Application Report
SPRAAF3–November 2006

TMS320DM64x Video Port to Video Port Communication

Neal Frager, Navaid Karimi, Bernard Thompson *Digital Customer Applications Team*

ABSTRACT

The video port is one of the essential features of the TMS320DM64x devices. This application report (1) covers the basics of video port communication and an implementation of communication between two DM64x video ports and (2) describes the hardware and software interfaces required to achieve communication between two DM64x video ports for BT.656-based video streams and RAW mode transfers. This will be helpful to those who are interested in the video port functionality and those who need a high-bandwidth communications interface on DM64x-based boards. The solution presented is ideal for anyone using multiple DM64x devices and looking for high-speed inter-processor communication.

Contents

List of Figures

List of Tables

1 Video Port Communication

This section covers the basics of video port communication, providing some of the background information required to implement a system where video ports are used to communicate with other video ports. The video port itself is designed specifically to communicate with other video devices such as video encoders and video decoders. Figure 1 shows a standard use of the video ports as seen on the DM642 Evaluation Module (DM642EVM). For additional information on the video port peripheral and the video port mini-driver, please see the documents listed in Section 4.

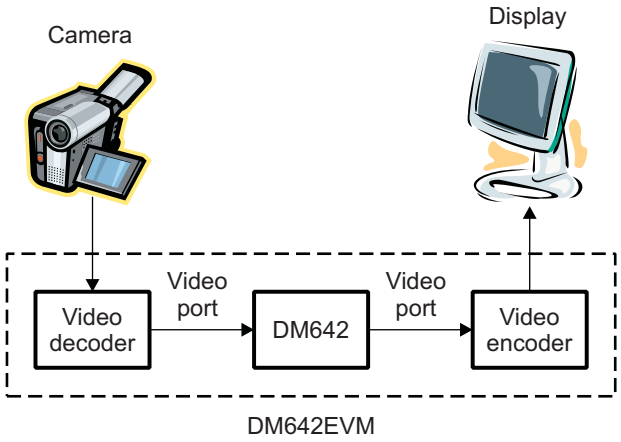

Figure 1. Basic Digital Video System

1.1 Video Port Hardware Architecture

The DM64x video port is a unidirectional, high-speed parallel interface designed primarily for communication with video encoders (video digital-to-analog converters) and decoders (video analog-to-digital converters). The DM64x has an advantage over other DSPs due to the video port, as this peripheral allows for easier video data transfer. It is more efficient than using the EMIF (external memory interface) and glue logic to capture and display video. The video port has up to 20 data lines (20-bit port); 3 control lines for synchronization; and 2 clock lines, one input and one output. Internally the video port contains a 5120-byte First In First Out (FIFO) memory which is serviced by the EDMA (enhanced direct memory access peripheral) only. A high-level block diagram is given in Figure 2. For additional details on the video port architecture, please see the *TMS320C64x DSP Video Port/VCXO Interpolated Control (VIC) Port Reference Guide* (SPRU629).

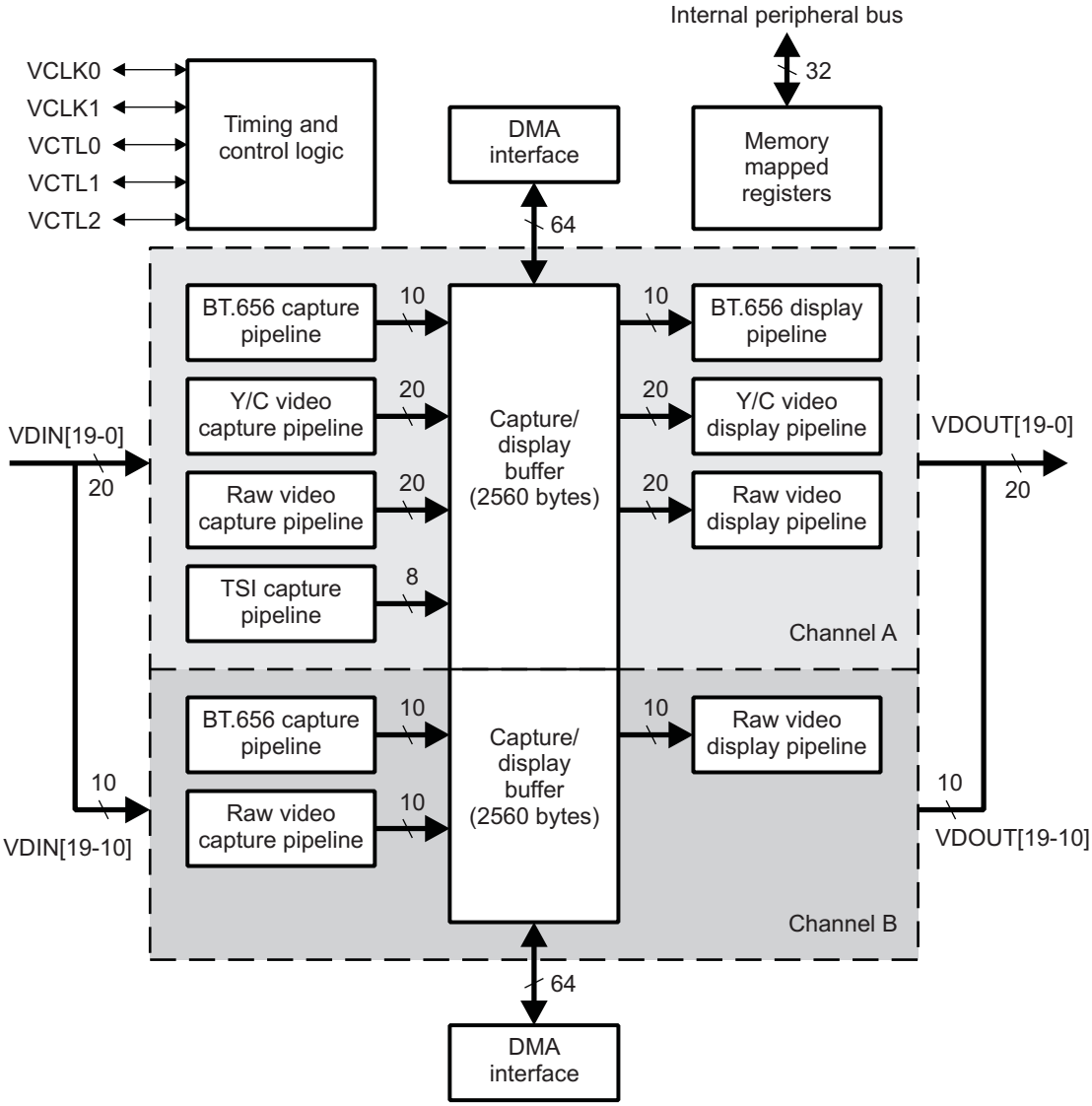

Figure 2. Video Port Block Diagram

1.2 Video Port Modes

The video port supports several modes of video transfer, including BT.656 and RAW. This application report discusses BT.656 and RAW mode transfers and how they can be used for communication between two DM64x devices.

1.2.1 BT.656

BT.656 is a format for sending video data digitally with the special feature of having embedded synchronization. Embedded synchronization allows the data to be transferred with no separate synchronization signals. BT.656 is the most common digital video standard used with many video encoders and decoders. Transferring digital video in Y/Cb/Cr format is rather simple with BT.656. A BT.656 video stream is composed of three types of data, which stream across the 8-bit or 10-bit bus (see Table 1).

Table 1. BT.656 Data Types

Data Type	Description
Blanking data	This is given by the transmitter any time a frame is not present. The blanking data is 0x10 for Y data and 0x80 for Cb/Cr data. Example: 0x10 0x80 0x10 0x80 0x10 0x80 0x10 0x80 0x10 0x80 0x10 0x80 0x10 …
Timing reference codes	These are what allow BT.656 to operate without hardware synchronization lines. They are of the form '0xFF 0x00 0x00 0xXY' where XY is the actual command.
Image data	This is the remainder of the data contained within the fields themselves, which consists of the Y/Cb/Cr values for each pixel forming the image itself. Example: 0xY 0xCb 0xY 0xCr 0xY 0xCb 0xY 0xCr 0xY 0xCb 0xY 0xCr 0xY 0xCb …

The embedded timing reference codes allow the video port to synchronize to the data stream using only the parallel bus and the clock line. Each code starts with '0xFF 0x00 0x00', and the final byte contains the actual command. The command is contained within three bits of the final byte; the other bits of the final byte are used for error checking. The three command bits are F for Field, V for Vertical Sync, and H for Horizontal Sync. The final byte has the form '0b1FVH$P_3P_2P_1P_0$'. The four P bits are protection bits which act like a checksum for the F, V, and H bits. The F bit determines which field is being transmitted for interlaced video, field 1 (F = 0) or field 2 (F = 1). The V bit is 1 when blanking data is coming next. The H bit determines if the code is a start of active video code (H = 0) or an end of active video code (H = 1).

The F, V, and H bits are used to form three basic commands which apply to either field. The commands are listed below. Figure 3 puts these elements together to show how the timing codes work together to transmit an interlaced video frame.

- Start of active video (SAV) indicates the beginning of a line of video. In this case, V = 0, as you are no longer in vertical blanking; H = 0 for SAV.
- End of Active Video (EAV) indicates the end of a line of video. V is also 0 in this case, as you are not yet entering vertical blanking. H = 1 for EAV.
- Last End of Active Video indicates the end of a field. Here, V = 1, as you are returning to vertical blanking. H = 1 to indicate EAV.

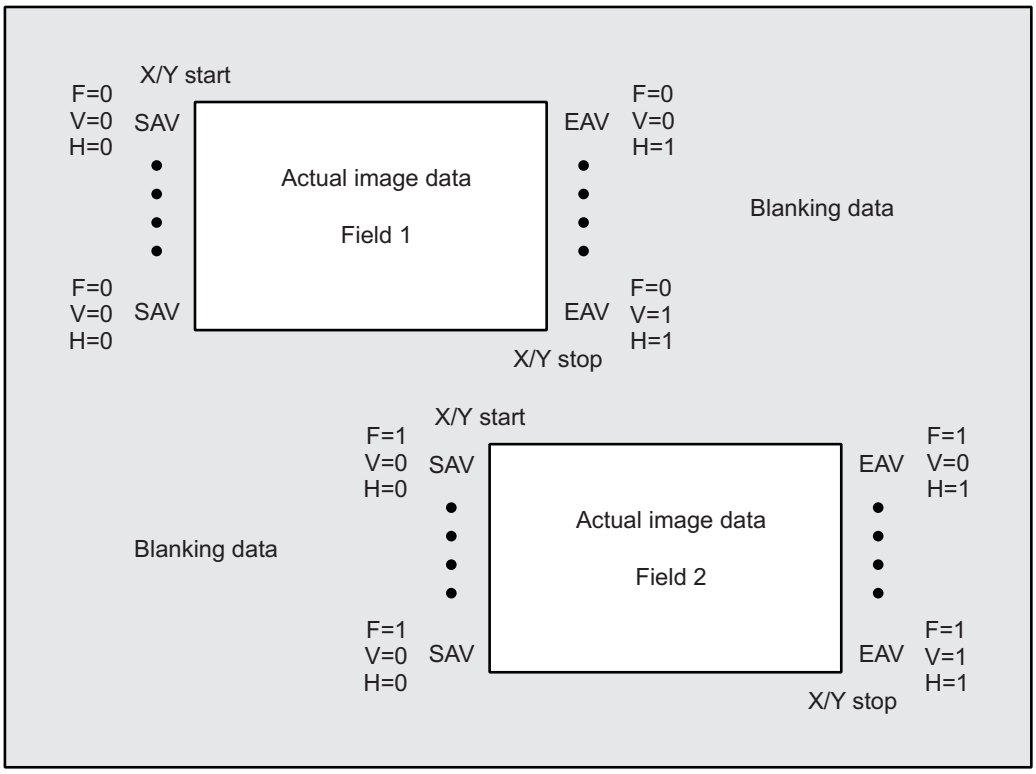

Figure 3. BT.656 Frame Diagram

1.2.2 RAW Mode

RAW mode is a video port configuration in which data is captured and not formatted. RAW mode allows for almost any video or other data to be brought into or out of the device, as it does not impose any formatting restrictions. This allows the port to be used for high-speed communications and enables compatibility with video devices that do not follow any other supported standard of the video port, such as BT.656. RAW mode can be implemented between two video ports and is quite useful for moving video as well as non-video data across the interface. RAW mode requires only one control line between the transmitting and receiving video ports for the most basic operation that constitutes a progressive frame. The control line connects the transmitter's active video (AVID) signal and the receiver's capture enable (CAPEN) signal. As data is streamed across the bus, the video port display indicates that there are valid data with the AVID signal. The receiver captures data only when the CAPEN signal is asserted. The receiver can synchronize using this single line by waiting for a period of blanking (AVID = 0) of a particular length, so that it knows when the start of a new field begins. Figure 4 shows a RAW frame diagram with an interlaced setup that uses two control lines, both the active video line and the field ID line.

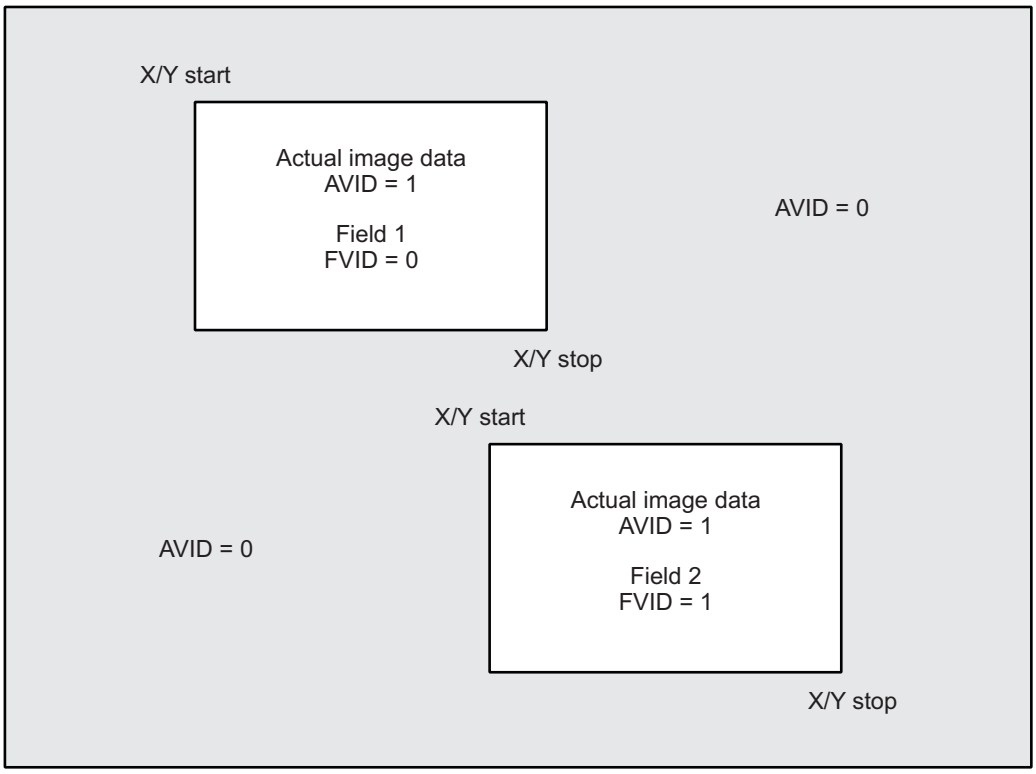

Figure 4. RAW Frame Diagram

1.3 Video Port Driver Software

The video port mini-driver is software provided by TI which allows the use of the video port from relatively simple APIs (application programming interfaces). The functions of the APIs allow the developer to quickly implement a video port configuration, taking what could be hundreds of individual register writes and turning them into just a few simple API calls. The mini-driver operates by configuring the EDMA to handle the transfer of frames from the video port into memory, as well as applying a register configuration to the video port. This means that the driver itself incurs minimal CPU overhead while it is running. The only CPU overhead comes from the calls made by the application software to allow it to swap frame buffers between what the application sees and what the driver is capturing or displaying. Figure 5 gives a visual overview of the video port driver functionality. For additional details on the video port mini-driver, please see *The TMS320DM642 Video Port Mini-Driver* (SPRA918).

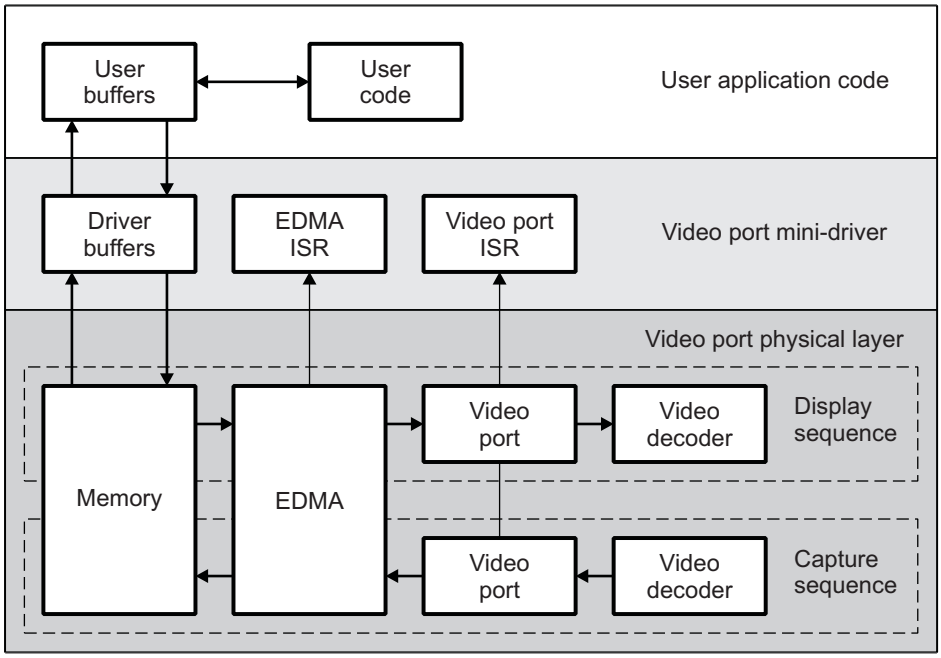

Figure 5. Video Port Block Diagram

2 Video Port to Video Port Communication

This section covers the usage of a video port to communicate with another video port. While the video port is typically connected to a video encoder or decoder, this section discusses sending arbitrary data in place of the typical video data the peripheral was originally designed to handle. This section also covers possible uses for this concept, as well as the performance one can expect to find. Additional information on the topic, including hardware and software examples, can be found in the appendices.

2.1 Applications of Video Port to Video Port Communication

There are a number of applications that can benefit from this type of communication. In general, any application dealing with large volumes of streaming data can utilize this interface. This leaves other, more traditional interfaces such as the HPI (host port interface), EMIF (external memory interface), or EMAC (Ethernet MAC) free to add additional differentiation in the end product. Because the DM642 supports video streams with high bandwidth, the same interface can be used to provide a high-speed interface for other data.

For example, one can use video port to video port (VP-VP) communication between devices in a video transcoder. Here, a single device can be used to decompress an incoming video stream, sending it to multiple devices that can compress the video stream in other formats. Figure 6 shows a block diagram of such a system allowing for a single format to be converted into three different formats.

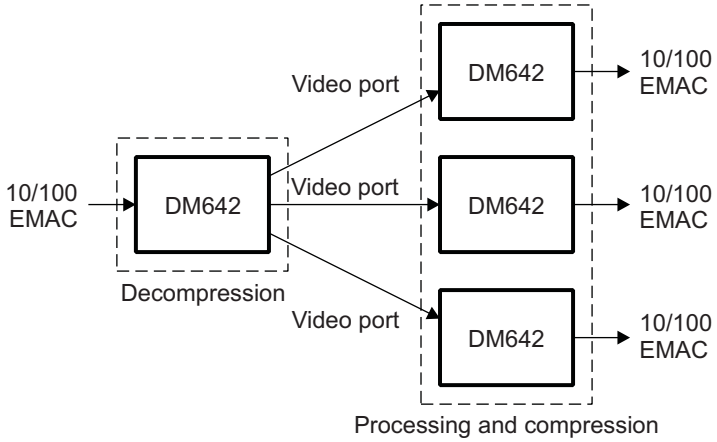

Figure 6. Video Transcoder

Another possible application would be a High Definition (HD) video compression system. Here, the video ports can be used to send data between devices for processing, as the requirements for HD compression typically take several DSPs. Figure 7 shows a block diagram of a theoretical HD compression system utilizing multiple VP-VP links, including VP-VP links for the transfer of compressed data.

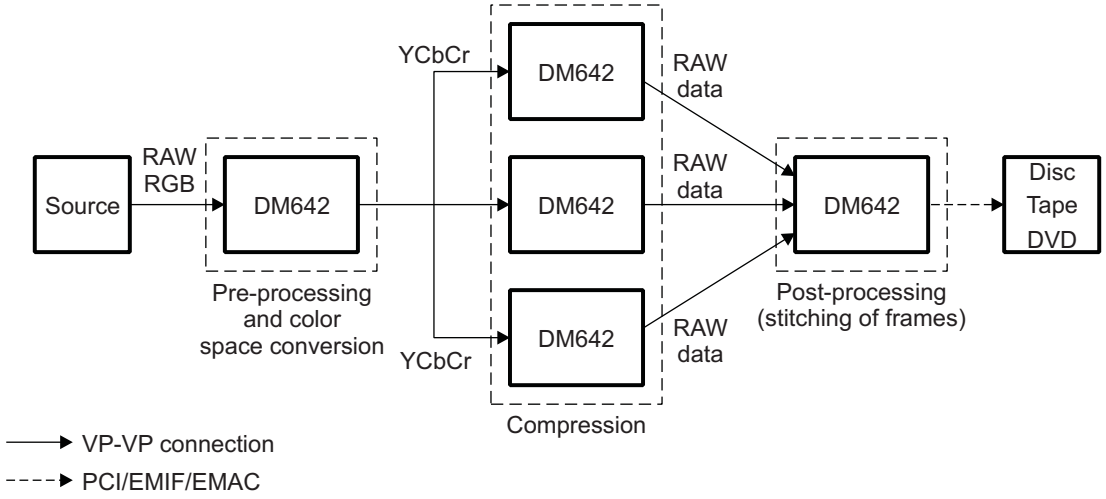

Figure 7. High Definition Compression System

2.2 Utilizing a Video Transmit Stream for Data

The key to a fast and easy implementation of a data transfer over the video ports is to utilize the hardware and software provided by TI as they are designed. What this means is to have the video port and the video port mini-driver function as if they are transferring video data while having the arbitrary data encapsulated in the video data at a higher level. This encapsulation concept is much like many networking concepts where higher level protocols are embedded within lower level protocols. This allows the lower level video port operations and hardware communication to be encapsulated by a much simpler higher level communication protocol for arbitrary data transmission. Figure 8 shows a block diagram of basic DSP to DSP communication by way of the video ports.

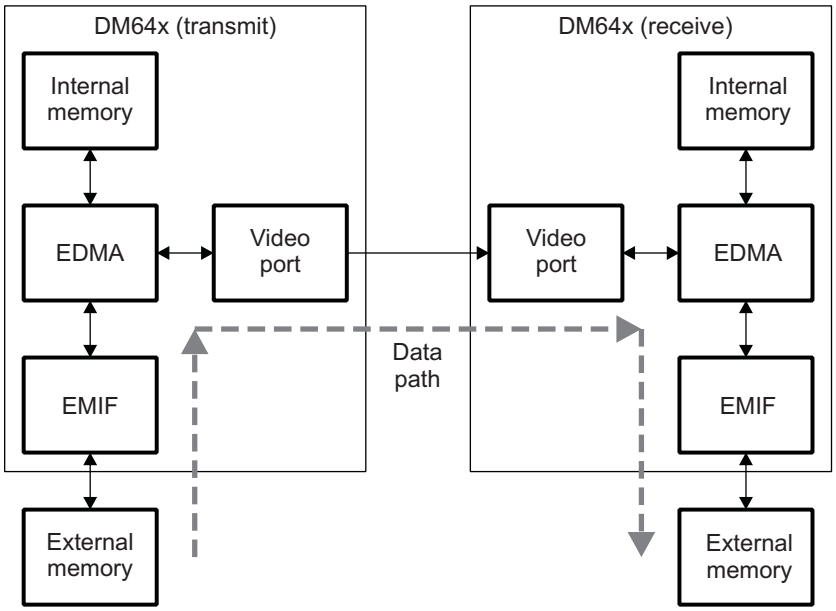

Figure 8. Video Port to Video Port Block Diagram

Video data is designed to be sent in frames, providing a time-based division of data that also can be used effectively for the purpose of transmitting arbitrary data. This allows each frame to be used as a packet of data where the arbitrary data and any required header information can be placed. The video port mini-driver allocates buffers to hold these frames. Because of the buffers, the driver can fill (capture) or transmit (display) a frame, and the application programmer can simultaneously access a different frame. This buffer scheme enables the transfer of frames containing arbitrary data with a simple video port configuration by the transmitting device. Once one can successfully transfer a frame, the arbitrary data and header information can be added to the frame for data transmission. The concept is similar for the receiving side, where the driver is used to collect frames. The application software must check each received frame and interpret any header information to know how to handle the data. An example implementation of this concept is given in Appendix A.

The header information passed through the video port can be application specific. In most cases, including the example given in Appendix A, a minimum of two words of data are required for each block of data to be sent or received. In particular, a destination address and length will allow the receiver to easily parse the data and place it at the proper destinations in the receiver's memory. A length of 0 can be used to inform the receiver that there are no more valid blocks of arbitrary data contained in a particular frame. In addition to these two pieces of information per block, there is also a key value per frame that is used to define whether a frame contains valid data. The key can be any value, as long as both the sender and receiver agree on the value. This key value allows a frame to be quickly ignored if there are no valid blocks of data contained within the frame. With these header constructs, one can send several blocks of data in a single frame that the video ports will treat as a video frame. Figure 9 shows the basic packetization scheme used for this application report.

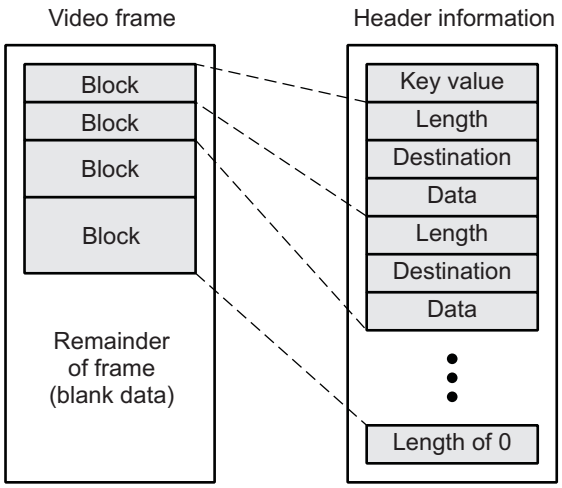

Figure 9. Basic Packetization Scheme for a Frame

2.3 Performance Considerations

Because the video port is such a high-speed parallel interface, it can achieve a fairly high bandwidth. The theoretical maximum bandwidth of a video port is about 1.6 Gbps, which comes from the 20-bit bus running at 80 MHz. However, a more typical maximum bandwidth, as used in the example code, is the 1.28 Gbps available when a 16-bit bus is used. The advantage of using the 16-bit bus is that the buffers and the EDMA transfers end up much cleaner due to how the video port FIFOs function. Using a 20-bit bus requires additional CPU overhead to sort the data. There is a RAW mode discussion on this in the *TMS320C64x DSP Video Port/VCXO Interpolated Control (VIC) Port Reference Guide* (SPRU629).

There are two primary limitations which would prevent achieving the full bandwidth of the interface. The first limitation is the header information that needs to be sent with the data such that the receiver can interpret it properly. This overhead is minimal for large blocks of data, but increases as the size of the blocks that need to be transferred decreases. For example, if you have two words of header information per block, as is the case with the example provided, and you are sending only blocks of one word each, you will cut your bandwidth to one third of the maximum. On the other hand, if you were to send one block that is very close to the size of the frame, the ratio of header information to actual data becomes much smaller, bringing the bandwidth very close to the maximum.

The second limitation is that when transferring these large amounts of data within the DSP, these data take up a significant amount of the EDMA bandwidth, and also the EMIF bandwidth if the buffers are in external memory. Every video port you have active brings in or moves out large amounts of data, up to 1.6 Gbps. The EDMA is typically able to handle such high-speed transfers; however, it will not take many of them before the overhead is too great and data is lost, particularly if external SDRAM is used. The more individual blocks there are in the frame, the more time the CPU will have to spend parsing the header information to copy the data blocks to their final destinations, adding to the bandwidth. Because of the increased EDMA bandwidth, the example in the above paragraph applies here, too. If you have many small blocks of data, you will require many small CPU-initiated transfers to handle them, adding additional overhead, whereas a single large block in a frame would be handled by the CPU very efficiently via the EDMA.

3 Conclusion

This application report explains the basic interface for using the RAW and BT.656 modes between video ports. With this concept, the application programmer can easily interface multiple processors and create a high-speed, glueless parallel bus to communicate between DSPs, allowing more traditional interfaces to be used for product differentiation. This interface is ideal for applications which require a high-bandwidth, streaming data interface.

4 References

1. *TMS320C64x DSP Video Port/VCXO Interpolated Control (VIC) Port Reference Guide* (SPRU629)
2. *The TMS320DM642 Video Port Mini-Driver* (SPRA918)

Appendix A Video Port to Video Port Software

The demonstration software configures the transmitter and receiver video ports to communicate with each other using a RAW or BT.656 configuration.

Since the video port works with continuous streams of data, the simplest way of communicating is to design the transfers to function on top of the existing video port mini-driver software. Following this, we have created two functions:

- VPVP_Xmit() takes pointers to data and creates a data structure with header information. The data and header information are placed inside the transmitter's frame buffer.
- VPVP_Recv() is a function that is run every time a new frame is received. It checks the frame buffer for the header created by VPVP_Xmit() and copies data out accordingly.

These functions are described in Section A.1 and Section A.2. A description of the header format is given in Table A-1.

Table A-1. Video Port to Video Port Software Header Format

Word	Purpose
Key value	This key value tells the receiving code that the frame contains valid data.
$Length_1$	The length in bytes tells the receiver how much data is valid.
$Destination_1$	The destination address tells the receiver where the data should be placed.
$Data_1$	The payload to be transmitted to the receiver.
$Length_2$	The second length value for the second data block in the frame.
$Destination_2$	The second destination value for the second data block in the frame.
$Data_2$	The second payload of data.
...	Additional data blocks.
$Length_n$	The final length value needs to be 0 to indicate the end of valid data in the frame.

A.1 VPVP_Xmit ( ) Function

Description: This function takes pointers to the data to be transmitted and prepares a display frame buffer of data for transmission using the structure described in Table A-1.

Function:

```
int VPVP_Xmit (   FVID_Frame *FVID_DisplayBuffer,
                  char msgs[][],
                  int *msg_dests,
                  short *msg_lengths,
                  int msg_count);
```

Arguments:

FVID_DisplayBuffer	The video mini-driver display buffer. The VPVP_Xmit() function fills the current display buffer using the structure documented in Table A-1.
msgs	The input 2D array containing each of the data packets that are to be packed into the current display buffer.

msg_dests	An array of the destination addresses to which the data packets are to be copied within the receiver's memory map.
msg_lengths	An array of length values containing the length (in bytes) of each message to be transmitted.
msg_count	The number of data packets to be transmitted. It should equal the length of the msg_dests and msg_lengths array arguments.

Return Value: The number of messages that the VPVP_Xmit( ) function was able to pack into the FVID_DisplayBuffer.

If the return value equals msg_count, then the VPVP_Xmit() function was successful.

If the return value is less than msg_count, then not all of the messages were able to fit in a single FVID_DisplayBuffer. The remaining messages will need to be transmitted with the next FVID_DisplayBuffer.

Example Pseudo-Code TSK Function:

```
FVID_create();
FVID_control();
FVID_alloc();
while(1)
{
     If(Data_Ready_To_Send)
        {
            VPVP_Xmit();
            FVID_exchange();
        }
    else
        {
            Set 32-bit key value to 0.
            FVID_exchange();
        }
}
```

A.2 VPVP_Recv ( ) Function

Description: This function checks the currently received frame for the key value and extracts the data based on the header information if a valid frame is found.

Function:

```
int VPVP_Recv(    FVID_Frame *FVID_CaptureBuffer,
                  char msgs[][],
                  int *msg_dests,
                  short *msg_lengths);
```

Arguments:	FVID_CaptureBuffer	The video port mini-driver buffer that just received data. The VPVP_Recv( ) function parses this buffer to see if there is valid data inside.
	msgs	A 2D array that is to be filled with each of the messages received inside the capture buffer.

msg_dests	An array of pointers to each of the messages received.
msg_lengths	An array of message lengths corresponding to the length (in bytes) of each message received.

Return Value: The number of messages received in the FVID_CaptureBuffer.

If the return value is 0, then the key value was not present at the start of the capture buffer, and the data inside the buffer can be ignored.

If the return value is greater than 0, then it corresponds to the number of messages received.

If the return value is negative, then an error has occurred.

Example Pseudo-Code TSK Function:

```
FVID_create();
FVID_control();
FVID_alloc();
while(1)
{
        FVID_exchange();
        msg_count = VPVP_Recv();
        If(msg_count) Process(msg_count);
}
```

Appendix B Video Port to Video Port Hardware

The DM642EVM daughter card allows for communication between video port 2 and video port 1 on the board or to another DM642EVM with daughter card by way of a ribbon cable. There are three categories of settings on the board: video port control lines setting, made with jumpers JP1, JP2, and JP3; clock frequency setting, made with jumper JP4; and data flow setting, made with dip switches SW1 and SW2. A block diagram of the daughter card is shown in Figure B-1, and descriptions of the setting options are given in Table B-1.

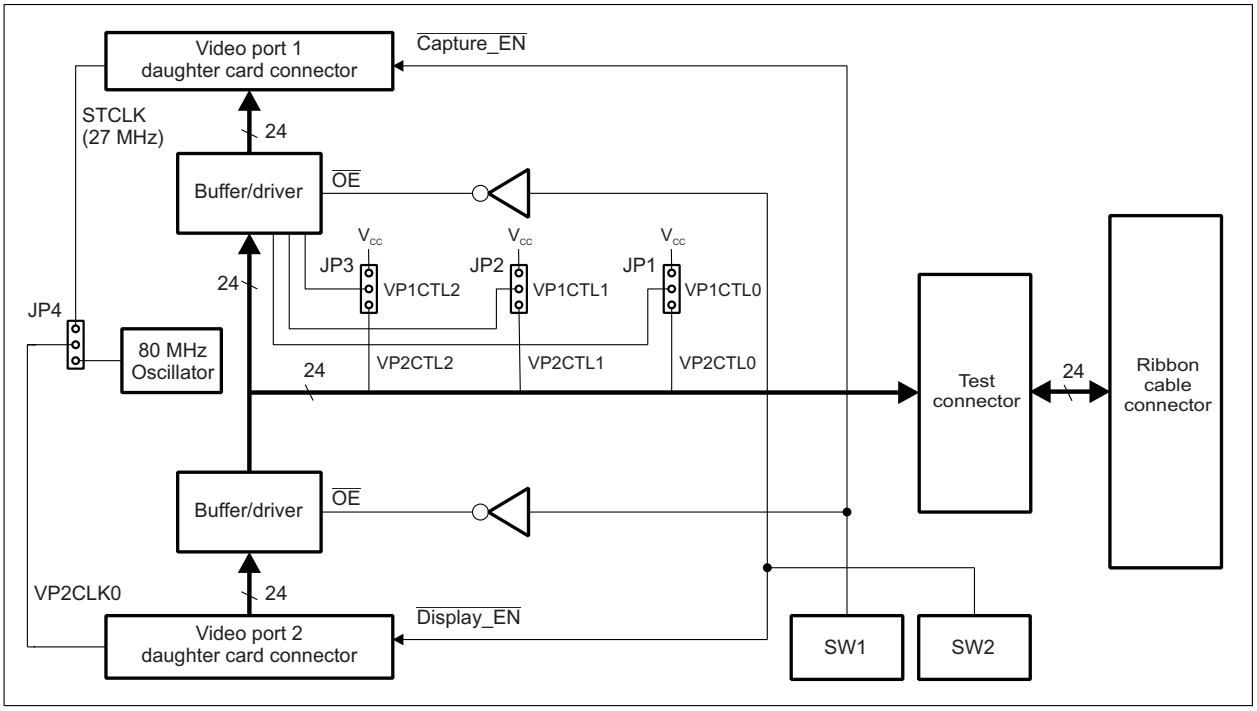

Figure B-1. Daughter Card Block Diagram

Table B-1. Daughter Card Jumper/Switch Control Lines

Setting Category	Jumper/Switch	Description
Video port control lines	JP1	Allows the Video Port 1 Control 0 line to be pulled high for BT.656 mode or connected to the Video Port 2 Control 0 line for RAW mode.
	JP2	Allows the Video Port 1 Control 1 line to be pulled high for BT.656 mode or connected to the Video Port 2 Control 1 line for RAW mode.
	JP3	Allows the Video Port 1 Control 2 line to be pulled high for BT.656 mode or connected to the Video Port 2 Control 2 line for RAW mode.
Clock frequency	JP4	Allows the Video Port 2 Clock 0 line to be driven either by the DM642EVM on-board 27-MHz STCLK signal, or from the on-daughter board oscillator at 80MHz.
Data flow	SW1	Enables the Video Port 1 signals with the buffer.
	SW2	Enables the Video Port 2 signals with the buffer.

Appendix C Video Port to Video Port Examples

The first example included with this application report uses the DM642EVM daughter cards to connect two DM642EVMs via a video port bus in 8-bit BT.656 mode. The second example uses the DM642EVM daughter cards to connect two DM642EVMs via a video port bus in 16-bit RAW mode.

C.1 BT.656 Example for Video Transmission

In this example, the transmit DM642EVM uses video port 0 to capture the incoming video with the on-board video decoder. It then displays this video using video port 2 configured in 8-bit BT.656 mode. Video port 2 is connected to the daughter card for transmission of the video frames to the receiving DM642EVM.

The receive DM642EVM uses video port 1 to receive the incoming video from the daughter card, and then displays this video using its video port 2 connection to the on-board video encoder.

Fundamentally, this example creates a video pass-through using two DM642 processors.

To run the BT.656 example for video transmission, the following steps should be performed in the order listed here:

1. Set up the DM642EVM daughter cards according to Table C-1 for the transmit and receive boards.
2. Use the Code Composer Studio™ integrated development environment (IDE) to load the **VPVP_BT656_CAPTURE_XMIT.out** program onto the transmit DM642 processor. This will configure the transmit DM642 processor to begin capturing video from a camera and displaying it across the daughter card interface.
3. Use the Code Composer Studio IDE to load the **VPVP_BT656_RECV_DISPLAY.out** program onto the receive DM642 processor. This configures the receive DM642 processor to begin receiving incoming video from the daughter card, and to begin displaying the video via the NTSC display output of the DM642EVM.

Table C-1. BT.656 Example Project Daughter Board Settings

Jumper/Switch	Setting for BT.656 Transmitter	Setting for BT.656 Receiver
JP1	V_{CC}	Vcc
JP2	V_{CC}	Vcc
JP3	V_{CC}	V_{CC}
JP4	27 MHz	Disconnected / Open
SW1	OFF	ON
SW2	ON	OFF

C.2 RAW Example for Text Messaging

In this example, the transmit DM642EVM uses a UART to collect messages typed into a terminal window. It prompts the user for the number of messages to transmit. When the user hits the Enter key on the final message for transmission, the VPVP_Xmit() function uses the captured UART messages to create a video buffer containing RAW data for display with video port 2 connected to the daughter card.

The receive DM642EVM uses video port 1 to receive the incoming messages from the daughter card. It uses the VPVP_Recv() function to check whether valid data has been received. When valid data has been received, the receive DM642 processor prints the messages received to a terminal window via the UART connection.

Fundamentally, this example demonstrates the ability to transmit non-video asynchronous data between two DM642 processors using the video port configured in 16-bit RAW mode.

To run the RAW example for Text Messaging, follow these steps in the order given:

1. Set up the DM642EVM daughter cards according to Table C-2 for the transmit and receive boards.
2. Make the RS232 UART connection from the transmit DM642EVM to the transmit PC COM port.
3. Make the RS232 UART connection from the receive DM642EVM to the receive PC COM port.
4. Start the terminal program on both the transmit and receive PCs with the following settings:
   - Baud Rate = 115,200 bps
   - Data Bits = 8 bits per word
   - Stop Bits = 1 stop bit
   - Parity = No
   - Hardware Flow Control = None
5. Use Code Composer Studio to load the **VPVP_RAW_MESSENGER_XMIT.out** program onto the transmit DM642 processor. This configures the transmit DM642 to begin communicating with the transmit PC terminal program via the RS232 UART connection.
6. Use Code Composer Studio to load the **VPVP_RAW_MESSENGER_RECV.out** program onto the receive DM642 processor. This configures the receive DM642 processor to begin receiving text messages from the transmit board via the daughter card, and begins printing these messages to the terminal window on the receive PC.

Note: Both transmit and receive programs wait for user input into the terminal program before transmission occurs across the VP-VP interface.

Table C-2. RAW Example Project Daughter Board Settings

Jumper/Switch	Setting for RAW Transmitter	Setting for RAW Receiver
JP1	Vcc	VP2CTL0
JP2	Vcc	Vcc
JP3	Vcc	Vcc
JP4	80 MHz	Disconnected / Open
SW1	OFF	ON
SW2	ON	OFF

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
Low Power Wireless	www.ti.com/lpw	Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265