Diamond Video Engine Controller Code Example

A tutorial on developing control code for the 388VDO Engine

Application Note

Tensilica, Inc.
3255-6 Scott Blvd.
Santa Clara, CA 95054
(408) 986-8000
Fax (408) 986-8919
www.tensilica.com

April 2008

Doc Number: AN08-815-00

Document Change History:

Contents

Abstract

This application note describes how to write basic control code to control video decoding functionality using the 388VDO Diamond Video Engine. The *Diamond 388VDO Software Guide,* provided along with the 388VDO Diamond Video Engine, provides a complete reference of the Application Programming Interface (API). This application note supplements the *Diamond 388VDO Software Guide* by providing a thorough explanation of how the API is used to set up and control the Diamond Video Engine to perform basic video decoding.

1 Introduction

The Tensilica Diamond Video (388VDO) Engine combines hardware and software IP package to provide a complete video compression/decompression package. It is fully programmable and supports all popular VGA and standard definition (SD or D1) video codecs with resolutions up to 720x480 (NTSC) and 720x576 (PAL) pixels. Lower resolutions such as QCIF, QVGA, CIF and VGA are also supported. The Diamond Video Engine supports H.264 Main Profile, VC-1 Main Profile, MPEG-4 Advanced Simple Profile (ASP), and MPEG-2 Main Profile among others. The Diamond Video Engine performs all key video codec functions in software – including the network abstraction layer, picture layer, slice layer, and entropy decoding and encoding – running on two integral processor cores.

The Diamond Video Engine is designed to offload processor-intensive video codec functions in video systems. However, it cannot function independently. It requires a separate controller to initialize and control it. Once the Diamond Video Engine is initialized, it requires very little processing bandwidth from the controller.

The Diamond Video Engine comes with the *Diamond 388VDO Software Guide* that describes the Diamond Video Engine Application Programming Interface (from here on referred to as the API) and Inter-Processor Communication Protocol (from here on referred to as IPCP). This application note is supplemental to these guides and teaches the use of the API and IPCP. As you read this document, you should keep the *Diamond 388VDO Software Guide* handy as a reference to the API and IPCP.

This application note describes a simple system consisting of the Diamond 338VDO Video Engine and a Diamond DC_B_330HiFi_be processor as the controller. The DC_B_330HiFi_be processor initializes and controls the Diamond Video Engine to perform MPEG-4 decoding. The video control application is purposefully kept simple to facilitate learning. It is not intended to be a "drop-in" or reference control code for any video system. However, the control code can be used as a platform to experiment with the API, and a starting point towards creating a full-featured controller code for video based systems.

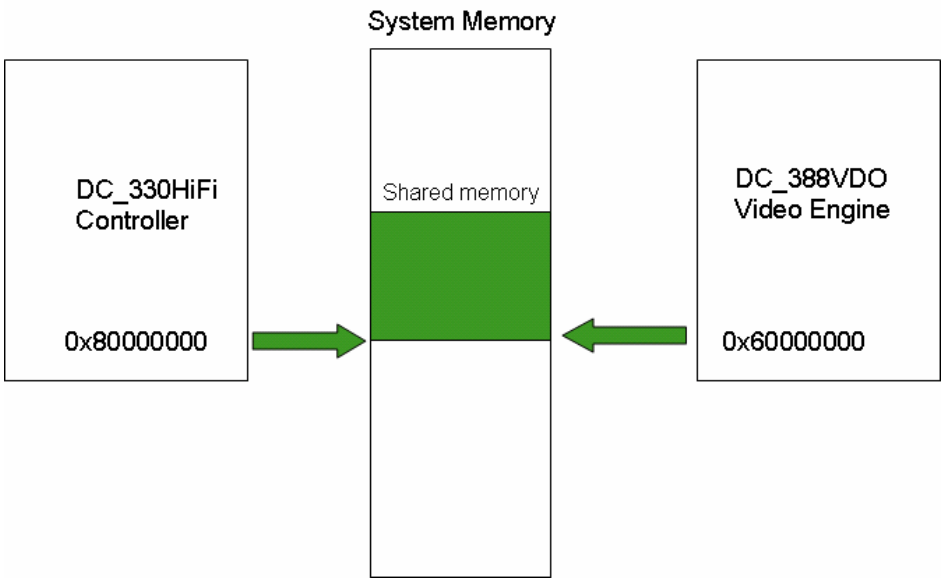
Prerequisites for this application note include basic understanding of video decode systems, and general use of the Xplorer Diamond Edition Software Development environment. Also, this application note assumes the readers have either read the *Diamond 388VDO Software Guide* and *Diamond Video Hardware User's Guide* or have attended some training to acquire some basic knowledge of the Diamond Video Engine.

This application note contains 5 software control examples, starting with a very simple "test for life" program that checks whether the controller can access the video engine – and then further examples gradually add additional control functionality. An Xplorer Diamond Edition software workspace, `VDO_control_examples.xws` can be requested from the Tensilica technical support team. This workspace contains all the video control examples described in this application note. Note that the code provided in the workspace may differ slightly from the code examples in this document.

To compile and simulate the examples provided in this application note, the Xplorer Diamond Edition Software Development tools (ver. 2.1 or later) is required. The 388VDO Video Engine Simulators (ver 0.4) and Software Package (ver. 2.4) must be installed after Xplorer. Refer to the *Diamond 388VDO Software Guide* for product installation instructions.

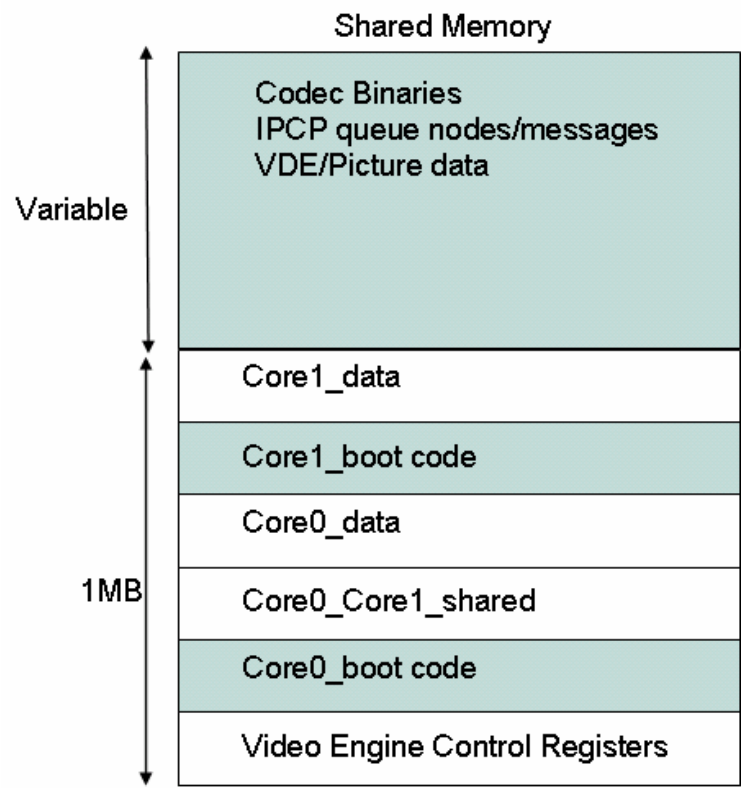
2 Sample Video System and Memory Map

This application note describes example control code based upon a simple abstract video system comprised of a Diamond DC_B_330HiFi_be (Big Endian) processor, a shared system RAM, and the Diamond 388VDO Video Engine (from here on, the DC_330HiFi_be will be referred to as the *system controller* and the 388VDO will be referred to as the *video engine)*. The shared memory holds the Diamond Video Engine's boot and codec binary code. All control and video data is passed between the system controller and video engine through the shared RAM memory.

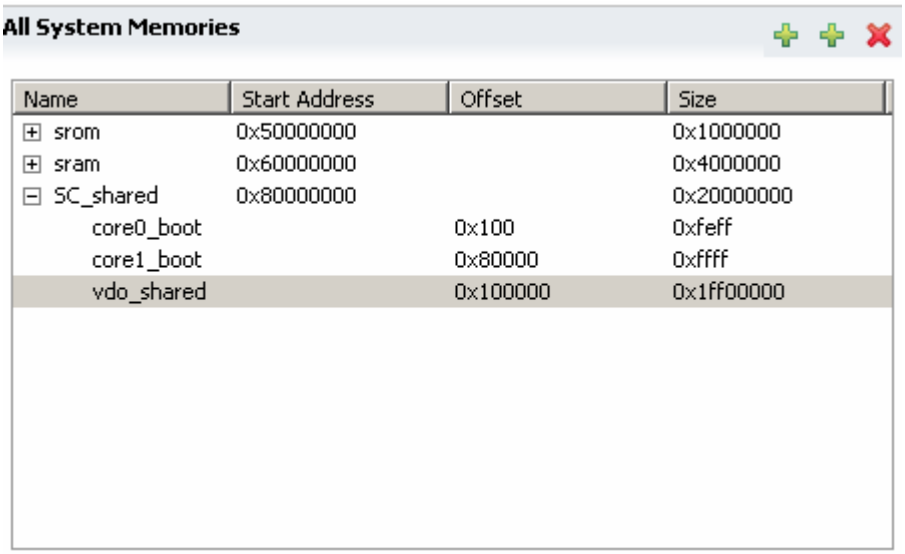


From the perspective of the video engine, the shared memory space is in a fixed physical address region of 0x60000000 to 0x7fffffff. From the perspective of the system controller, the shared memory space is mapped to 0x80000000 to 0x9fffffff. Although this sample system has 512MB of shared RAM memory, a typical design will require much less memory (a few MB may be sufficient for simple video systems). In a video system, this shared memory space is mapped to a shared DDR memory.

The first MB of the shared memory is the video engine memory space. This memory space is used to hold video engine control registers, boot code, and also serves as a working data space for the video engine. The remaining shared memory above 1MB is used to hold the codec binaries for the video engine and also serves to hold IPCP software queue nodes, along with picture/VDE (Video Decode Entities) data. The size of this shared memory varies depending on the codecs used, resolution, and buffering requirements. The shared memory space and memory segments are shown in following diagram (memory segment sizes are not to scale).



The memory map of shared memory is shown in the diagram above (refer to table 5.7 in the *Diamond 388VDO Software Guide* for a detailed memory map). Note that several memory segments of the shared memory are used for interaction between the system controller and the video engine (shaded in the diagram above). The Core1_boot and Core0_boot segments must contain the boot code for the video engine and are typically loaded by the system controller. For this example system, the system controller image links the video engine boot code binaries into core0 and core1 boot code space. The system controller must also manage the loading of video engine codec binaries, queue, and video data. In this example, a corresponding linker memory map from the perspective of the system controller is as follows:



In the core0_boot memory segment, a section named vdoboot0.data is defined with a memory address of 0x8000100. The bootloader for video engine core 0 (core0_main) will be linked to this section. Note that the memory sections are not shown in the preceding diagram.

In the core1_boot memory segment, a section named vdoboot1.data is defined with a memory address of 0x80080000. The bootloader for video engine core 1 (core1_main) will be linked to this section.

In the vdo_shared memory segment, a section named vdoshared.data is defined with a memory address of 0x80100000. Codec binaries, queue messages, and video data will be transferred in this memory section. Following the vdoshared.data section is a section named vdo_alloc.data that is allocated to the video engine by the system controller (more on this later).

The Xplorer workspace that accompanies this application note contains a modified diamond configuration target (derived from the DC_B_330HiFi_be processor) named SC_330HiFi with the memory map described above. This configuration target is used for all the examples provided in this application note.

3 Example 1: First Sign of Life

The first example will simply check that the video engine is alive and that the system controller can communicate with the video engine through shared memory. This test is one of the first verification tests that should be performed on a system with the video engine.

Prior to allowing the video engine to boot from reset, the boot code must be loaded into the core0_boot and core1_boot memory segments. There are various techniques that can be employed to transfer the boot code from ROM to the appropriate memory segments. In this example system, the system controller links the boot code binaries to the boot code segments. This is done by defining arrays for the boot code and assigning these arrays to be linked to the vdoboot0.data and vdoboot1.data segments.

The video engine software package provides the bootcode in elf-32 format. The xt-objcopy utility available in the Diamond Software toolkit converts this to binary format using the command shown below. Note that an Xtensa command line tool environment is needed to run xt-objcopy, however the selection of XTENSA_CORE is not relevant, and can be set to any Diamond core.

```
>xt-objcopy –O binary core0_main.out core0_main.bin
>xt-objcopy –O binary core1_main.out core1_main.bin
>xxd – i core0_main.bin > core0_main.h
>xxd – i core1_main.bin > core1_main.h
```

The Linux xxd utility is used to convert the binary into unsigned char array and saved into a header file. The header file is modified to link the array in a specific memory address. The code below shows how the core0 arrays are assigned to the vdoboot0.data segment. The same is done to link the core1 array to the vdoboot1.data segment.

```
#define VDO_BOOT0   __attribute__ ((section ("vdoboot0.data")))
unsigned char core0_main_bin[] VDO_BOOT0 = {
  0x60, 0x00, 0x1c, 0x00, 0x22, 0x22, 0x11,  .  .  .
```

Now, when the `core0_main.h` and `core1_main.h` header files are included in the software project, the boot code for the video engine will be linked to the vdoboot0.data and vdoboot1.data memory segments respectively.

```
#include "core0_main.h"
#include "core1_main.h"
#include "typedefs.h"
#include "sys_ctrl_data_org.h"
```

The Diamond Video Engine software installation contains a set of header files that expose queue access functions and video engine message definitions. **These header files are to be used in all Diamond Video engine control applications.** These header files are found in the following directory:

<Diamond Video Install Path>/388VDO/Software/System/SC/include

The examples provided in this application note include all IPC header files in the **ipc_queue** project. Therefore, all example projects will include the `ipc_queue` folder as part of the include path upon building.

The `sys_ctrl_data_org.h` header file contains definitions of video engine registers and the `typdefs.h` provides basic type definitions used throughout the example code.

The main example code is shown below.

```
main() {

System_Controller_Init();
Semaphore_Check();
printf("Video Engine is alive\n");
}
```

The system controller must first do necessary initialization. In this example, the system controller has a memory management unit. The memory management unit is initialized such that the shared memory region is uncached by using a XTHAL function. This will

assure that all writes to shared memory region are immediately visible to the video engine instead of being held in the system controller's cache memory.

```
System_Controller_Init() {
xthal_set_region_attribute(0x80000000, 0x20000000, 0, 0);
}
```

Afterwards, a video engine check is done by testing the video engine semaphore. The semaphore check routine writes out an 'S' character to the video engine semaphore register. When the video engine completes its boot sequence, it will check the semaphore register for 'S' and then return a 'V'. When the system controller notices the 'V', it can be certain that the video engine is executing the boot code and is alive. This sequence must be performed as the first initialization step prior to passing any further messages to the Diamond Video Engine.

```
Semaphore_Check() {
volatile UWORD32 *pu4_vpm_spm_sync_var;
pu4_vpm_spm_sync_var = (UWORD32 *)SPM_VPM_SYNC_VAR;
//Write ASCII S
*pu4_vpm_spm_sync_var = 'S';
while(1)
{
    //check for ASCII V
    if('V' == *pu4_vpm_spm_sync_var)
    break;
  }
//Write ASCII S
*pu4_vpm_spm_sync_var = 'S';
}
```

Note that the pointer for the video engine semaphore register (SPM_VPM_SYNC_VAR) is mapped as a volatile pointer. This assures that the compiler does not optimize away writes to this register or perform code scheduling optimizations that result in out-of-order accesses to the register. The video engine's SPM_VPM_SYNC_VAR register address, along with other video engine registers, are defined in `sys_ctrl_data_orgs.h` file shown below. This file has been simplified from the original header file by removing conditional code not relevant to this example. Ignore the TRNS_PTR2_OFF and TRNS_OFF_2_PTR macros for now (they will be covered later).

The SYSTEM_CTRL_DATA_START address is the base address of the shared memory with respect to the system controller. It is set to 0x80000000 for the XTVMP software model. Note that the SYSTEM_CTRL_DATA_START address must be user modified to work for the actual video system design.

```
#define SYSTEM_CTRL_DATA_START 0x80000000 /* sharedmem base address */


#define TRNS_PTR_2_OFF(x) ((WORD32)x - SYSTEM_CTRL_DATA_START)
#define TRNS_OFF_2_PTR(x) \
            ((void*)((WORD32)x + SYSTEM_CTRL_DATA_START))

#define VPSM_OFFSET_START (IPC_SHARED_SYSTEM_CTRL_REGISTER)
```

```
#define SPM_INT_CAUSE     (IPC_SHARED_SYSTEM_CTRL_REGISTER  + 0x0)
#define VPM_INT_CAUSE     (IPC_SHARED_SYSTEM_CTRL_REGISTER  + 0x4)
#define IPC_QUEUE_BLK_PTR (IPC_SHARED_SYSTEM_CTRL_REGISTER  + 0x8)
#define SPM_VPM_SYNC_VAR  (IPC_SHARED_SYSTEM_CTRL_REGISTER  + 0xc)
#define VPM_SESSION_ID    (IPC_SHARED_SYSTEM_CTRL_REGISTER  + 0x10)
```

The code for this example is provided in the accompanying Xplorer workspace as project VDO_EX1. This project can be built using the SC_330HiFi target. The next section describes how the operation of this example is simulated using the XTMP software system simulation model, XTVMP, that is provided with the Diamond Video Engine Software Tools.

4 System Simulation using XTVMP

The Diamond Video Engine Software Tools deliverable set includes a system simulation model, XTVMP (Xtensa Video Multi-Core Simulator) that is based on the XTMP (Xtensa Modeling Protocol) feature of Tensilica Processor cores. XTVMP has several uses:

- **Video Engine Evaluation Tool**: XTVMP is used in the evaluation stages to test for compatibility with various video streams (conformance testing) and understanding how system design choices, such as clock rate, memory bandwidth/latency affect video codec timing (profile testing).
- **Early Phase System Modeling**: XTVMP is created from a XTMP model of 388VDO. The source code for XTVMP includes the 388VDO Video Engine model, along with a DDR memory model, and a generic system controller (the Diamond DC_330HiFi processor is used). In the architecture phase of a video subsystem design, the XTVMP source can be modified to more closely resemble the actual target system. For example, the memory controller and model can be made truer to the target memory subsystem. Custom XTMP models of other system components can be added to provide closer modeling of end systems. This capability allows early system exploration by allowing designers to experiment how design choices affect system performance in software.
- **IPCP learning/testing platform**: XTVMP contains a generic system controller (a Diamond 330HiFi processor is used) to run the video engine driver. You can experiment with the Diamond Video Inter-Processor Control Protocol (IPCP) and video engine messages, create driver software in C language, and check for correct functionality.

  In this application note, we will experiment using XTVMP as an IPCP learning tool (c). You can create VDO driver code, compile it for the Diamond 330HiFi, and then easily simulate the system controller code on XTVMP.

  Before proceeding, you may need to build the XTVMP simulator on your platform. Refer to the *Diamond 388VDO Internals Guide* for instructions on building the XTVMP simulator. The following steps describe how you can build the first example and test it on XTVMP.

  1. Open any Diamond processor Shell (*i.e.,* Start Menu ->RB.2007.2 Diamond->DC_108mini CMD shell). Set the XTENSA_SYSTEM environment variable to point to the Diamond processor Registry path (*i.e.,* `C:\usr\xtensa\XtDevToolsDE\XtensaRegistry\RB-2007.2-win32`). Set the XTENSA_VIDEO environment variable to point to the Video installation path (*i.e.,* `C:\Tensilica\DC_388VDO`).

  2. In the same console, start Xplorer-DE

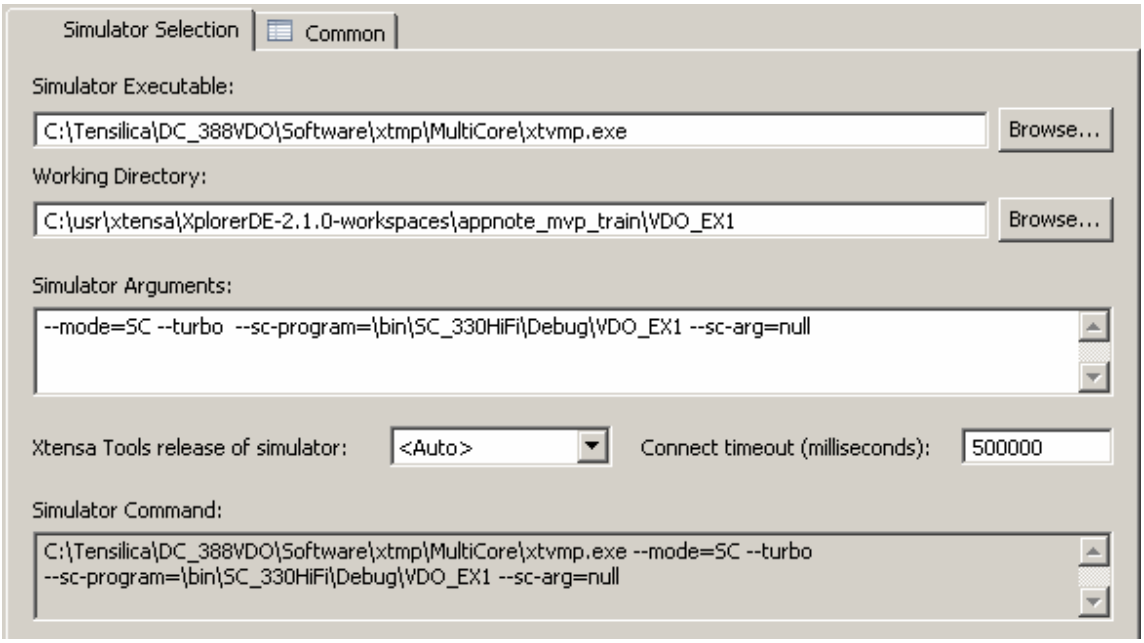
     ```
     >c:\usr\xtensa\XplorerDE-2.1.0\xplorer &
     ```

3. In Xplorer, import the VDO_training.xws workspace. Click on "Select All" for each dialog box to import everything needed for this tutorial.

4. Build the VDO_EX1 project using the **Debug** target for the **SC_330HiFi** configuration.

   Note the path of the binary that has been built. You can determine the path by finding the binary under the "Binaries" list in the VDO_EX1 project and then right-clicking on properties. Copy (CTRL+C) the path in the Locations Field in the properties dialog .

5. Click the Run pull-down menu and select Run... Then, click on MP Launch:VDO_EX1 on the left side of the Run dialog. A diagram of the dialog is shown below.



Notes:

- –mode=SC, sets the XTVMP operation mode for standard codec operation. In this mode, the XTVMP magically loads the core0_main.out and core1_main.out. So, it isn't necessary to include `core0_lib.h` and `core1_lib.h` files for the simulation to work. However, it is done redundantly in this example to illustrate what would be required on an actual system. The video engine bootloader header files are omitted in following examples.
- If you are using a Linux system, you will need to create an empty file named null in your VDO_EX1 project.
- The –turbo simulator argument speeds up simulation performance at the expense of cycle accuracy. If you remove the –turbo argument, the cycle counts reported on the console are valid estimates. Also, the memory bandwidth will be reported. Note that these estimates are based on a 32-cycle memory latency. Refer to the *Diamond 388VDO Internals Guide* for more information on performance/bandwidth analysis.

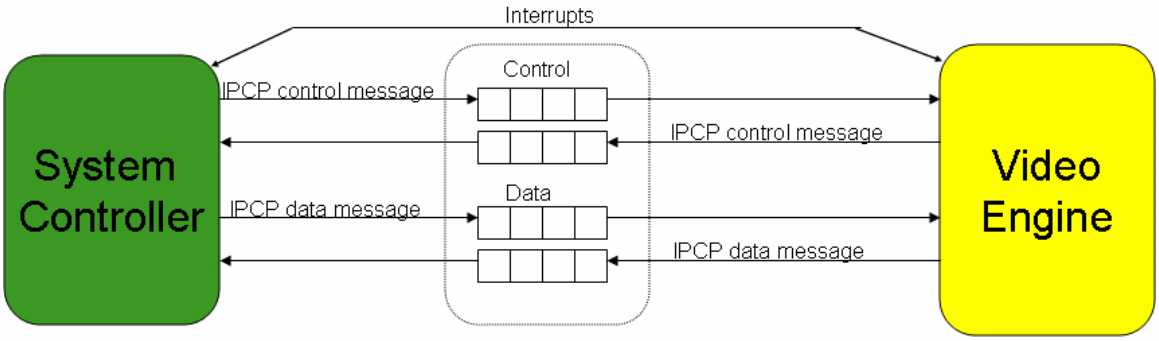
6. Set the Simulator Executable to the path of the XTVMP simulator and set the working directory to the directory of your workspace.

7. Click on the Run button on the bottom of the Run dialog. Note that it may take several seconds for the XTVMP simulator to start. Check that the console output is similar to what is shown below.

```
****VDO EXAMPLE 1 STARTED****
System_Controller_Init complete
Semaphore_Check complete
****VDO TEST FINISHED****
```

5 Example 2: Communicating with the Video Engine

The second example demonstrates the initialization of the IPCP and verifying communications with the video engine by sending a "PING" message. For the system controller to communicate with the video engine, it first needs to setup the IPCP software queues. The diagram below shows four software queues used to pass control/video messages between the system controller and the video engine. Once the IPCP queues are set up, the system controller can transfer control/data messages with the video engine across the queues.



The Diamond Video Engine Software Tools package is shipped with an IPCP library and a set of header files necessary to use IPCP functions. The IPCP library is provided in source code form (`ipc_queues.c`) so that it can be ported to any system controller. However, this code should not be modified. The IPCP library contains functions to initialize, check status, and transfer messages across the software queues. The IPCP functions are described in the *Diamond 388VDO Software Guide*.

The Xplorer workspace that accompanies this application note contains a library project named ipc_queue. The Xplorer software project for this example (and the examples that follow) sets library dependencies such that the ipc_queue library is linked along with the application binary (Library Dependencies are set in Xplorer by right clicking on the project name in the C/C++ projects view and selecting properties/Library Dependencies). The ipc_queue project also contains all IPC header files necessary to building video system controller applications.

The example code statically defines four IPCP software queues contexts.

- spm_ctrl_queue – control message queue from system controller to video engine
- vpm_ctrl_queue – control message queue from video engine to system controller
- spm_data_queue – data message queue from system controller to video engine
- vpm_data_queue – data message queue from video engine to system controller

The queue contexts are instances of structure ipc_queue and are defined in the `ipc_queues.h` file (provided in the Diamond Video Engine software installation), and hold queue state information along with a pointer to a table of queue node pointers.

The video engine uses 2 control queues to share control information with the system controller. The video engine may process one or more video sessions. For example, the video engine may be set up to handle 2 sessions, one encode session and one decode session, in a teleconferencing system. Each video session has its own set of data queues. In this example, we will only use a single session with 2 data queues.

The example code defines three control or data nodes for each of the queues. The control nodes are instances of the ipc_ctrl_msg structure defined in `ipc_data_msgs.h` , and the data nodes are instances of the ipc_data_msg structures defined in `ipc_ctrl_msgs.h`.

Note that the queue contexts, their node pointer tables, and nodes are linked to the shared memory section, vdoshared.data. This is important because the video engine only has access to shared memory.

```
#include "ipc_queues.h"
#include "ipc_data_msgs.h"
#include "ipc_ctrl_msgs.h"

#define VDO_SHARED   __attribute__ ((section ("vdoshared.data")))

//Create Queues
ipc_queue spm_ctrl_queue  VDO_SHARED;
ipc_queue spm_data_queue  VDO_SHARED;
ipc_queue vpm_ctrl_queue  VDO_SHARED;
ipc_queue vpm_data_queue  VDO_SHARED;

//Create Queue Node pointer tables
void* spm_ctrl_queue_nodes[3] VDO_SHARED;
void* vpm_ctrl_queue_nodes[3] VDO_SHARED;
void* spm_data_queue_nodes[3] VDO_SHARED;
void* vpm_data_queue_nodes[3] VDO_SHARED;

//Create IPC data/control message nodes
ipc_ctrl_msg spm_ctrl[3] VDO_SHARED;
ipc_ctrl_msg vpm_ctrl[3] VDO_SHARED;
ipc_data_msg spm_data[3] VDO_SHARED;
ipc_data_msg vpm_data[3] VDO_SHARED;
```

This example defines three nodes for each of the queues. There are three entries in each of the queue node pointer tables that correspond to the three ipc_ctrl_msg/ipc_data_msg nodes for each of the queues.

Three nodes per queue are arbitrarily used in this example. However, a system may use more than three nodes for each of the queues to improve system performance.

The queue node pointer tables are initialized by assigning the address of each queue node to an entry of the queue node pointer table. This is done in a loop for all nodes of each of the queues.

```
//Initialize Queue pointer table
for (i=0; i<3 ;i++)
{
spm_ctrl_queue_nodes[i] = (void*)TRNS_PTR_2_OFF(&spm_ctrl[i]);
vpm_ctrl_queue_nodes[i] = (void*)TRNS_PTR_2_OFF(&vpm_ctrl[i]);
spm_data_queue_nodes[i] = (void*)TRNS_PTR_2_OFF(&spm_data[i]);
vpm_data_queue_nodes[i] = (void*)TRNS_PTR_2_OFF(&vpm_data[i]);

}
```

Note that a macro TRNS_PTR_2_OFF modifies the address for the queue pointer tables. Recall that the shared memory base address from the perspective of the video engine is always at 0x60000000, whereas the shared memory base address from the perspective of the system controller is 0x80000000. Since shared memory is mapped differently, you cannot simply pass pointers between the system controller and video engine. The video engine API requires that all shared memory references are provided as offsets to the base address of shared memory. The `sys_ctrl_data_orgs.h` header defines translation macros TRNS_PTR_2_OFF (translate pointer to offset) and TRNS_OFF_2_PTR (translate offset to pointer) to perform pointer/offset translation.

Whenever the system controller passes shared memory pointers to the video engine, the pointer must first be converted to an address offset relative to the base of shared memory (0x80000000). For example, if the system controller passes a memory address of 0x80100000 to the video engine, the address must be translated to an offset of 0x00100000 (0x80100000 – 0x80000000).

The `sys_ctrl_data_orgs.h` also has a TRNS_OFF_2_PTR macro used to translate an offset from the perspective of the video engine to a pointer from the perspective of the system controller. The TRNS_OFF_2_PTR macro will be used in later examples.

Once the queue pointer tables are initialized, the queue contexts are initialized using the ipc_queue_init function from the IPCP library. This function initializes the queue context specified in the first argument, assigns the node pointer table from the second argument, and specifies the number of nodes in the third argument.

```
//Initialize Queue Contexts
ipc_queue_init(&ps_spm_ctrl_queue, spm_ctrl_queue_nodes, 3);
ipc_queue_init(&ps_spm_data_queue, spm_data_queue_nodes, 3);
ipc_queue_init(&ps_vpm_ctrl_queue, vpm_ctrl_queue_nodes, 3);
ipc_queue_init(&ps_vpm_data_queue, vpm_data_queue_nodes, 3);
```

Before IPCP messages can be passed on the control queues, the video engine needs to be informed about where the control queue contexts are in memory. The video engine has a register, IPC_QUEUE_BLOCK_PTR that must be initialized by the system controller that points to a queue block structure in shared memory. The queue_block_structure is shown below.

```
typedef struct _spm_queues_block
{
   ipc_queue *ps_spm_ctrl_queue;
    ipc_queue *ps_vpm_ctrl_queue;
} spm_queues_block;
```

```
//create queue block structure for VDO
spm_queues_block queue_block_structure VDO_SHARED;
```

The queue_block_structure contains pointers to the control queue contexts and is statically linked in the vdoshared.data memory section. The queue_block_structure containing pointers to the queue contexts defined previously. The queue_block_structure is initialized in the code below.

```
//initialize control queue block structure for VDO
queue_block_structure.ps_spm_ctrl_queue=(void*)TRNS_PTR_2_OFF(&ps_spm_
ctrl_queue);
queue_block_structure.ps_vpm_ctrl_queue=(void*)TRNS_PTR_2_OFF(&ps_vpm_
ctrl_queue);
```

Once the queue block structure is initialized, the offset to the queue block structure is set to the video engine's IPC_QUEUE_BLK _PTR register in the code below.

```
volatile UWORD32 *pu4_ipc_queue_blk;
pu4_ipc_queue_blk = (UWORD32 *)IPC_QUEUE_BLK_PTR;
*pu4_ipc_queue_blk = TRNS_PTR_2_OFF(&queue_block_structure);
```

Once the queues are set up, the system controller can perform the semaphore check described in example 1. When the semaphore check is complete, the system controller can begin passing control messages to the video engine. The code below shows an example of an IPC_PING message passed to the video engine through the IPCP queue.

```
#include "ipc_ctrl_msgs.h"

ipc_ctrl_msg *spm_control_message;

//memory for ping messages
char src_ping_msg[] VDO_SHARED = "Hello World";
char dst_ping_msg[sizeof(src_ping_msg)] VDO_SHARED;

int size_ping_msg = sizeof(src_ping_msg);

//get a free node from spm control queue
spm_control_message=ipc_queue_get_free_node(&spm_ctrl_queue);

//write ipc_ping message into node
spm_control_message->e_msg_code = IPC_PING;
spm_control_message->u4_size = sizeof(ipc_ping);
ipc_ping *ping_message =
            &spm_control_message->s_ipc_ctrl_msg.s_ipc_ping;

ping_message->u4_size=size_ping_msg;
ping_message->pv_src_msg = (void*)TRNS_PTR_2_OFF(src_ping_msg);
ping_message->pv_dst_msg = (void*)TRNS_PTR_2_OFF(dst_ping_msg);
```

```
//produce node to VDO
ipc_queue_set_node_produced(&spm_ctrl_queue);
//Drive spm message interrupt
spm_interrupt_vpm(CTRL_MSG_FROM_SPM);
```

First, the system controller obtains a free node from the spm_ctrl_queue using the ipc_queue_get_free_node function. The function returns a free node and then the system controller writes an IPC message to the node. The IPC message consists of the message code (IPC_PING), the size of the message payload and the message payload itself. The `ipc_ctrl_msgs.h` file and the *Diamond 388VDO Software Guide* provide a complete listing and description of IPC messages, along with descriptions of the corresponding structures that define the message payload.

In the example code, the ping_message pointer points to the IPC_PING message payload section of the IPC message. The s_ipc_ctrl_msg is a union of all the control message payload structures, of which the s_ipc_ping structure is a structure associated with the IPC_PING message. The IPC_PING message is initialized with an offset to a ping message in shared memory (src_ping_msg) and an offset to a destination for which the video engine will write back the ping message (dst_ping_msg) .

Once the IPC message is written to the node, the system controller "commits" the node back to the queue using the ipc_queue_set_node_produced function. The system controller must also alert the video engine of the new control message by asserting an interrupt with the spm_interrupt_vpm function shown below.

```
#define SCINTGENADDR0                          0xa0000000

void spm_interrupt_vpm(UWORD32 u4_int_cause)
{
   UWORD32 *pu4_int_cause_register = (UWORD32 *)SPM_INT_CAUSE;
   *pu4_int_cause_register = u4_int_cause;

#pragma flush
   {
      UWORD32 *pu4_sc_intgen_addr = (UWORD32 *)SCINTGENADDR0;
      *pu4_sc_intgen_addr = 1;
   }
}
```

The video engine SPM_INT_CAUSE register is set to CTRL_MSG_FROM_SPM (enum type defined in `sys_ctrl_data_org.h` that signals a control message interrupt from the system controller). The XTVMP system model has an interrupt pulse generator register, SCINTGENADDR0, assigned to the system controller's physical address space at 0xa0000000 (defined in `spm_interrupts.h`). When the system controller writes a '1' to this register, an interrupt pulse is generated to the video engine.

The code example above shows the use of #pragma flush to prevent the XCC compiler from scheduling the assertion of the interrupt before writing the SPM_INT_CAUSE register.

When the video engine receives the IPC_PING message:

- The video engine responds by copying the ping message from src_ping_msg to dst_ping_msg, and then generates an IPC_PING_ACK message on the vpm_ctrl_queue.

- The ipc_queue_get_node_produced function blocks on the vpm_ctrl_queue until a node is produced by the video engine, and then returns the message node. After which, the ping message “Hello World” is displayed on the console.
- Finally, the ipc_queue_set_node_consumed function releases the message node for use by the video engine. This sequence is shown in the code below.

```
//get ipc_ping_ack message from VDO

vpm_control_message=ipc_queue_get_node_produced(&ps_vpm_ctrl_queue);

printf("\nVPM CTRL message = %x \n", vpm_control_message->e_msg_code);
printf(" Ping message = %s \n", dst_ping_msg);

//Mark the VPM message as consumed
ipc_queue_set_node_consumed(&ps_vpm_ctrl_queue);

//clear interrupt line
check_clear_vpm_int();
```

When a control message is produced by the video engine, it signals the system controller through an interrupt. This example does not use interrupts, but instead uses the ipc_queue_get_node_produced to block until the control message is available. Even though the interrupt is ignored by the system controller, the interrupt needs to be cleared. The check_clear_vpm_int function shown below clears the video engine interrupt by writing 0 to the VPM_INT_CAUSE register. Note that the video engine interrupt cause is returned by this function, but it is not used in this example.

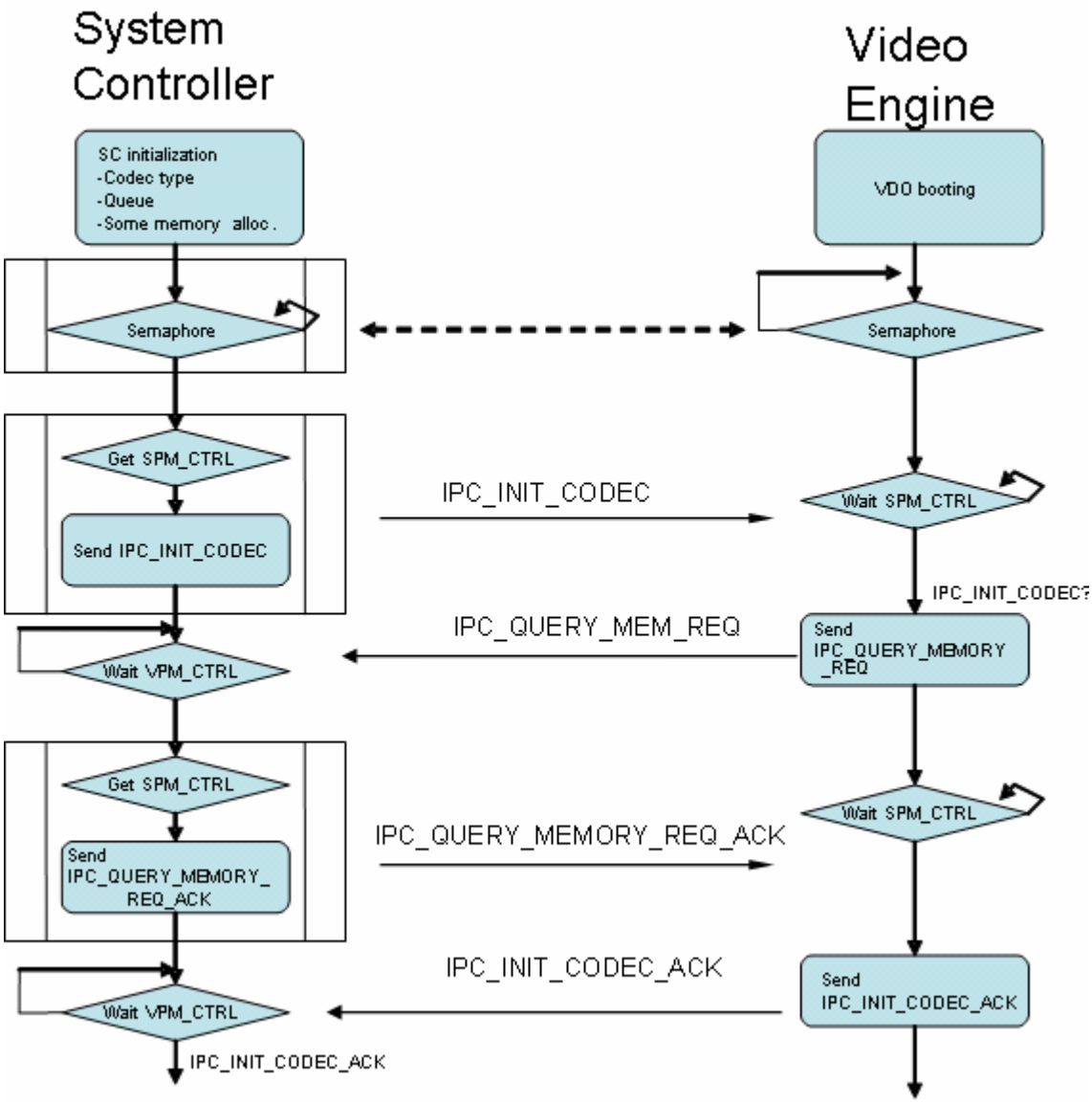
```
UWORD32 clear_vpm_int()
{
//clear interrupt line
volatile UWORD32 *pu4_int_var;
UWORD32 temp;
pu4_int_var = (UWORD32 *)VPM_INT_CAUSE;
temp=*pu4_int_var;
*pu4_int_var=0;
return (temp) ;
}
```

The code for this example is provided in the accompanying Xplorer workspace as project VDO_EX2. This project can be built using the SC_330HiFi configuration target and simulated as described in section 4. The reader is advised to build the project, step through the code using the XplorerDE debugger while examining code variables, to better understand this example.

6 Example 3: MPEG-4 Decoder Initialization

Once the bootloader has been loaded and control queues are set up, the system controller is able to send/receive control messages to the video engine, the video engine can be initialized to process video. In this example, an MPEG-4 decoder will be initialized. Other codecs (H.264, MPEG-2, VC1) are initialized in the same manner.

The codec initialization is described in the following diagram.



After the software queues and semaphore check described in the previous example is complete, the system controller starts initialization of the video engine by passing a IPC_INIT_CONTROL message to the video engine. The payload of the IPC_INIT_CONTROL message contains offsets to the codec binaries in the shared memory space, offsets to the data queues, and video processing parameters associated with the video processing session.

The codec binaries for the video engine are linked into the shared memory region using the same technique as used for the video engine boot code in the first example. The codec binaries are delivered in binary format (as opposed to elf-32) so no conversion is needed. The Linux xxd utility is used to create unsigned char arrays from the binary files and they are assigned to the vdoshared.data section using the code similar to what is shown below.

```
#define VDO_SHARED   __attribute__ ((section ("vdoshared.data")))
unsigned char mpeg4_lib0_o[] VDO_SHARED = {
  0x7f, 0x45, 0x4c, 0x46, 0x01, 0x02, 0x01, . . .
unsigned char mpeg4_lib1_o[] VDO_SHARED = {
  0x7f, 0x45, 0x4c, 0x46, 0x01, 0x02, 0x01, . . .
```

The formatting and sending of the IPC_INIT_CODEC message is shown in the code below. The IPC_INIT_CODEC message payload initializes offsets to the mpeg4 binary images, sets the session number to zero, and initializes offsets to the data queue contexts for this session. The IPC_INIT_CODEC message payload also contains settings such as maximum resolution, buffering, *etc*. Note that the u4_xtra_display_buffers is set to the same number of nodes available in vpm_data_queue.

```
ipc_ctrl_msg *spm_control_message;

//get a free node from spm control queue
spm_control_message=ipc_queue_get_free_node(&spm_ctrl_queue);

//write ipc_init_codec message into node
spm_control_message->e_msg_code = IPC_INIT_CODEC;
spm_control_message->u4_size = sizeof(ipc_init_codec);

ipc_init_codec *init_p =
          &spm_control_message->s_ipc_ctrl_msg.s_ipc_init_codec;
init_p->e_codec_type = IPC_MPEG4_DEC;
init_p->pv_codec0_image = (void*)TRNS_PTR_2_OFF(mpeg4dec_lib0_o);
init_p->pv_codec1_image = (void*)TRNS_PTR_2_OFF(mpeg4dec_lib1_o);
init_p->u4_session_id = 0;
```

```
ipc_dec_cfg_params *init_dec_p =
                &init_p->s_init_codec_prms.s_dec_cfg_params;
init_dec_p->u4_max_image_width = 176;
init_dec_p->u4_max_image_height = 144;
init_dec_p->u4_max_level_num = 0;
init_dec_p->ps_ipc_vde_queue =
                    (void*)TRNS_PTR_2_OFF(&ps_spm_data_queue);
init_dec_p->ps_ipc_disp_queue =
                    (void*)TRNS_PTR_2_OFF(&ps_vpm_data_queue);
init_dec_p->u4_display_queue_dir = 0;
init_dec_p->u4_xtra_display_buffers = NODES;
init_dec_p->u4_display_interrupt_on = 0;
init_dec_p->e_yuv_format = IPC_YUV_420_UV_INTERLEAVE_H;
```

Once the spm_control_message is written with the IPC_INIT_CODEC message, it is sent to the video engine using ipc_queue_set_node_produced() and by driving the spm message interrupt. When the video engine consumes the message node, it will mark the node as consumed. The system controller blocks until the message is consumed using ipc_queue_check_node_consumed and then frees the node using ipc_queue_set_node_released so that the node may be reused.

```
//produce node to VDO
ipc_queue_set_node_produced(&ps_spm_ctrl_queue);
//Drive spm message interrupt
spm_interrupt_vpm(CTRL_MSG_FROM_SPM);
//block until node is consumed
while (ipc_queue_check_node_consumed(&ps_spm_ctrl_queue)==0) {}
//free consumed node
ipc_queue_set_node_released(&ps_spm_ctrl_queue);
```

The video engine loads the codec binaries and determines the video processing memory requirements from the video processing parameters provided in the IPC_INIT_CONTROL message. The video engine sends its memory requirements in an IPC_QUERY_MEM_REQ message to the system controller. The payload of this message contains the number of memory blocks requested, and a pointer to a MemRec structure. This structure contains size/alignment requirements for each of the memory blocks requested.

```
UWORD32 mem_blocks_requested;
MemRec *BlockTable;
ipc_ctrl_msg *vpm_control_message;

//expect to get IPC_QUERY_MEMORY_REQ message
vpm_control_message=ipc_queue_get_node_produced(vpm_ctrl_queue);

mem_blocks_requested = vpm_control_message->
      s_ipc_ctrl_msg.s_ipc_query_mem_req.u4_mem_tab_requests;
printf("\nVDO requests #%d memory blocks \n",mem_blocks_requested);
```

```
BlockTable = TRNS_OFF_2_PTR(vpm_control_message->
             s_ipc_ctrl_msg.s_ipc_query_mem_req.ps_ext_mem_rec);

check_clear_vpm_int();
//Mark the VPM message as consumed
ipc_queue_set_node_consumed(&ps_vpm_ctrl_queue);
```

The system controller is responsible for allocating the requested memory blocks from shared memory and populating the pv_base fields of the MemRec structure with the offset for each of the requested memory blocks.

A system typically uses a custom malloc routine to allocate memory from the shared memory space. However, in this simple example, the working memory for the video engine is allocated from the vdoalloc.data section. A dummy variable (codec_working_memory_start) is declared in this section. The address of this dummy variable will be used as the start of the codec working memory region.

```
#define VDO_ALLOC   __attribute__ ((section ("vdoalloc.data")))
#define ALIGNED16   __attribute__ ((aligned(16)))

//codec alloc working memory space start – 16byte aligned
UWORD8 codec_working_memory_start VDO_ALLOC ALIGNED16;

UWORD32 mem_size, mem_alignment, mem_attribute;
UWORD8 *mem_ptr;
UWORD32 mem_alloc_ptr=0;

for (i=0;i<mem_blocks_requested;i++) {

//extract size alignment attribute from BlockTable
mem_size = BlockTable[i].u4_size;

mem_ptr = (void*)TRNS_PTR_2_OFF(&codec_working_memory +
mem_alloc_ptr);
BlockTable[i].pv_base = mem_ptr;
//update mem alloc ptr and force 16 byte alignment
mem_alloc_ptr = mem_alloc_ptr + mem_size;
while ((mem_alloc_ptr % 16)!=0) mem_alloc_ptr++;
}
```

After the MemRec structure is populated with the allocated memory offsets, the system controller passes a IPC_QUERY_MEMORY_REQ_ACK message to the video engine. The formatting and setting of the IPC_QUERY_MEMORY_REQ_ACK message is shown below.

```
//send IPC_QUERY_MEMORY_REQ_ACK
spm_control_message=ipc_queue_get_free_node(&ps_spm_ctrl_queue);
spm_control_message->e_msg_code = IPC_QUERY_MEMORY_REQ_ACK;
spm_control_message->u4_size = sizeof(ipc_query_memory_req_ack);
```

```
ipc_query_memory_req_ack *mem_ack_message =
      &spm_control_message->s_ipc_ctrl_msg.s_ipc_query_mem_req_ack;
mem_ack_message->e_status = IPC_SUCCESS;
mem_ack_message->u4_mem_tab_requests = mem_blocks_requested;
mem_ack_message->ps_ext_mem_rec = (void*)TRNS_PTR_2_OFF(BlockTable);

//produce and release ctrl node
ipc_queue_set_node_produced(&ps_spm_ctrl_queue);
spm_interrupt_vpm(CTRL_MSG_FROM_SPM);
while (ipc_queue_check_node_consumed(&ps_spm_ctrl_queue)==0) {}
ipc_queue_set_node_released(&ps_spm_ctrl_queue);
```

Finally, the video engine sends an IPC_INIT_CODEC_ACK message to signal that initialization is complete and that the video engine is ready to start processing video.

```
vpm_control_message=ipc_queue_get_node_produced(&vpm_ctrl_queue);

//expect to get IPC_INIT_CODEC_ACK message #21
printf("Got vpm ctrl message %x\n",vpm_control_message->e_msg_code);

//Mark the VPM message as consumed
check_clear_vpm_int();
ipc_queue_set_node_consumed(&vpm_ctrl_queue)
```

This code example results in the message “Got vpm ctrl message 21” (the control message codes are found in the `ipc_ctrl_msgs.h` file). Now the Diamond Video Engine is ready to decode MPEG-4 video.

The code for this example is provided in the accompanying Xplorer workspace as project VDO_EX3. This project can be built using the SC_330HiFi configuration target and simulated as described in section 4. The reader is advised to build the project to better understand this example.

7 Example 4: Decoding MPEG-4 Video

Once the Diamond Video Engine has been completely initialized for MPEG-4 decode, an MPEG-4 video bitstream data can be sent to the video engine. This example explains the system controller code necessary to decode MPEG-4 video with the video engine. Much of this example is applicable to decoding video using other supported coding standards; however, the VDE formatting requirements (discussed later) will be significantly different from one codec standard to another.

In this example code, the compressed MPEG-4 video bitstream file is converted to a unsigned char array named video_stream using the Linux xxd utility.

```
unsigned char video_stream[] VDO_SHARED = {
  0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x01, 0x20, 0x00, 0x84, 0x40,
0x03, . . .
```

The main loop for decoding the MPEG-4 video stream is a while loop composed of three parts. Part A produces video stream data to the spm_data_queue if there is a free node on this queue and releases any consumed node on the queue. Part B reads a message from the vpm_ctrl_queue if there is a node available on this queue. Part C consumes a picture from the vpm_data_queue if there is a node available on this queue. This loop continues until there is no longer any remaining video stream data. The main loop and Part A for this example is shown below.

```
UWORD8 *ptr_video = video_stream;
UWORD32 video_stream_bytes = sizeof(video_stream);
UWORD32 vde_size;
UWORD32 vde_offset=0;
UWORD32 vde_sent=0;
UWORD32 frames_received=0;
UWORD32 finished = 0;

while (1)
{ //PART A:  Produce VDEs
   //get a free node (non-blocking) from spm data queue
   spm_data_message=ipc_queue_get_free_node(&ps_spm_data_queue);
   //if a node is available fill it with VDE
   if ((spm_data_message) && !finished)
      {
      //find vde size in video_stream
        vde_size=find_vde_size(ptr_video, video_stream_bytes,
vde_offset);
       //Send SPM_DATA message if there is a vde
       if (vde_size!=0)
         {
         spm_data_message->e_msg_code = IPC_DEC_VDE_DATA;
         spm_data_message->u4_size = sizeof(ipc_dec_vde_data);
         spm_data_message->u4_pack_num = vde_sent;

         ipc_dec_vde_data *vde_message =
               &spm_data_message->s_ipc_data_msg.s_ipc_dec_vde;
         vde_message->pu1_vde_start_pos =
                  (void*)TRNS_PTR_2_OFF(ptr_video+vde_offset);
         vde_message->u4_vde_length = vde_size;
         vde_message->u4_vde_time_stamp_hi = 0;
         vde_message->u4_vde_time_stamp_low = vde_sent;

         ipc_queue_set_node_produced(&ps_spm_data_queue);
         //increment for next vde
         vde_offset+=vde_size;
         vde_sent++;
         }
   //else end of file
   else {
```

```
      //Send Zero Byte VDE
      ipc_dec_vde_data *vde_message =
            &spm_data_message->s_ipc_data_msg.s_ipc_dec_vde;
      vde_message->u4_vde_length = 0;
      ipc_queue_set_node_produced(&ps_spm_data_queue);

      //Close codec to flush out remaining pictures
      //send IPC_CLOSE_CODEC
      spm_control_message=ipc_queue_get_free_node(&ps_spm_ctrl_queue);
      spm_control_message->e_msg_code = IPC_CLOSE_CODEC;
      spm_control_message->u4_size = sizeof(ipc_close_codec);

      ipc_close_codec *close_message =
            &spm_control_message->s_ipc_ctrl_msg.s_ipc_close_codec;
      close_message->u4_process_pending_data_msgs = 1;
      close_message->u4_session_id = 0;

      //produce and release ctrl node
      ipc_queue_set_node_produced(&ps_spm_ctrl_queue);
      spm_interrupt_vpm(CTRL_MSG_FROM_SPM);

      finished=1;
      }
   }

//Release consumed node from spm_data_queue
   ack_data_message=ipc_queue_check_node_consumed(&ps_spm_data_queue);
   if (ack_data_message)
   {ipc_queue_set_node_released(&ps_spm_data_queue);}
. . .
```

In part A of the code, the spm_data_queue is checked for a free node using ipc_queue_get_free_node. If there is a free node, video stream data is provided to the video engine. However, the video engine cannot take the video stream in one contiguous block of memory. Instead, the bitstream needs to be provided in separate blocks referred to as Video Decode Entities (VDEs). The definition of a VDE is described in the Diamond 388VDO Software Guide and differs for each video standard.

For this example, we have an input array holding the elementary stream as a contiguous bitstream. A simple function, find_vde_size parses the array and returns the size of an MPEG-4 VDE by looking for MPEG-4 start code (0x000001) and returns the number of bytes until the start code is found. The function is shown below. The arguments to this function are a pointer to the video bitstream (ptr_video), the total size in bytes of the bitstream (video_stream_bytes), and the number of bytes previously consumed (vde_consumed_bytes).

```
UWORD32 find_vde_size(void* ptr_video, WORD32 video_stream_bytes,
UWORD32 vde_consumed_bytes)
{

UWORD8 *stream_ptr = ptr_video + vde_consumed_bytes;
```

```
UWORD32 consumed=0;

//advance stream pointer past start codes
//8 bytes seem to work as minimum VDE size
stream_ptr=stream_ptr+8;
consumed = 8;

//check for end of stream
   if ((vde_consumed_bytes + consumed) >=  video_stream_bytes)
   {
      //printf("\nEND OF STREAM\n\n");
      return 0;
   }

//find next start code
while(1){

//check for end of stream
   if ((vde_consumed_bytes + consumed) >=  video_stream_bytes)
   {
      return consumed;
   }

//check for new start code
   if ((stream_ptr[0]==0) && (stream_ptr[1]==0) &&
(stream_ptr[2]==0x01))
   {
         return consumed;
   }
stream_ptr++;
consumed++;

  }
}
```

The vde_size function works in this simple MPEG-4 decode example, however is not suitable across a wide variety of video streams. Developers may reference the VDE extractor functions in the Diamond Video Software tools package (`\Software\System\SC\src\vde.c`) for examples of code used to extract VDEs for all of the supported decoder standards.

One final note on the VDEs: The video engine is particularly sensitive when it comes to processing VDEs. The video engine assumes that the system controller has correctly parsed the video bitstream and correctly formatted the VDEs (there is only minimal VDE checking performed on behalf of the video engine). If the formatting of VDEs is not done correctly, the video engine may not detect this, resulting in functional errors. Therefore, it is important to verify correct functionality of VDE formatting, by testing across a large variety of test streams.

Let's go back to Part A of the while loop, the vde_size function is used to parse the video stream and returns the size of the VDE. An offset to the video stream data is passed, along with the VDE size as part of the IPC_DEC_VDE_DATA message. If the end of the video stream is reached, a zero byte VDE is sent across the spm_data_queue. The zero byte VDE is a special VDE that signals the end of stream. The zero byte VDE will cause the

MPEG-4 decoder to flush any remaining reference pictures. Then the IPC_CLOSE_CODEC message is sent to end the decode session.

The end of Part A shows the spm_data_queue is checked for any nodes already consumed by the video engine. When the video engine consumes an IPC_DEC_VDE_DATA node from the spm_data_queue, it will write some acknowledgement information. The acknowledgement information includes picture type, error flags, bitstream bytes consumed, etc. Once the system controller reads the acknowledgement information, it is important for the system controller to release any consumed nodes so that they can be reused for new VDEs.

```
//PART B:  Check for VPM control messages
//If there is a node in vpm_ctrl_queue, consume the node
   if (ipc_queue_check_active_nodes(&ps_vpm_ctrl_queue)!=0)
      {
      UWORD32 ctrl_msg_code;
  vpm_control_message=ipc_queue_get_node_produced(&ps_vpm_ctrl_queue);
   ctrl_msg_code = vpm_control_message->e_msg_code;
      printf("\nRecieved VPM_CTRL_MSG %x\n",ctrl_msg_code);
      //Mark the VPM message as consumed
      check_clear_vpm_int();
      ipc_queue_set_node_consumed(&ps_vpm_ctrl_queue);
      if (ctrl_msg_code==0x24) break;
      }
```

Part B simply checks the vpm_ctrl_queue for messages from the video engine and prints these messages. If an IPC_CLOSE_CODEC_ACK message (0x24) occurs, it sets the variable closed to one, and causes an exit out of the while loop.

```
//PART C:  Get Decoded Frames
//If there is a node in vpm_data_queue, consume the node
   vpm_data_message=ipc_queue_check_node_produced(&ps_vpm_data_queue,
1);
   if (vpm_data_message)
      {

   printf("\nFRAME %d RECEIVED\n",frames_received++);
      //NOTE THE TRANSLATION FROM OFFSET TO POINTER NEEDED FOR SYSTEM
CONTROLLER
      unsigned char *ptr_pic= (unsigned char*)
         TRNS_OFF_2_PTR(vpm_data_message-
         >s_ipc_data_msg.s_ipc_dec_pic.pu1_buf1);
      printf("IPC_VDE_DATA Y buffer pointer = %x \n", ptr_pic);
      printf("IPC_VDE_DATA message corrupt flag = %d \n",
vpm_data_message->s_ipc_data_msg.s_ipc_dec_pic.s_display_info.
u4_data_corrupt_flag);

      ipc_queue_set_node_consumed(&ps_vpm_data_queue);
      check_clear_vpm_int();

} //END OF WHILE LOOP
```

Part C checks the vpm_data_queue for IPC_PICTURE_DATA messages and prints a message about the generated picture. Note the use of the TRNS_OFF_2_PTR macro to convert the shared memory offset to a pointer for the system controller. In this simple example, the picture data is simply discarded.

In a typical video system, the system controller would take image pointers from the IPC_PICTURE_DATA message and send the pointers to a display controller in the order received. The display controller must copy the image data from shared memory space before the system controller issues the ipc_queue_set_node_consumed command. Once the node is consumed, the corresponding video picture buffers are released back to the video engine for reuse.

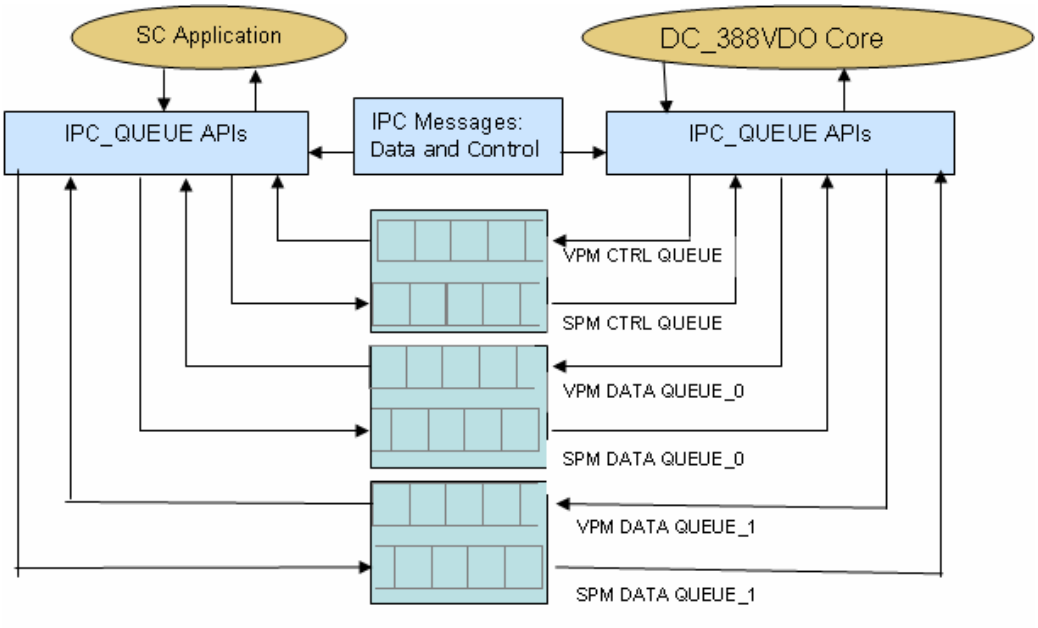
The while loop described in this section will iterate until there are no further VDEs to be decoded (vde_size==0). Part A of the while loop shows that the IPC_CLOSE_CODEC message is sent to the video engine through the spm_ctrl_queue. The video engine will respond the the IPC_CLOSE_CODEC message with an IPC_CLOSE_CODEC_ACK message. In Part B of the while loop, shows that the controller breaks out of the while loop when it receives the IPC_CLOSE_CODEC_ACK (ctrl_msg_code==0x24). After the while loop, the controller should flush all remaining pictures from the vde_data_queue.

The code for this example is provided in the accompanying Xplorer workspace as project VDO_EX4. This project can be built using the SC_330HiFi configuration target and simulated as described in section 4. The reader is advised to build and simulate the project to better understand this example.

8 Example 5: Multi-Instance Decoder

Some video applications require simultaneous execution of video codecs. For example, a video conference system requires simultaneous video encoding and decoding. Likewise, a digital television with picture-in-picture support may decode two video streams. The Diamond Video Engine IPCP has been designed to support multiple instances of video codecs that run simultaneously.

To handle multiple codec instances, the IPCP allows for multiple data queues. The following diagram shows the IPCP communication block diagram of a system that supports two decoders. One set of control queues are shared between both decoders, while each decoder has a dedicated set of data queues.



The video engine (DC_388VDO) core program automatically switches between the two codec instances when there are no further VDEs to decode on the current instance. The system controller may also send an IPC message to the video core program to force a codec switch.

It is important to note that each codec instance must be initialized atomically starting from the IPC_INIT_CODEC message until the IPC_INIT_CODEC_ACK message is received. In other words, the SC application cannot partially initialize one codec instance, and then partially initialize the other codec instance.

The VDO_EX5 project provides an example of simultaneous decode of two MPEG-4 video streams. This code is based on the previous VDO_EX4 project with a simple duplication of data queues and codec initialization to support the second codec. The main loop (Parts A, B, and C of the previous section) are moved to a function named codec_0_iteration. This function is duplicated to codec_1_iteration, to support the second codec instance. The main loop is shown below.

```
while ((closed_0==0) || (closed_1==0))
{
   if (codec_0_go && !closed_0) codec_0_iteration();
   if (codec_1_go && !closed_1) codec_1_iteration();

   if (((iteration_count++)%7000)==0)
   {
      if (codec_0_go) codec_0_go=0 ;else codec_0_go=1;
      if (codec_1_go) codec_1_go=0 ;else codec_1_go=1;
   }
}
```

While either of the codecs are active (not closed) either codec_0_iteration or codec_1_iteration function will be called. Let's say codec_0_go is set, then codec_0_iteration will be called. After 7000 calls of codec_0_iteration, the codec_1_iteration will be called, and codec instance zero will be starved of VDEs. After 7000 more iterations, the codec_0_iteration will be called, and codec instance one will be

starved of VDEs. And so on, until there are no more VDEs to process for either codec and both codecs are closed.

You may examine the complete VDO_EX5 project code, build, and simulate it using the VDO_EX5 launch. The output console output (excerpt) is shown below. The untabbed console output are from codec instance zero, while the un-tabbed console output is from codec instance one. This shows that the video engine core automatically switches to another codec instance when starved of VDEs.

```
STARTING MAIN DECODE LOOP
      **************FRAME 0 RECEIVED***************
      **************FRAME 1 RECEIVED***************
      **************FRAME 2 RECEIVED***************
      **************FRAME 3 RECEIVED***************
**************FRAME 0 RECEIVED***************
**************FRAME 1 RECEIVED***************
**************FRAME 2 RECEIVED***************
**************FRAME 3 RECEIVED***************
      **************FRAME 4 RECEIVED***************
      **************FRAME 5 RECEIVED***************
      **************FRAME 6 RECEIVED***************
      **************FRAME 7 RECEIVED***************
**************FRAME 4 RECEIVED***************
**************FRAME 5 RECEIVED***************
**************FRAME 6 RECEIVED***************
**************FRAME 7 RECEIVED***************
```

9 Debugging System Controller Code

If the video engine seems to be malfunctioning, chances are there is some error in the system controller code. This section covers some of the common errors and methods to mitigate these errors.

1) **Incomplete IPC messages:** A common error is failing to initialize some important element in the IPC message structure. It is good practice to initialize every element of an IPC message structure, even if it seems to be irrelevant.

2) **Ignored Error signals:** When a video processing error occurs, it may appear that the video engine has suddenly "hung". It is quite likely that the video engine has determined an error condition and has attempted to flag the system controller of the error condition. However if the system controller ignores these error flags, and continues sending the video engine additional IPC messages, the video engine may appear to "hang". These error flags include:

- IPC_ERROR (0x25) message received from video engine
- VPM interrupt with VPM_INT_CAUSE register equal to VPM_EXCEPTION_C0 (0x02) & VPM_EXCEPTION_C1 (0x03)
- Error flags in IPC messages
    - s_ipc_ctrl_msg.s_ipc_init_codec_ack.u4_error_code
    - s_ipc_data_msg.s_ipc_dec_vde.s_ipc_dec_vde_ack.u4_error_code
    - s_ipc_data_msg.s_ipc_dec_pic.s_display_info. u4_data_corrupt_flag

Therefore, it is imperative that system controller code listen for these error flags to allow the system to alert the developer of such problems and exit gracefully if such problems occur in deployed systems.

3) **Memory Access Errors:** If the video engine appears to "hang" without raising any error flags the source of the problem may be due to problems in memory access. Examples of these errors include:

- Incorrect usage or failing to use memory translation macros (TRNS_PTR_2_OFF & TRNS_OFF_2_PTR).
- Programming errors in the allocation of shared memory during IPC_MEMORY_REQ
- Hardware design errors in the bus interfacing logic of the video engine and system controller
- Incorrect byte reordering in the case of systems that use a little endian system controller (the Diamond Video Engine is big endian)

To rule out hardware sources of memory access problems, system verification should include an exhaustive check of all shared memory regions and communication with the system controller.

4) **Improper VDE formatting:** The video engine requires that video elementary streams are parsed and formatted as specified by the *Diamond 388VDO Software Guide* (Section 5.5). Programmers should verify the formatting routines carefully, since even subtle errors in formatting will lead to decode errors.

Another effective aid to debugging system controller code is to generate IPC dumps to check the data of all IPC messages for correct values. The VDO_EX3 and VDO_EX4 examples contain conditional code to display the data of IPC messages. When these examples are compiled with #define IPC_DEBUG, the IPC message data is displayed.

```
#ifdef IPC_DEBUG
printf ("\nIPC_INIT_CODEC\n");
print_ipc_message((void*)spm_control_message, sizeof(ipc_ctrl_msg));
#endif
```

When compiled with IPC_DEBUG, the print_ipc_message function is called to dump the contents of the IPC message structure.

```
void print_ipc_message (UWORD8 *message_addr, int message_size)
{
   UWORD8 data;
   int i;
   int byte = 0;
   while (byte < message_size )
   {
      printf("%x:  ",(void*)TRNS_PTR_2_OFF(message_addr));
      for (i=0; i<16 ; i++)
         {
            data = *message_addr++;
      if (data < 0x10) printf("0%x ",data); else printf ("%x ",data);
            byte++;
         }
```

```
            printf("\n");
        }
    printf("\n");
}
```

Example output of IPC message dumps is shown below.

```
IPC_INIT_CODEC
10dff0:  00 00 00 4c 00 00 00 01 00 00 00 02 00 00 00 b0
10e000:  00 00 00 90 00 00 00 00 00 10 df 50 00 00 00 00
10e010:  00 10 df 90 00 00 00 01 00 00 00 01 00 00 00 01
10e020:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10e030:  00 00 00 00 00 00 00 00 00 10 00 00 00 10 9b 70
10e040:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

IPC_QUERY_MEMORY_REQ
10e0f0:  00 00 00 00 00 00 00 23 00 00 00 10 00 01 54 70
10e100:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10e110:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10e120:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10e130:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10e140:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

IPC_QUERY_MEMORY_REQ_ACK
10e044:  00 00 00 0c 00 00 00 03 00 00 00 00 00 00 00 10
10e054:  00 01 54 70 00 00 00 00 00 00 00 00 00 00 00 00
10e064:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10e074:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10e084:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10e094:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

IPC_INIT_CODEC_ACK
10e144:  00 00 00 10 00 00 00 21 00 00 00 00 00 00 00 02
10e154:  00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 00
10e164:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10e174:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10e184:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10e194:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

It is good practice to use a similar means to generate IPC dumps in your system control code. If you should require assistance with debugging codec problems, please generate IPC dumps and send them along with your support enquiry.

5) Incompatible library/boot/codec versions: It is important to make sure that the IPC code/headers, the boot code, and codecs are from the same release. It is possible to mistakenly use IPC headers from previous releases with boot code and codecs from more current releases (or vice-versa). These types of mismatch will likely cause malfunctions.

10 Conclusion

This application note serves as a tutorial to the Diamond Video Engine API and IPCP software queues used to develop Diamond Video Engine control code. The examples described in this application note are provided in an accompanying XplorerDE workspace, `VDO_examples.xws`. Upon understanding, compiling, and simulating the examples, you can experiment with the examples to learn more about the functionality of the Diamond Video Engine. You are encouraged to use the examples as a starting point towards developing code to control the Diamond Video Engine in your target system.

Happy coding!