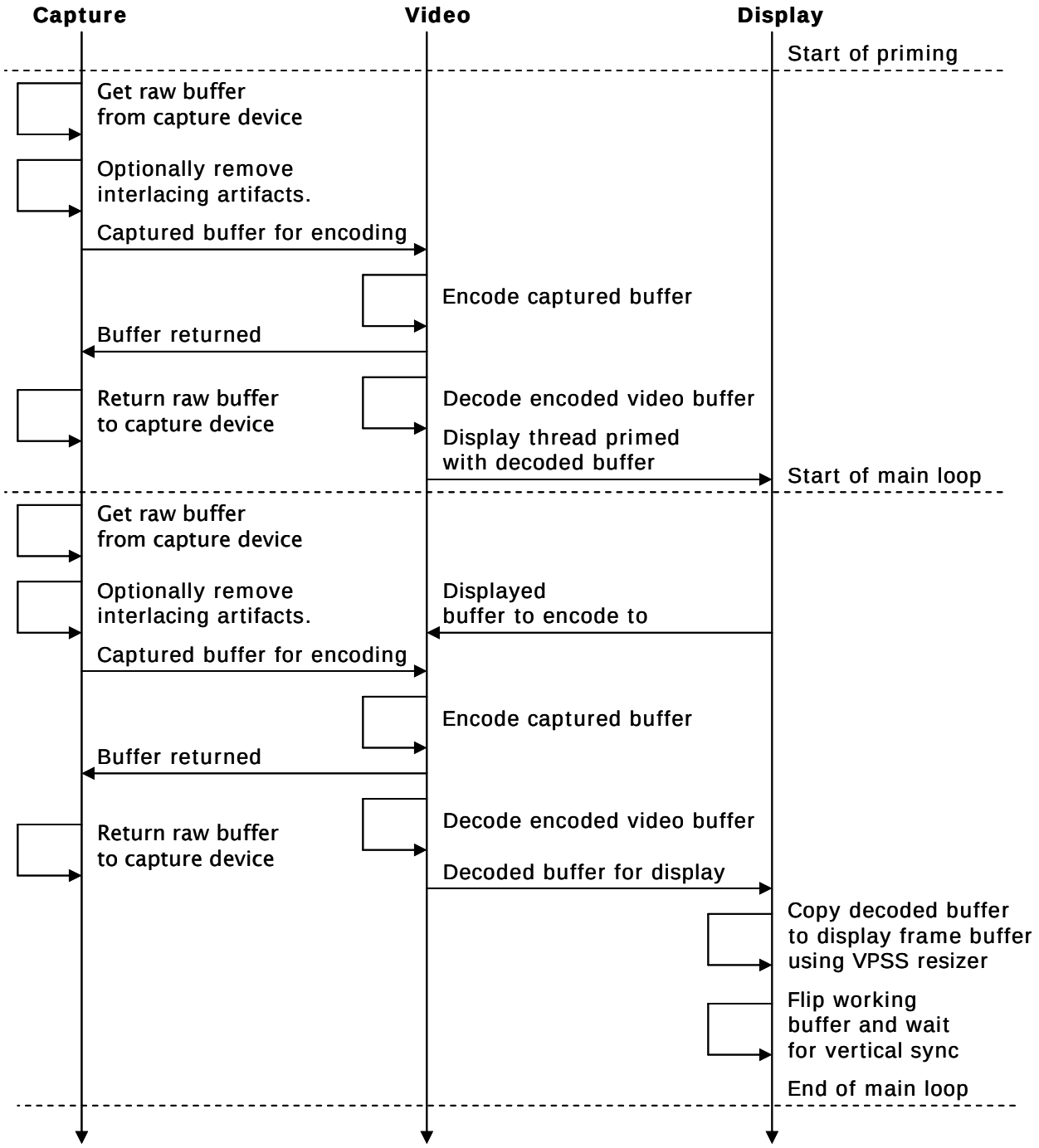

Figure 5. Video Thread Interactions

Before the main loop starts, the display thread is primed with video buffers. This ensures that all display buffers are owned by the display thread when the main loop starts. As a result, the DSP processing can be done in parallel with the copy to the display frame buffer, and the system can recover from occasional expensive frames (see Section 2.3). The frames are encoded and subsequently decoded into the display buffers and are then sent to the display thread using `FifoUtil_put()`.The priming of the display thread is synchronized using a Rendezvous object.

The capture thread main loop starts by dequeuing a captured frame buffer from the capture device using the `VIDIOC_DQBUF` ioctl, and a destination buffer is received from the video thread using `FifoUtil_get()`. The captured frame buffer is then optionally (depending on command-line parameters) processed to remove interlacing artifacts in the destination buffer before being sent to the video thread for encoding. The capture thread then returns the captured video buffer to the capture device driver using the `VIDIOC_QBUF` ioctl. Having the removal of interlacing artifacts using the VPSS resizer module on the DM6446 in a separate thread allows this operation to be done in parallel with DSP encode and decode video processing.

The video thread receives a frame buffer from the capture thread and a display buffer from the display thread using `FifoUtil_get()`. The captured frame buffer is encoded on the DSP using the `VIDENC_process()` call, and then it is returned to the capture thread using the `FifoUtil_put()` call. The newly encoded buffer is then decoded on the DSP using the `VIDDEC_process()` call, and the resulting decoded buffer is sent to the display thread using `FifoUtil_put()` to be displayed on the video display (VPSS back end).

The display thread receives the decoded raw buffer using `FifoUtil_get()` and copies it to the FBDev display device driver frame buffer using the VPSS resizer module and the `Rszcopy_execute()` call. When the display thread has finished copying the buffer, it makes the new frame buffer the new display buffer on the next vertical sync using the `FBIOPAN_DISPLAY` ioctl. It then waits on the next vertical sync using the `FBIO_WAITFORVSYNC` ioctl. Note that while the display thread is doing this, the video thread is free to encode and decode the next frame, leaving both the ARM and DSP cores fully utilized.

3 Replacing the Encode and Decode Algorithms with Other Codecs

This section shows how to replace the encoder and decoder algorithms used by the encodedecode demo (h.264). The 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 encodedecode.cfg file needs to be edited. This file contains the configuration of the Codec Engine for the encodedecode demo. First the copy codec packages needs to be pulled in and made available using the following statements:

```
var VIDDEC_COPY = xdc.useModule('codecs.viddec_copy.VIDDEC_COPY');
var VIDENC_COPY = xdc.useModule('codecs.videnc_copy.VIDENC_COPY');
```

The declarations of the H264ENC and H264DEC 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("encodedecode", [
    {name: "videnc_copy", mod: VIDENC_COPY, local: false},
    {name: "viddec_copy", mod: VIDDEC_COPY, local: false}
]);
```

Again, the lines for h264enc and h264dec 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" to "videnc_copy", and from "h264dec" to "viddec_copy", the video.c file of the demo needs to be changed to reflect this. In video.c, find the line where the `VIDENC_create()` Codec Engine call is made, and modify this line so it reads as follows:

```
hEncode = VIDENC_create(hEngine, "videnc_copy", &params);
```

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

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

```
hDecode = VIDDEC_create(hEngine, "viddec_copy", &params);
```

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

Your altered encodedecode demo will capture raw video and display raw video, since the copy codecs merely do a copy and no compression or decompression. Essentially, the image you captured should be shown on the display with little to no degradation in quality.

4 More Information

For more information, see the following documentation:

- *Encode Demo for the DVEVM/DVSDK 1.2* (SPRAA96A)
- *Decode Demo for the DVEVM/DVSDK 1.2* (SPRAAG9A)
- EncodeDecode Demo readme file.
  $(DVEVM_INSTALL_DIR)\demos\encodedecode\encodedecode.txt.
  Contains information on how to invoke the demo from the command line.
- Decode Demo readme file. $(DVEVM_INSTALL_DIR)\demos\decode\decode.txt.
- Encode Demo readme file. $(DVEVM_INSTALL_DIR)\demos\encode\encode.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