APPLICATION NOTE

SAA7195A (VMC+)
For
Video Capture Applications

AN96059

Philips
Semiconductors

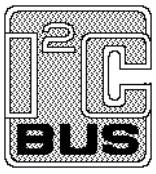
PHILIPS

SAA7195A (VMC+) for Video Capture Applications

Application Note AN96059

Abstract

SAA7195A is the core device used in many multimedia PC add-on cards that were designed to demonstrate the performance of various desktop video chipset from Philips Semiconductors. Typical applications of these cards are video display and capture in PC environment, image processing in medical and scientific equipment. SAA7195A is a highly integrated and complicated device. This note tries to explain some of those complex functions and disclose some tricks on how to efficiently use this device. Nevertheless, this note only serves as a reference for users of SAA7195A. Depending on applications, there might be some discrepancies between the content in this note and the actual programming of the chip under that applications. Once such discrepancies arise, users should always resolve problems according to the data sheet of SAA7195A and seek technical support from Philips Semiconductors.



Purchase of Philips I^2C components conveys a license under the Philips I^2C patent to use the components in the I^2C system, provided the system conforms to the I^2C specifications defined by Philips.

APPLICATION NOTE

SAA7195A (VMC+) For Video Capture Applications AN96059

Author(s):

Bruce Lei, Jeffrey Chang

Regional Sales Office - Southern East Asia

Industrial Regional Application Laboratory

Taipei-Taiwan

Keywords

VMC+
DPC7167
CCIR
NTSC, PAL & SECAM
Color Key
VLUT
Extended Memory
UMA

Date: 01, September, 1996

Summary

SAA7195A (also nicknamed as “VMC+”) plays as the heart in a video capture subsystem under PC environment. There are more than 70 registers inside performing the functions of video in/out format conversion, scan rate conversion, video sizing and scaling, frame buffer control, host interface and video capturing back to host. Except that it can not be booted up by itself, VMC+ actually acts like a special-purpose CPU.

Instead of the basic programming of VMC+ which can be found in the data book, this note will focus on the more advanced techniques in programming this chip. One of the main purposes of this note is trying to resolve some tricky problems that users could encounter and are not able to find out the solutions from data book. Therefore, in writing this note and explaining the techniques, we assume that the readers already have some experience in using this device and basic knowledge of video capturing in a PC system.

Although VMC+ can be used in many different applications and systems (e.g. DOS & MS-Windows in PC, MAC and medical equipment), we will illustrate those techniques by examples in MS-Windows application only, since that is the most popular application in the PC world for video display and capture now.

CONTENTS

1. INTRODUCTION

Video display and capture in a PC system is one of the most important functions in the multimedia world now. There are many ways with a lot of various devices from famous chip vendors that can achieve this job. For the end users, it is getting much less hassle of the current architecture than the one used 3 years ago. However, from the programmers' and system designers' point of view, none of them are simple or easy. SAA7195A is definitely not the best solution for video capture now. But it does play a special role among the devices of the desktop video chipset from Philips Semiconductors. Since VMC+ is used in so many desktop video application boards(DTV6, SAA7168 demo board, DPC7167, DPC7140 and VideoBlaster from Creative Lab,...etc.), therefore fully understand how this device work will be very important to the programmers and system developers of those application boards.

In this note, we often use example codes from the software of DPC7167 demo board running under Video For Windows (VFW) application and we program this device based on the DPC7167 hardware configuration. We do so not only that it's convenient to us but also because VFW is a recognized standard software for video capturing and DPC7167 is a typical traditional video capture card. However, users should keep in mind that all of these codes and setup should be easily changed and adapted into your proprietary systems. Please refer to application note AN95056 - "User's Manual: DPC7167 Demo Board" and AN95058 - "Source Code VFW Driver for DPC7167 Demo Board" for questions specific to the DPC7167 board.

Contents in this note are grouped in different topics (chapters) by their property and start from simpler to more complex ones, from programming within device to the whole system. There are total six topics as follows:

1) Basic video display and window positioning by VMC+.

2) Scaling and zooming.

3) Applications of video look up tables in VMC+.

4) Video capturing to PC systems via memory mapping approach.

5) Video capturing to PC systems via I/O mapping approach.

6) Miscellaneous adjustments of VMC+ on various system configuration.

2. BASIC VIDEO DISPLAY AND WINDOW POSITIONING BY VMC+

This chapter shows how to use VMC+ to display video overlaid with VGA graphics and how to position the video window under MS-Windows environment.

2.1. Basic Video Display

One of the main tasks of SAA7195A is to do scan rate conversion so that low scan rate interlace video can be overlaid with the high scan rate noninterlace graphics. To perform this job, a video memory (frame buffer) that can hold a full size of video frames is needed. VMC+ will store the input video fields into frame buffer with the video clock and sync first, then read out the video data from this buffer with graphics clock and sync. During this process, video cropping, scaling and zooming are accomplished at the same time as well. Figure 2.1 simplifies and illustrates this process.

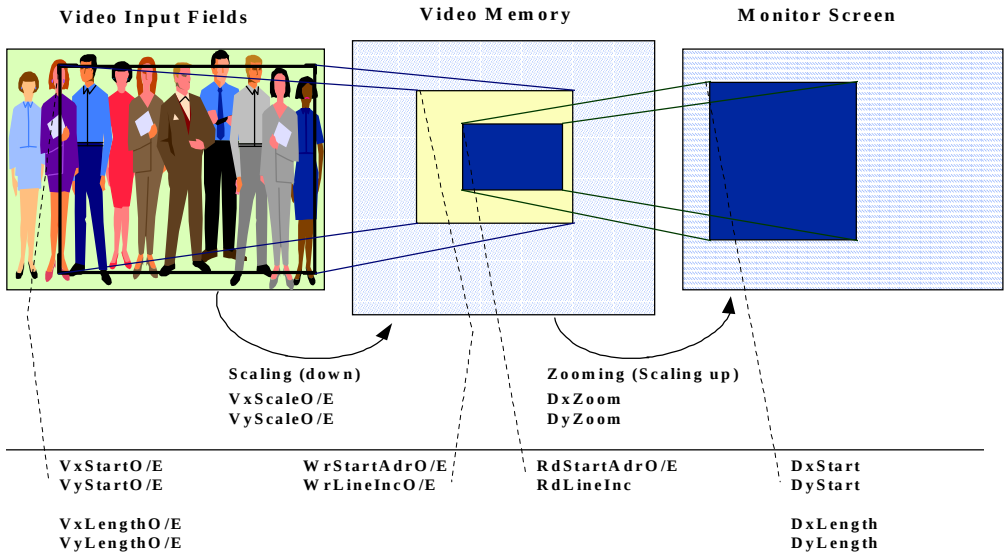


Figure 2.1

For most video, a complete frame (the actual picture we see on TV screen) is composed of two interlace fields. VMC+ takes interlace fields as input and stores in video memory as noninterlace frame format[1]. It assigns a reference origin (0,0) point to every input video frame. This origin is defined as the upper left corner of a complete video frame. It is located at the starting point of HREF (signal from video decoder) in horizontal direction and the next line after Vertical-Sync (from video decoder as well) in vertical direction.

[1] This is not the only way to store video in frame buffer. Video fields can be stored differently for even and odd fields in a more complex method. For illustration purpose, we use this simplified architecture that we used in DPC7167 software

With this origin as a reference, register “VxStartO/E” (reg. index 65 & 85)[2] and “VyStartO/E”[3] (index 6B & 8B) specify the upper left corner of the rectangle area that we want to process and store in the frame buffer; while “VxLengthO/E” and “VyLengthO/E” indicate the length and width of this rectangle area. The so-called video cropping is performed with these four parameters programmed.

A physical origin of the whole frame buffer is supposed to be at the point with both row and column address being 0. Hence, “WrStartAdrO/E” specify the starting point where the actual video pixel (from either odd or even fields) data should be stored with reference to the origin of this buffer. Since the video pixels sent to this buffer are actually a continuos stream for each scan line in a video field, instead of defining a rectangle area, defining a “Line Increment” parameter is better in this case. “WrLineIncO/E” define the incremental values for writing a new scan line with reference to the starting point of the previous line. One more advantage of the use of this line increment is to do interlace and non-interlace conversion. With “WrLineIncO/E” specified as double the length of a scan line, a interlace video can be stored in the buffer as the following example (Figure 2.2):

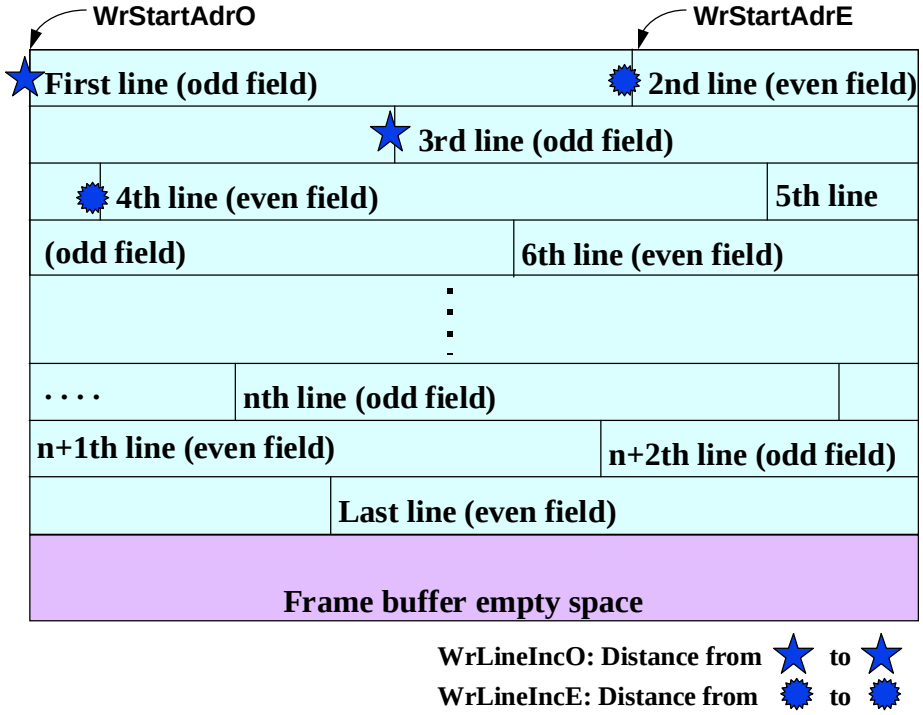


Figure 2.2

[2] “VxStartO/E” means VxstartO and VxStartE two different registers, for our example, and in most cases, they will be programmed with the same value to have a consistent video picture.

[3] All the registers related to video acquisition and storing i.e. Vx(y)StartO/E, Vx(y)LengthO/E, WrStartAdrO/E, WrLineIncO/E, RdStartAdrO/E and RdLineInc are counted in video pixels.

With sequential reading from the starting point at first line to the ending at the last line, a non-interlace video is formed. While reading the video data from the frame buffer to the display device, another video cropping process can be performed as shown in Figure 2.1. “RdStartAdrO/E” specify the upper left corner of the video area intended for display. “RdLineInc” indicates the increment value calculated in the way as “WrLineIncO/E”.

For the final display in a non-interlace mode, “RdStartAdrO” and “RdStartAdrE” are usually programmed with the same value, while “RdLineInc” will only be half of the value in “WrLineIncO/E” as in our previous example. For interlace mode, “RdStartAdrO” and “RdStartAdrE” are to be filled with the starting point of the intended display area in the odd fields and even fields respectively, usually “RdStartAdrE” is at the same position but next line after “RdStartAdrO”, while “RdLineInc” will be programmed the same value as “WrLineIncO/E” as in our previous example. This is illustrated in the following figure (Figure 2.3).

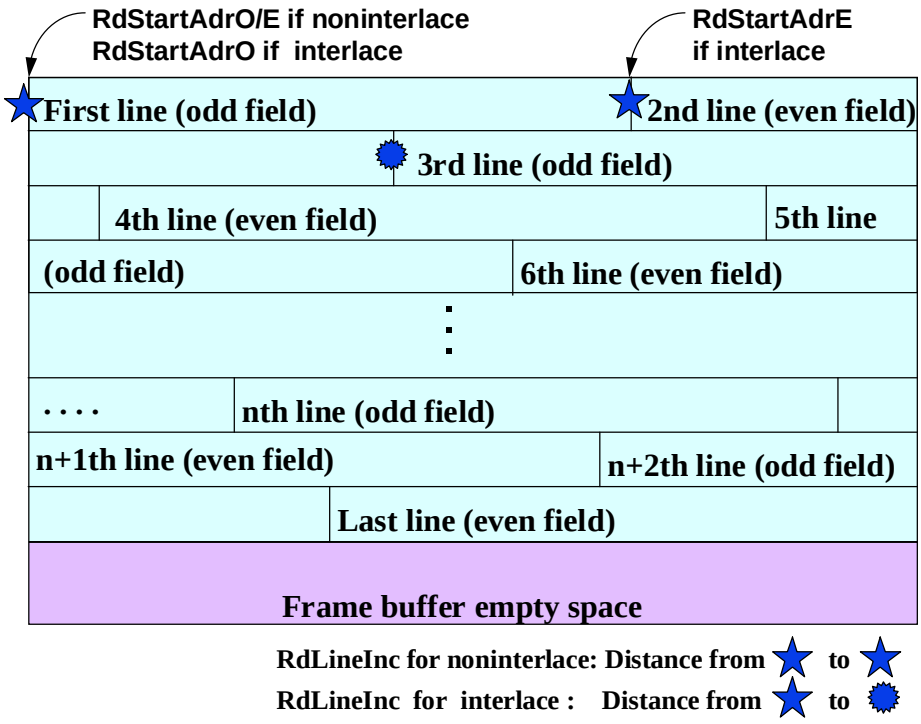


Figure 2.3

2.2. Window Positioning

The final display position on monitor screen is controlled by the registers “DxStart”, “DyStart”, “DxLength” and “DyLength”[4] with reference to the upper left corner of the screen of graphics display. Please refer to Figure 3.3.4 on page 119 of SAA7195A data book for details of this positioning control. Note that there are two auxiliary registers “DXOffset” and “DYOffset” for global control of this positioning. i.e. For different graphics mode, the distances between the origin (upper left corner) of the screen to the inactive edges of DHS and DVS (for graphics display) are different in most cases. By adjusting these two registers, the video window can be right at the desired location and these registers need to be set only once for a specific graphics mode. Please refer to Figure 3.3.12 on page 126 in SAA7195A data book for more details. We now summarize the use of both these two sets of position control registers in the following figure (Figure 2.4).

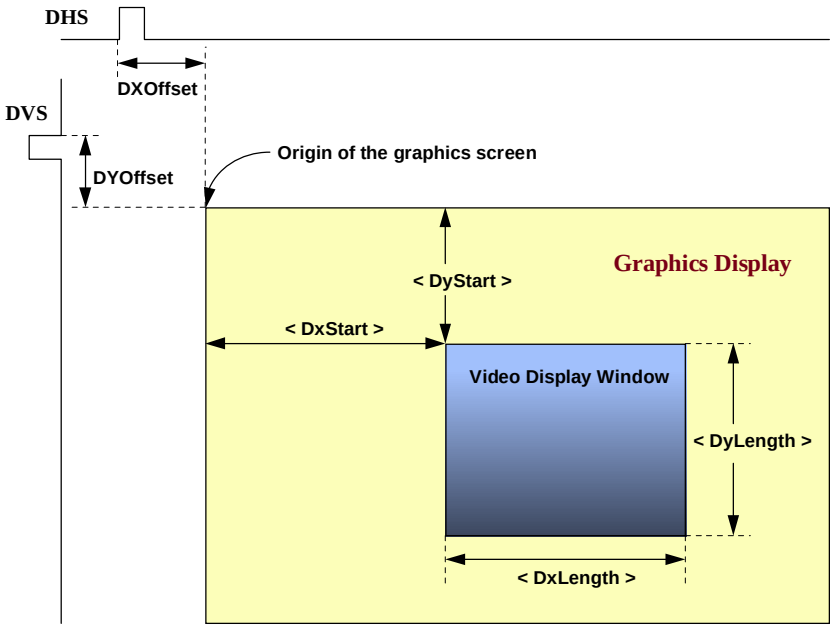


Figure 2.4

The video window overlaid with graphics is supposed to move around free within the whole screen. However, when the left border of the video window moves and exceeds the left border of the screen; “DxStart” must be set to zero. All of these registers need to be set with positive number. Since there is no 2’s complements interpretation, setting an negative number in the high level program will be mistaken as a huge positive number into the chip.

[4] All the registers related to the display i.e. Dx(y)Start, Dx(y)Length, DX(Y)Offset are counted in graphics pixels (equivalent to one cycle of DCLK).

Besides the above mentioned registers, three more registers are related to the display position control; they are reg. 42, 43 & 44. Bit 0 - 4 in reg. 42 (CKeyMode) control the delay of the color key. By adjusting this parameter, it can compensate the difference of delay in RAMDAC on the VGA board and the Mixer-DAC on the video board. When there is mis-alignment on the video with graphics (e.g. a graphics shadow with the color set in “CKeyComp” register on the cursor, when cursor moves into the video window); That is what we mean mis-alignment. By tuning the value of this register, it would eventually make the video and graphics align together, i.e. the shadow in the video should be gone.

Register 43 (“HrefDel”) controls the delay of the signal “HREFOUT” (pin 50) from VMC+. Increasing the value of this register will shift the left edge of the video to right one pixel a step. Since the active edge of this signal it to indicate the starting point (first pixel) of a video scan line to be displayed under the graphics mode, therefore when YUV format is being used for the video in frame buffer, only one out of two adjacent number (either odd or even number depending on hardware configuration) can be set into this register to have the correct color display.

If there are still mis-alignment of the graphics color on the edges of the video window after adjusting the above two registers, try changing the value of register 44 (“VidSelDel”). This will allow you to shift the starting point (left border) of the color key to match with the border of the video window exactly. With programming to the above registers, color key activation, active video starting point and the border of the video window can be all matched together in the horizontal direction.

It is much simpler in aligning the video and graphics in the vertical direction. Unless the user want to do video cropping, otherwise there are only “DYOffset” and “DyStart” two registers can be adjusted to serve this purpose. The function and meaning of these two registers have been discussed in the previous paragraphs.

3. SCALING AND ZOOMING

For SAA7195A, scaling is done in the video acquisition process i.e. after VMC+ gets the video data and before it places into the frame buffer (please refer to Figure 2.1). VMC+ is only able to scale down the picture at this stage. Scaling up (named as zooming for this device) is done when VMC+ reads out the data from frame buffer to the video backend devices (usually a DAC with a mixer); at this stage, only scaling up is possible by altering the speed of SCLK.

3.1. Setting Up For Basic Scaling And Zooming

Unless in some special applications, Vx(y)ScaleO and Vx(y)ScaleE for odd and even fields are usually programmed with the same value just like Vx(y)StartO/E to get a consistent video display. While VxScaleO/E control the scaling factor in horizontal direction, VyScaleO/E control the vertical direction. All these four registers are evaluated in the following formula to get the actual scaling factor:

SF = (1024 - R)/1024

where **SF** is the scaling factor, **R** is the value programmed in the registers

For example, programming a value of 512 into VxScaleO/E will get an SF of 1/2 and result in a video of half of the original size horizontally. With these control registers, a pseudo linear scaling is implemented. By “linear”, we mean that VMC+ is able to scale the video into any random size users want. However, the actual scaling performed is to scale down the video to 1023/1024, 1022/1024, 1021/1024..... 3/1024, 2/1024, 1/1024 of the original size in both horizontal and vertical direction.

Instead of the whole frame of the original video, the area of the video to be scaled down is defined by Vx(y)StartO/E and Vx(y)LengthO/E. This rectangle has to be confined within the original frame, otherwise junk pixels may appear in the scaled video. When no cropping of video is needed, then these four registers would be programmed with the exact size of one field of the original video. For a typical NTSC video, this size would be 640 by 240 [5] for each odd and even field.

When scaling down to certain small size in horizontal direction, artifacts may occur in the video picture. Resetting **bit 3** in **register 2B** to zero will activate the decimation filter for horizontal scaling which can smooth the coarse image artifacts phenomenon. This is due to the fact that artifacts are caused by high frequency components while this decimation filter is actually a low pass filter. However, this reduction of artifacts is not free. It’s under the price of sacrificing the sharpness of the image. Therefore, unless a real small picture need to be displayed, usually we don’t use this decimation filter.

[5] Usually we program VyStartO/E with a value of 10, indicating that we drop the first 10 lines out from the original video, since the real active video starts from line 22 for each field; and the counting start point for VyStatO/E is at around the 10th line in a field.

Basic zooming is proceeded in a similar way as scaling except the equation of zooming factor is different as follows:

ZF = 1024/(1024 - R)

Where **ZF** is the zooming factor and **R** is the value setting in registers “DxZoom” and “DyZoom”. For example, setting DxZoom to 768 will enlarge the video four times in the horizontal direction. Unlike scaling, there is no filter in VMC+ to reduce the artifacts (blocky effect) caused by zooming. The more you zoom, the worse the blocky effect is.

The area to be zoomed and displayed is controlled by the registers “RdStartAdrO/E” and “RdLincInc” (refer to Figure 2.1). Again, this rectangle area defined by these registers has to be confined within the rectangle defined by “WrStartAdrO/E” and “WrLineIncO/E” to avoid unexpected video to get in. The whole process from input video to final display can be illustrated more by the following figure (Figure 3.1).

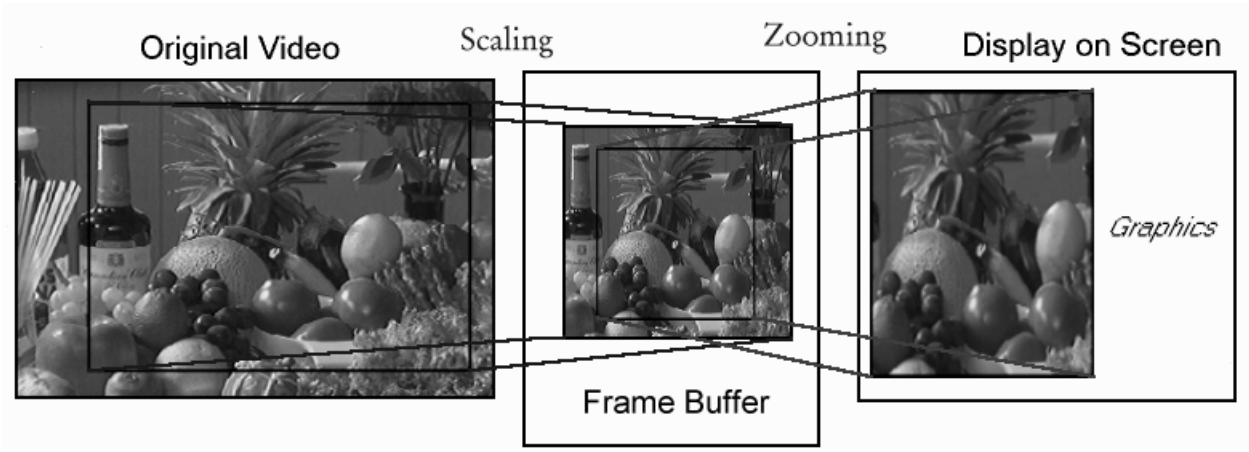


Figure 3.1

From time to time, when scaling down the video vertically to certain sizes (usually to less than half of the original height), mis-alignment of video might occur and cause incorrect interlace of the original odd and even fields. When this happens, you may try to increase the “WrStartAdrO/E” to 3 times of the length of one scan line, if the original setting is the length of one scan line. This is due to some discrepancy of calculation inside the VMC+ scaler.

3.2. Scaling And Zooming Under Different System Configuration

The size of frame buffer under VMC+ control is usually one Mbytes (for 16-bit/pixel in dual bank mode). The advantages of using memory in dual bank mode are not just to double the addressing space of VMC+ but also to reduce the display clock to half, so that the video subsystem controlled by VMC+ can be run at a higher display clock (usually from VGA via feature connector). In terms of PC graphics, the video subsystem can work with VGA card at a much higher graphics resolution mode than with the single bank mode. However, single bank mode costs much less than dual bank mode (since it only uses half of the VRAM in dual bank mode). Another problem arises when using single bank mode, which can only store up to 512x512 pixels in the buffer. A regular video input frame (e.g. 720x480 for a CCIR NTSC format) needs more space than that to be stored. In such case, the trick to display the full size of the picture is as follows:

1). store only one field (either odd or even) in the frame buffer (only take 720x240). This could be done by setting register **30 "VdAcqMode"** bit 6 or 7 to **zero**. (please refer to page 129 in data book).

2). then zoom up twice in the vertical direction when read it out from the frame buffer. This could be done by setting register **B0** and **B1 "DYZoom"** to **512**.

We do not lose any video information since we didn't scale down the original video. We might see some artifacts in vertical direction; but that's a tradeoff in terms of cost.

The size of a video input frame depends on the output format of the frontend video decoder. For Philips SAA7110, square pixel format is the standard (640x480 for NTSC, 768x576 for PAL & SECAM). For SAA7111, it can output the video in either CCIR-601 or CCIR-656 formats. Both formats will give a full frame in the size of 720x480 for NTSC or 720x576 for PAL & SECAM. There are some other video formats from other chip vendors which are different from the above mentioned. Although they are all different in pixel clock frequency, this does not bother VMC+. What need to be cared are the scaling factors for different format.

When VMC+ is used in the PC environment, 4:3 aspect ratio need to be maintained to avoid any deformation of the shapes of the objects in the original video. Except the square pixel format from SAA7110, which has already output in 4:3 aspect ratio, all the other formats need to be scaled or zoomed somehow to match the aspect ratio, when a full size of frame is to be displayed. For example, a CCIR-601 PAL format (720x576) needs to be either scaled down vertically from 576 to 540 or zoomed up horizontally from 720 to 768 to maintain the aspect ratio. (please refer to Figure 3.2).

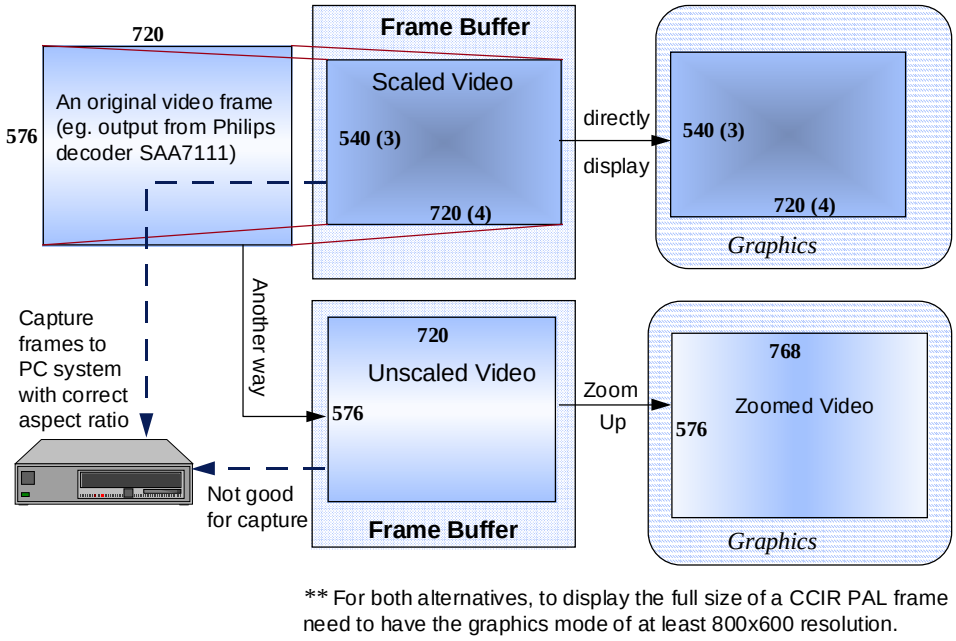


Figure 3.2

We would recommend to always use scaling for this kind of adaptation of video formats instead of zooming for the following reasons:

1). Scaling down the video then storing in the frame buffer requires less space in memory than storing the original video in buffer first then zooming up.

2). If the video pictures are to be captured into the system besides display, then doing scaling before saving into the frame buffer is a necessary procedure. Otherwise, the pictures captured will not have a correct aspect ratio.

3). There is a decimation filter in scaling that can be used to smooth the picture; while no filter is there for zooming.

Since video capture is done by reading the video data from the frame buffer to the system main memory, therefore the size, alignment and format (with scaling or not) for the video data stored in the frame buffer are very important to the correct capture. This issue will be discussed in later chapters.

4. APPLICATIONS OF VIDEO LOOK UP TABLES IN VMC+

There are two Video Look Up Tables (VLUT) inside VMC+. Lots of applications can utilize these VLUTs for special features. One of the best applications is to adjust color saturation, luminance brightness and contrast via programming of these VLUTs.

The VLUTs in VMC+ consist of two 256 x 8 bit RAMs. One RAM is located in the luminance signal path which allows to set brightness and contrast. The other one is located in the chrominance signal path (which means it effects the C_b and C_r signals in the same manner) where the color saturation can be adjusted.

These luminance and chrominance RAMs can be loaded individually. The starting address (set in the register VLUTIndex, reg. 27) indicates the location of the first byte to be written into the RAM, while the data to be written are stored in VLUTDataY (reg. 28) or VLUTDataC (reg.29). The following data bytes will be written into the RAM in an auto-increment mode with the increment value of one.

The “Brightness” adjusts the luminance intensity of the video picture. In general, this parameter will affect all the pixels in the picture with a equal magnitude on the luminance values.

The “Contrast” controls the relative difference between higher and lower intensity luminance values of the video picture. Higher values of “Contrast” cause pixels with darker luminance to tend toward black and lighter luminance to tend toward white. Lower values of “Contrast” cause luminance values of all pixels to move toward medium luminance values.

As mentioned above, the output luminance (Y_o) is determined by the following equation:

$$Y_o = C * (Y_i - 128) / 128 + B$$

where

Y_o = Output luminance [0, 255],

Y_i = Input luminance [0, 255],

B = Brightness [0, 255],

C = Contrast [0, 255]

The “Saturation” controls the color intensity of the video picture. Higher values of “Saturation” cause larger color gain, while lower values cause smaller color gain. The output chrominance (UV_o) is determined by the following equation :

$$UV_o = S * (UV_i - 128) / 128 + 128$$

where

UV_o = Output chrominance [0, 255],

UV_i = Input chrominance [0, 255],

S = Saturation [0, 255]

For using the VLUT of VMC+ in the above application, the following C sample code is included here for reference.

```
/* ------------------------------------------------
   VMC_SetVLUT -
  ------------------------------------------------ */
BOOL FAR PASCAL EXPORT VMC_SetVLUT (
WORD    wBright,            // [0, 255] - (default: 128)
WORD    wContrast,          // [0, 255] - (default: 128)
WORD    wSaturate           // [0, 255] - (default: 128)
)
{
      WORD    wIdx;
      float fLuminance;
      float fChrominance;
      long lData;
      DWORD dwTick;         // Added by Jeffrey DEC 11 '95

      if (pSAA7195A == NULL)
            pSAA7195A = VmcOpen();

      if (pSAA7195A == NULL)
            return( FALSE );

      VMC_FreezeVideo();
```

```
dwTick = GetTickCount();
while (GetTickCount() < dwTick + 100)            /* Delay 100ms */
      ;

for (wIdx = 0; wIdx < 256; wIdx++)
{
      VmcSetReg(pSAA7195A, VLUTIndex, (long) wIdx);  // VLUT Address

      // Luminance (Y)
      // Y = (C / 128) * X + B - C,
      //   = C * (X - 128) / 128 + B,
      // where    Y: Luminance,   [16,235] for CCIR-601 standard.
      //          C: Contrast,    [0, 255]
      //          B: Brightness,  [0, 255]

      fLuminance = (float) wContrast / 128.0F * wIdx + wBright -
                  wContrast;
      lData = (long) max(min(fLuminance, 235), 16);
      VmcSetReg(pSAA7195A, VLUTDataY, lData);    // Y (Luminance)

      // Chrominance (UV)
      // UV = (S / 128) * X + 128 - S
      //    = S * (X - 128) / 128 + 128
      // where    S: Saturation,  [0, 255]
      //          UV: Chrominance [16, 240] for CCIR-601 standard

      fChrominance = (float) wSaturate / 128.0F * wIdx + 128.0F -
            wSaturate;
      lData = (long) max(min(fChrominance, 240), 16);
      VmcSetReg(pSAA7195A, VLUTDataC, lData);   // Chrominance
} /* for wIdx */
```

```
    dwTick = GetTickCount();
    while (GetTickCount() < dwTick + 100)          /* Delay 100ms */
        ;

    VMC_UnFreezeVideo();
    return( TRUE );
} /* VMC_SetVLUT */
```

5. VIDEO CAPTURING TO PC SYSTEMS VIA MEMORY MAPPING APPROACH

To capture motion video down to PC systems via SAA7195A is a complicated task, which involves the understanding of VMC+, DOS, MS-Windows and the memory schemes under PC environment. Lots of coding also involved to accomplish this job. We will first explain the memory schemes in the current PCs, then discuss the capturing to system via extended and upper memory approach respectively.

5.1. Memory Schemes In The Current PC

The content of this section is out of our scope here. However, in order to explain well the capturing of video via VMC+, this memory architecture still needs to be understood first. With the following figure (Figure 5.1), we illustrate the memory schemes used by the current PC. The readers should refer to other books dedicated in this topic for more details.

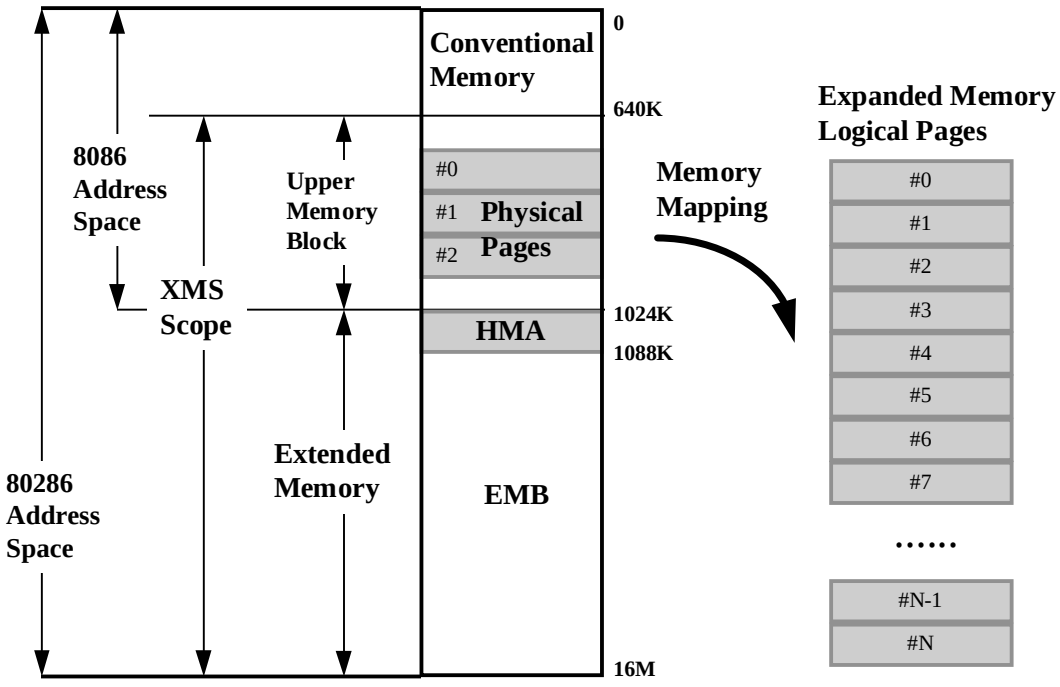


Figure 5.1: EMS/XMS Diagram

The Extended Memory Specification (XMS) arbitrates the use of memory lying outside the conventional DOS domain of the first 1Mbytes. Of particular concern as computing hardware increases in sophistication is the potentially vast amount of extended memory that may be installed above the 1Mbytes boundary of the 8086 address space. The 80286 and more modern processors can form addresses with more than the 20 bits to which the 8086 is limited, but not by following the real-mode rule (for 80286 or later CPU) for combining 16-bit segments and offsets. With only a few exceptions, the protected mode of these newer processors is required to get at extended memory, putting this practically unbounded region outside the range of normal DOS memory management.

Furthermore, a real-mode environment is ideal for many purposes requiring moderate resources and only minimal assistance from an operating system. The XMS acts as a supplement, presenting the additional resource of extended memory without the trapping that may make a protected-mode operating system seem extravagant. When extra power is needed, the XMS is an essential agent in the smooth transition to protected mode, offering the means through which the protected-mode software may preserve extended memory already being used by real-mode programs. In short, the XMS is the means by which DOS may usefully serve as a launching pad to protected mode but not lose touch with its basic configuration in real mode.

The interface of VMC+ supports two different PC-bus configuration the ISA-bus and the NuBus. It is used to control the functions of the VMC+ as well as to read or write data from/to the video frame buffer. The ISA-bus interface of VMC+ supports memory access and I/O access. Its interface configuration can be used in any Intel 80x86 AT-bus design system (IBM compatibles) down to IBM-XT computers which are not able to run in the protected mode. In order to overcome the 16MBytes memory address range problem, another alternative is built in via access of the UMA (Upper Memory Area), also named as UMB (Upper Memory Block). We will illustrate both these memory mapping approach by examples in the next two sections.

5.2. Extended Memory Mapping Approach

Normally, we access video memory using XMS (Extended Memory Specification) approach. The starting address of the frame buffer on the video subsystem could be located at 1 Mega, 2 Mega, 3 Mega ... to 15 Mega when mapping to main memory on systems. However, it will not be permitted to map the video memory to the location that could overlap with the main memory. Let’s demonstrate the method by the following example.

Example 1: If there are 1M bytes video memory (in mux mode) on video board and 8M bytes main memory in our computer, we choose 15 Mega as the starting address of the video memory. The programming of VMC+ is as follows :

Step 1. Set Command (0x00) to 0xD5 (refer to page 104, VMC+ data book) since we access dual bank (mux mode) video memory by memory mapping approach.

Step 2. Set MEMCtrl (0x17) to 0x03 (refer to page 112, VMC+ data book) to use XMS (extended memory) approach.

Step 3. Set MemBaseBank (0x10) to 0x0F (refer to page 109, VMC+ data book), since the frame buffer address starts from 15 Mega of main memory. Value set in the bits 4-5 of this register represents the bank number of the physical memory on the video subsystem. This physical memory is local to the VMC+ and up to 4 Mbytes. For most system with only 1 Mbytes or less local video memory, these bits should be set to 0.

The following sample program will show you how to set the registers of VMC+ when using XMS approach.

```
/* ----------------------------------------------------
   VMC_SetVRAMBaseAddr -
  ---------------------------------------------------- */
BOOL FAR PASCAL EXPORT VMC_SetVRAMBaseAddr (
BYTE BaseAddr,
BYTE Range
)
{
      if (pSAA7195A == NULL)
            pSAA7195A = VmcOpen();

      if (pSAA7195A == NULL)
            return( FALSE );

      // [1] p109 Memory base address and memory banking
      //  7 6 5 4 3 2 1 0
      // |  | | |  |_|_|_|_ Memory base address [23:20]
      // |  | |_|_________ Memory bank
      VmcSetReg(pSAA7195A, MemBaseBank, BaseAddr);
```

```
    // [1] p112 Memory control
    //  MemCtrl     0 0 := 16 KByte;    UMA only
    //              0 1 := 32 KByte;    UMA only
    //              1 0 := 64 KByte;    UMA only
    //              1 1 := 1  MByte;    Extended Memory only
    VmcSetReg(pSAA7195A, MemCtrl, 0x03);


  return( TRUE );
} /* VMC_SetVRAMBaseAddr */
```

If there are 16M bytes (or more) of main memory on the system, the above method for video capturing won't work. To resolve this problem, the programmers need to use either UMA mapping or I/O access unless there is some way to disable a range (1 Mbytes for example) of the main memory. On some latest PC, they do have an option in the BIOS setup to disable one Mbytes of the main memory for video subsystem to use (called memory hole, usually at 15 to 16 Mbytes). Turning on this option in BIOS setup will enable us to use extended memory for video capture again.

5.3. UMA Mapping Approach

When using UMA mapping, a segment of memory in this UMA region (from 640K to 1Mbytes) needs to be reserved as a window for mapping the video memory to main memory. The segment size can be programmed in the VMC+ as 16K, 32K or 64K.

As an example, the following table (Table 5-1) shows some available segments (in 16K page size) in the UMA region in a typical PC. The reason we choose 16K as page size is that the smaller the page size is, the more the segments could be available. These available segments can be different from PC to PC; or even in the same PC, it could be different from time to time due to changes in the system setup. Note that all the addresses in this table are "segment address", which is used by CPU to access to a physical memory location by adding this segment address with the offset address. Both the "segment" and "offset" address are only 16 bits and specific to Intel's 80x86.

Table 5-1: UMA address and range

Frame Buffer Base Address	Address Excluding Rang
B000	B000 - B3FF
B400	B400 - B7FF
D000	D000 - D3FF
D400	D400 - D7FF
D800	D800 - DBFF
DC00	DC00 - DFFF
E000	E000 - E3FF
E400	E400 - E7FF
E800	E800 - EBFF
EC00	EC00 - EFFF

In the application software of DPC7167 demo board (Figure 7.1), we provide the above 10 entries for video capturing via UMA access. Again, they are only for reference. The system programmers of VMC+ should find out and make his/her own list of available segments.

Nevertheless, the segment you choose for VMC+ for video capturing today may be invalid and cause conflicts with other application software tomorrow due to change in UMA arrangement. Therefore a better solution for this issue is to reserve a piece of memory in UMA every time when you turn on your machine; and set up your video application software to always use this piece of memory. This could be done by adding a statement in the top of your CONFIG.SYS file under the root directory. For example;

DEVICE = C:\DOS\EMM386.EXE RAM X=D000-D3FF

After the execution of this CONFIG.SYS, UMA ranged from D000 to D3FF has been reserved and no other application (device drivers) can use until the booting process is finished. Thus ensure this range of memory can be used by our video application.

Following two examples will show you how to use the UMA-feature registers of VMC+ to do video capturing. Please refer to page 110-111 of VMC+ data book for more information.

Example 2: To set up VMC+ for video capturing to use UMA ranged from DC00 to DCFF (16K bytes of page size) for a total video memory 1Mbytes. The UMA-feature registers can be programmed as follows :

Step 1. Set MemBaseBank (0x10) to 0 (refer to page 109, VMC+ data book), since we only have 1 Mbytes video memory.

Step 2. Set MEMCtrl (0x17) to 0 (refer to page 112, VMC+ data book) to select 16K bytes as the page size for UMA access.

Step 3. Set UMAbase (0x18) to 0x17 (refer to page 112, VMC+ data book) to form a base starting address at DC00 (absolute address DC000)

UMA area base = 512K bytes + UMAbase * 16K

UMAbase = (UMA area base - 512K) / 16K
= (0xDC000 - 0x80000) / 0x4000
= 0x17

Step 4. Set UMAbank (0x19) from 0 to “PMAX” once a time, step by step with increment value of 1. “PMAX” is the last page number and calculated as:

PMAX = (total video memory / page size) - 1

For the above example, PMAX is (1Mbytes / 16Kbytes) - 1 = 63. Thus the programmer needs to set this register to 0 at first time, wait for the first block transfer to complete, then set to 1 at second time, wait for transfer to complete, set to 2 at the third time... and finally set to 63 at the last time and then transfer the last block.

Example 3: If there are 512K bytes video memory and the page size is 32K bytes which are to be mapped to UMA ranged from B000 to B7FF. For video capturing on such system, the UMA-feature registers will be programmed as follows :

Step 1. Set MemBaseBank (0x10) to 0 (refer to page 109, VMC+ data book), since we have less than 1 Mbytes video memory local to VMC+.

Step 2. Set MEMCtrl (0x17) to 1 (refer to page 112, VMC+ data book) to select 32K as page size.

Step 3. Set UMAbase (0x18) to 0x06 (refer to page 112, VMC+ data book). This value is obtained by the following calculation:

UMA area base = 512K bytes + UMAbase * 32K bytes

UMAbase = (UMA area base - 512K) / 32K
= (0xB0000 - 0x80000) / 0x8000
= 0x06

Step 4. Set UMAbank (0x19) from 0 (the 1st bank) to [(512K bytes / 32K bytes) - 1 = 15] (the last bank) for a complete and sequential transfer of the entire frame buffer to main memory on the system.

The following sample program shows how to set those registers of SAA7195A for UMA access approach.

```
/* ************************************************************
    fpCopyBuffer          : Main memory pointer
    glpFrameBuffer        : Video memory pointer
    fpCopyBufferT             : Main memory pointer
    glpUMAFrameBuffer     : Main memory (UMA) pointer
 ************************************************************ */
    SAA7195A_SetReg(MemBaseBank, 0x00);      // [1]p109
    SAA7195A_SetReg(MemCtrl, 0x00);          // [1]p112
    SAA7195A_SetReg(UMAbase, gwUMABase);     // [1]p112
```

```
for (i = 0; i <= PMAX ; i++)
{
      SAA7195A_SetReg(UMAbank, i);
      FullCopyBytes((BYTE huge*)fpCopyBufferT + (LONG)(0x4000L * i),
            glpUMAFrameBuffer, 0x1000L);
} /* for i */

 RectCopyBytes((BYTE huge*)fpCopyBuffer, gwWidth * 2,
      (BYTE huge*) fpCopyBufferT, 640 * 2,
      0, 0, gwWidth * 2, gwHeight);                  /* MAPA.ASM */
```

6. VIDEO CAPTURING TO PC SYSTEMS VIA I/O MAPPING APPROACH.

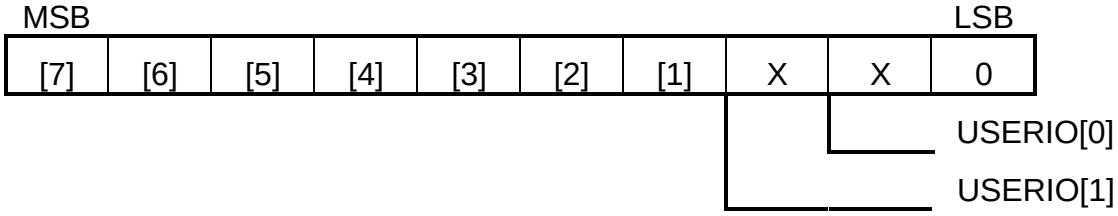
All the registers in VMC+ are accessed by host system via I/O port access. However, before using this device, it has to be initialized as follows: Up to four SAA7195A can be handled in one system. The status of the USERIO[1:0] lines in the RESET phase (i.e. when the RESET pin is held HIGH) determines the Chip-ID which is part of the base address (see below). While latching the USERIO[1:0] status in the RESET phase, these pins are automatically set to input. After RESET these pins are freely programmable as input or output ports.

After RESET the PC-Interface is in idle condition. It will not react on any input. In order to get the interface working the port address (IO-Base address) has to be set first. This is done by transmitting a special pattern (see below) to the parallel printer (Centronics) port address (278h).

Parallel Port Address	1. Header	2. Header	3. Header	4. Header	5. Header	6. Header	7. Port address
278	CD	48	EA	67	C6	EB	XX

Note: all numbers in hexadecimal representation.

So the VMC+ port address is configured from of the bits [7:1] of the 7th header byte (above figure) and the USERIO[1:0] settings as follows:



Note: The numbers in brackets are the bit numbers of the header. Bit[0] is not used, the LSB of the port address is always set to 0. The status of USERIO[1:0] is latched during RESET only.

Example 4: If we set the USERIO[1:0]=00 of VMC+ and choose 0x318 as port address, we should transmit the address (see below) within a special pattern to the parallel printer. Since

Port address = 0x318 = 11 0001 1000 (in binary)

so

MSB									LSB
[7]	[6]	[5]	[4]	[3]	[2]	[1]	X	X	0
1	1	0	0	0	1	1	0		

0xC6

Pattern as follows:

Parallel Port Address	1. Header	2. Header	3. Header	4. Header	5. Header	6. Header	7. Port address
278	CD	48	EA	67	C6	EB	C6

In order to set the port address of VMC+, we should transmit the pattern to parallel port as the following C demo program :

```
// Name:            WriteIOBase
// Description: Set VMC's I/O base address
// Parameter:       Base I/O base address
// Note:            Use "base | 0x400" for version 1
void CI2cDrv::WriteIOBase(unsigned base)
{
    if ((base != 0) && (m_nVmc != -1))
    {
        _disable();                              // Disable interrupts

        unsigned lpt2 = LPT2 | (base & 0xFC00);    // get upper bits...

        _outp(lpt2, 0xCD);                  // Write header to LPT2:
        _outp(lpt2, 0x48);
```

```
            _outp(lpt2, 0xEA);
            _outp(lpt2, 0x67);
            _outp(lpt2, 0xC6);
            _outp(lpt2, 0xEB);

            _outp(lpt2, (base >> 2) & 0xFF);          // Write Port address

            _enable();                                // Enable interrupts
        }
}
```

In order to write data to any control register, their index (address of the register) has to be selected first. The index is set in the port address (IO-Base address). The contents (data) of the register are written via the ("port address" + 1) (as shown in Figure 6.1) in the WRITE slow or fast mode defined by register "Command" (reg. 01, bit 2).

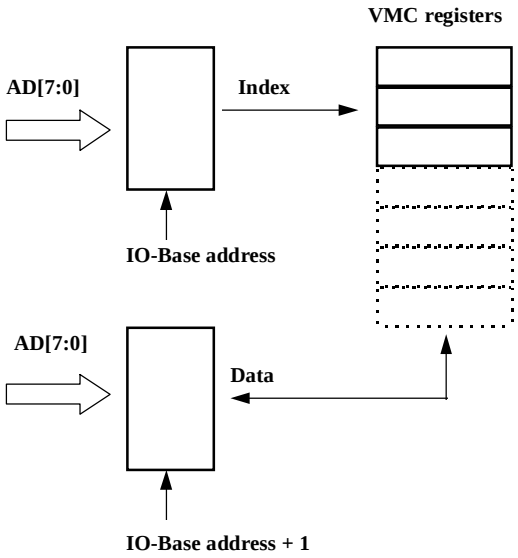


Figure 6.1 Registers access via I/O port

If we use IO-mapped approach to access video memory, it is necessary to program IO-feature registers of VMC+. For examples: MemIndex, MemData and MemInc. The following figure describes the principle operation.

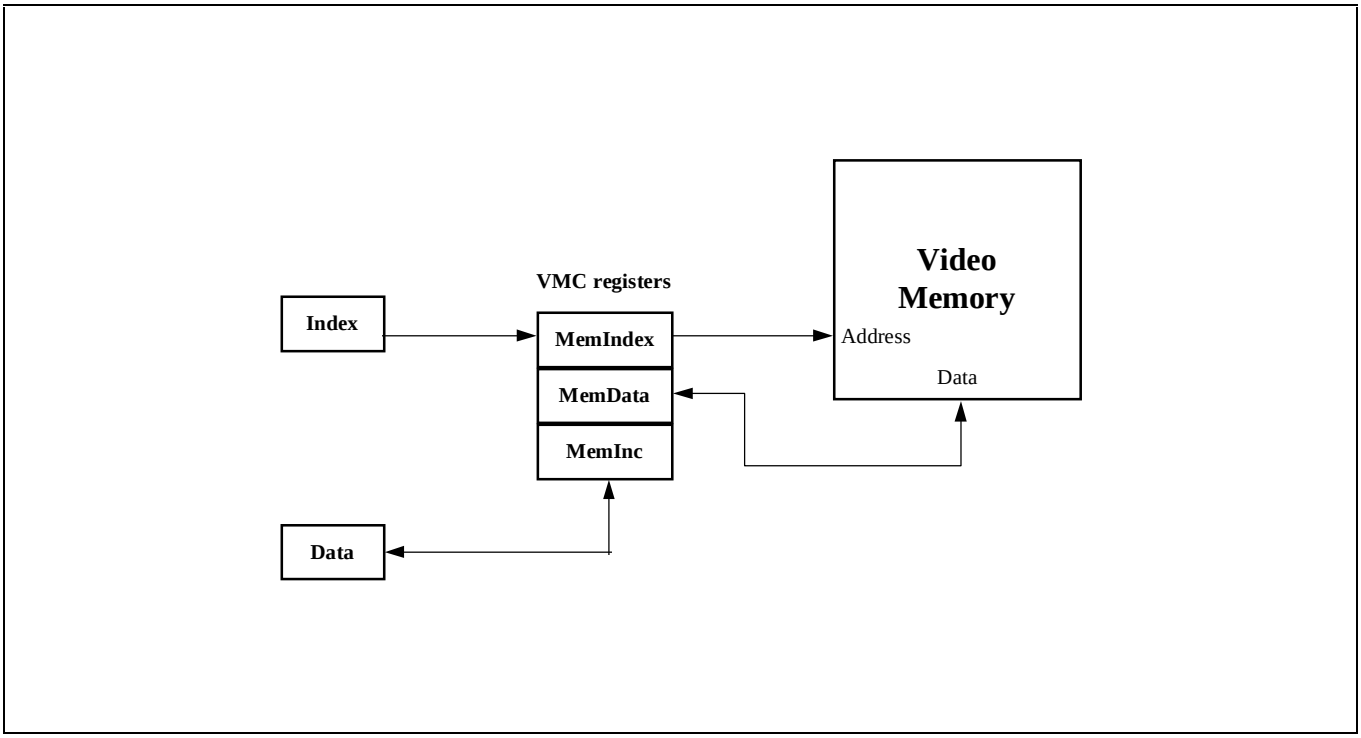


Figure 6.2

Example 5: If there are 1M bytes video memory in single bank mode, we should program the registers of VMC+ as follows:

Step 1. Set Command (0x00) to 0x44 (refer to page 104, VMC+ data book), since we access single bank video memory using IO-mapped approach.

Step 2. Set MemInc (0x14 and 0x15) to 2 (refer to page 97 and 107, VMC+ data book). For normal case, set this register to 2 for single bank mode and 1 for mux mode.

Step 3. Set MemIndex (0x11, 0x12 and 0x13) to the location in the video memory which the application software wants to access for video capturing.

Step 4. Check the bit 7 of Status register (0x01) (refer to page 105, VMC+ data book). Read memory data from the following register only when this bit is set to 1.

Step 5. Get the video data via MemData register (0x16) (refer to page 107, VMC+ data book).

The following sample program (in Macro Assembly) will show you how to set the registers of VMC+ for video capturing with IO-mapping approach.

```
; %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;   IO1BankCopy -
; %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
cProc   IO1BankCopy, <NEAR, PASCAL, PUBLIC>, <SI,DS>
        parmD    pDst       ; Pointer to dest memory
        parmW    wDstWidth      ; Width of a dest scanline (in bytes)
        parmW    wIOPort  ; I/O base port (Default: 318h)
        parmW    wSrcWidth      ; Width of a src. scan line (in bytes)
        parmW    xSrc       ; byte offset into scan
        parmW    ySrc       ; scan to start from
        parmW    dxSrc      ; number of bytes (per scan) to copy
        parmW    dySrc      ; number of scans to copy

cBegin
        CLD
        PUSH     ESI
        PUSH     EDI

        MOVZX    EDI, DI
        LES      DI, pDst

; Prepare for memory increment [1]p107
; for MemInc  14 <- 2
        MOV              DX, wIOPort
```

```
        MOV         AL, 14h
        OUT         DX, AL

        INC         DX              ; Data Port
        MOV         AL, 02h   ; [1]p97 Single bank pixel increment = 2
        OUT         DX, AL

; for MemInc  15 <- 0
        MOV         DX, wIOPort
        MOV         AL, 15h
        OUT         DX, AL

        INC         DX              ; Data Port
        MOV         AL, 0
        OUT         DX, AL
; Finish memory increment

; Initial memory index for VMC+ [1]p107
        MOV         ESI, 0
        MOV         EBX, ESI

        MOV         CX, dxSrc           ; 1 Pixel = 2 bytes
        SHR         CX, 1

NextLineIO1:
        PUSH        CX

; Prepare for memory index [1]p107
; for I/O base MemIndex  11 <- Memory Index (Address) low byte
        MOV         DX, wIOPort
```

```
    MOV     AL, 11h
    OUT     DX, AL

    INC     DX          ; Data Port
    MOV     AL, BL
    OUT     DX, AL
    SHR     EBX, 8

; for I/O base MemIndex  12 <- Memory Index (Address) middle byte
    MOV     DX, wIOPort
    MOV     AL, 12h
    OUT     DX, AL

    INC     DX          ; Data Port
    MOV     AL, BL
    OUT     DX, AL
    SHR     EBX, 8

; for I/O base MemIndex  13 <- Memory Index (Address) high byte
    MOV     DX, wIOPort
    MOV     AL, 13h
    OUT     DX, AL

    INC     DX          ; Data Port
    MOV     AL, BL
    OUT     DX, AL
; Finish memory address

NextPixelIO1:
    MOV     DX, wIOPort
```

```
    MOV     AL, 01h     ; misc state register
    OUT     DX, AL

    INC     DX          ; Data Port
NotReady1:
    IN      AL, DX
    TEST    AL, 80h
    JNZ     NotReady1

; Get a pixel data
    MOV     DX, wIOPort ; Read 2 bytes
    MOV     AL, 16h
    OUT     DX, AL

    INC     DX          ; Data Port

    IN      AL, DX      ; 1st byte (2 bytes per pixel)
    MOV     ES:[EDI], AL
    INC     EDI
;   INSB                ; Please tell me why it does not work !

    IN      AL, DX      ; 2nd byte (2 bytes per pixel)
    MOV     ES:[EDI], AL
    INC     EDI
;   INSB                ; Please tell me why it does not work !

    LOOP    NextPixelIO1

; Prepare memory index for next line VMC+ [1]p107
    MOVZX   EBX, wSrcWidth  ; (in bytes) 1 pixel = 2 bytes
```

```
    ADD         ESI, EBX            ; [1]p97 Single bank pixel increment = 2
    MOV         EBX, ESI

    POP         CX

    DEC         dySrc
    JNZ         NextLineIO1

    POP         EDI
    POP         ESI
cEnd        ; IO1BankCopy
;  ---------------------------------------------------------------
```

7. MISCELLANEOUS ADJUSTMENTS OF VMC+ ON VARIOUS SYSTEM CONFIGURATION

Due to the complexity of current PC systems and to make the end users be adapted with a video capture card more convenient, sometimes it's necessary to have the automatic detection of some system configuration in the application software. Following sections are two typical examples. Although they are not directly related to the VMC+, they are important from a system point of view.

7.1. Detection Of The Frontend Video Decoders

Currently there are two main types of frontend video decoder from Philips Semiconductors which can convert the analog CVBS signal to the digital YUV or RGB data; i.e. SAA7110 and SAA7111. A typical video subsystem with SAA7110/11 and SAA7195A based on IBM-PC main system is as follows

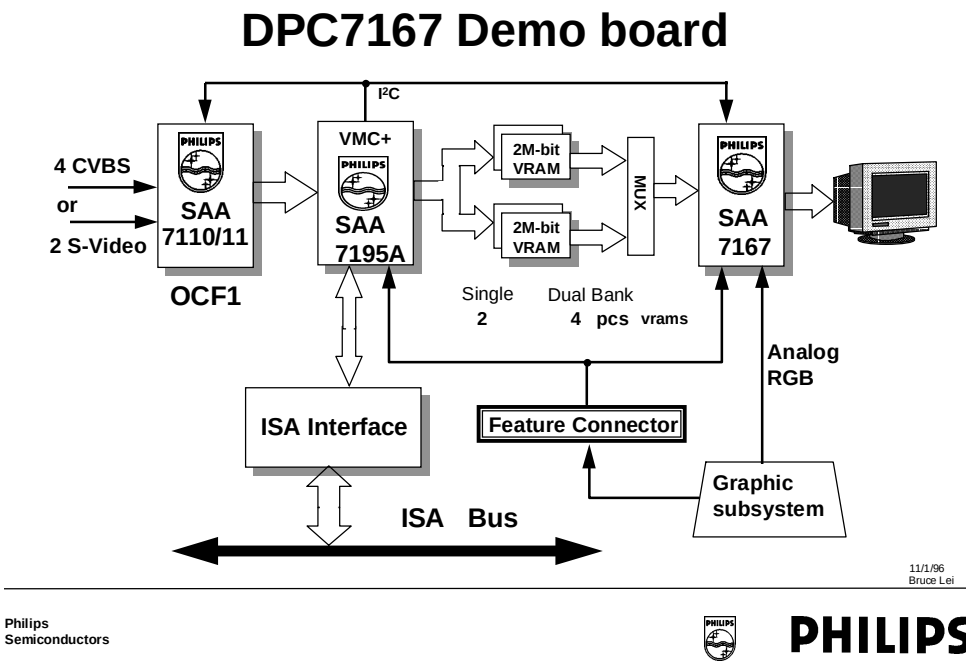


Figure 7.1

We will use the architecture of this demo board as an example to illustrate the methods we adopted in this chapter. The setup and programming of SAA7110/11 and SAA7195A may be different from what we discuss here if the system configuration is not the same as above one.

On this DPC7167 demo board, either SAA7110 or SAA7111 can be plugged in. They are fully compatible in hardware; but are completely different in register setting and slave address (for I^2C access, also named module address). Therefore the simplest way is to detect which one by sending data to its specific slave address. If no "acknowledge" signal comes back from the desired slave device, then this device is not present on the board[6]. The software should switch to the other slave address for testing. Unless there are hardware problems on board, otherwise one of the slave addresses (for SAA7110 or SAA7111) should respond to this detecting procedure.

This process can be extended if there are more than above mentioned devices or if there are I^2C devices of which the slave address unknown. We can simply do a address scan from "00" to "FF" (for current I^2C device, there is only one byte for the address). While scanning to any specific address, if there is an "ACK" signal received, then that specific device is confirmed to be on the board. Following is a piece of sample code to do the address scan:

```
/*  ------------------------------------------------------------------------------------------------------------
----
I2C_Scan -
Return:
A pointer to an global array which holds all the slave addresses of the I2C devices it found in that
particular system.
------------------------------------------------------------------------------------------------------------
*/
static  int     bI2C_addr[256], *pTmp;    /*  bI2C_addr[ ] is an global array that holds all the addresses
                                              it found in a ascending order, last one in this array is followed
                                              by consecutive zero to the end of the array. */
int* FAR PASCAL I2C_Scan (void)
{
   int     iMad;
   BOOL    bError;

   pTmp = bI2C_addr;
```

[6] When there is no "ACK" signal back from the desired device, it means either this device is not present or it's there but there is hardware problem on the board. We exclude the latter case since we assume that all the boards shipped to users are good boards.

```
  for (iMad = 0x0; iMad < 0x100; iMad++)
  {
        I2C_Start();                    /* perform a "START" condition  */
        I2C_TxByte((BYTE)iMad);    /* Send a slave address on I2C bus  */
        bError = I2C_GetACK();       /* Check response (any "ACK" signal on I2C bus); if bError is
                                        FALSE, that means there is an I2C device with this slave
                                          address in this system.   */
        I2C_Stop();                   /* perform a "STOP" condition, this is necessary whether there is
                                        a slave device or not, otherwise bus might be halted.  */
        if (! bError)
          *pTmp++ = iMad;
  } /* for iMad */

 return (bI2C_addr);
} /* I2C_Scan */
```

We don't want to list the codes of those functions of I2C_Start(), I2CGetAck(), I2CTxByte()... etc. in this note, since they are lengthy and out of the scope of our topics. However, all of these information can be found in the VMC+ data book from page 184 thru page 193. For more information of how to use I^2C or its protocol, please refer to the Philips IC12 data handbook - I^2C peripherals.

7.2. Detection Of The Graphics Modes And Adjustments Of VMC+ In PC Environment

When VMC+ is used in PC environment, usually it needs to get information from the graphics subsystem (via feature connectors in most cases) to perform the correct scan rate conversion and color key overlay tasks. However, VMC+ won't be able to tell the current graphics mode (pseudo color or high color) by the information it receives from the feature connector. Instead of sending the information about the current graphics mode by users, we implement this procedure in the imbedded code of our application software. When the software is activated, this code will get the VGA mode information from the system and set up VMC+ accordingly. VMC+ can support color key in pseudo color (16 or 256 colors) and high color (32k or 64k color) modes (not true color yet). There are mode 1 and 2 of high color modes; mode 1 is an older and less popular mode in the systems nowadays. The setup of VMC+ could be almost the same as pseudo color. Therefore, when we say high color; we mean mode 2. A simple timing diagram for these graphics modes is as follows (Figure 7.2).

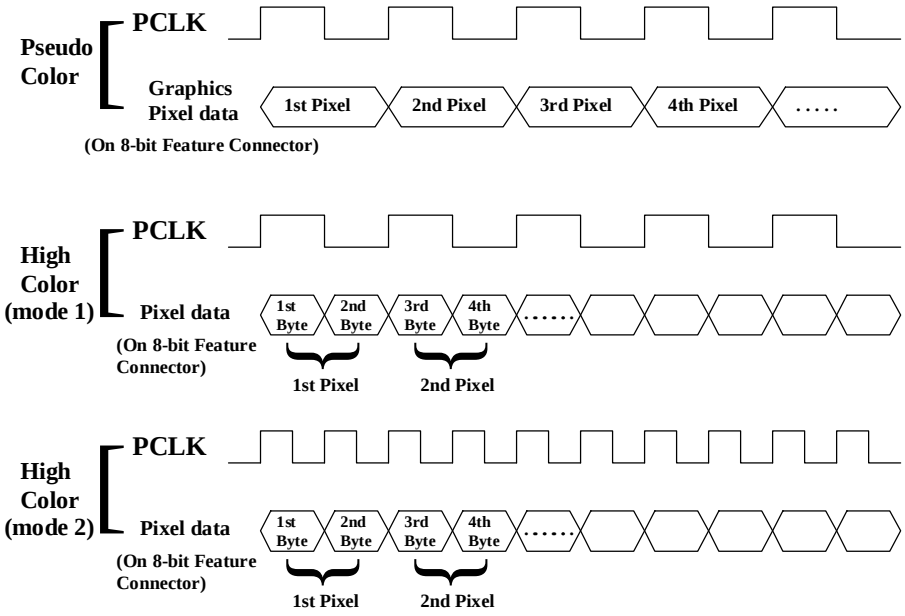


Figure 7.2

* Special remarks on Hi-color mode:

1). Hi-color mode 1: The first kind of Hi-color mode introduced to the market and less popular now, which utilizes both edges of PCLK to latch one byte of pixel data (one pixel is composed of 2 bytes data in Hi-color mode). Therefore, one complete PCLK cycle can carry two-byte (which is one pixel) information. And the PCLK frequency is then the same as in Pseudo-color mode. For the above reasons, while in Hi-color mode 1, we can do all the programming and H/W setup as in Pseudo-color mode (except the color key value).

2). Hi-color mode 2: the more popular one being used now. As a regular clocking system, each byte is transferred on each clock cycle. Hence one pixel needs two PCLK cycles to transfer. Therefore the PCLK frequency is usually doubled than in Pseudo-color mode for the same screen resolution.

The following sample code shows you how to use Windows Standard API to find out the correct graphics modes. Unfortunately, there is no way to find out its mode number when the system is in high color mode. This software always assumes mode 2 if high color mode is detected. After executing the following code, VMC+ will be set up properly (by another piece of code) according to the status the software detects. If after the whole process is done and high color mode is detected; but the display is still not correct (scale up horizontally), then it's very likely that it is high color mode 1. Consequently, the users need to do some corrective setup to VMC+ manually to fix the problem.

```
/* ------------------------------------------------------------------------------------------------------------
----
The function of DPC7167_VgaMode_Detect( ) can only get the "Bits" and "Planes" info from the system, the desired mode information need to be evaluated from these two numbers in a certain way as showing in the following codes.

 Examples of "Bits" and "Planes" values of various graphics modes.
        Modes                   Bits    Planes
        Standard VGA              1       4
        VGA (v3.0)                1       4

        Modes                           Bits    Planes
        640 x 480 x 16   (pseudo color)   1       4
        640 x 480 x 256 (pseudo color)    8       1
        640 x 480 x 32K (high color)      16      1
        640 x 480 x 64K (high color)      16      1
        640 x 480 x 16.8M (true color)    24      1
------------------------------------------------------------------------------------------------------------
*/

UINT FAR PASCAL EXPORT DPC7167_VgaMode_Detect (void)
{
```

```
   HDC hDC;
   int iBits;
   int iColorPlanes;

   hDC = CreateIC("DISPLAY", NULL, NULL, NULL);   /* MS-Windows API */
   iBits         =   GetDeviceCaps(hDC, BITSPIXEL);      /* MS-Windows API */
   iColorPlanes   =   GetDeviceCaps(hDC, PLANES);   /* MS-Windows API */
   DeleteDC(hDC);                                          /* MS-Windows API */

   switch (iBits * iColorPlanes)
   {
      case 4:
      case 8:            /* 16 or 256 colors, Pseudo Color */
         return( 0 );
         break;

      case 16:           /* 64K colors, Hi-Color */
         return( 1 );
         break;

      case 24:           /* 16M colors, True Color */
      default:
         return( 2 );
         break;
   }  /* end of “switch” */
}   /*  end of routine */
```

After successfully detecting the graphics modes, the software will need to program “CKeyMode” (reg. 42, bit 5&6) of VMC+ with the correct mode first; then set up the color-key registers “CKeyComp” (reg. 49) & “CKeyCompHi” (reg. 4A) with the correct value carefully. Except the standard VGA mode from Microsoft, the color values of all the other modes really depend on the VGA chip vendors. Although there is no standard for this, for your reference, the following values are found in many graphics subsystems:

	Reg.49	Reg.4A	Bit 5&6 of Reg. 42
Standard VGA mode:	2D	X	01
16-color pseudo-color :	0D	X	01
256-color pseudo-color:	FD	X	01
32K-color Hi-color (mode1):	1F	X	01
64K-color Hi-color (mode1):	1F	X	01
32K-color Hi-color (mode2):	1F	7C	1x
64K-color Hi-color (mode2):	1F	F8	1x

For DPC7167 demo board, In case that SAA7195A is not able to generate the color key on boards. The key signal required for the mixer in SAA7167 can still be generated by SAA7167 internally (please refer to Figure 7.1). This process could be done by programming the registers of SAA7167 properly. However, while in hi-color (or true-color) mode, SAA7167 can only generate the color-key with all the same values of each bytes within that pixel.

8. REFERENCE

1. “Video and Memory Controller (plus) - SAA7195A” data book, Mar. 1994.
2. Application Note - AN95056 “User’s Manual DPC7167 Demo Board”
3. Application Note - AN95058 “Source code DPC7167 Demo Board Video for Windows”
4. 1995 Data Handbook IC22, Philips Semiconductors.
5. “DOS Internals” - Geoff Chapell, Addison-Wesley, 1994.