Freescale Semiconductor
Application Note

Document Number: AN4043
Rev. 0, 03/2010

Getting Started with WinCE 6.0 VPU Applications

by *Multimedia Applications Division*
Freescale Semiconductor, Inc.
Austin, TX

This application note focuses on the loopback test application, which is a video processing unit (VPU) demonstration application.

Hardware accelerated codecs are the components of the VPU. The hardware accelerated codec libraries and drivers, reside on the i.MX27 processor and Linux® and Windows® Embedded CE board support packages (BSPs), respectively.

The loopback test application captures images through a camera, encodes those images and decodes them again. Finally, the decoded images are displayed in a VGA panel.

Contents

freescale semiconductor

1 Loopback Project

Figure 1 shows the loopback solution explorer.

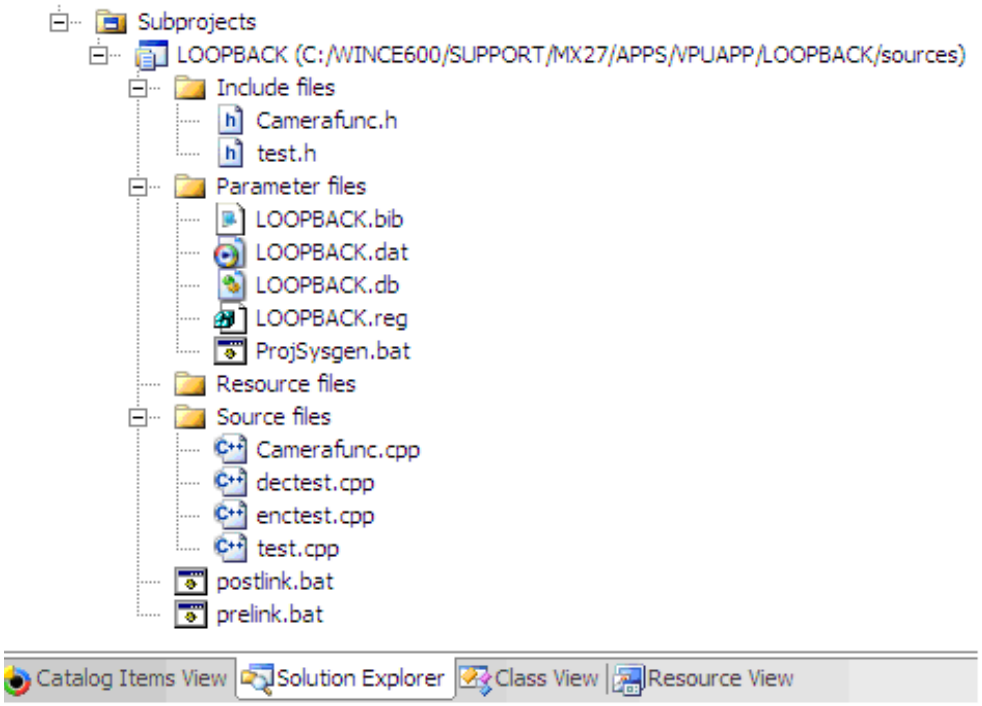

Figure 1. Loopback Solution Explorer

The loopback project is divided into the following four source files:

Test.cpp	This file is responsible for all the major functionalities of this project. It integrates the camera and the encoder and decoder files into one application.
Camerafunc.cpp	This file implements all the functionalities with respect to the camera driver, which includes loading the camera driver, initializing, starting and stopping the camera view, getting the physical address per frame, unloading the camera driver, and so on.
Dectest.cpp	This file provides the decoder implementation using the VPU API. It also demonstrates the decoding process in detail.
Enctest.cpp	This file provides the encoder implementation using the VPU API.

2 Explaining the Source Code

The following sections explain the source code.

2.1 Test.cpp File

At the beginning of the main function, `bitstreambuf` structure is initialized to zeros.

```
VPUMemAlloc bitstreambuf = {0};
```

The QueryPerformanceFrequency function retrieves the frequency of the high resolution performance counter, if it exists. The frequency remains unchanged when the system is running. The parameter is a pointer to the variable that receives the current performance counter frequency, in counts per second. If the installed hardware does not support a high resolution performance counter, then it must be set to zero.

```
QueryPerformanceFrequency(&liFre);
```

The QueryPerformanceCounter function retrieves the current value of the high resolution performance counter. The parameter is a pointer to the variable that receives the current performance counter value, in counts.

```
QueryPerformanceCounter(&litmp0);
QueryPerformanceCounter(&litmp1);
```

The main function is declared in the test.cpp file. First, the main function retrieves the VPU version using vpu_GetVersionInfo() function. Then, a switch case control sequence—assigns the product string description to productstr variable.

```
switch(ipprjnum){
case (PRJ_TRISTAN & 0x0f):
strcpy(productstr, "TRISTANeX");
break;
case (PRJ_TRISTAN_REV & 0x0f):
strcpy(productstr, "TRISTANeX-Rev");
break;
case (PRJ_PRISM_CX & 0x0f):
strcpy(productstr, "PRiSM-CX");
break;
case (PRJ_SHIVA & 0x0f):
strcpy(productstr, "Shiva");
break;
case (PRJ_PRISM_EX & 0x0f):
strcpy(productstr, "PRiSM-EX");
break;
default: break;
}
```

Later, it allocates the bitstream buffer for decoder through the vpu_AllocPhysMem function. This function physically allocates the contiguous memory. Here, the first parameter is cbSize, which is the number of bytes to allocate. The second parameter is VPUMemAlloc, which is a pointer to a physical address that stores the physical address of the memory allocation.

```
if (RETCODE_SUCCESS != vpu_AllocPhysMem(CODEC_BITSTREAM_SIZE, &bitstreambuf)) {
goto MAIN_CLEANUP;
}
```

The function returns RETCODE_SUCCESS for success, and RETCODE_FAILURE, if it fails. The program goes to MAIN_CLEANUP, if it does not allocate the number of bytes. The function then stores the virtual address and the physical address into pBitStream and BitstreamPhy variables respectively.

```
pBitStream = (UINT8 *)bitstreambuf.VirtAdd;
BitstreamPhy = bitstreambuf.PhysAdd;
```

VPUMemAlloc structure contains three member variables. The first is the physical address, the second is the virtual address, and the third one called reserved is used internally by the driver.

```
typedef struct {
```

```
        ULONG PhysAdd;
        ULONG VirtAdd;
        UINT Reserved;

    } VPUMemAlloc;
```

If pBitStream is null, then go to MAIN_CLEANUP again.

```
    if(!pBitStream){
    goto MAIN_CLEANUP;
    }
```

The function then assigns CODEC_STD to the codestd variable and selects the CODEC to be used. In this case, it is AVC (advanced video coding) encoding or decoding.

```
    codestd = CODEC_STD;
```

codestd is an enumeration variable, which is used to select the encoder or decoder.

```
    typedef enum {
    STD_MPEG4 = 0,
    STD_H263,
    STD_AVC
    } CodStd;
```

The codestd parameter is passed to the DecodeTest function. This parameter does the actual work of demonstrating how to decode using the VPU API.

```
    DecodeTest(codestd);
```

2.2 Decoding Process

The DecodeTest function demonstrates how to decode using the VPU API. The function prototype is as follows:

```
    void DecodeTest(CodStd codestand);
```

The DecodeTest procedure initializes the decParam and decBufInfo variables, which belong to the DecParam and DecBufInfo types respectively.

```
    typedef struct {
      int prescanEnable;
      int prescanMode;
      int dispReorderBuf;
      int iframeSearchEnable;
      int skipframeMode;
      int skipframeNum;
      int chunkSize;
      int picStartByteOffset;
      PhysicalAddress picStreamBufferAddr;
      } DecParam;
      typedef struct{
      DecAvcSliceBufInfo avcSliceBufInfo;
      } DecBufInfo;
```

The DecodeTest function also declares and initializes the frameBuf and VframeBuf variables with NUM_FRAM_BUF macro statically.

```
FrameBuffer frameBuf[NUM_FRAME_BUF];
PVOID VframeBuf[NUM_FRAME_BUF];
```

`FrameBuffer` type keeps pointers to the physical address of the buffer Y, Cb, and Cr.

```
typedef struct {
PhysicalAddress bufY;
PhysicalAddress bufCb;
PhysicalAddress bufCr;
} FrameBuffer;
```

After this, the `DecodeTest` function declares some miscellaneous variables which are not used inside this function. This includes `exit`, `*pPSSaveBuffer`, `*pSliceSaveBuffer`, `AllocatedPhyPtr`, `rotStride`, and `dispIdx`.

```
int i;
int frameIdx;
int exit;
Uint32 fRateInfo;
int stride;
void *pPSSaveBuffer = NULL;
void *pSliceSaveBuffer = NULL;
PhysicalAddress AllocatedPhyPtr = 0;
int totalNumofErrMbs = 0, decodefinish = 0;
int needbuffercounter;
int rotStride = 0, dispIdx = 0;
int rotAngle = 0;
int fillendBs = 0;
```

The `DecodeTest` function then initializes the `bitstreambuf`, `framebuffers`, `pssavebuffer`, and `slicesavebuffer` variables that are of `VPUMemAlloc` type.

```
VPUMemAlloc  bitstreambuf = {0};
VPUMemAlloc  framebuffers = {0};
VPUMemAlloc  pssavebuffer = {0};
VPUMemAlloc  slicesavebuffer = {0};
```

The `CreateEvent` function creates or opens a named or unnamed event object. Since the first parameter is `NULL`, the handle cannot be inherited by the child processes. Since the second parameter is `FALSE`, the function creates an auto reset event object and the system automatically resets the event state to nonsignaled, after a single waiting thread is released. Since the third parameter is `FALSE`, the initial state of the event object is nonsignaled and the fourth parameter is the name of the event object. In this case, the macro is `VPU_INT_PIC_RUN_NAME`.

```
RunEvent = CreateEvent(NULL, FALSE, FALSE, VPU_INT_PIC_RUN_NAME);
if(RunEvent == NULL) {
        return ;
}
```

The `CreateThread` function creates a thread to execute within the virtual address space of the calling process. The first parameter of this function is `NULL`, which implies that the handle cannot be inherited. The second parameter is the initial size of the stack, in bytes and the system rounds this value to the nearest page. Since this parameter is zero, the new thread uses the default size for the executable. The third parameter is the pointer to the application defined function, which is to be executed by the thread. (In this case, the function is `EncodeProc`). The fourth parameter is a pointer to a variable, which is to be passed to the thread and this is an optional parameter. (In this case, it is `NULL`)

```
hEncodeThread = CreateThread(NULL, 0,
(LPTHREAD_START_ROUTINE)EncodeProc, NULL, 0, NULL);
if(hEncodeThread == NULL) {
 CloseHandle(RunEvent);
 return;
}
```

The fifth parameter is a flag that control the creation of the thread. Since it is zero, the thread runs immediately after it is created. The last parameter is a pointer to a variable that receives the thread identifier. This parameter is `NULL`, which implies that the thread identifier is not returned.

The `DecodeTest` function then declares a `PhysicalAddress` variable to use the physical address of the `framebuffer` directly and initializes `IMAGE_SHARE_FRAMEBUFFER_RAM_PA_START` address.

```
PhysicalAddress frameBufferPhysAddr;
frameBufferPhysAddr = IMAGE_SHARE_FRAMEBUFFER_RAM_PA_START;
```

`decOP` structure is initialized with the corresponding parameters for decoding. In this case, `decOP.bistreamFormat` is set to AVC format, but it can be modified to MPEG4 or H.263 format, if required.

```
decOP.bitstreamFormat = codestand;
decOP.bitstreamBuffer = BitstreamPhy;
decOP.virt_bitstreamBuffer = pBitStream;
decOP.bitstreamBufferSize = CODEC_BITSTREAM_SIZE;
paBsBufStart = decOP.bitstreamBuffer;
paBsBufEnd = decOP.bitstreamBuffer + decOP.bitstreamBufferSize;
decOP.reorderEnable = REORDER_ENABLE_VALUE;
decOP.filePlayEnable = 0;
```

`decOP.qpReport` member variable is set, but it is never used.

```
if(codestand == STD_MPEG4)
decOP.qpReport = 1;
else
decOP.qpReport = 0;
```

In this section of code, a physical memory is allocated according to the `PS_SAVE_BUFFER_SIZE` macro. `pssavebuffer` is the pointer returned to the physical memory. If memory allocation fails, then it goes to `ERR_DEC_OPEN`.

```
if(decOP.bitstreamFormat == STD_AVC) {
if(RETCODE_SUCCESS != vpu_AllocPhysMem(PS_SAVE_BUFFER_SIZE, &pssavebuffer)) {
          RETAILMSG(1, (_T("AllocPhysMem for Slice save buffer failed  \n")));
          goto ERR_DEC_OPEN;
         }
decOP.psSaveBuffer = pssavebuffer.PhysAdd;
decOP.psSaveBufferSize = PS_SAVE_BUFFER_SIZE;
}
```

`vpu_DecOpen` function opens a decoding processing instance. Here, the first parameter is a pointer to the storage that contains the handle, using which a decoder instance is referred. If no instance is available, `NULL` is returned. The second parameter is a pointer to `DecOpenParam` type, which describes the parameters necessary for decoding.

```
ret = vpu_DecOpen(&handle, &decOP);
if(ret != RETCODE_SUCCESS) {
DEBUGMSG(1, (_T("VPU_DecOpen failed Error code is 0x%x \n"),  ret));
```

```
goto ERR_DEC_INIT;
}
```

vpu_DecGetBitstreamBuffer function gets the information about the bitstream for the decoder. The first parameter is a pointer to the handle obtained from the vpu_DecOpen function. The second parameter is a pointer to the memory location that stores the physical address at which the bit processor gets the bitstreams. The third parameter is a pointer to the storage that stores the physical address at which the bitstreams are given as input.

```
ret = vpu_DecGetBitstreamBuffer(handle, &paRdPtr, &paWrPtr, &size);
if(ret != RETCODE_SUCCESS) {
DEBUGMSG(1, (_T("VPU_DecGetBitstreamBuffer failed Error code is 0x%x \n"),  ret));
goto ERR_DEC_OPEN;
}
```

While loop calls vpu_DecGetBitstreamBuffer function and waits for 10 ms, as long as the size is bigger than (CODEC_BITSTREAM_SIZE - 1024*2).

```
while(size > (CODEC_BITSTREAM_SIZE - 1024*2)) {
vpu_DecGetBitstreamBuffer(handle, &paRdPtr, &paWrPtr, &size);
Sleep(10);
}
```

The vpu_DecSetEscSeqInit function is provided while executing the vpu_DecGetInitialInfo function, to let the application escape from the hung state. The first parameter is a pointer to the handle obtained from vpu_DecOpen. The second parameter is enabled to escape from the hung state.

```
vpu_DecSetEscSeqInit(handle, 1);
```

The vpu_DecGetInitialInfo function gets the bitstream header information such as picture size. The first parameter is a pointer to the handle obtained from vpu_DecOpen function. The second parameter is a pointer to DecGetInitialInfo data structure that contains the header info. It returns RETCODE_SUCCESS for successful closure.

```
ret = vpu_DecGetInitialInfo(handle, &initialInfo);
if(ret != RETCODE_SUCCESS) {
DEBUGMSG(1, (_T("vpu_DecGetInitialInfo failed Error code is 0x%x \n"),  ret));
goto ERR_DEC_OPEN;
}
```

The vpu_DecSetEscSeqInit function is disabled to escape from the hung state.

```
vpu_DecSetEscSeqInit(handle, 0);
```

The RETAILMSG macro conditionally generates an output which is a printf style formatted message. The macro prints the pictures' width or height and the frames' rate, resolution, or division.

```
fRateInfo = initialInfo.frameRateInfo;
RETAILMSG(1,(_T("picWidth: %u, picHeight: %u, frameRate: %.2f, frRes: %u, frDiv: %u\n"),
initialInfo.picWidth, initialInfo.picHeight, (double)(fRateInfo & 0xffff)/
((fRateInfo >> 16) + 1), fRateInfo & 0xffff, fRateInfo >> 16));
```

This section of code sets the initialInfo structure with a recalculated picture width and height. It also changes the YFrameSize variable, multiplying it by a factor of 1.5.

```
initialInfo.picWidth = ((initialInfo.picWidth + 15) & ~15);
initialInfo.picHeight = ((initialInfo.picHeight + 15) & ~15);
stride = initialInfo.picWidth;
```

```
YFrameSize = initialInfo.picWidth * initialInfo.picHeight;
tmp = (int) (YFrameSize * 1.5);
needbuffercounter = initialInfo.minFrameBufferCount + 2;
```

This section of code allocates enough memory for the `framebuffer` and this variable is the address of the allocated memory.

```
if(RETCODE_SUCCESS != vpu_AllocPhysMem(tmp * needbuffercounter, &framebuffers)) {
printf("AllocPhysMem failed\n");
goto ERR_DEC_OPEN;
}
```

This for loop calculates and initializes the start address for Y, Cb, and Cr buffers. `FrameBuf` belongs to a FrameBuffer variable type.

```
pFrameBuf = (UINT8*)framebuffers.VirtAdd;
FrameBufPhy = framebuffers.PhysAdd;
for (i = 0; i < needbuffercounter; i++) {
frameBuf[i].bufY  = FrameBufPhy + tmp * i ;
frameBuf[i].bufCb = frameBuf[i].bufY + YFrameSize;
frameBuf[i].bufCr = frameBuf[i].bufCb + YFrameSize/4;
VframeBuf[i] = (PVOID) (pFrameBuf + tmp * i) ;
}
```

`vpu_AllocPhyMem` allocates memory for the slice save buffer. Size can be `initialInfo.worstSliceSize` or `initialInfor.normalSliceSize`. This time, worst case is used.

```
if (decOP.bitstreamFormat == STD_AVC) {
if(RETCODE_SUCCESS != vpu_AllocPhysMem(initialInfo.worstSliceSize*1024,
&slicesavebuffer)) {
          RETAILMSG(1, (_T("AllocPhysMem for Slice save buffer failed  \n")));
          goto ERR_DEC_OPEN;
         }
 decBufInfo.avcSliceBufInfo.sliceSaveBuffer = slicesavebuffer.PhysAdd;
decBufInfo.avcSliceBufInfo.sliceSaveBufferSize = initialInfo.worstSliceSize*1024;
}
```

The `vpu_DecRegisterFrameBuffer` function is recommended to register frame buffers by the decoder. This function registers the frame buffers requested by the `vpu_DecGetInitialInfo`. The first parameter is a pointer to the handle, which is obtained from the `vpu_DecOpen` function. The second parameter is a pointer to the first element of an array of the FrameBuffer. The third parameter is the number of elements in the array and the fourth parameter is the stride value of the frame buffers that are registered.

```
ret = vpu_DecRegisterFrameBuffer(handle, frameBuf, initialInfo.minFrameBufferCount,
stride, &decBufInfo);
if(ret != RETCODE_SUCCESS) {
DEBUGMSG(1, (_T("vpu_DecRegisterFrameBuffer failed Error code is 0x%x \n"),  ret));
goto ERR_DEC_OPEN;
}
```

The `decParam` structure is set according to the decoding configuration.

```
decParam.dispReorderBuf = 0;
decParam.prescanEnable = PRESCAN_ENABLE_VALUE;
decParam.prescanMode = 0;
decParam.skipframeMode = 0;
decParam.skipframeNum = 0;
decParam.iframeSearchEnable = 0;
```

The `PPOpenHandle` function creates a handle to the post procesoor (PP) stream driver and returns a handle to the PP stream driver, which is set in this function.

```
ghPp = PPOpenHandle();
if (ghPp == NULL) {
DEBUGMSG(1, (_T("vPPOpenHandle is failed\r\n")));
goto ERR_DEC_OPEN;
}
```

This section of code sets the image orientation according to the information in the `initialInfo` structure.

```
UINT16 inWidth, inHeigh;
inWidth = rotAngle == 90 || rotAngle == 270 ? initialInfo.picHeight :
initialInfo.picWidth;
inHeigh = rotAngle == 90 || rotAngle == 270 ? initialInfo.picWidth :
initialInfo.picHeight;
```

If `codestand` is in MPEG4 format, then it activates the `Deblock` functionality.

```
if(codestand == STD_MPEG4)
gPpConfig.bDeblock = TRUE;
else
gPpConfig.bDeblock = FALSE;
```

This section of code continues to configure the `gPpConfig` structure.

```
gPpConfig.bDering = FALSE;
gPpConfig.inputSize.width = inWidth;
gPpConfig.inputSize.height = inHeigh;
gPpConfig.inputStride = 0;
gPpConfig.outputSize.width = 240;//176;//240;//
gPpConfig.outputSize.height = 240*IMAGE_HEIGHT/IMAGE_WIDTH;//120;//180;//
gPpConfig.outputStride = 240*2;
gPpConfig.outputFormat = ppCSCOutputFormat_RGB16;
gPpConfig.outputPixelFormat.component0_width = 5;
gPpConfig.outputPixelFormat.component1_width = 6;
gPpConfig.outputPixelFormat.component2_width = 5;
gPpConfig.outputPixelFormat.component0_offset = 11;
gPpConfig.outputPixelFormat.component1_offset = 5;
gPpConfig.outputPixelFormat.component2_offset = 0;
gPpConfig.CSCEquation = ppCSCEquationA_1;
```

The `gPpConfig` variable belongs to the `ppConfigData` type and is used for post processing configuration.

```
typedef struct {
BOOL bDeblock;
BOOL bDering;
// Input
ppFrameSize inputSize;
UINT16 inputStride; // In pixels
// Output
ppFrameSize outputSize;
UINT16 outputStride; // In bytes
ppCSCOutputFormat outputFormat;
ppPixelFormat outputPixelFormat;
ppCSCEquation CSCEquation;
} ppConfigData, *pPpConfigData;
```

The PPConfigure function configures the post processor using the parameters passed by the caller. The first parameter is a handle to the PP driver and the second parameter is a pointer to the PP configuration data structure.

```
PPConfigure(ghPp, &gPpConfig);
```

The PPStart function starts the post processor processing. The parameter is a handle to the PP driver.

```
PPStart(ghPp);
```

The application ensures that the initialInfo.frameRateInfo member is set with a proper value. Otherwise, it assigns 30 as the default frame rate.

```
if((initialInfo.frameRateInfo<=0) || (initialInfo.frameRateInfo > 30))
initialInfo.frameRateInfo = 30;
```

The CreateEvent function creates or opens a named or unnamed event object. The first parameter is a pointer to a SECURITY_ATTRIBUTES structure. Since this parameter is NULL, the handle cannot be inherited by the child processes. Since the second parameter is FALSE, the function creates an auto reset event object and the system automatically resets the event state to nonsignaled after a single waiting thread is released. If the second parameter is TRUE, the function creates a manual reset event object, which requires the usage of the ResetEvent function to set the event state to nonsignaled. The third parameter is FALSE, which implies that the initial state of the event object is nonsignaled. The last parameter is NULL, which implies that the event object is created without a name.

```
hRateTimerEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
```

The timeSetEvent function starts a specified timer event and the multimedia timer runs in its own thread. After the event is activated, it calls the specified callback function or sets or pulses the specified event object. This function is obsolete. New applications use CreateTimerQueueTimer to create a timer queue timer. The first parameter is the event delay, in milliseconds. The second parameter is the resolution of the timer event, in milliseconds. A resolution of 0 indicates that the periodic events occur with the greatest possible accuracy.

```
TimerId = timeSetEvent(1000/initialInfo.frameRateInfo,
0,(LPTIMECALLBACK)hRateTimerEvent, 0, TIME_CALLBACK_EVENT_SET|TIME_PERIODIC);
```

To reduce system overhead, use the maximum value, appropriate for the application. The third parameter is a pointer to the callback function that is called once upon the expiration of a single event or periodically upon the expiration of the periodic events. The last parameter specifies the timer event type.

The CeSetThreadPriority function sets the priority for a real time thread, on a thread by thread basis. The first parameter is a handle to the thread and the second parameter is a priority to set the thread. This value ranges from 0 through 255, with 0 as the highest priority.

```
CeSetThreadPriority(GetCurrentThread(), 240);
```

If PROFILE_TIME is defined and frameIdx equals to 100, the QueryPerformanceCounter function is executed. This function retrieves the current value of the high resolution performance counter. The parameter is a pointer to the variable that receives the current performance counter value, in counts.

If frameIdx equals to PROFILE_FRMAE_END_NUM, the QueryPerformanceCounter function is executed and it substracts litmp.QuadPart from liTime.QuadPart.

```
#ifdef PROFILE_TIME
if(frameIdx == 100)
QueryPerformanceCounter(&liTime);
else if(frameIdx == PROFILE_FRMAE_END_NUM) {
litmp = liTime;
QueryPerformanceCounter(&liTime);
liTime.QuadPart -= litmp.QuadPart;
}
#endif
```

If ENC_DEC_SYNC is defined, then the WaitForSingleObject function is called. It waits until the specified object is in the signaled state or untill the time-out interval lapses. The object is hDecoding and since the second parameter is INFINITE, the function's time-out interval never lapses.

```
#ifdef ENC_DEC_SYNC
WaitForSingleObject(hDecoding, INFINITE);
#endif
```

The vpu_DecStartOneFrame function starts decoding one frame at a time. The first parameter is a handle obtained from the vpu_DecOpen function and the second parameter is a structure of decoding parameters.

```
ret = vpu_DecStartOneFrame(handle, &decParam);
```

If PROFILE_TIME is defined and if frameIdx ranges from 100 to 4100, the QueryPerformanceCounter function is executed to update the liStartTime.

```
#ifdef PROFILE_TIME
if((frameIdx >= 100) && (frameIdx < 4100))
QueryPerformanceCounter(&liStartTime);
#endif
```

If ret is not equal to RETCODE_SUCCESS, it goes to ERR_DEC_OPEN.

```
if(ret != RETCODE_SUCCESS) {
DEBUGMSG(1, (_T("vpu_DecStartOneFrame failed Error code is 0x%x \n"),  ret));
goto ERR_DEC_OPEN;
}
```

The function now waits for the RunEvent object to complete its execution.

```
WaitForSingleObject(RunEvent, INFINITE);
```

If PROFILE_TIME is defined, the application compares if frameIdx is greater than or equal to 100 and if frameIdx is less than PROFILE_FRMAE_END_NUM. If it falls under this range, the application calls the QueryPerformanceCounter function and updates the liStopTime variable. It also updates the liHWTime.QuadPart variable.

```
#ifdef PROFILE_TIME
if((frameIdx >= 100) && (frameIdx < PROFILE_FRMAE_END_NUM)) {
QueryPerformanceCounter(&liStopTime);
liHWTime.QuadPart += (liStopTime.QuadPart - liStartTime.QuadPart);
}
#endif
```

The vpu_DecGetOutputInfo function gets the information of the decoding output. The first parameter is a handle obtained from the vpu_DecOpen function and the second parameter is a pointer to the DecOutputInfo data structure.

```
ret = vpu_DecGetOutputInfo(handle, &outputInfo);
```

If `ret` is not equal to `RETCODE_SUCCESS`, it goes to `ERR_DEC_OPEN`.

```
if(ret != RETCODE_SUCCESS) {
DEBUGMSG(1, (_T("vpu_DecGetOutputInfo failed Error code is 0x%x \n"),  ret));
goto ERR_DEC_OPEN;
}
```

If `outputInfo.notSufficientPsBuffer` variable is not equal to 0, it prints a message `PS buffer overflow` on the debug console and sets the `decodefinish` variable to 1.

```
if(outputInfo.notSufficientPsBuffer) {
DEBUGMSG(1, (_T("PS Buffer overflow framdIdx %d, Exit decoding... \n"), frameIdx));
decodefinish = 1;
}
```

If `outputInfo.notSufficientSliceBuffer` is not equal to 0, it prints the message `Slice buffer overflow` on the debug console.

```
if(outputInfo.notSufficientSliceBuffer) {
DEBUGMSG(1, (_T("Slice Buffer overflow framdIdx %d \n"), frameIdx));
}
```

If `outputInfo.indexFrameDisplay` is -1, then the decoding is complete. If `outputInfo.indexFrameDisplay` is greater than `needbuffercounter` and if `outputInfo.prescanresult` is not equal to 0, it sets the `decodefinish` variable to 1.

```
if(outputInfo.indexFrameDisplay == -1)
decodefinish = 1;
else if((outputInfo.indexFrameDisplay > needbuffercounter) && (outputInfo.prescanresult
!= 0))
decodefinish = 1;
```

If `decodefinish` variable is not equal to 0, the decoding process is complete.

```
if(decodefinish)
break;
```

If `outputInfo.prescanresult` variable equals to 0, the program execution is continued.

```
if(outputInfo.prescanresult == 0) {
continue;
}
```

If `outputInfo.indexFrameDisplay` variable is -3 or -2, then the bit processor does not have any picture to display and hence, it continues with the program execution.

```
if(outputInfo.indexFrameDisplay == -3 || outputInfo.indexFrameDisplay == -2)
continue;
```

If `ret` equals `RECODE_SUCESS`, the buffers Y, Cb, Cr, and the physical address of the framebuffer is assigned to the `gPpInputPtr` structure. The function then waits for the `hRateTimerEvent` object. The `PPAddBuffers` function allows the caller to add work buffers to the PP buffer queues. The first parameter is a handle to the PP driver. The second parameter is a pointer to a structure that contains the input and output buffers. It now waits infinitely for `ghPpDisplayEvent` object.

```
if (ret == RETCODE_SUCCESS) {
gPpInputPtr.InYBuf = (PVOID) frameBuf[outputInfo.indexFrameDisplay].bufY;
gPpInputPtr.InUBuf = (PVOID) frameBuf[outputInfo.indexFrameDisplay].bufCb;
gPpInputPtr.InVBuf = (PVOID) frameBuf[outputInfo.indexFrameDisplay].bufCr;
```

```
gPpInputPtr.OutBuf = (PVOID) frameBufferPhysAddr;
        if(codestand == STD_MPEG4)
        gPpInputPtr.InQPBuf = (PVOID) outputInfo.phys_qpInfo;
WaitForSingleObject(hRateTimerEvent, INFINITE);
PPAddBuffers(ghPp, &gPpInputPtr);
WaitForSingleObject(ghPpDisplayEvent, INFINITE);
}
```

If `outputInfo.numOfErrMBs` is not equal to 0, it increments the `totalNumofErrMbs` variable and prints the number of error Mbs and the frame index on the debug console.

```
if(outputInfo.numOfErrMBs) {
totalNumofErrMbs += outputInfo.numOfErrMBs;
DEBUGMSG(1, (_T("Num of Error Mbs : %d, in Frame : %d \n"), outputInfo.numOfErrMBs,
frameIdx));
}
```

The function now increments the `frameIdx` variable.

```
frameIdx++;
```

If `frameIdx` variable equals to `TEST_FRAME_NUM` macro, then `g_fLoopTest` is set to `FALSE`, to complete the while loop execution and thereby the demonstration application.

```
if(frameIdx == TEST_FRAME_NUM)
g_fLoopTest = FALSE;
```

Once the while loop completes its execution, it prints the total frames used for the decoding and encoding processes. The while loop also prints the average time of hardware and the average total time, on the screen.

```
#ifdef PROFILE_TIME
printf("Dec: total frames:%d", frameIdx);
printf("Dec: Average time of HW:
%f(ms)\n",liHWTime.QuadPart/(fHighFre*PROFILE_FRMAE_NUM)*1000);
printf("Dec: Average total time :
%f(ms)\n",liTime.QuadPart/(fHighFre*PROFILE_FRMAE_NUM)*1000);
#endif
```

Now, the while loop waits until the encoding thread ends.

```
if(hEncodeThread) {
g_fLoopTest = FALSE;
Sleep(200);
TerminateThread(hEncodeThread, 0);
CloseHandle(hEncodeThread);
hEncodeThread = NULL;
}
```

The `timeKillEvent` function cancels the specified timer event.

```
timeKillEvent(TimerId);
```

Now, close the open instance if the decoding process is complete.

```
ERR_DEC_OPEN:
vpu_DecClose(handle);
if(bitstreambuf.PhysAdd)
        vpu_FreePhysMem(&bitstreambuf);
if(framebuffers.PhysAdd)
```

```
        vpu_FreePhysMem(&framebuffers);
if(pssavebuffer.PhysAdd)
vpu_FreePhysMem(&pssavebuffer);
if(slicesavebuffer.PhysAdd)
        vpu_FreePhysMem(&slicesavebuffer);
if (ghPp != NULL) {
PPStop(ghPp);
PPCloseHandle(ghPp);
ghPp = NULL;
}
if (ghPpDisplayEvent != NULL) {
CloseHandle(ghPpDisplayEvent);
ghPpDisplayEvent = NULL;
}
if(RunEvent) {
CloseHandle(RunEvent);
RunEvent = NULL;
}
CloseHandle(hRateTimerEvent);
ERR_DEC_INIT:
return ;
```

2.3 Encoding Process

The `EncodeProc` function is called from the encoding thread in the dectest.cpp file.

```
DWORD WINAPI EncodeProc(LPVOID lpParameter)
```

In this function, the `EncodeTest` function is called according to the following parameters:

```
CodStd stdMode;
int picWidth,picHeight, bitRate;
stdMode = CODEC_STD;
picWidth = IMAGE_WIDTH;
picHeight = IMAGE_HEIGHT;
bitRate = 512;
EncodeTest(stdMode, picWidth, picHeight, bitRate);
return 0;
```

The `EncodeTest` function demonstrates how to encode using the VPU API. The function prototype is as follows:

```
static void EncodeTest(CodStd stdMode, int picWidth, int cpicHeight, int bitRate);
```

At the beginning of the function, it declares the required variables for the encoding process.

```
EncHandle handle;
EncOpenParam encOP;
EncParam encParam;
SearchRamParam searchPa = { 0 };
EncHeaderParam encHeaderParam = { 0 };
EncInitialInfo initialInfo;
EncOutputInfo outputInfo;
RetCode ret;
PhysicalAddress bsBuf0;
Uint32 size0;
```

The `EncodeTest` function also declares and initializes the variables used for the profile time measurement.

```
#ifdef PROFILE_TIME
LARGE_INTEGER liStartTime = {0}, liStopTime = {0};
LARGE_INTEGER liTime = {0};
LARGE_INTEGER liHWTime = {0};
LARGE_INTEGER litmp = {0};
#endif
```

The `FrameIdx` variables are used to keep track of the frame index in the application.

```
int srcFrameIdx, secondsrcFrameIdx = 0, tmpFrameIdx = 0;
UINT YFrameSize;
int tmp;
```

The `FrameBuffer` and `VPUMemAlloc` variables are declared, as they are used in the encoding process.

```
PRP_BUFFER PrpBuffer = {0};
FrameBuffer frameBufTmp = {0};
FrameBuffer frameBuf[NUM_FRAME_BUF];
int i;
int frameIdx;
int exit;
int stride;
HANDLE RunEvent = NULL;
#ifndef USE_PRP_PHYSICAL_ADDRESS
HANDLE hFillBufIST = NULL;
#endif
VPUMemAlloc  bitstreambuf = {0};
VPUMemAlloc  framebuffers = {0};
```

`RunEvent` is created and the name of the event object is VPU Pic Run Command.

```
RunEvent = CreateEvent(NULL, FALSE, FALSE, VPU_INT_PIC_RUN_NAME);
if(RunEvent == NULL){
return ;
}
```

Now, the decoding event is created. This event object does not have a name.

```
#ifdef ENC_DEC_SYNC
hDecoding = CreateEvent(NULL, FALSE, FALSE, NULL);
if(hDecoding == NULL){
CloseHandle(RunEvent);
return ;
}
```

Since `USE_PRP_PHYSICAL_ADDRESS` is not defined, the Fill Buffer and the Fill Buffer events are created.

```
#ifndef USE_PRP_PHYSICAL_ADDRESS
hFillBufEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
if(RunEvent == NULL) {
 CloseHandle(RunEvent);
        return ;
}
hFillBufEventFinish = CreateEvent(NULL, FALSE, FALSE, NULL);
if(hFillBufEventFinish == NULL){
CloseHandle(RunEvent);
CloseHandle(hFillBufEvent);
return ;
}
```

The thread to fill the YUV image is created.

```
hFillBufIST = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)FillYUVThread, NULL, 0,
NULL);
if(hFillBufIST == NULL) {
CloseHandle(RunEvent);
CloseHandle(hFillBufEvent);
CloseHandle(hFillBufEventFinish);
return;
}
#endif
```

The thread invokes the `FillYUVThread` function and the `ThreadPriority` is set to 121. It now waits infinitely for the `hFillBufEvent` to execute. The `PerFrameCameraCapture` function outputs one frame through the capture pin of the camera. The parameter passed is the address of the output data buffer.

```
static DWORD WINAPI FillYUVThread(LPVOID lpParameter)
{
CeSetThreadPriority(GetCurrentThread(), 121);
while(1) {
WaitForSingleObject(hFillBufEvent, INFINITE);
PerFrameCameraCapture(pFrameBuf + (pAvailableFrameBuf->bufY - FrameBufPhy));
SetEvent(hFillBufEventFinish);
}
return 0;
}
```

Then, call `LoadCamDriver` function to load the camera driver and initiate the preview and capture pins. The camera driver is loaded succesfully, if it returns `true`.

```
if(!LoadCamDriver()) {
CloseHandle(RunEvent);
return;
}
```

In this section of code, a physical memory is allocated to the `bitstreambuffer`. The virtual and physical address of the allocated memory which is returned, is assigned to the `pBitStream` and `BitstreamPhy` pointer variables.

```
if (RETCODE_SUCCESS != vpu_AllocPhysMem(CODEC_BITSTREAM_SIZE, &bitstreambuf)) {
goto ERR_ENC_OPEN;
}
pBitStream = (UINT8 *)bitstreambuf.VirtAdd;

BitstreamPhy = bitstreambuf.PhysAdd;
```

Fill the parameters of `encOP` structure according to the encoding requirements.

```
encOP.bitstreamBuffer = BitstreamPhy;
encOP.bitstreamBufferSize = CODEC_BITSTREAM_SIZE;
encOP.bitstreamFormat = stdMode;
encOP.picWidth = picWidth;
encOP.picHeight = picHeight;
encOP.frameRateInfo = 30;
encOP.bitRate = bitRate;
encOP.initialDelay = 0;
encOP.vbvBufferSize = 0;
encOP.enableAutoSkip = 1;
encOP.gopSize = 0;
```

```
encOP.slicemode.sliceMode = 0;
encOP.slicemode.sliceSizeMode = 0;
encOP.slicemode.sliceSize = 0;
encOP.intraRefresh = 0;
encOP.sliceReport = 0;
encOP.mbReport = 0;
encOP.mbQpReport = 0;
encOP.rcIntraQp =-1;
```

The remaining members of the `encOP` structure are set according to the specific encoding format.

```
if(stdMode == STD_MPEG4 ) {
encOP.EncStdParam.mp4Param.mp4_dataPartitionEnable = 0;
encOP.EncStdParam.mp4Param.mp4_reversibleVlcEnable = 0;
encOP.EncStdParam.mp4Param.mp4_intraDcVlcThr = 0;
encOP.EncStdParam.mp4Param.mp4_hecEnable= 0;
encOP.EncStdParam.mp4Param.mp4_verid = 2;
}
else if(stdMode == STD_H263) {
encOP.EncStdParam.h263Param.h263_annexJEnable = 0;
encOP.EncStdParam.h263Param.h263_annexKEnable = 0;
encOP.EncStdParam.h263Param.h263_annexTEnable = 0;
}
else if(stdMode == STD_AVC) {
encOP.EncStdParam.avcParam.avc_constrainedIntraPredFlag = 0;
encOP.EncStdParam.avcParam.avc_disableDeblk = 2;
encOP.EncStdParam.avcParam.avc_deblkFilterOffsetAlpha = -3;
encOP.EncStdParam.avcParam.avc_deblkFilterOffsetBeta = -5;
encOP.EncStdParam.avcParam.avc_chromaQpOffset = 4;
encOP.EncStdParam.avcParam.avc_audEnable = 0;
encOP.EncStdParam.avcParam.avc_fmoEnable = 0;
encOP.EncStdParam.avcParam.avc_fmoType = 0;
encOP.EncStdParam.avcParam.avc_fmoSliceNum = 0;
}
else {
DEBUGMSG(1, (_T("Invalid bitstream format mode \n")));
goto ERR_ENC_INIT;
}
encOP.ringBufferEnable =  0;
encOP.dynamicAllocEnable = 0;
```

The `vpu_EncOpen` function opens an encoding processing instance. The first parameter is a pointer to the storage that contains a handle through which an encoder instance is referred. `NULL` is returned, if no instance is available. The second parameter is a pointer to the `EncOpenParam` type, which describes the parameters necessary for decoding.

```
ret = vpu_EncOpen(&handle, &encOP);
if(ret != RETCODE_SUCCESS) {
DEBUGMSG(1, (_T("vpu_EncOpen failed Error code is 0x%x \n"),  ret));
goto ERR_ENC_INIT;
}
```

The `searchPa.searchRamAddr` variable is set with the `SEARCHRAM_ADDR` command. The `vpu_EncGiveCommand` function provides commands to the encoder. The first parameter is a handle obtained from the `vpu_EncOpen` function. The second parameter is a command sent to the encoder and the third parameter is used for some specific commands. `RETCODE_SUCCESS` is returned for successful closure.

```
searchPa.searchRamAddr = DEFAULT_SEARCHRAM_ADDR;
ret = vpu_EncGiveCommand( handle, ENC_SET_SEARCHRAM_PARAM, &searchPa);
if(ret != RETCODE_SUCCESS) {
DEBUGMSG(1, (_T("VPU_EncGiveCommand failed Error code is 0x%x \n"),  ret));
goto ERR_ENC_OPEN;
}
```

The `vpu_EncGetInitialInfo` function retrieves the bitstream header information. The first parameter is a pointer to the handle obtained from the `vpu_EncOpen` function. The second parameter is a pointer to the `EncInitialInfo` data structure. `RETCODE_SUCCESS` is returned for successful closure.

```
ret = vpu_EncGetInitialInfo(handle, &initialInfo);
if(ret != RETCODE_SUCCESS) {
DEBUGMSG(1, (_T("vpu_EncGetInitialInfo failed Error code is 0x%x \n"),  ret));
goto ERR_ENC_OPEN;
}
```

`YFrameSize` is resized according to the width and height of the picture. `SrcFrameIdx` is set with the `minFrameBufferCount` member of the `initialInfo` structure.

```
YFrameSize = encOP.picWidth * encOP.picHeight;
srcFrameIdx = initialInfo.minFrameBufferCount;
tmp = (int) (YFrameSize*1.5);
```

Since `USE_PRP_PHYSICAL_ADDRESS` is not defined, the code allocates memory for the `framebuffers`. The addresses of Y, Cb, and Cr buffers are calculated according to the `framebuffers` pointer, returned from the `vpu_AllocPhysMem` function.

```
#ifdef USE_PRP_PHYSICAL_ADDRESS
//allocate memory for framebuffer
pFrameBuf = (UINT8*)vpu_AllocPhysMem((tmp * (initialInfo.minFrameBufferCount)),
&FrameBufPhy);
for (i = 0; i < initialInfo.minFrameBufferCount; i++) {
#else
if(RETCODE_SUCCESS != vpu_AllocPhysMem((tmp * (initialInfo.minFrameBufferCount+2)),
&framebuffers)) {
printf("AllocPhysMem failed\n");
goto ERR_ENC_INIT;
}
pFrameBuf = (UINT8*)framebuffers.VirtAdd;
FrameBufPhy = framebuffers.PhysAdd;
for (i = 0; i < initialInfo.minFrameBufferCount+2; i++) {
#endif
frameBuf[i].bufY  = FrameBufPhy + tmp * i ;
frameBuf[i].bufCb = frameBuf[i].bufY + YFrameSize;
frameBuf[i].bufCr = frameBuf[i].bufCb + YFrameSize/4;
}
```

The `vpu_EncRegisterFrameBuffer` function registers the frame buffers requested by `vpu_EncGetInitialInfo`.

```
stride = picWidth;
ret = vpu_EncRegisterFrameBuffer(handle, frameBuf, initialInfo.minFrameBufferCount,
stride);
if( ret != RETCODE_SUCCESS ) {
DEBUGMSG(1, (_T("vpu_EncRegisterFrameBuffer failed Error code is 0x%x \n"),  ret));
goto ERR_ENC_OPEN;
}
```

The vpu_EncRegisterFrameBuffer function initializes frameIdx, exit, encParam.forceIPicture, and encParam.skipPicture to 0. The pointer to the first element of an array of FrameBuffer is assigned to the encParam.sourceFrame member variable.

```
exit = 0;
frameIdx = 0;
encParam.sourceFrame = &frameBuf[srcFrameIdx];
encParam.quantParam = 30;
encParam.forceIPicture = 0;
encParam.skipPicture = 0;
```

The encoder header varies with respect to the stdMode variable set.

```
if(stdMode == STD_MPEG4) {
encHeaderParam.headerType = VOL_HEADER;
vpu_EncGiveCommand(handle, ENC_PUT_MP4_HEADER, &encHeaderParam);
if(encOP.ringBufferEnable == 0)
SaveBSBuffer((pBitStream +(encHeaderParam.PhysBuf - BitstreamPhy)),
(size_t)encHeaderParam.size);
}
else if(stdMode == STD_AVC) {
encHeaderParam.headerType = SPS_RBSP;
vpu_EncGiveCommand(handle, ENC_PUT_AVC_HEADER, &encHeaderParam);
if(encOP.ringBufferEnable == 0)
SaveBSBuffer((pBitStream +(encHeaderParam.PhysBuf - BitstreamPhy)),
(size_t)encHeaderParam.size);
encHeaderParam.headerType = PPS_RBSP;
vpu_EncGiveCommand(handle, ENC_PUT_AVC_HEADER, &encHeaderParam);
if( encOP.ringBufferEnable == 0 )
SaveBSBuffer((pBitStream +(encHeaderParam.PhysBuf - BitstreamPhy)),
(size_t)encHeaderParam.size);
}
```

In this section of code, the parameters are updated.

```
encParam.slicemode.sliceMode = encOP.slicemode.sliceMode;
encParam.slicemode.sliceSizeMode = encOP.slicemode.sliceSizeMode;
encParam.slicemode.sliceSize = encOP.slicemode.sliceSize;
encParam.intraRefresh = encOP.intraRefresh;
encParam.hecEnable = 0;
```

The StartCameraCapture function starts the camera capture pin. If it is FALSE, then use the physical address. The PerFrameCameraCapture function outputs one frame through the capture pin of the camera. If it returns TRUE, then the operation is successful. The SetEvent function sets the specified event object to a signaled state. CeSetThreadPriority increments the priority of the current thread.

```
#ifdef USE_PRP_PHYSICAL_ADDRESS
StartCameraCapture(TRUE);
#else
StartCameraCapture(
);
if(!PerFrameCameraCapture(pFrameBuf + (encParam.sourceFrame->bufY - FrameBufPhy)))
goto ERR_ENC_OPEN;
secondsrcFrameIdx = srcFrameIdx + 1;

SetEvent(hFillBufEventFinish);
#endif
CeSetThreadPriority(GetCurrentThread(), 120);
```

The while loop continues to execute as long as the g_fLoopTest variable does not equal 0.

```
while (g_fLoopTest) {
#ifdef PROFILE_TIME
if(frameIdx == 100) {
QueryPerformanceCounter(&liTime);
}
else if(frameIdx == PROFILE_FRMAE_END_NUM) {
litmp = liTime;
QueryPerformanceCounter(&liTime);
liTime.QuadPart -= litmp.QuadPart;
}
#endif
```

Since USE_PRP_PHYSICAL_ADDRESS is not defined, the WaitForSingleObject function waits until the hFillBufEventFinish method completes its execution. The current frame buffer is assigned to the pAvailableFrameBuf pointer. The vpu_EncStartOneFrame function starts encoding one frame at a time.

```
#ifdef USE_PRP_PHYSICAL_ADDRESS
GetPhysicalAddrPerFrame(&PrpBuffer);
frameBufTmp.bufY  = (PhysicalAddress)PrpBuffer.pPhysAddr;
frameBufTmp.bufCb = frameBufTmp.bufY + YFrameSize;
frameBufTmp.bufCr = frameBufTmp.bufCb + YFrameSize/4;
encParam.sourceFrame = &frameBufTmp;
#else
WaitForSingleObject(hFillBufEventFinish, INFINITE);

pAvailableFrameBuf = &frameBuf[secondsrcFrameIdx];
SetEvent(hFillBufEvent);
encParam.sourceFrame = &frameBuf[srcFrameIdx];
#endif
ret = vpu_EncStartOneFrame(handle, &encParam);
```

If the frameIdx variable ranges from 100 to 4100, the liStartTime variable is updated with the performance counter value.

```
#ifdef PROFILE_TIME
if((frameIdx >= 100) && (frameIdx < 4100))
QueryPerformanceCounter(&liStartTime);
#endif
```

If vpu_EncStartOneFrame encodes a single frame, then it returns RETCODE_SUCCESS.

```
if( ret != RETCODE_SUCCESS ) {
DEBUGMSG(1, (_T("vpu_EncStartOneFrame failed Error code is 0x%x \n"),  ret));
goto ERR_ENC_OPEN;
}
```

The application waits for the RunEvent object to be executed.

```
WaitForSingleObject(RunEvent, INFINITE);
```

If the frameIdx variable is greater than or equal to 100 and PROFILE_FRMAE_END_NUM is constant, the liStopTime variable is updated with the performance counter value. It also recalculates the liHTTime.QuadPart variable depending on liStopTime and liStartTime.

```
#ifdef PROFILE_TIME
if((frameIdx >= 100) && (frameIdx < PROFILE_FRMAE_END_NUM)) {
QueryPerformanceCounter(&liStopTime);
```

```
liHWTime.QuadPart += (liStopTime.QuadPart - liStartTime.QuadPart);
}
#endif
```

It prepares to fill the next YUV data.

```
#ifdef USE_PRP_PHYSICAL_ADDRESS
ReturnPhysicalAddrPerFrame(&PrpBuffer);
#else
tmpFrameIdx = secondsrcFrameIdx;
secondsrcFrameIdx = srcFrameIdx;
srcFrameIdx = tmpFrameIdx;
#endif
```

The `vpu_EncGetOutputInfo` function match the `vpu_EncStartOneFrame` function with the same handle. No other API functions intervenes between these two functions with any other handle, except for `vpu_IsBusy()`, `vpu_DecGetBistreamBuffer()`, and `vpu_DecUpdateBitstreamBuffer()`.

```
ret = vpu_EncGetOutputInfo(handle, &outputInfo);
if(ret != RETCODE_SUCCESS) {
DEBUGMSG(1, (_T("vpu_EncGetOutputInfo failed Error code is 0x%x \n"),  ret));
goto ERR_ENC_OPEN;
}
```

The function prints a warning message if the BitStream buffer is wrapped around.

```
if(outputInfo.bitstreamWrapAround  == 1)
printf("Warnning!! BitStream buffer wrap arounded. prepare more large buffer \n");
```

Now, the bitstream for the current frame is available.

```
bsBuf0 = outputInfo.bitstreamBuffer;
size0 = outputInfo.bitstreamSize;
```

In this section of code, the `BSBuffer` is saved.

```
if((SaveBSBuffer((pBitStream +(bsBuf0 - BitstreamPhy)), (size_t)size0) != size0)) {
RETAILMSG(1, (_T("SaveBSBuffer failed! \n")));
break;
}
```

The function sets the `hDecoding` event object to a signaled state.

```
#ifdef ENC_DEC_SYNC
SetEvent(hDecoding);
#endif
It prints the current frameIdx.
RETAILMSG(1, (_T("frame %d\n"),frameIdx++));
If PROFILE_TIME is defined, prints the statistics of the application.
#ifdef PROFILE_TIME
printf("Enc: total frames:%d", frameIdx);
printf("Enc: Average time of HW:
%f(ms)\n",liHWTime.QuadPart/(fHighFre*PROFILE_FRMAE_NUM)*1000);
printf("Enc: Averate total time:
%f(ms)\n",liTime.QuadPart/(fHighFre*PROFILE_FRMAE_NUM)*1000);
#endif
```

The open instance is closed if the encoding is complete.

```
ERR_ENC_OPEN:
StopCameraCapture();
```

```
vpu_EncClose(handle);
if(bitstreambuf.PhysAdd)
        vpu_FreePhysMem(&bitstreambuf);
if(framebuffers.PhysAdd)
        vpu_FreePhysMem(&framebuffers);
if(RunEvent) {
CloseHandle(RunEvent);
RunEvent = NULL;
}
#ifndef USE_PRP_PHYSICAL_ADDRESS
if(hFillBufEvent) {
CloseHandle(hFillBufEvent);
hFillBufEvent = NULL;
}
if(hFillBufEventFinish) {
CloseHandle(hFillBufEventFinish);
hFillBufEventFinish = NULL;
}
if(hFillBufIST) {
TerminateThread(hFillBufIST, 0);
CloseHandle(hFillBufIST);
hFillBufIST = NULL;
}
#endif
ERR_ENC_INIT:
UnloadCamDriver();
```

3 Conclusion

The encoding and the decoding format used in this project is hard coded for AVC. However, this can be modified to MPEG-4 or H.263 format if the operation of this demonstration application is understood. The codec format can also be changed at run time. The VPU driver facilitates the implementation of codecs for the i.MX27 processor by hiding the details of the encoding or decoding process.

4 Revision History

Table 1 provides a revision history for this application note.

Table 1. Document Revision History

Rev. Number	Date	Substantive Change(s)
0	03/2010	Initial Release

THIS PAGE INTENTIONALLY LEFT BLANK

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
1-800-521-6274 or
+1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064
Japan
0120 191014 or
+81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
1-800 441-2447 or
+1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

Document Number: AN4043
Rev. 0
03/2010