Graphics applications with TRAVEO™ T2G family

Cluster 2D series automotive microcontrollers

About this document

Associated part family

The following series of TRAVEO™ T2G family automotive microcontrollers:

- CYT4DN revision A

Associated driver software

- TRAVEO™ T2G family automotive microcontroller sample driver library (SDL) release V7.1.0
- TRAVEO™ T2G family automotive microcontroller graphics driver V0E.6.0

Scope and purpose

This document describes how to create specific applications which use the graphics functions of TRAVEO™ T2G family cluster 2D series automotive microcontrollers. See the table of contents to to locate applications described in this document. This document also describes topics which are not included in the graphics driver manual, such as clock routing. In addition to this document, there are other documents and tutorials that describe graphics applications that cover topics that are not covered here.

You should read the TRAVEO™ T2G graphics driver manual **[5]** before reading this document.

See also AN233199 - TRAVEO™ T2G family cluster series automotive microcontrollers – Video subsystem hardware overview**[6]** to learn about the TRAVEO™ T2G family automotive microcontroller graphics hardware.

This document explains how to use the sample driver library (SDL)**[4]** APIs to set the peripheral hardware if the APIs support it.

Intended audience

- Engineers who finished tutorials provided in the TRAVEO™ T2G family automotive microcontroller graphics driver manual.
- Customers who develop graphics applications which have specific functions described in this document with TRAVEO™ T2G family cluster 2D series automotive microcontrollers
- Internal/external engineers who learn graphics application with TRAVEO™ T2G family cluster 2D series automotive microcontrollers

Table of contents

Table of contents

1 Graphics application

This chapter describes the graphics application to be introduced in this document. The application uses two displays, called “Display 0” and “Display 1”.

Display 0 is intended to be used for automotive clusters, where scene switching can be done on the display. See **Cluster display (Display 0)**.

Display 1 is intended to be used for head-up displays, where dynamic distortion calibration can be done on the display. See **Head-up display (Display 1)**.

1.1 Cluster display (Display 0)

Display 0 is intended to be used as automotive cluster, where a background image and contents can be shown. The scene of the cluster is switched at an interval by sliding the second scene in from the right of the display, and sliding the first scene out to the left at the same time. See **Figure 1**.

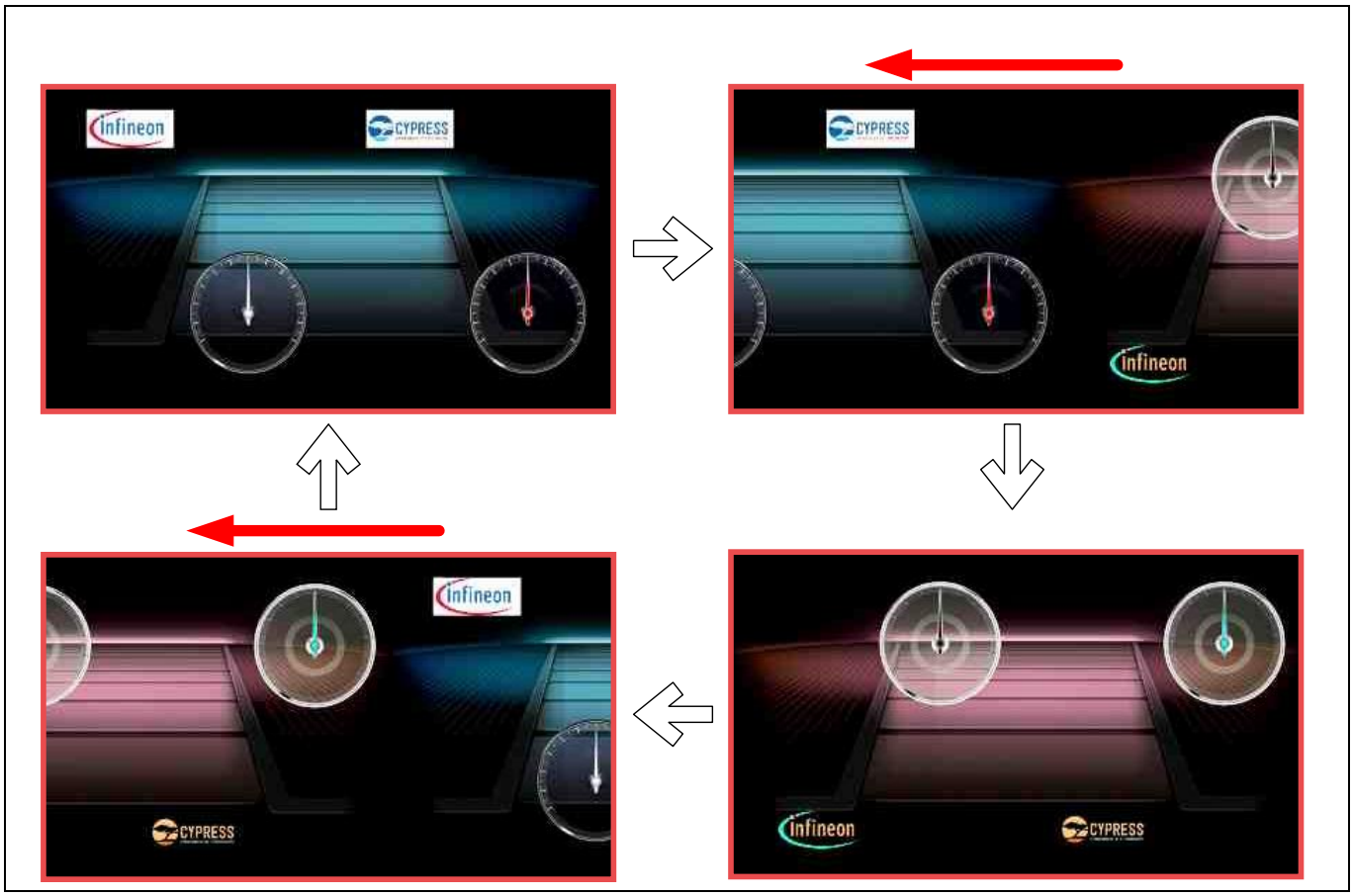

Figure 1 Cluster display overview

Table 1 Display 0 parameters

Parameter	Value	Comments
Resolution (pixel x pixel)	1280 x 720	Width total = 1650 Height total = 750
Frame rate (fps)	60	
Interface	Single LVDS	
Pixel clock frequency (Hz)	74,250,000	= 1650 x 750 x 60 = Source clock frequency / 2
Source clock frequency (Hz)	148,500,000	PLL400_4 (Path5)
Window count	3	Background window 1 Background window 2 Contents window

Figure 2 shows the data flow of this example. At first, all images used in this example are stored in external flash as shown in the figure. “2D GFX Core” copies the contents into VRAM if needed to render the next frame. The 2D GFX Core renders frames using the contents stored in the VRAM. At the same time, Video I/O reads background images from the external memory directly to the display.

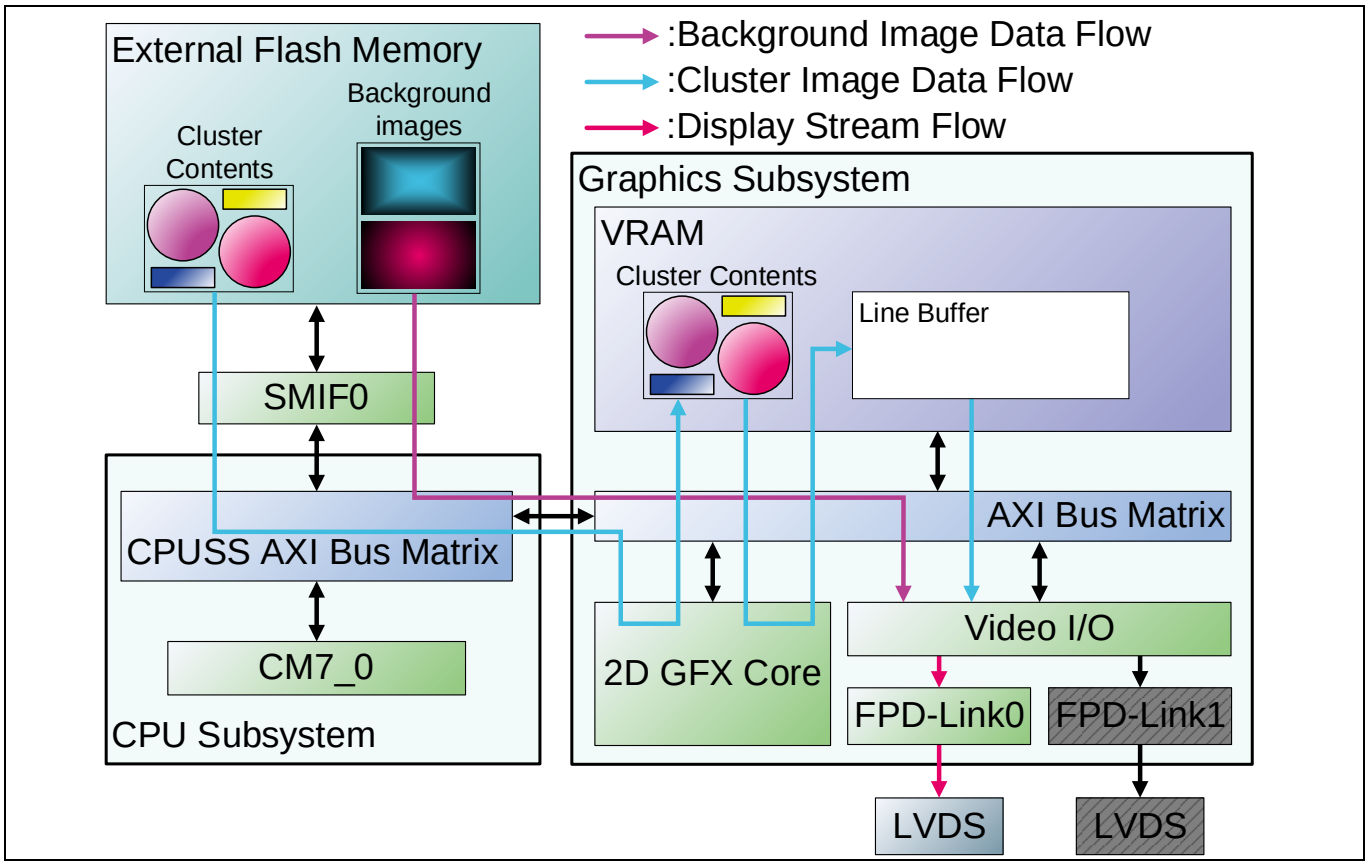

Figure 2 Cluster display data flow overview

1.2 Head-up display (Display 1)

Display 1 is intended to be used as head-up display (HUD). Display 1 normally shows only the most relevant information for a driver, called “normal mode”. You can use the buttons to change the mode to distortion calibration mode for calibrating the distortion by moving points on the grid (**Figure 3**).

The calibration is supposed to be done when the automobile is not moving. The reason is that it is not possible for a real-time driving application on both Display 0 and 1 to work properly because it would take up large amounts of VRAM and CPU resources. The distortion change is normally done in real-life applications to compensate a changed angle from the driver to the HUD (for example, when the driver’s seat position changes.)

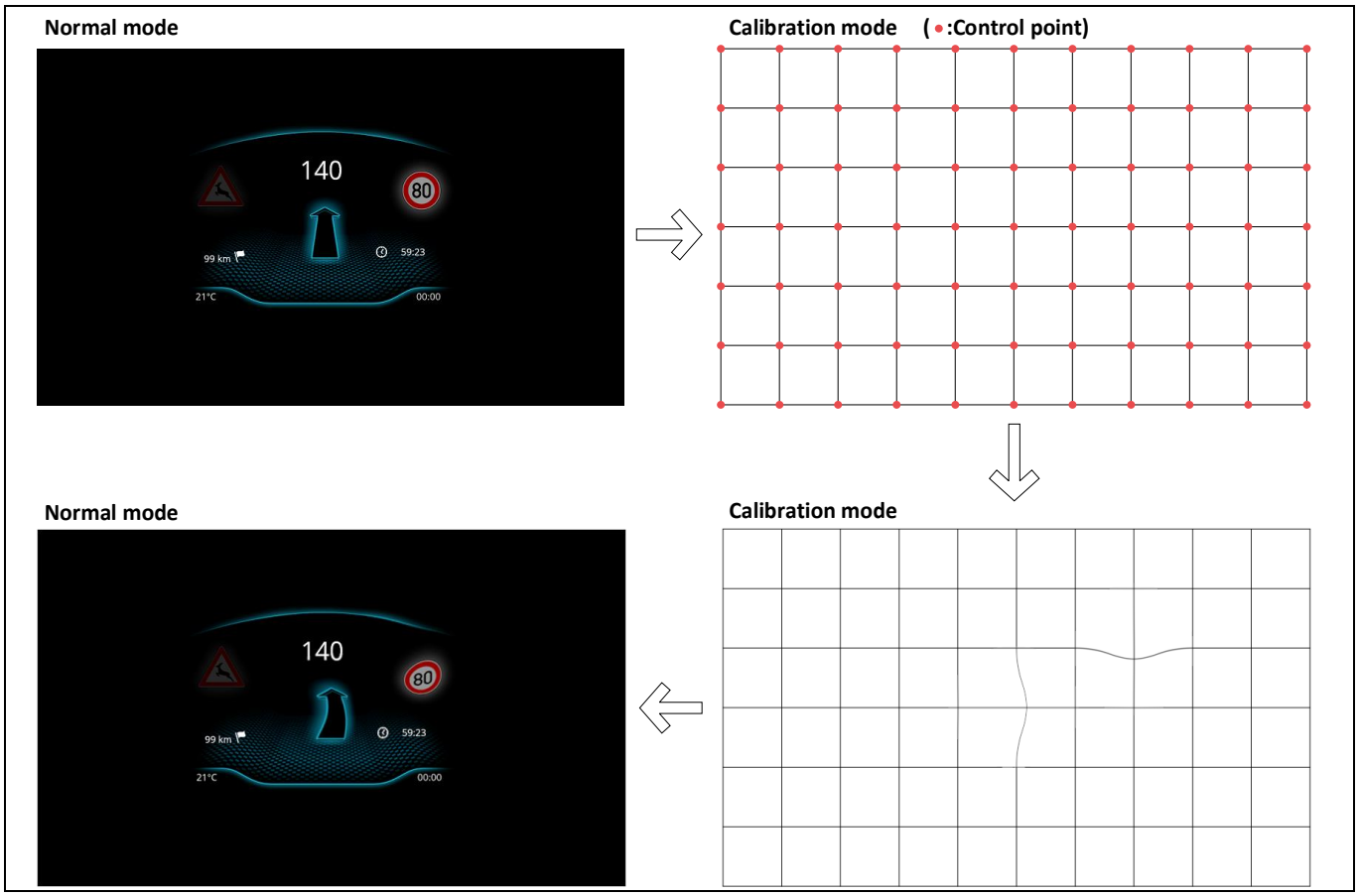

Figure 3 HUD overview

Table 2 Display 1 parameters

Parameter	Value	Comments
Resolution (pixel x pixel)	800 x 480	Width total = 992 Height total = 500
Frame rate (fps)	60	
Interface	Single LVDS	
Pixel clock frequency (Hz)	29,760,000	= 992 x 500 x 60 = Source clock frequency / 4
Source clock frequency (Hz)	119,040,000	PLL200_2 (Path8)
Window count (normal mode)	1	Warping Contents window
Window count (calibration mode)	2	Control pointer window Warping Grid window

Figure 4 shows the data flow of this example in normal mode. 2D GFX Core copies HUD contents stored in the external flash once at the very first time of the application. 2D GFX Core renders frames for the HUD during the application run time. Video I/O reads the Coordinate Buffer stored in the internal flash and distorts the frames using the coordinate data.

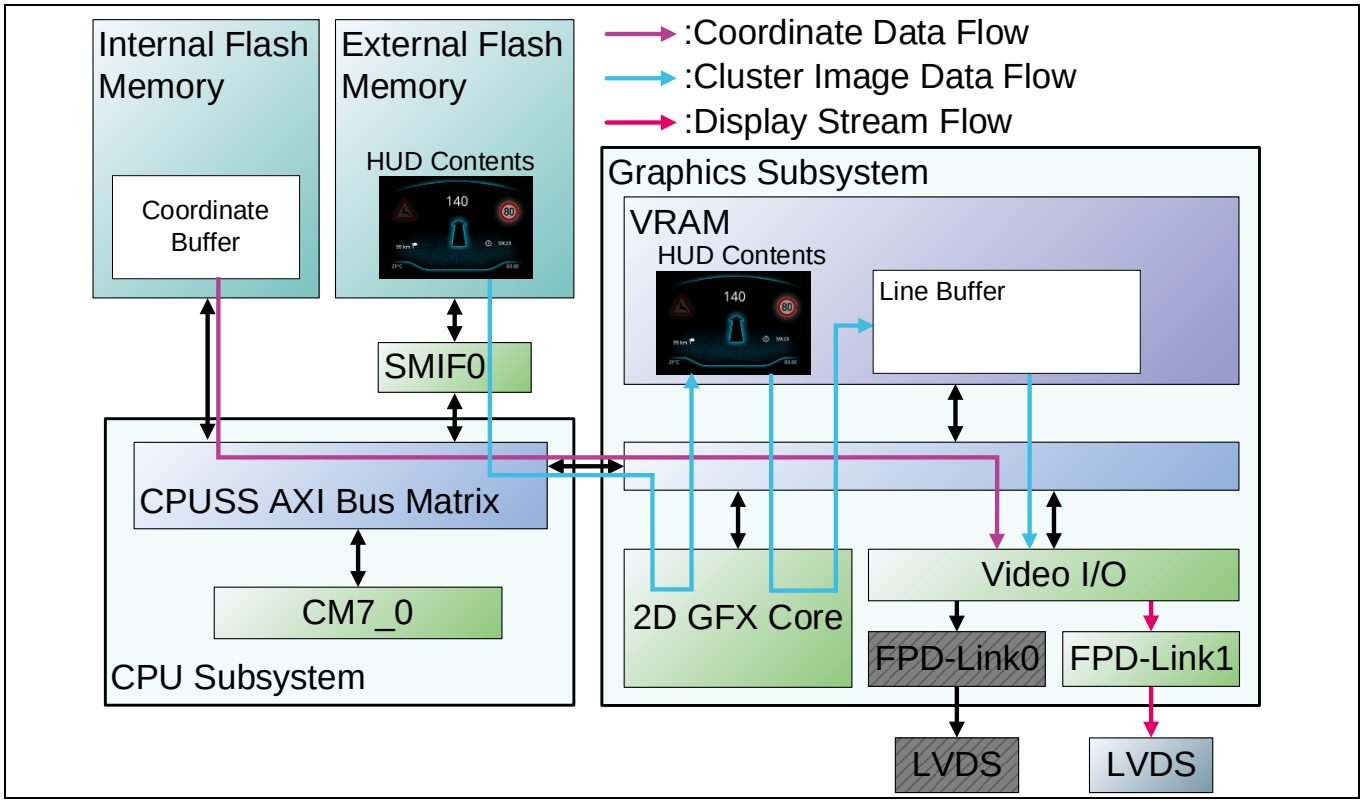

Figure 4 Warping display normal mode data flow

Figure 5 shows the data flow of this example in calibration mode. Two coordinate buffers are stored in the VRAM. The CM7_0 is modifying one and the Video I/O is using the other one. When next frame to be rendered, CM7_0 starts modifying the coordinate buffer which Video I/O had been using in the previous frame generation and vice versa.

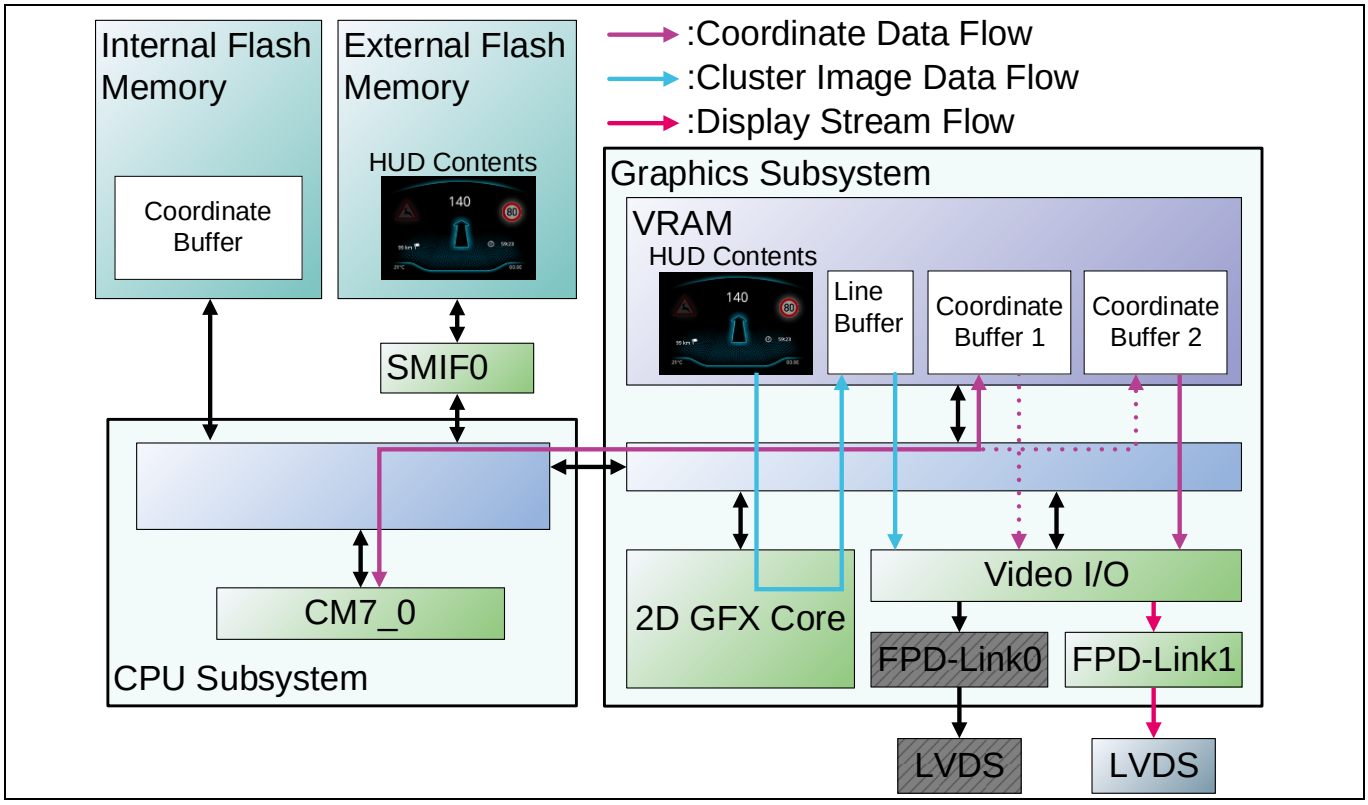

Figure 5 Warping display calibration mode data flow overview

2 Graphics environment

This section describes basic settings for graphics environment that are not covered by TRAVEO™ T2G family automotive microcontroller graphics driver manual. In addition to settings described in this section, you should set up interrupt for graphics. See the TRAVEO™ T2G family automotive microcontroller graphics driver **[5]** tutorial and application note **[3]** for interrupt and interrupt routines.

2.1 Enabling graphics hardware

At first, you should enable the graphics hardware to use TRAVEO™ T2G graphics functionalities using the following registers:

- SUBSS_VIDEOSSCFG_CTL.ENABLED
- SUBSS_VIDEOSSCFG_GFX2D_CTL.GFX2D.ENABLED

SUBSS_VIDEOSSCFG_CTL.ENABLED is the enable bit for the whole video sub system. SUBSS_VIDEOSSCFG_GFX2D_CTL.GFX2D.ENABLED is the enable bit for Graphics 2D core, which contains the blit engine, draw engine, and the command sequencer in the video subsystem. **Code Listing 1** shows an example code to enable the graphics hardware.

Code Listing 1 Enabling the graphics subsystem

```
VIDEOSS0_SUBSS0_VIDEOSSCFG->unIPCTRL.stcField.u1CTLENABLED=1;
VIDEOSS0_SUBSS0_VIDEOSSCFG->unIPCTRLGFX2D.stcField.u1CTLENABLED_GFX2D=1;
```

2.2 Clock settings

You should let the system provide the maximum-frequency clock to the graphics subsystem so that the graphics system works in the maximum performance level. Also, the system provides a specific clock frequency to the display engine so that the display engine can realize target resolution and fps. See the “Clock specifications” section in the datasheet **[1]** of the MCU you are using, and find indexes and maximum frequencies of the high-frequency clocks (CLK_HF) which are provided to the video subsystem, Display #0, and Display #1. **Figure 6** is a part of the datasheet of the CYT4DN family series; settings for other MCUs can vary.

26.7 Clock Specifications

Table 26-19. Root and Intermediate Clocks[64, 66]

Root Clock	Typ. frequency with ECO (MHz)[65]	Typ. frequency with IMO (MHz)	Source	Description
CLK_HF0	200	192	PLL200#0	Root clock for CPUSS, PERI (CLK_MEM, CLK_SLOW, CLK_PERI)
CLK_HF1	320	312	PLL400#0	CM7 CPU Core#0, CM7 CPU Core#1 clock
CLK_HF2	100		PLL200#1	Peripheral clock root other than CLK_PERI
CLK_HF3	100		PLL200#0	Event generator, clock output on EXT_CLK pins (when used as output)
CLK_HF4	50		PLL200#0	Ethernet Channel#0. Internal clock 50 MHz for RMII, External PHY provides 25 MHz for MII and 125 MHz for RGMII.
CLK_HF5	196.608		PLL400#1 / PLL400#2 / EXT_CLK	Sound Subsystem #0 root clock, ETH0 TSU clock (CLK_IF_SRSS0)
CLK_HF6	196.608			Sound Subsystem #1 root clock (CLK_IF_SRSS1)
CLK_HF7	196.608			Sound Subsystem #2 root clock (CLK_IF_SRSS2)
CLK_HF8	400		PLL400#2	SMIF#0 root clock
CLK_HF9	400		PLL400#2	SMIF#1 root clock
CLK_HF10	250		PLL400#3	Video Subsystem root clock
CLK_HF11	220		PLL200#2 / PLL400#4	Display#0 root clock
CLK_HF12	220		PLL200#2 / PLL400#4	Display#1 root clock
CLK_HF13	NA		ILO	CSV Dedicated
CLK_FAST_0	320	312	NA	CM7 CPU Core#0, intermediate clock
CLK_FAST_1	320	312	NA	CM7 CPU Core#1, intermediate clock
CLK_MEM	200	192	NA	Intermediate clock for SMIF, Flash, Ethernet, VIDEOSS
CLK_SLOW	100	96	NA	Generated by clock gating CLK_MEM, intermediate clock for CM0+, P-DMA, M-DMA, Crypto, SMIF, VIDEOSS
CLK_PERI	100	96	NA	Generated by clock gating CLK_HF0, intermediate clock for IOSS, TCPWM0, CPU trace, SMIF, Sound Subsystem, Ethernet

Figure 6 Clock specification

As you can see in the datasheet, CLK_HF10 drives the video subsystem, CLK_HF11 drives Display #0, and CLK_HF12 drives Display #1. The maximum frequencies of CLK_HF10, CLK_HF11, and CLK_HF12 are 250 MHz, 220 MHz, and 220 MHz respectively.

Note that PLL400#3 is the source of CLK_HF10, while PLL200#2 or PLL400#4 can be the source of CLK_HF11 and CLK_HF12. The minimum frequency of the PLLs are limited by FPD-Link input clock frequency specification. See “FPD-Link” section of the TRM [2] shown in **Figure 7**.

35.1 Features

- Input clock frequency 110 to 220 MHz

Figure 7 FPD-Link specification

Per the “FPD-Link” section of the TRM, the minimum frequency of the input clock of FPD-Link is 110 MHz.

The video subsystem root clock shown in **Figure 6** is used in the modules of the video subsystem. You should let this clock have its maximum frequency to realize the best graphics performance. Therefore, set the frequency of PLL400#3 to 250 MHz (its maximum frequency) and route it to CLK_HF10. See the application note [3] for detailed information on setting the clocks.

Display#0 root clock and Display#1 root clock shown in **Figure 6** are used for the pixel clock of each displays. The frequencies depend on the target resolution and frame rate (fps) of the display. **Figure 8** shows the clock flow for Display #0 if using the LVDS output. Note that this is a simplified diagram for ease of demonstration.

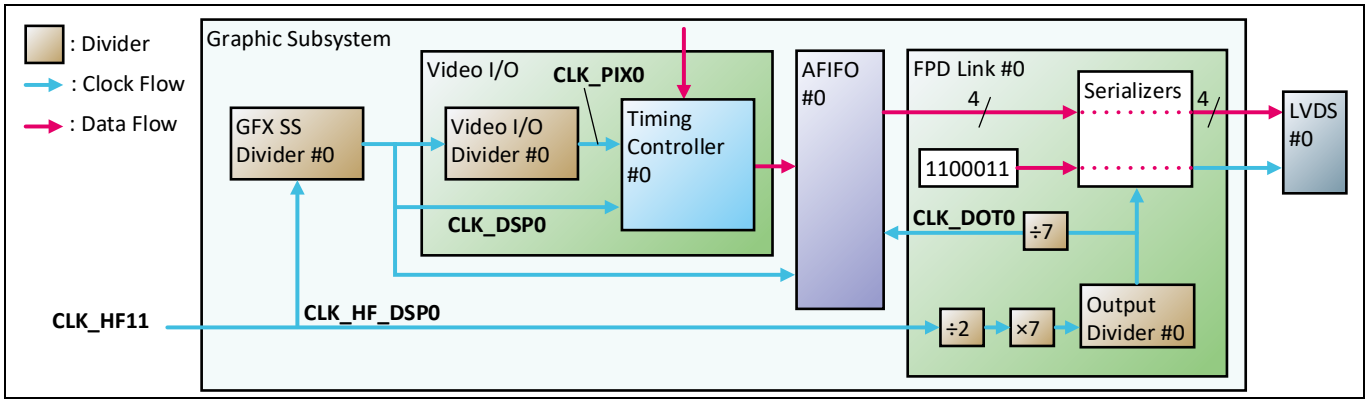

Figure 8 Clock flow for LVDS Display #0

As an example, it is assumed that the source of CLK_HF11 is PLL400#4. The relationship between clocks and dividers is shown in the **Figure 8**. There are conditions which clock frequencies must satisfy. These are listed in the following formula where FREQ_<*clock name*> represents frequency of <*clock name*>.

$$FREQ\_CLK\_DSP0/2 = FREQ\_CLK\_DOT0 = pixel\ per\ seconds \quad (1)$$

$$FREQ\_CLK\_PIX0 = FREQ\_CLK\_DSP0/2 \quad (2)$$

$$110[MHz] \leq FREQ\_CLK\_HF11 \leq 220[MHz] \quad (3)$$

Formula (1) is a condition for the graphics subsystem to achieve the display resolution and fps without overflow and underflow in the FIFO buffer (AFIFO # 0 in the **Figure 8**) located between Video I/O and FPD-Link. Formula (2) is a condition when using Single LVDS. Formula (3) is from maximum/minimum frequency of the CLK_HF, which can be confirmed by checking the datasheet and the TRM.

The dividing ratio of GFX SS Divider #0, Video I/O Divider #0, and FPD Link Divider #0 are derived from these formulas. Note that the dividing ratio of GFX SS Divider #0 and FPD Link Divider #0 can be selected from 1, 2, 4, and 8.

At first, note the dividing ratio of Video I/O Divider #0 is 2 from the formula (2).

Pixel per seconds is 74,250,000 (1650 x 750 x 60) according to the resolution and fps of the Display #0 (**Table 1**). Thus, FREQ_CLK_DSP0 is 148,500,000 Hz from the formula (1). Provided that GFX SS Divider is 1, you can derive that FREQ_CLK_HF11 is 148,500,000 Hz. Because this value satisfies the formula (3), the dividing ratio of 1 of GFX SS Divider is accurate.

Because FREQ_CLK_DOT0 must be 74,250,000 Hz in accordance with the formula (1), the dividing ratio of FPD Link Divider #0 should also be 1.

Here is a summary of clock settings for Display #0:

- Set the frequency of PLL400#4 as 148,500,000 Hz.
- Set the source of CLK_HF11 as PLL400#4.
- Set the dividing ratio of GFX SS Divider #0 to 1.
- Set the dividing ratio of Video I/O Divider #0 to 2.
- Set the dividing ratio of FPD Link Divider #0 to 1.

See application note[3] to learn more about PLL and CLK_HF settings.

Code Listing 2 shows an example of setting GFX SS Divider #0.

Code Listing 2 Setting GFX SS Divider #0

```
/* Set GFX SS Divider #0 so that it divides its input clock by 1 */
VIDEOSS0_SUBSS0_VIDEOSSCFG->unCLKDSP0CFG.stcField.u2DIVVAL0=0;
```

Code Listing 3 shows an example of setting Video I/O Divider #0.

Code Listing 3 Setting Video I/O Divider #0

```
/* Set Video I/O Divider #0 so that it divides its input clock by 2 */
VIDEOSS0_VIDEOIO0_DISENG00_DISENGCFG0->unCTL.stcField.u1DSPCLKDIVIDE=1;
```

See the section **FPD-Link settings** to learn how to set FPD-Link Divider #0.

You can configure clock settings for Display #1 in the same way as clock settings of Display #0:

- Set the frequency of PLL200#2 as 119,040,000 Hz
- Set the source of CLK_HF12 as PLL200#2.
- Set the dividing ratio of GFX SS Divider #1 to 2.
- Set the dividing ratio of Video I/O Divider #1 to 2.
- Set the dividing ratio of FPD-Link Divider #1 to 2.

2.3 FPD-Link settings

An example code setting FPD-Link 0 is shown in **Code Listing 4**. Enter the value which represents the dividing ratio derived in **Clock settings** as the second parameter of `CY_Fpdlink_FastInit`. Here, the dividing ratio is entered as 1.

Code Listing 4 Setting the FPD Link with an API in the SDL

```
/* Set the FPD-Link so that FPD Link Divider #0 divides its input clock by 1 */
Cy_Fpdlink_FastInit(Fpdlink0, FpdlinkPllOutDiv1);
```

Set FPD-Link 1 in the same way as FPD-Link 0.

2.4 TCON settings

This section describes how to set TCON0 which decides the LVDS output data mapping. For LVDS display, the input format is shown in **Figure 9** as defined in the OpenLDI specification. TCON is set such that TRAVEO™ T2G MCU can drive this display.

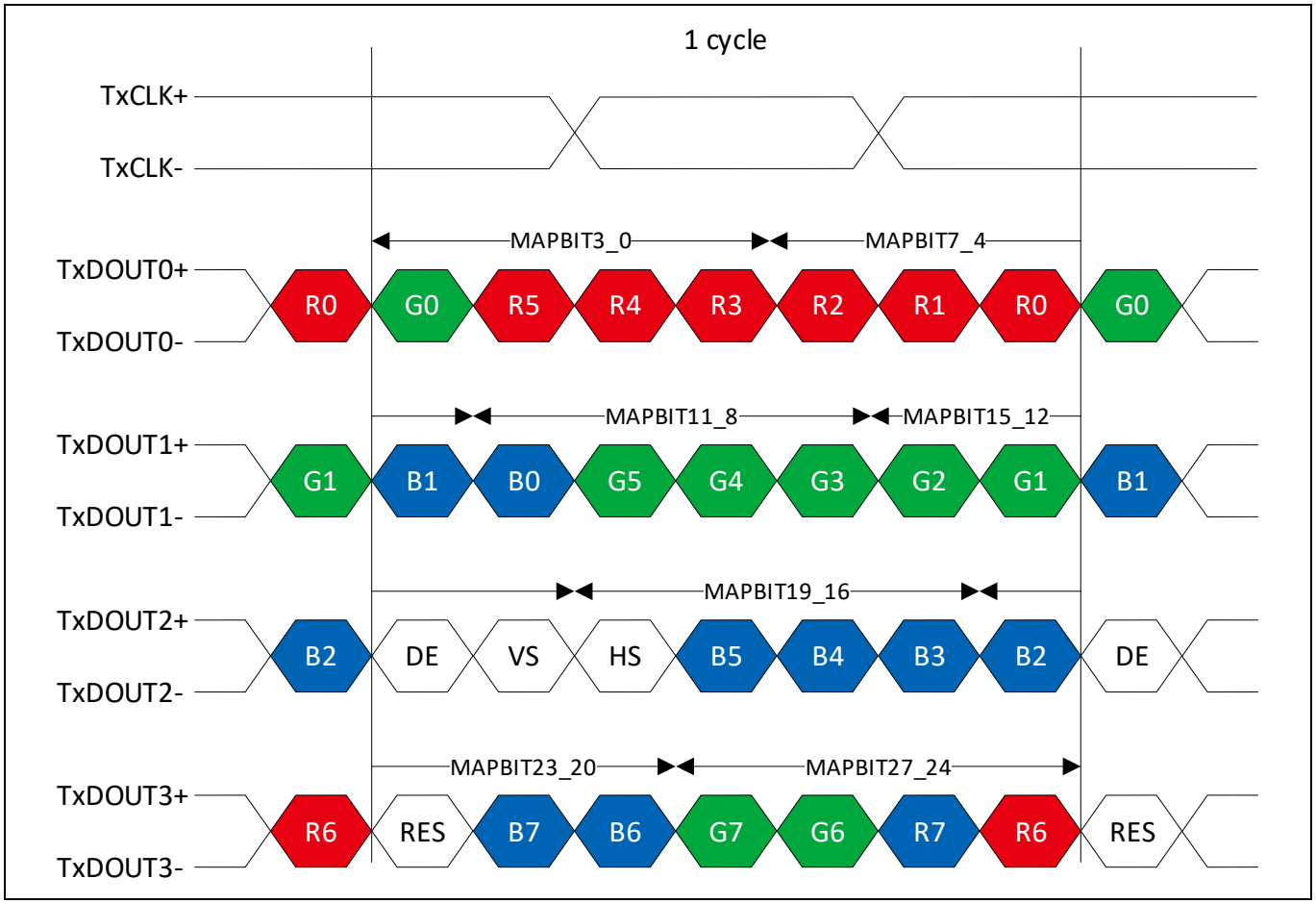

Figure 9 LVDS display map bit example

MAPBIT<*xx*>_<*yy*> in **Figure 9** represents the TRAVEO™ T2G MCU register, which sets the LVDS map bit as follows:

A mapping between the display map bit and register value is shown in **Code Listing 5**.

Code Listing 5 Map LVDS bit and MAPBIT register value

```
#define _R0_ ( 0u)
#define _R1_ ( 1u)
#define _R2_ ( 2u)
#define _R3_ ( 3u)
#define _R4_ ( 4u)
#define _R5_ ( 5u)
#define _R6_ ( 6u)
#define _R7_ ( 7u)
#define _G0_ ( 8u)
#define _G1_ ( 9u)
#define _G2_ (10u)
#define _G3_ (11u)
#define _G4_ (12u)
#define _G5_ (13u)
#define _G6_ (14u)
#define _G7_ (15u)
#define _B0_ (16u)
#define _B1_ (17u)
#define _B2_ (18u)
```

Code Listing 5 Map LVDS bit and MAPBIT register value

```
#define _B3_ (19u)
#define _B4_ (20u)
#define _B5_ (21u)
#define _B6_ (22u)
#define _B7_ (23u)
#define _HS_ (24u) // horizontal sync
#define _VS_ (25u) // vertical sync
#define _EN_ (26u) // data enable
#define _RS_ (27u) // reserved
```

An example code which sets TCON 0 is shown in **Code Listing 6**. stc_VIDEOSS_VIDEOIO_DISENG0_TCON0_t is a TCON register structure type and VIDEOSS0->VIDEOIO[0].DISENG0[0].TCON0 is an instance of the register. Both are defined in I/O header of SDL [4].

Code Listing 6 Setting LVDS MAPBIT registers

```
/* Declare a pointer to the TCON register structure */
volatile stc_VIDEOSS_VIDEOIO_DISENG0_TCON0_t* p;

/* Set the top address of the TCON0 register to the pointer */
p = &VIDEOSS0->VIDEOIO[0].DISENG0[0].TCON0;

/* Set TCON mapping registers */
p->unMAPBIT3_0.u32Register   =_G0_|(_R5_<<8)|(_R4_<<16)|(_R3_<<24);
p->unMAPBIT7_4.u32Register   =_R2_|(_R1_<<8)|(_R0_<<16)|(_B1_<<24);
p->unMAPBIT11_8.u32Register  =_B0_|(_G5_<<8)|(_G4_<<16)|(_G3_<<24);
p->unMAPBIT15_12.u32Register =_G2_|(_G1_<<8)|(_EN_<<16)|(_VS_<<24);
p->unMAPBIT19_16.u32Register =_HS_|(_B5_<<8)|(_B4_<<16)|(_B3_<<24);
p->unMAPBIT23_20.u32Register =_B2_|(_RS_<<8)|(_B7_<<16)|(_B6_<<24);
p->unMAPBIT27_24.u32Register =_G7_|(_G6_<<8)|(_R7_<<16)|(_R6_<<24);

/* Set TCON channel mode to single mode(0) */
p->unTCON_CTRL.stcField.u2CHANNELMODE = 0;

/* Set TCON bypass mode to bypass mode(1) */
p->unTCON_CTRL.stcField.u1BYPASS = 1;
/* Enable the TCON */
p->unTCON_CTRL.stcField.u1ENLVDS = 1;
```

2.5 Disabling the graphics hardware

This section describes how to disable the graphic subsystem safely. When the graphic subsystem is powered off while there are outstanding AXI transactions from CPU (and/or other sources) to VRAM, it will hang up the system interconnect. Thus, before disabling the graphics subsystem, ensure that there are no outstanding AXI accesses from CPUSS to the VRAM.

- Ensure that M-DMA(AXI-DMA) and Ethernet MAC have completed their transfers to/from VRAM, and do not set up any new transfers to/from VRAM or with the descriptor in VRAM.
- Ensure that CM7 is not executing code from VRAM.
- Ensure that no VRAM addresses are cached in the CM7's caches and that the VRAM address range is configured to non-cacheable.
- Ensure that there are no outstanding transactions from the CM7 CPU to VRAM.

After ensure no transfer to/from VRAM, execute the following registers:

- SUBSS_VIDEOSSCFG_CTL.ENABLED

- SUBSS_VIDEOSSCFG_GFX2D_CTL.GFX2D.ENABLED

SUBSS_VIDEOSSCFG_CTL.ENABLED is the enable bit for the whole video sub-system. SUBSS_VIDEOSSCFG_GFX2D_CTL.GFX2D.ENABLED is the enable bit for graphics 2D core, which contains the blit engine, draw engine, and the command sequencer in the video subsystem. shows an example code to disable the graphics hardware.

Code Listing 7 Disabling the graphics hardware

```
SUBSS_VIDEOSSCFG_CTL.ENABLED = 0;
SUBSS_VIDEOSSCFG_GFX2D_CTL.GFX2D.ENABLED = 0;
```

3 Scene switching

This section describes the scene switching application which works on Display #0. In this scene switching application, scene 1 shifts to the left and goes out of the display. At the same time, scene 2 shifts left in from the right of the display. This application is roughly divided into two portions: Procedure for background window and procedure for cluster contents window.

3.1 Data flow of the scene switching application

Data flow when the scenes switch is shown in **Figure 10**. A bus master means an I/O controlling hardware that directs data traffic on the bus or input/output paths. Three layers are used: Decode Layer 1 and Decode Layer 2 for background image, and OTF Layer for cluster contents.

During scene switching, the composition engine must fetch both background image 1, which is being moved out left and background image 2, which is being moved in right. Two decode layers read each image from the flash memory.

The blit engine copies the cluster contents from the flash memory to the VRAM. This copying must be done in the frame one before the frame in which the contents appear. The blit engine renders the contents and saves them into the line buffer. The on-the-fly (OTF) layer fetches the line buffer.

The CPU frees the VRAM space when its contents are moved completely out of the display.

See **Shifting the cluster contents** for details.

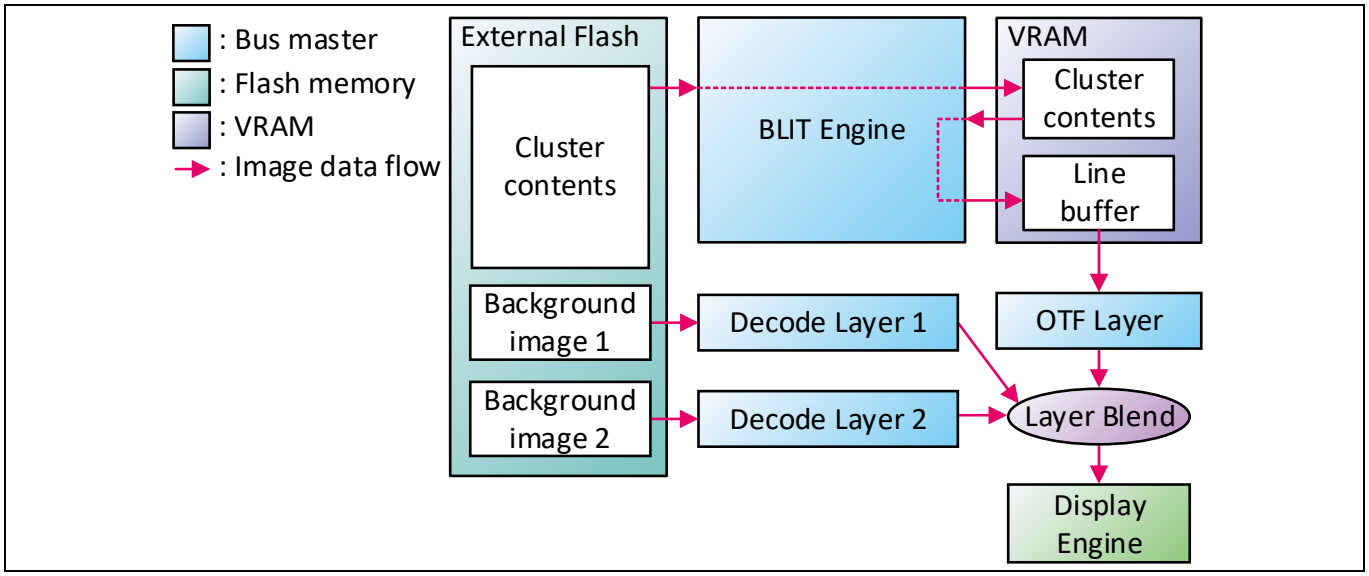

Figure 10 Data flow during scene switching

3.2 Shifting the background window

Background images are typically saved in the flash memory in a compressed state. The decode layer fetches, decompresses, and transfers them to the display engine on-the-fly. This is because typically, the background image size is large; thus, it is not suitable to copy it to the VRAM. In addition, because the background image does not change dynamically, saving the background image to the VRAM is not required. Note that this application does not use the blit engine for outputting the background image. Thus, there are no blit mode.

A way to shift the background image is changing the coordinates of the window itself. You cannot change the coordinate of the image in the window.

The method to realize the background shift affects the number of layers used and flash access. Depending on the method, you can realize the shifting background with only one layer or you can reduce flash access. This section introduces a basic method to realize the background shift. A method only with one layer is introduced in **Appendix: Background shift with one layer**. A method with reduced flash access is introduced in **Appendix: Background shift with less flash access**.

3.2.1 Basic procedure of background shifting

This section describes a basic method to the shift background window. This procedure uses two layers. These layers must have decompressing functionality.

You need to prepare two compressed background images into flash memory named “Background 1” and “Background 2” shown in **Figure 11**.

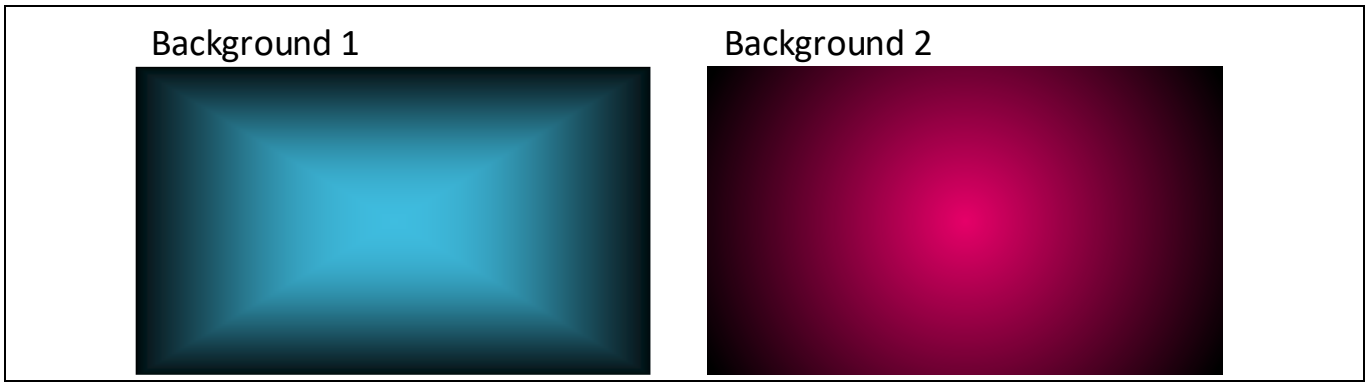

Figure 11 Prepared background image for basic shift

The procedure in which the Background 1 slides out and the Background 2 slides in the display is shown in **Figure 12**. Switching the other way around can be done in the same way.

1. Display shows Background 1 on Window 0.
2. Start scene switching. This creates Window 1 with Background 2 at the right, out of the display.
3. Move the windows to the left by changing the coordinates of Window 0 and Window 1.
4. The switching is complete. The display shows Background 2 on Window 1. Window 0 is completely out of the display.
5. Delete Window 0.

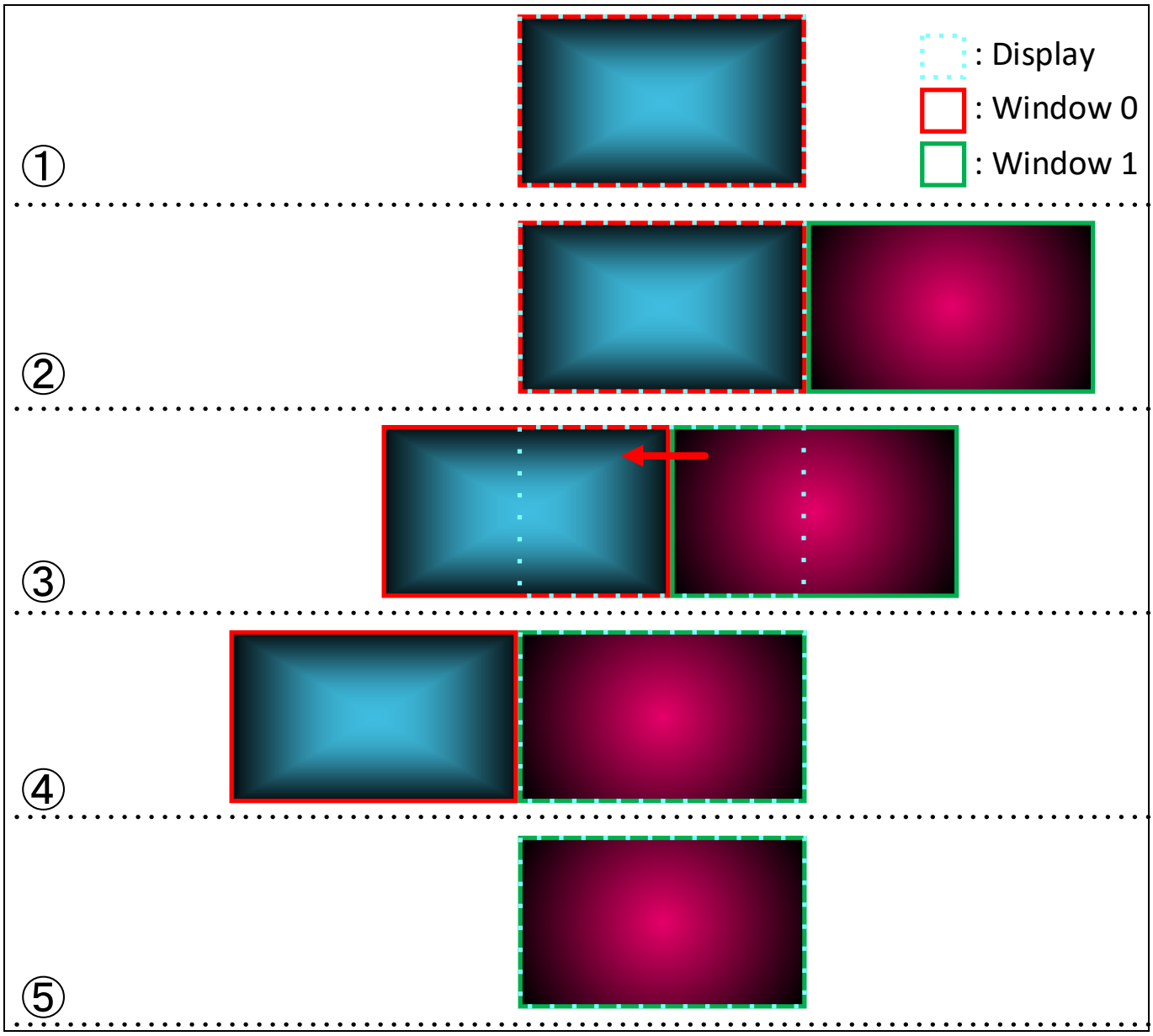

Figure 12 Basic shift from Background 1 to Background 2

In this procedure, two decoding layers and flash accessing for two background images are required. To reduce them, see **Appendix: Background shift with one layer** and **Appendix: Background shift with less flash access**.

3.3 Shifting the cluster contents

This section describes the implementation of shifting cluster contents. An OTF window is used to show the cluster contents. In the contents shift procedure, the window itself does not move unlike in background window shifting. Instead, the coordinates of the contents in the window change. See **Figure 13**. Images numbered 1 - 4 are supposed to be shown in scene 1 and images numbered 5 - 8 are supposed to be shown in scene 2. In the way scene switch, images 1 - 4 will shift out to the left of the display and images 5 ~8 will shift in from the right of the display. Before the switching starts, the data of images 1 - 4 located on the VRAM and being shown on the display while data of images 5 - 8 are not located on the VRAM. The coordinates and address on the flash memory of images 5 - 8 are managed by the application.

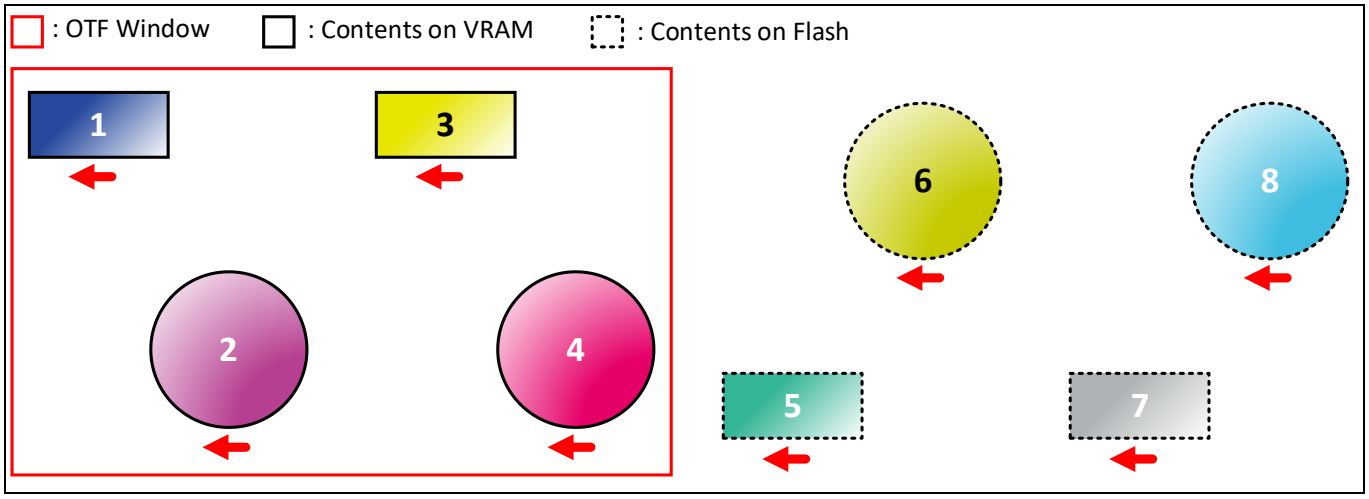

Figure 13 Initiating contents shift

As images shift, image 5 will get inside the window. After the CPU detects the entrance of image 5 in the previous frame, the CPU reserves the VRAM area where the data of image 5 will be stored into and the CPU makes instructions which let the blit engine copy the data from the flash memory to the VRAM area, and transfer the image to the line buffer until image 5 appears (step 1 of **Figure 14**).

As the shifting continues, image 1 will get out of the window. After the CPU detects that image 1 has got out of the window completely, the CPU stops showing image 1 and frees the area on the VRAM where the data of image 1 was located (step 2 of **Figure 14**).

Like this, during the shifting, the VRAM area for the image being moved into the window is reserved and the VRAM area for the image that leaves the window are freed. It reduces the VRAM usage during the shift compared to reserving VRAM areas for both scene 1 and scene 2 (graph on the right of **Figure 14**).

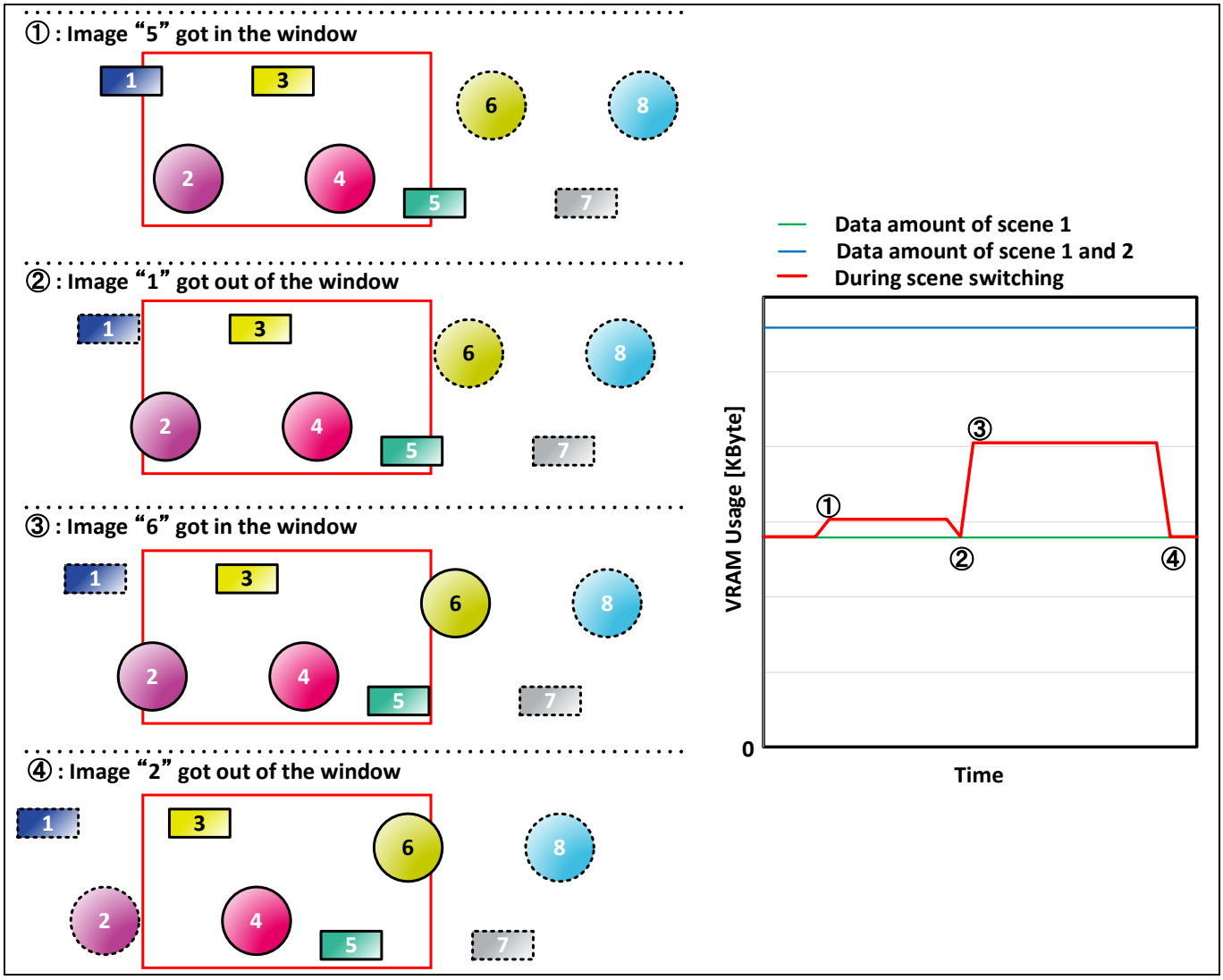

Figure 14 Data usage of contents shift

3.3.1 Data flow of contents shifting

This section describes details of procedure when an image is getting into the window. See **Figure 15**. The blit engine copies the image from the flash memory into VRAM area and transfers it to the line buffer. `extSurf` is a surface of the compressed image stored in the flash memory. `interSurf` is a surface of the decompressed image located on the VRAM area. `strSurf` is a surface which represents the line buffer of the OTF window. `TASK_MEM` represents a task in which the blit engine decompresses `extSurf` and copies the image into `interSurf`. This task would be done once when the image is getting inside the window. `TASK_WIN` represents an OTF task which transfers the image data from `interSurf` to `strSurf` as synchronizing with video I/O. `TASK_MEM` and `TASK_WIN` are non-OTF task and OTF task respectively; thus you must assign different tasks to them. See **3.3.5** to learn how `TASK_MEM` and `TASK_WIN` synchronize with each other.

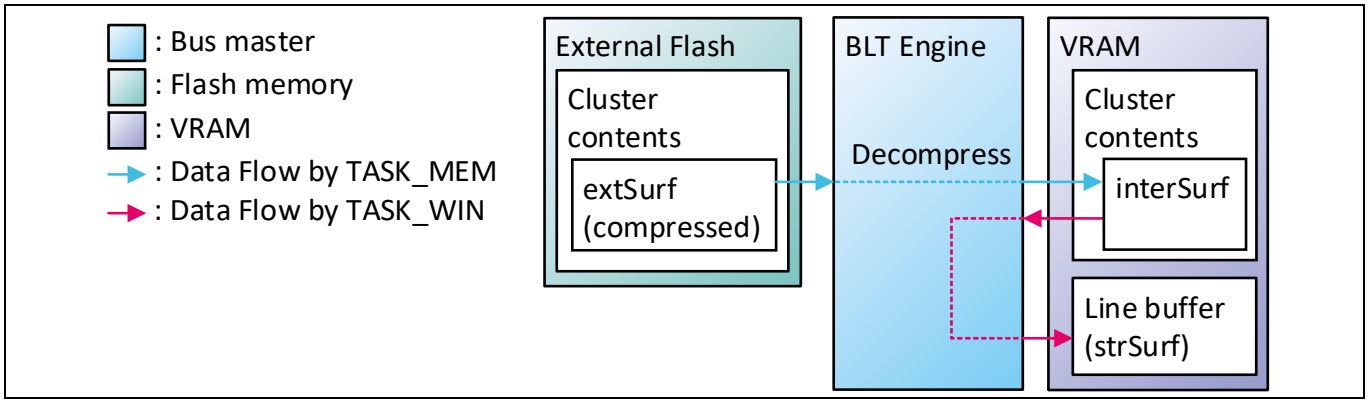

Figure 15 Data flow of preparing new contents image

3.3.2 Initializing the OTF window

This section introduces example codes for creating the OTF window. Although there are some undefined variables in the code examples, these examples assume that these variables are defined beforehand. See the TRAVEO™ T2G graphics driver manual [5] to learn about the usage of graphics APIs.

The function `void* VideoAlloc(size, alignment)` in the following code snippets allocates the memory area in the VRAM. The first input is the size in byte to allocate. The second input is the alignment in byte. The return value is the top address of the allocated area.

Creating `strSurf` is shown in **Code Listing 8**.

Code Listing 8 Creating store surface for contents shift

```
/* Allocate VRAM area for the line buffer */
pBuffer = VideoAlloc(lineBufferSize, 32, NULL);

/* Check if the previous allocation succeeded */
CY_ASSERT(pBuffer != NULL);

/* Reset the store surface object which represents the line buffer */
CyGfx_SmResetSurfaceObject(&strSurf);

/* Assign VRAM area to the surface object */
CyGfx_SmAssignBuffer(&strSurf, w, lineCnt, format, pBuffer, 0);

/* Assign the store surface to the window */
CyGfx_DispWinSetSurface(winHandle, CYGFX_DISP_BUFF_TARGET_COLOR_BUFF,&strSurf);
```

Creating and assigning `TASK_WIN` are shown in **Code Listing 9**. As shown in the code list, `beOtfCtx` describes the `TASK_WIN` task in this document.

Code Listing 9 Creating a blit task for the OTF window

```
/* Allocate VRAM area for the command buffer of the TASK_WIN*/
pOtfCmdSeqBuffer = VideoAlloc(otfInstBufSize, 32, NULL);

/* Check if the previous allocation succeeded */
CY_ASSERT(pOtfCmdSeqBuffer != NULL);

/* Assign the command buffer to TASK_WIN */
beOtfTaskType = CYGFX_BE_TASK_WIN_PRIO_0;
```

Code Listing 9 Creating a blit task for the OTF window

```
CyGfx_BeSetTaskInstructionBuffer(beOtfTaskType, pOtfCmdSeqBuffer,
   otfInstBufSize);

/* Reset the blit context object for TASK_WIN */
CyGfx_BeResetContext(&beOtfCtx);

/* Set the task type to the context object */
CyGfx_BeSetAttribute(&beOtfCtx, CYGFX_BE_CTX_ATTR_TASK, beOtfTaskType);

/* We use a coordinate system starting in the upper left corner (like display).
   */
CyGfx_BeSetAttribute(&beOtfCtx, CYGFX_BE_CTX_ATTR_ZERO_POINT,
   CYGFX_BE_ATTR_ZERO_TOP_LEFT);
```

Creating and assigning `TASK_MEM` are shown in **Code Listing 10**. As shown in the code list, `beCopyCtx` describes the `TASK_MEM` task in this document.

Code Listing 10 Creating a blit task for copying from the flash to the VRAM

```
/* Allocate VRAM area for the command buffer of TASK_MEM */
pCopyCmdBuffer = VideoAlloc(copyInstBufSize, 32, NULL);

/* Check if the previous allocation succeeded */
CY_ASSERT(pCopyCmdBuffer != NULL);

/* Assign the command buffer to TASK_MEM */
beCopyTaskType = CYGFX_BE_TASK_MEM_PRIO_0;
CyGfx_BeSetTaskInstructionBuffer(beCopyTaskType, pCopyCmdBuffer,
   copyInstBufSize);
CyGfx_BeSetTaskCopses(pWinParam->copyTaskType, 64);

/* The context must be the reset blit context object for TASK_MEM */
CyGfx_BeResetContext(&beCopyCtx);

/* Set the task type to the context object */
CyGfx_BeSetAttribute(&beCopyCtx,CYGFX_BE_CTX_ATTR_TASK,beCopyTaskType);

/* We use a coordinate system starting in the upper left corner (like display).
   */
CyGfx_BeSetAttribute(&beCopyCtx, CYGFX_BE_CTX_ATTR_ZERO_POINT,
   CYGFX_BE_ATTR_ZERO_TOP_LEFT);
```

3.3.3 Making a surface out of the window

This section introduces code examples that prepare the contents out of a window. Creating `extSurf` is shown in **Code Listing 11**.

Code Listing 11 Creating an external compressed image surface

```
/* Reset the surface object which represents an image on the external flash
   memory */
CyGfx_SmResetSurfaceObject(&extSurf);

/* Assign the external image area to the surface object. 'pExtBuffer' points to
   the address of the external image. */
CyGfx_SmAssignBuffer(&extSurf, w, h, format, pExtBuffer, 0);

/* Set the commpression format of the external image */
```

Code Listing 11 Creating an external compressed image surface

```
CyGfx_SmSetAttribute(&extSurf, CYGFX_SM_ATTR_COMPRESSION_FORMAT, compType);
```

3.3.4 Preparing the surface before moving into the window

This section introduces code examples which demonstrate how an image is prepared before being moved into the window from outside.

Creating `interSurf` is shown in **Code Listing 12**.

Code Listing 12 Creating internal VRAM image surface

```
/* Allocate VRAM area for the internal surface */
pBuffer = VideoAlloc(size, 32, NULL);

/* Check if the previous allocation succeeded */
CY_ASSERT(pBuffer != NULL);

/* Reset the surface object which represents an image on the VRAM */
CyGfx_SmResetSurfaceObject(&interSurf);

/* Assign the VRAM area to the surface object */
CyGfx_SmAssignBuffer(&interSurf, w, h, format, pBuffer, 0);

/* Set the compression format of the surface as non-compression */
CyGfx_SmSetAttribute(&interSurf, CYGFX_SM_ATTR_COMPRESSION_FORMAT,
   CYGFX_SM_COMP_NON));
```

Code Listing 13 is a code example which lets the blit engine transfer the image from `extSurf` to `interSurf` and from `interSurf` to `strSurf`.

Code Listing 13 Preparing a blit task for copying and synchronizing the OTF window

```
/* Set the external image surface as 'source' of TASK_MEM */
CyGfx_BeBindSurface(&beCopyCtx, CYGFX_BE_TARGET_SRC, &extSurf);

/* set the internal VRAM image surface as 'store' of TASK_MEM */
CyGfx_BeBindSurface(&beCopyCtx, CYGFX_BE_TARGET_STORE, &interSurf);

/* Start the blit operation of TASK_MEM */
/* Executing this function does not guarantee that the operation has been
   finished. Use 'CyGfx_BeGetSync' to make sure of the completion. */
CyGfx_BeBlt(&beCopyCtx, 0.0, 0.0);

/* Set the internal VRAM image surface as the 'source' of TASK_WIN */
CyGfx_BeBindSurface(&beOtfCtx, CYGFX_BE_TARGET_SRC, &interSurf);

/* set the OTF line buffer surface as 'dest' and 'store' of TASK_WIN */
CyGfx_BeBindSurface(&beOtfCtx, (CYGFX_BE_TARGET_STORE|CYGFX_BE_TARGET_DST),
   &strSurf);

/* Schedule the blit operation of TASK_WIN */
/* Executing this function does not trigger the OTF blit operation until
   'CyGfx_DispWinCommit' is executed. */
CyGfx_BeBlt(&beOtfCtx, posX, posY);
```

3.3.5 Synchronizing TASK_MEM and TASK_WIN

`CyGfx_Blt (& beCopyCtx, 0.0, 0.0)` in **Code Listing 13** instructs the blit engine to copy from the external flash to the VRAM, and `CyGfx_BeBlt (& beOtfCtx, posX, posY)` gives the blit engine instructions to transfer the copied content to the line buffer on the VRAM.

The latter `beOtfCtx` must be done after the former `beCopyCtx` is finished. The OTF mode task (`beOtfCtx`) would work when a window committing function (`CyGfx_DispWinCommit`) was called. Therefore, before calling `CyGfx_DispWinCommit`, you need to ensure that the `beCopyCtx` task was completed.

Code Listing 14 and **Code Listing 15** allow this application to know that the blit engine task has finished.

The `CyGfx_BeGetSync` function in **Code Listing 14** assigns the address to an area where the GFX driver manages the status of the task (called a “sync object”), to a variable `copySync`. Also, in LBO mode, the previous instruction made by the `CyGfx_Blt` function may be still remaining without executed. The `CyGfx_BeGetSync` function triggers this remaining instruction to be executed.

Code Listing 14 Getting a sync object (triggering the remaining copy task in LBO mode)

```
/* Get the sync object */
/*Trigger copying task which is remaining in LBO mode */
CyGfx_BeGetSync(&beCopyCtx, &copySync);
```

Code Listing 15 checks the value of the sync object to verify that the blit engine has completed the copy task. If the copy operation was complete, the `CyGfx_SyncWaitSync` function returns `GYGFX_OK`.

Code Listing 15 Ensuring that the copy operation from external flash to VRAM is already finished

```
/* Check if the copying operation has finished */
if(CyGfx_SyncWaitSync(&copySync, 0) == CYGFX_OK)
{
    /* Copying has been finished. */
}
else
{
    /* Copying hasn't been finished yet. */
}
```

4 Dynamic distortion calibration

This chapter describes the dynamic distortion calibration application using the warping function which works on Display 1. In this chapter, the following terms are used: “warping map”, “reference coordinate”, “warping layer”, “distortion grid nodes”, and “normal grid nodes”. See **Appendix: Image warping**, **Appendix: Warping layer and coordinate buffer**, and **Appendix: Distortion calibration procedure** to learn more about these terms and the basic function of warping. The application described in this section calibrate images shown on distorted display so that users can see the image normally. Thus, the aim of this application is to calibrate distorted images, not to distort image. See **Head-up display (Display 1)**.

4.1 Data flow of dynamic distortion calibration

This sub-section describes data flow during normal mode, calibration mode and when mode transferring. See **Head-up display (Display 1)**.

4.1.1 Normal mode

Data flow during normal mode is shown in **Figure 16**. The coordinate buffer is located on the flash memory; the warping layer fetches it. The frame buffer for the contents is located in the VRAM as a line buffer because the warping layer is in OTF window mode.

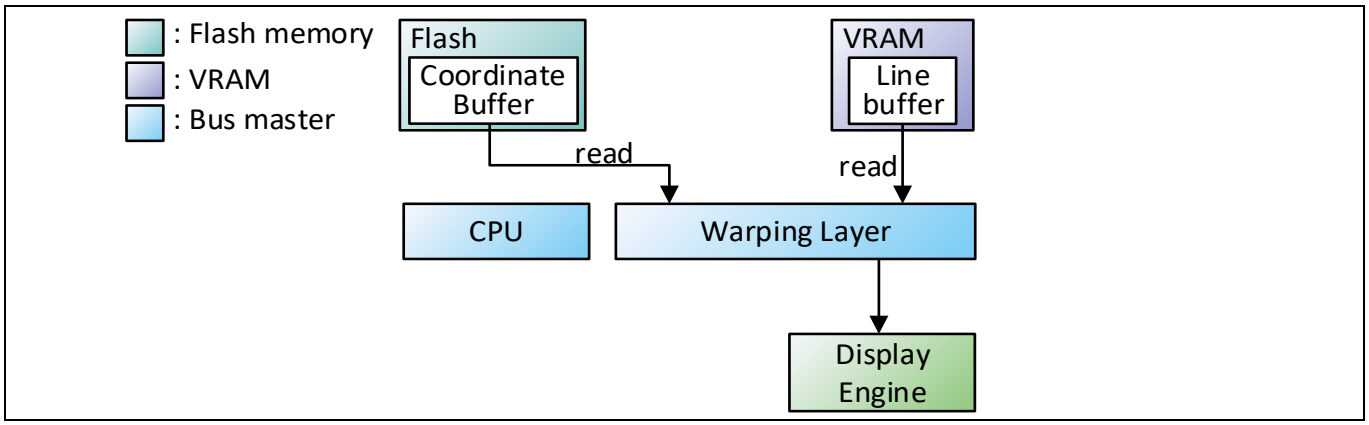

Figure 16 Dataflow of warping display in normal mode

4.1.2 Transfer to calibration mode

Data flow during transferring to calibration mode is shown in **Figure 17**. In this mode, the coordinate buffer is modified dynamically; therefore, it must be located on the VRAM. The CPU copies the coordinate buffer from the flash memory to the VRAM when transferring to calibration mode from normal mode. After transferring, the warping layer starts fetching the coordinate buffer from the VRAM.

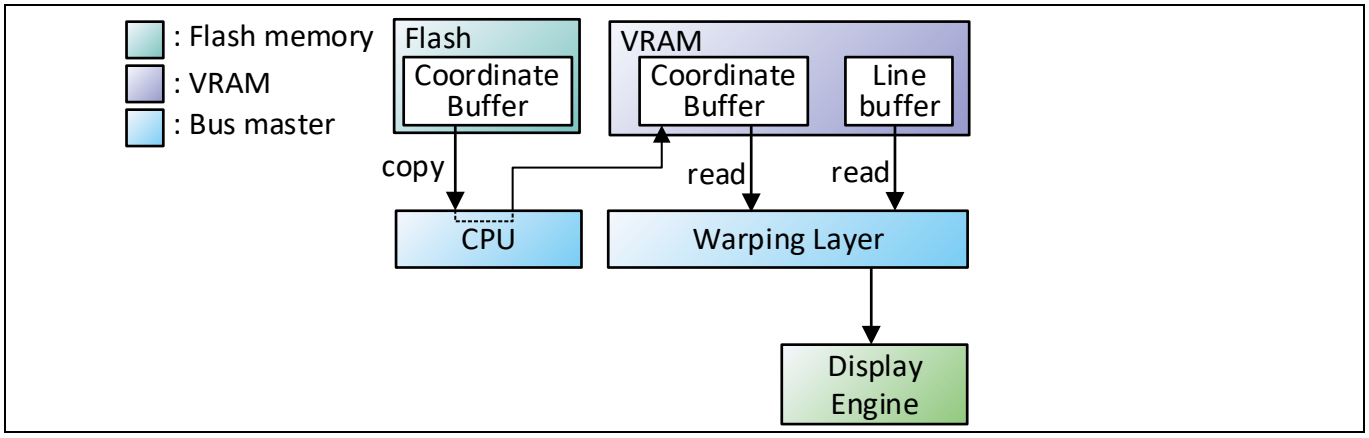

Figure 17 Data flow of warping display from normal mode to calibration mode

4.1.3 Double buffering

Note that if the CPU modifies the coordinate buffer when the warping layer is reading the coordinate buffer, the output image would be disturbed (**Figure 18**).

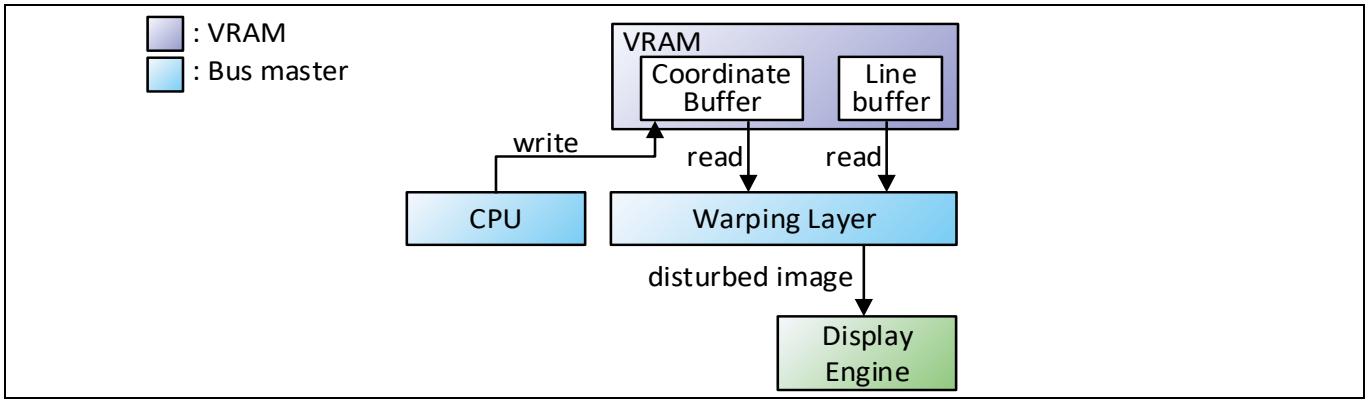

Figure 18 Data flow of warping display in calibration mode with a single buffer

To avoid this, you should have double buffered area for the coordinate buffer. While the CPU is modifying one coordinate buffer, the warping layer would be reading the other coordinate buffer. After the CPU finishes the modification, switch the written coordinate buffer and the read coordinate buffer (**Figure 19**).

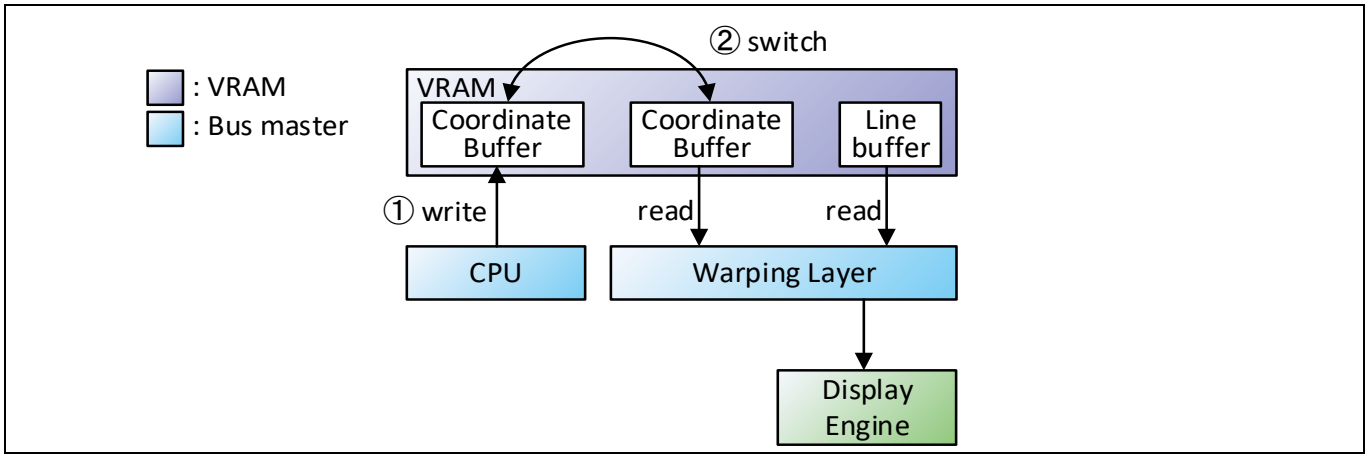

Figure 19 Data flow of warping display in calibration mode with a double buffer

Therefore, you must reserve twice amount of the coordinate buffer on the VRAM to output non-disturbed image while dynamic warping.

4.1.4 Transform to normal mode

Data flow when transferring to normal mode from calibration mode is shown in **Figure 20**. After the calibration is done, the CPU programs the data of the VRAM coordinate buffer into the flash memory and frees the reserved VRAM area for the coordinate buffer.

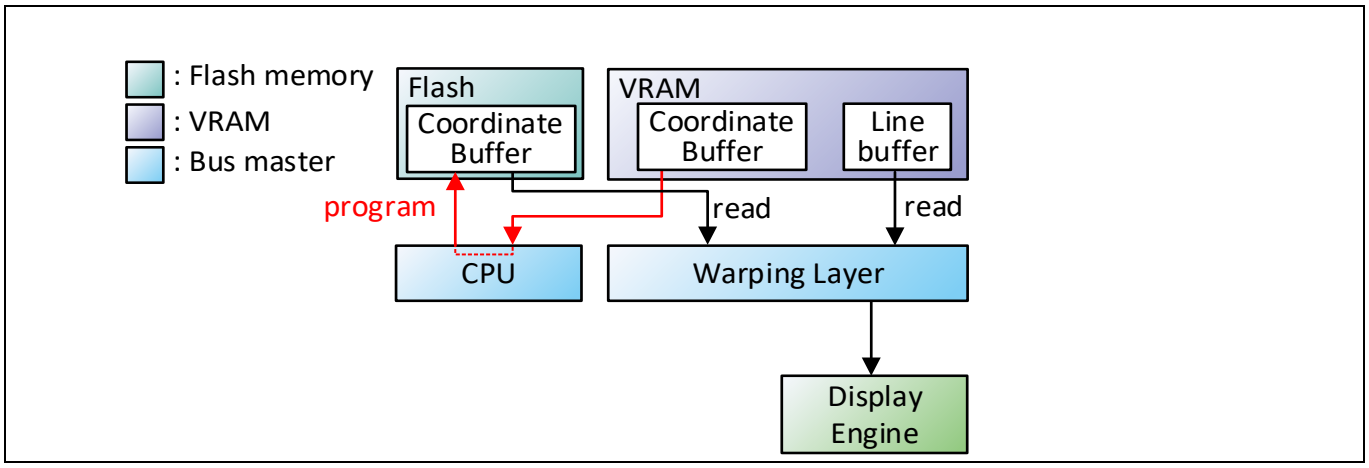

Figure 20 Data flow of warping display from calibration mode to normal mode

4.2 Warping for distortion calibration

This sub-section describes a way to generate a coordinate buffer which calibrates distortion. Warping layer mode is delta vector increments mode. See **Appendix: Warping layer** to learn about warping layer mode. The coordinate buffer length is 8 bpp. See **Appendix: Distortion calibration procedure**.

4.2.1 Giving distortion grid nodes

In the application described in **Head-up display (Display 1)**, 11 x 7 distortion grid nodes are given; it is assumed that you can select and move control points one by one using buttons (**Figure 3**). See **Appendix: Distortion calibration procedure** to learn about distortion grid nodes and normal grid nodes.

Attention: ***Note that control points do not represent distortion grid nodes. Distortion grid nodes define the distortion of the display surface. This example "offsets" the distortion; that is, if a distortion grid was located in the upper right from the normal grid, the calibrated image would be the original image distorted to the lower left (Figure 21). Therefore, the image is distorted to opposite direction of distortion grid nodes. If the application displayed the control points in the same position as the distortion grid nodes, the image would be distorted in the opposite direction of the control points when the control points are moved. This is counterintuitive. For this reason, the application displays the control points at the opposite position of the distortion grid nodes.***

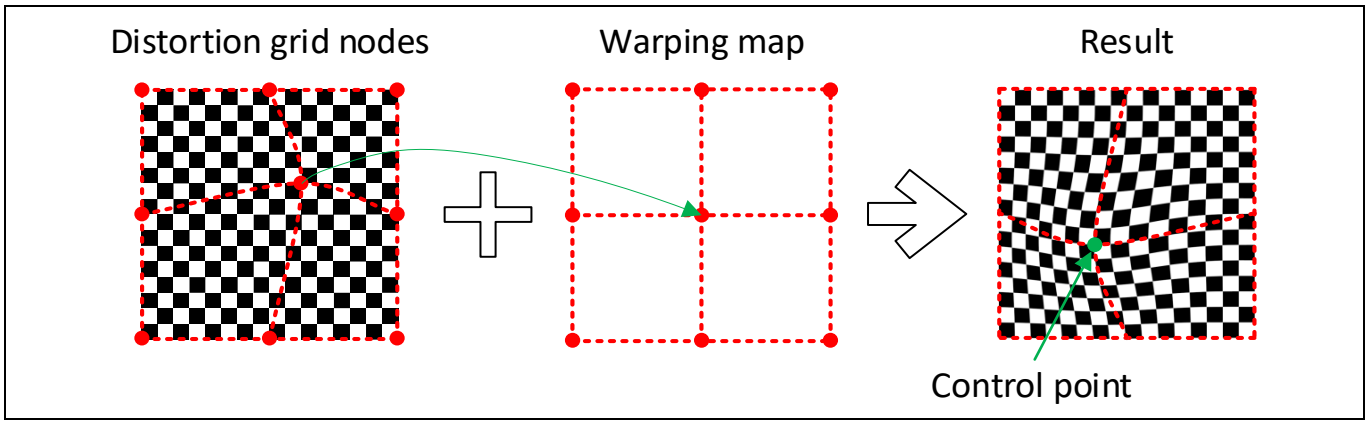

Figure 21 Distortion grid nodes and Control points

4.2.2 Spline interpolation among distortion nodes and making a warping map

While the application has 11 x 7 distortion grid nodes, 4 x 4 distortion grid nodes are introduced here for simplicity of description. Assume that the distortion of the display was represented by a 4 x 4 distortion grid as shown in **Figure 22**. Now, consider generating a warping map which offsets this distortion. w*i* (*i* = 0~15) represents coordinate of normal grid nodes on the warping map. p*i* (*i* = 0 ~ 15) represents coordinate of distortion grid nodes. x element and y element of w*i* and p*i* are described as w*i* = [w*i*.x, w*i*.y], p*i* = [p*i*.x, p*i*.y]. And reference coordinate value on (x, y) of the warping map is described as WM(x, y). Think of WM as a two-dimensional array of reference coordinates, the size of which is IMAGE_WIDTH x IMAGE_HIGHT. You know only the discrete 16 elements of WM(x, y) as shown in the following formula in *i* = 0 ~ 15.

$$\mathrm{WM}(\mathrm{w}i.\mathrm{x}, \mathrm{w}i.\mathrm{y}) = [\mathrm{p}i.\mathrm{x}, \mathrm{p}i.\mathrm{y}] \quad (4)$$

In this section, this WM(x, y) is completed using spline interpolation.

See **Appendix: Image warping** to learn about warping map and reference coordinate.

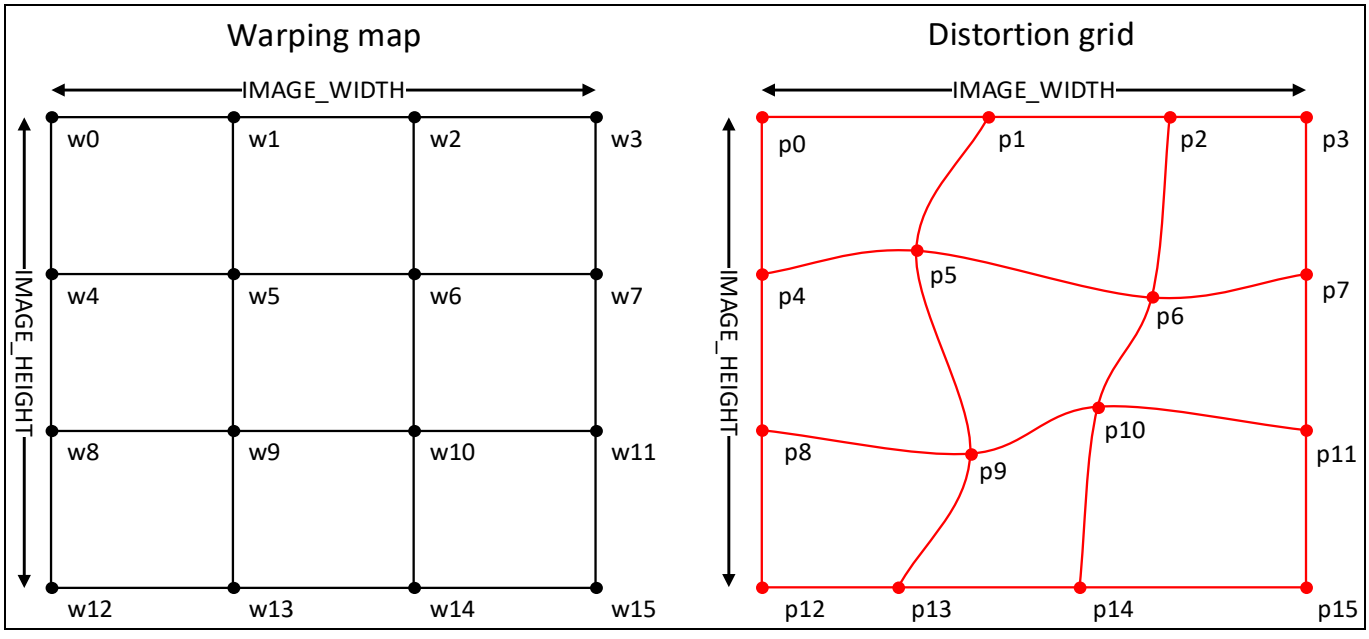

Figure 22 Example of warping map and 4 x 4 distortion grid

At first, you interpolate the warping map vertical way for all columns of the grid. Spline interpolation of column index 1 is shown in **Figure 23**.

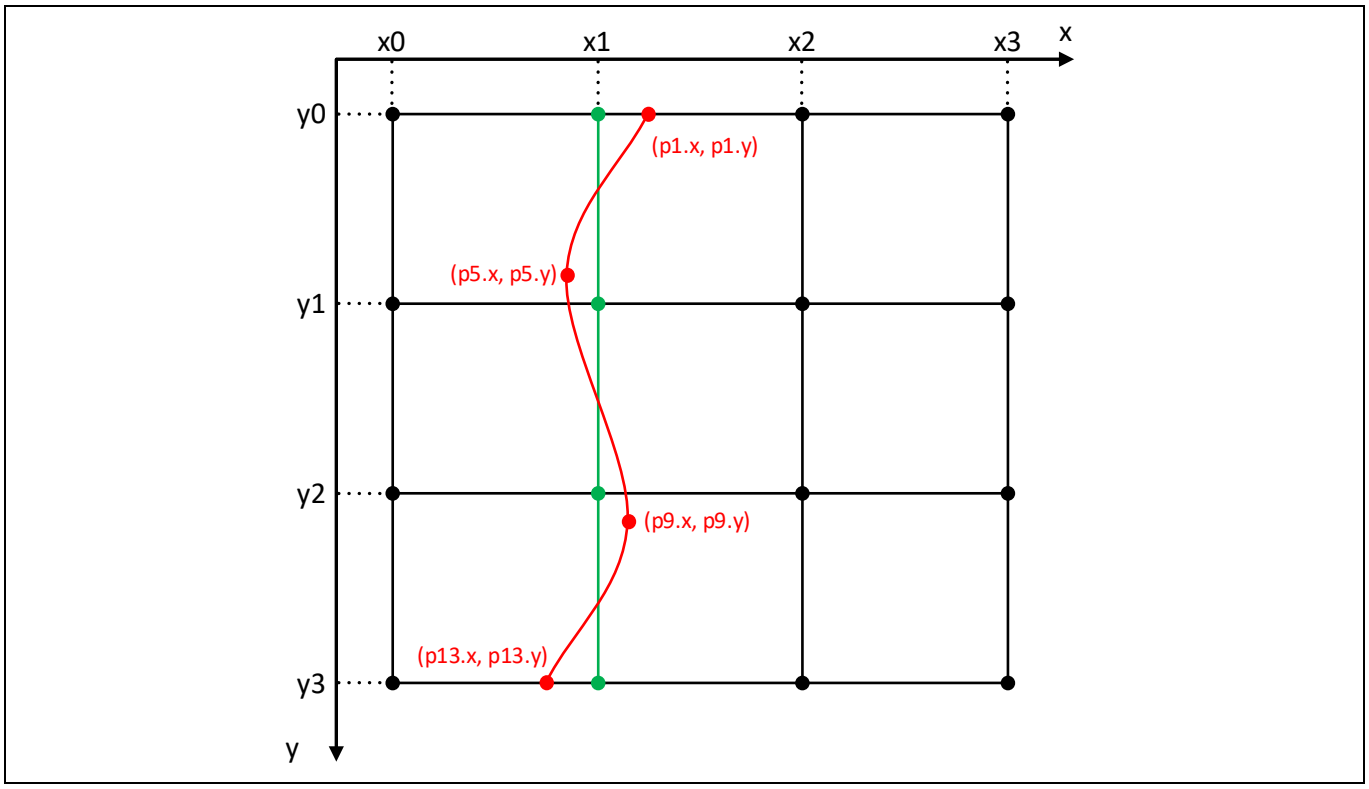

Figure 23 Mapping from normal grid vertical line to distortion grid vertical line

Suppose the x coordinate of each column of the normal grid nodes are x0, x1, x2, and x3 and the y coordinate of each line of the normal grid nodes is y0, y1, y2, and y3 (e.g. w0 = [x0, y0], w5 = [x1, y1]). The coordinates value of column index 1 of the distortion grid nodes are [p1.x, p1.y], [p5.x, p5.y], [p9.x, p9.y], and [p13.x, p13.y] respectively. According to formula (4), the warping map value on coordinates of normal grid nodes column 1 are expressed as formulas as follows:

- WM(x1, y0) = [p1.x, p1.y]
- WM(x1, y1) = [p5.x, p5.y]
- WM(x1, y2) = [p9.x, p9.y]
- WM(x1, y3) = [p13.x, p13.y]

Assume that the x element and the y element of these are independent and do spline interpolation among the discrete value of y (**Figure 24**).

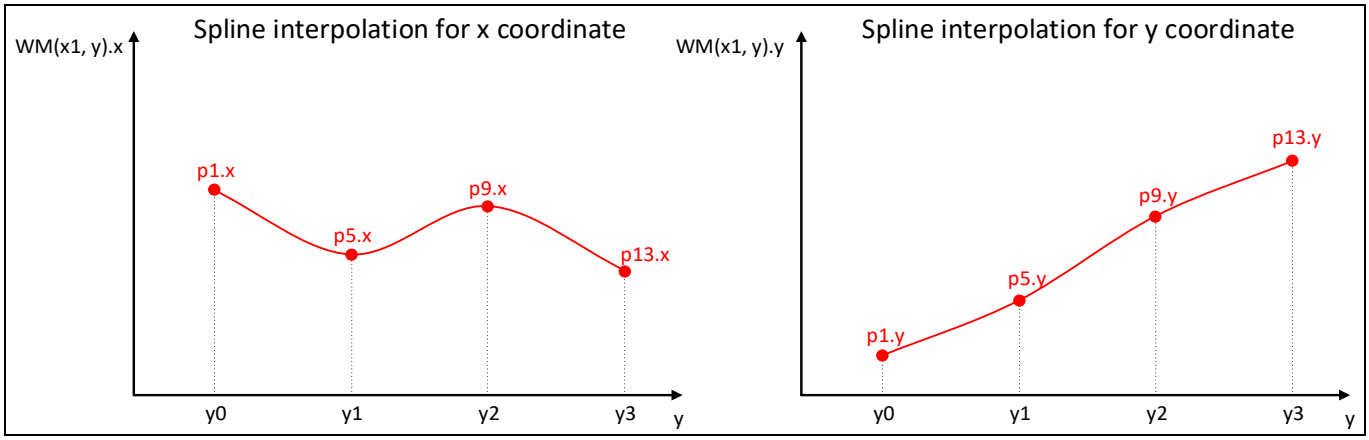

Figure 24 Vertical splining reference coordinates x and y

Do this spline interpolation for all columns of the grid and keep the results. Now you could know the warping map for all normal grid columns, for any y coordinate, WM(x0, y), WM(x1, y), WM(x2, y), WM(x3, y).

Next, do the horizontal spline interpolation (see **Figure 25**). The warping map value of the intersection points of line y = y' (y' = integer between y0 to y3) and columns of the normal grid is calculated as WM(x0, y'), WM(x1, y'), WM(x2, y'), and WM(x3, y').

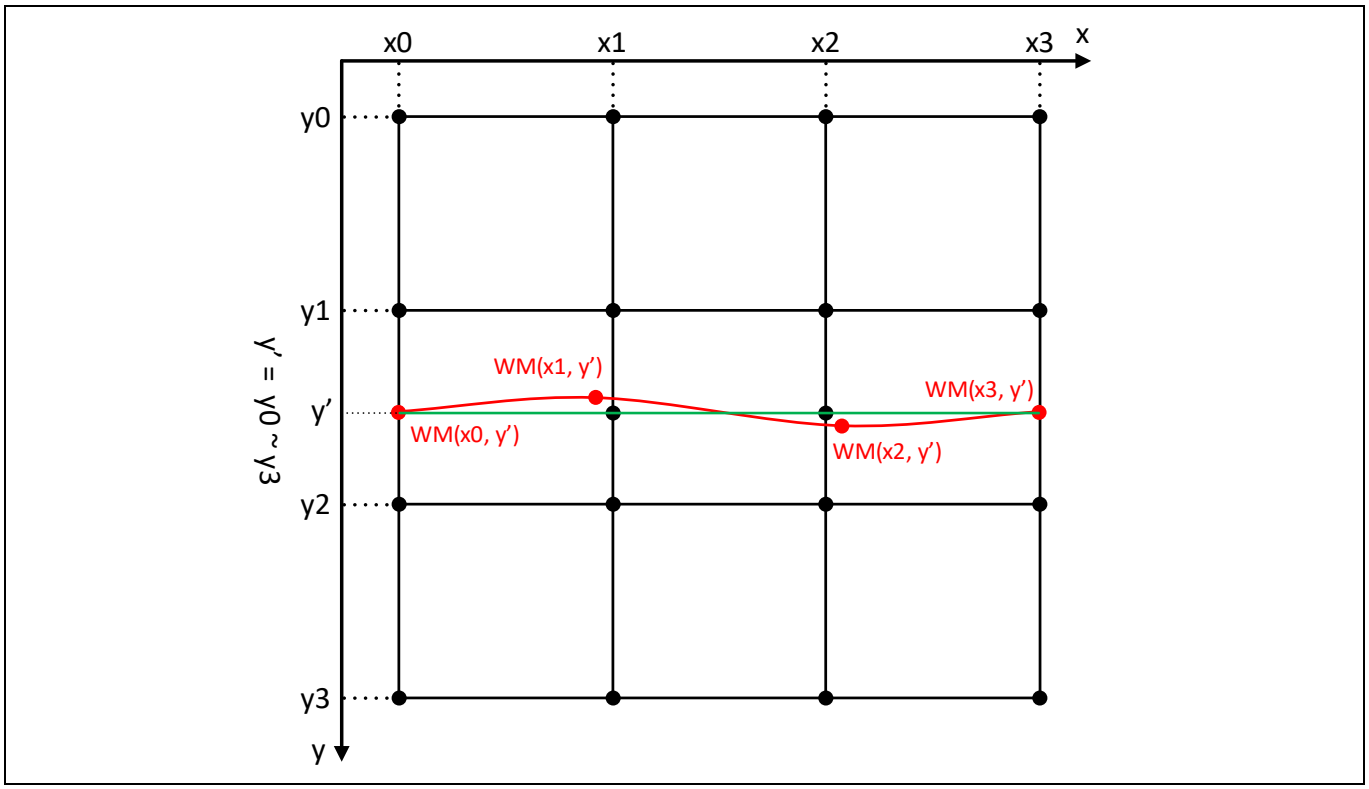

Figure 25 Mapping from normal grid horizontal line to distortion grid horizontal line

By spline interpolation of these points shown in **Figure 26**, you can determine the warping map value with any x and y = y' (WM(x, y')). By iterating this for every y integer value from y0 to y3, you can arrive at the warping map value WM(x, y) at any integer x and y.

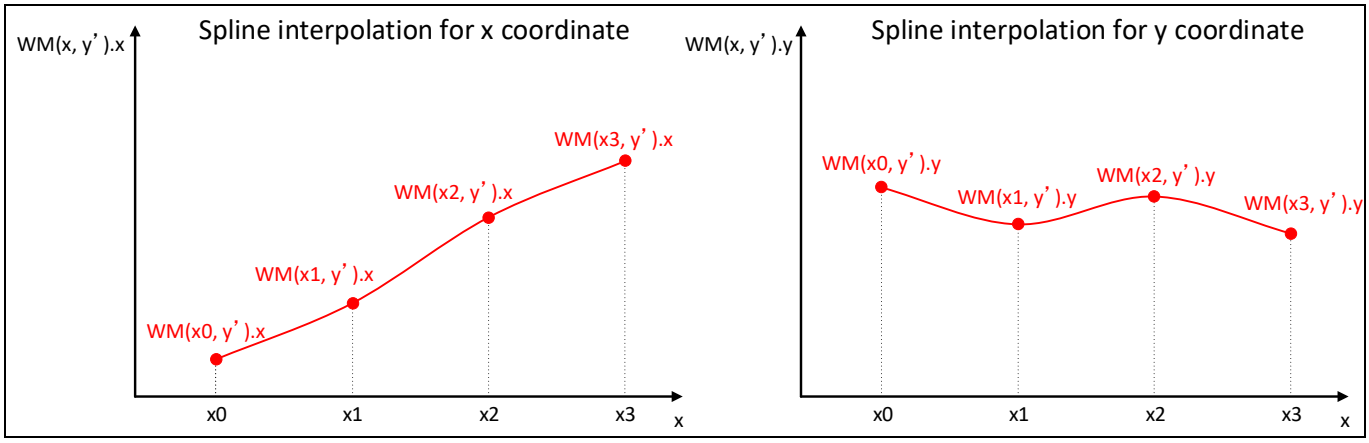

Figure 26 Horizontal splining reference coordinate x and y

4.2.3 Packing into the coordinate buffer

Although the warping layer can process the warping map calculated in the previous section directly as a coordinate buffer, in such cases, the coordinate buffer size would be huge. The reason is that x and y values require more than 16 bits (that is, 32 bpp) because the value of the warping map represents the coordinates of the window, and typically, the window width and height are longer than 255 (maximum of 8bits) pixels.

Therefore, in this section, you reduce the size of the coordinate buffer by packing the warping map into a coordinate buffer that is in delta vector increment mode. In the delta vector increment, the warping map value can be expressed with 8-bpp data normally (4 bits for x and y). See **Delta vector increments mode and the coordinate buffer** in **Appendix: Warping layer**.

The procedure described in this section will make the warping map line-by-line and pack it into the coordinate buffer to reduce the VRAM usage for generating the coordinate buffer. This also increases the performance of the program. This section also describes how to handle errors if it was impossible to represent the exact reference coordinate with 4-bit data.

4.2.3.1 Setting for registers which indicate the initial condition

As described in **Delta vector increments mode and the coordinate buffer**, in delta vector increments mode, the offset coordinate, offset vy, and offset vx must be defined with the register value. In this example, offset vy, and offset vx are defined with a constant value as offset vy = (0.0, 1.0), offset vx = (1.0, 0.0). Also, the first coordinate buffer value d0 is defined with a constant value of (0.0, 0.0). If the value of the warping map (0,0) is p0 (x0, y0), the offset coordinate = (x0, y0 – 1.0) – see **Figure 27**.

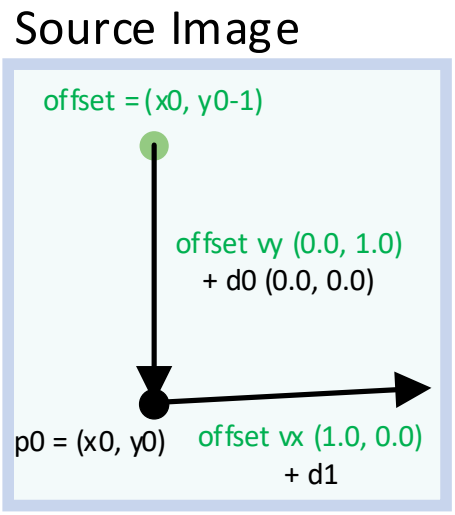

Figure 27 Predefined warping register values

As you can see in **Figure 27**, “p0” is already expressed by offset, offset vy, and d0.

4.2.3.2 Making the coordinate buffer

The basic concept of making a coordinate buffer is described with a flowchart (**Figure 42**) in **Delta vector increments mode and the coordinate buffer**. However, here the whole warping map value is not saved; instead, the warping map is calculated line-by-line and packed into the coordinate buffer to reduce the VRAM or System RAM usage.

See **Code Listing 1** for definitions of variable type used in code snippets in this section.

`fixP_12_4_t` is a 16-bit fixed point type, lower 4 bits of which indicate fractional parts. This is defined because lower 4 bits of the coordinate buffer indicate fractional parts.

`stc_coord_fixP_12_4_t` is a structure which contains x and y elements of type `fixP_12_4_t`.

`un_coord_fixP_12_4_t` is a union of `stc_fixP_12_4_t` and `uint32_t`. Processing x and y elements as `uint32_t` allows the CPU to do substitution and several operations with one instruction.

Code Listing 16 Definition for coordinate / vector type

```
/* Define signed 12.4 fixed-point type */
typedef int16_t fixP_12_4_t;

/* Define a structure for the variable which has elements (x, y)*/
typedef struct
{
    fixP_12_4_t x;
    fixP_12_4_t y;
} stc_coord_fixP_12_4_t;

/* Define a union for the variable which has elements (x, y) so that the CPU can
   operate x and y elements at the same time */
typedef union
{
    stc_coord_fixP_12_4_t coord;
    uint32_t              u32;
```

Code Listing 16 Definition for coordinate / vector type

```
    } un_coord_fixP_12_4_t;
```

See **Figure 28**. Assume the value p9 (WM[yIdx =1][xIdx = 4]) was input into a coordinate buffer packing routine. The routine must have saved the value listed here (shown in red letter in **Figure 28**).

- Previous position (p8)
- Previous vx (v8)
- Previous line offset (p5)
- Previous line offset vy (v5)
- Previous line offset vx (v6)

The values previous position and previous vx are required to calculate “d9”. The values of previous line offset, previous line offset vy, and previous line offset vx are not required to calculate d9, but are required to calculate d10 and d11 on the next line.

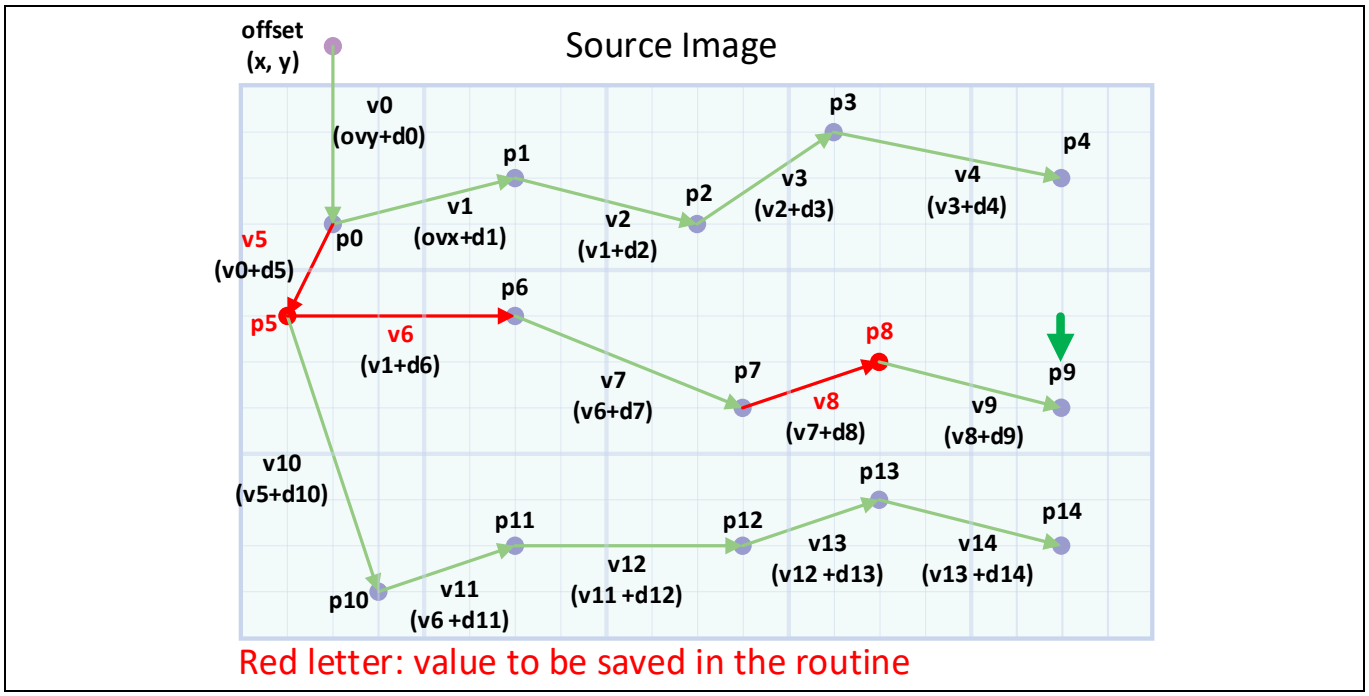

Figure 28 Packing reference coordinate into delta vector increments mode coordinate buffer

The following code snippet makes the coordinate buffer from the reference coordinate mentioned above.Definitions of five static variables which save the necessary value described in **Figure 28** are shown in **Code Listing 17**. Each variable type is `un_coord_fixP_12_4_t`.

Variable	Description
`gPreviousPosition`	Previous position
`gPreviousVx`	Previous vx
`gPreviousLineOffset`	Previous line offset
`gPreviousLineOffsetVy`	Previous line offset vy
`gPreviousLineOffsetVx`	Previous line offset vx

Code Listing 17 Definition for static variables

```
/* Previous position */
un_coord_fixP_12_4_t gPrevisouPosition;

/* Previous vx */
un_coord_fixP_12_4_t gPreviousVx;

/* Previous line offset */
un_coord_fixP_12_4_t gPreviousLineOffset;

/* Previous line offset vy */
un_coord_fixP_12_4_t gPreviousLineOffsetVy;

/* Previous line offset vx */
un_coord_fixP_12_4_t gPreviousLineOffsetVx;
```

The main routine which makes the coordinate buffer is shown in **Code Listing 18**. The following functions contain the spline algorithm. These functions are assumed to be defined by the user; the contents are not described in this document.

Function	Description
`UserSpline_GetDistortionGrid`	Returns the coordinate of the first distortion grid node
`UserSpline_AllGridColumns`	Splines the distortion grid node vertically. This function allocates the memory area to save the result of the vertical spline operation.
`UserSpline_Horizontal`	Splines horizontally. This function allocates the memory area to save the result of the horizontal spline operation.
`UserSpline_GetCoord`	Returns the spline result as a reference coordinate
`UserSpline_FreeMemoryHorizontal`	Frees the memory area reserved by `UserSpline_Horizontal`
`UserSpline_FreeMemoryVertical`	Frees the memory area reserved by `UserSpline_AllGridColumns`

Input parameters of `MakeWarpBuffer` are listed as follows:

Input parameters	Description
`void* pCoordBuf`	Pointer which has top address of the coordinate buffer to be made
`CYGFX_WARP_PARAM_S* pWarpStruct`	Pointer to a structure which controls the warping layer register. This document assumes that structure has been defined already and is input to this function. See the graphics driver **[5]** manual to learn about CYGFX_WARP_PARAM_S.

Code Listing 18 MakeWarpBuffer

```
/* Get the first distortion grid coordinate which indicates the reference
   coordinate of (0, 0) */
float targetX0, targetY0;
UserSpline_GetDistortionGrid(&targetX0, &targetY0);

/* Convert the float value into a fixed-point variable */
un_coord_fixP_12_4_t targetPos0;
targetPos0.coord.x = Float2Fix12_4(targetX0);
```

Code Listing 18 MakeWarpBuffer

```
    targetPos0.coord.y = Float2Fix12_4(targetY0);

    /* Initialize static parameters */
    InitStaticParamerter(targetPos0);

    /* Initialize register setting parameters */
    InitWarpStructure(pWarpStruct);

    /* Spline for all grid columns */
    UserSpline_AllGridColumns();

    /* Spline and pack the image line by image line */
    for(uint32_t yIdx = 0ul; yIdx < WIN_GRID_HEIGHT; yIdx+=1)
    {
        /* Spline for the image line  */    UserSpline_Horizontal(yIdx);

        /* Pack the splined data into the coordinate buffer pixel by pixel */
        for(uint32_t xIdx = 0ul; xIdx < WIN_GRID_WIDTH; xIdx+=1)
        {
            /* Get the warping map value created by the spline function */
            float targetX, targetY;
            UserSpline_GetCoord(xIdx, &targetX, &targetY);

            /* Convert the value into a fixed-point variable */
            un_coord_fixP_12_4_t targetPos;
            targetPos.coord.x = Float2Fix12_4(targetX);
            targetPos.coord.y = Float2Fix12_4(targetY);

            /* Pack into the coordinate buffer */
            PackCoordinateBuffer(targetPos, pCoordBuf, xIdx, yIdx);
        }

        /* Free the memory used for image line spline */
        UserSpline_FreeMemoryHorizontal();
    }

    /* Free the memory used for grid columns spline */
    UserSpline_FreeMemoryVertical();
```

The `InitStaticParameter` function, which initializes the static variables defined in **Code Listing 17** is described in **Code Listing 19**. As described in **Setting for registers which indicate the initial condition**, the offset (`gPreviousLineOffset`) indicates an upper pixel to the first reference coordinate; thus, it subtracts 0x10 from the y element (note that the lower 4 bits indicate the fractional parts, meaning sub-pixel). Offset vx (`gPreviousLineOffsetVx`), and Offset vy (`gPreviousLineOffsetVy`) are also initialized as described in **Setting for registers which indicate the initial condition**.

Input parameters of the `InitStaticParameter` function are listed as follows:

Parameter	Description
`un_coord_fixP_12_4_t p0`	The coordinate of the first value of the warping map. The value should be the same as (0,0) of the distortion grid node.

Code Listing 19 InitStaticParamerter

```
/* Initialize some static parameters */
gPreviousLineOffset.coord.x   = p0.coord.x + 0x00;
gPreviousLineOffset.coord.y   = p0.coord.y - 0x10;

gPreviousLineOffsetVx.coord.x = 0x10;
gPreviousLineOffsetVx.coord.y = 0x00;

gPreviousLineOffsetVy.coord.x = 0x00;
gPreviousLineOffsetVy.coord.y = 0x10;
```

The `InitWarpStructure` function which sets the variable of type `CYGFX_WARP_PARAM_S`, which manipulates the register value of the warping layer is shown in **Code Listing 20**. This code contains procedures for converting the fixed-point value, the lower 4 bits of which indicate a fractional part to a value which is defined by the hardware warping layer.

Input parameters of the `InitWarpStructure` function are listed as follows:

- `CYGFX_WARP_PARAM_S* pWarpStruct`: Pointer to a structure which controls the warping layer register. This document assumes that the structure is already defined and input to this function. See the graphics deriver **[5]** manual to learn about `CYGFX_WARP_PARAM_S`.

Code Listing 20 InitWarpStructure

```
/* Set offsetx: signed 16.5 fixed-point shifted by 11
    thus, shift by 12 bits to convert from .4 fixed point */
pWarpStruct->offsetx = gPreviousLineOffset.coord.x << 12ul;

/* Set offsety: signed 16.5 fixed-point shifted by 11
   thus, shift by 12 to convert from .4 fixed point */
pWarpStruct->offsety = gPreviousLineOffset.coord.y << 12ul;

/* Set dxx: signed 3.5 fixed-point shifted by 13
   thus, shift by 14 to convert from .4 fixed point */
pWarpStruct->dxx = gPreviousLineOffsetVx.coord.x << 14ul;

/* Set dxy: signed 3.5 fixed-point shifted by 13
   thus shift by 14 to convert from .4 fixed point */
pWarpStruct->dxy = gPreviousLineOffsetVx.coord.y << 14ul;

/* Set dxx: signed 3.5 fixed-point shifted by 13
   thus shift by 14 to convert from .4 fixed point */
pWarpStruct->dyx = gPreviousLineOffsetVy.coord.x << 14ul;

/* Set dxy: signed 3.5 fixed-point shifted by 13
   thus shift by 14 to convert from .4 fixed point */
pWarpStruct->dyy = gPreviousLineOffsetVy.coord.y << 14ul;
```

Function `Float2Fix12_4` which returns `fixP_12_4_t` value according to input float type value is shown in **Code Listing 21**.

Input parameters of "`Float2Fix12_4`" are listed as follows.

Parameter	Description
`float f`	Float type value to be converted into type `fixP_12_4_t`

The following is the output of `Float2Fix12_4`:

Parameter	Description
`fixP_12_4_t`	`fixP_12_4_t` type value according to the input value.

Code Listing 21 Float2Fix12_4

```
/* Shifting 4 bit converts the normal value to a .4 fixed-point value. Because
   the input is floating-point value, multiply by 16.0 instead. */
return (fixP_12_4_t)(f * 16.0);
```

The `PackCoordinateBuffer` function which generates the coordinate buffer element value according to the input reference coordinate is shown in **Code Listing 22**. Static variables defined in **Code Listing 17** are used in this function. Because x and y elements of the coordinate buffer have only 4-bit length, it saturates the calculated result so that the result can be expressed in a 4-bit integer. The value which was out of 4-bit length will be saved in a variable named “error” and used later.

Input parameters of `PackCoordinateBuffer` are listed as follows.

Parameter	Description
`un_coord_fixP_12_4_t targetPos`	Next reference coordinate value
`void* pCoord`	Pointer to the top address of the coordinate buffer to be generated
`uint32_t xIdx`	Index of the pixel column of the warping map
`uint32_t yIdx`	Index of the pixel row of the warping map

Code Listing 22 PackCoordinateBuffer

```
un_coord_fixP_12_4_t previous_v;

/* Check if it is the first column */
if(xIdx == 0ul) /* first column*/
{
    /* Initialize some parameters using the saved value */
    gPreviousPosition.u32 = gPreviousLineOffset.u32;
    gPreviousVx.u32       = gPreviousLineOffsetVx.u32;
    previous_v.u32        = gPreviousLineOffsetVy.u32;
}
else /* Not the first column */
{
    /* Initialize the previous vector value */
    previous_v.u32 = gPreviousVx.u32;
}

/* Calculate the target vector */
un_coord_fixP_12_4_t target_v;
target_v = SubCoord(targetPos, gPreviousPosition);

/* Calculate the target vector delta */
un_coord_fixP_12_4_t target_d;
target_d = SubCoord(target_v, previous_v);
```

Code Listing 22 PackCoordinateBuffer

```
/* Saturate the target delta and get the error */
un_coord_fixP_12_4_t error;
error = SaturateFor4bit(&target_d);

/* Pack the target delta into the coordinate buffer */
SetCoordBufDeltaInc(pCoord, xIdx, yIdx, target_d);

/* Error correction */
/* Check if the error is not zero */
if(error.u32 != 0ul) /*the error is not "0". */
{
    /* Correct the target vector */
    target_v = SubCoord(target_v,  error);

    /* Correct the target position also */
    targetPos = SubCoord(targetPos, error);
}

/* Save parameters for later */
/* Save the target position as the previous position */
gPreviousPosition.u32 = targetPos.u32;

/* Check if it is the first column */
if(xIdx == 0ul) /* First column */
{
    /* Save the target position as the previous line offset */
    gPreviousLineOffset.u32   = targetPos.u32;

    /* Save the target vector as the previous line offset vy */
    gPreviousLineOffsetVy.u32 = target_v.u32;
}
else if (xIdx == 1ul) /* Second column */
{
    /* Save the target vector as the previous vector */
    gPreviousVx.u32           = target_v.u32;

    /* Save the target vector as the previous line offset vx */
    gPreviousLineOffsetVx.u32 = target_v.u32;
}
else /* Neither first column nor second column */
{
    /* Save the target vector as the previous vector */
    gPreviousVx.u32 = target_v.u32;
}
```

The `SubCoord` function which subtracts the `un_coord_fixP_12_s_t` variable from the same type variable is shown in **Code Listing 23**.

Input parameters of the `SubCoord` function are listed as follows.

Parameter	Description
`un_coord_fixP_12_4_t a`	Value to be subtracted from
`un_coord_fixP_12_4_t b`	Value to be subtracted by

The output of `SubCoord` is the following:

Parameter	Description
un_coord_fixP_12_4_t	Result of a - b

Code Listing 23 SubCoord

```
/* Define the temporary variable to save the result */
un_coord_fixP_12_4_t result;

/* Substruct the x element */
result.coord.x = a.coord.x - b.coord.x;

/* Substruct the y element */
result.coord.y = a.coord.y - b.coord.y;

/* Return the result */
return result;
```

The SaturateFor4bit function which saturates the input value is shown in **Code Listing 24**. Because the coordinate buffer value of each x and y element only has 4-bit length, it saturates the input value from -8 to 7 and returns the rest as an error.

Input parameters of SaturateFor4bit are listed as follows:

Parameter	Description
un_coord_fixP_12_4_t* coord	Pointer to the un_coord_fixP_12_4_t type variable to be saturated

The following is the output of SaturateFor4bit:

Parameter	Description
un_coord_fixP_12_4_t	Indicates the offset from the 4-bits range (-8 ~ 7). For example, if 10 was input, then this value would be 3 (10 – 7).

Code Listing 24 SaturateFor4bit

```
/* Define a temporary variable to save the error and initialize it to zero */
un_coord_fixP_12_4_t error = {0};

/* Check if the input x element is larger than the maximum value of the  signed
   4-bit data, that is 7. Also check if the x element is smaller than the
   minimam value of the signed 4-bit data, that is (-8) */
if((coord->coord.x) > 7) /* It is larger than the maximum value */
{
    /* Save the error value */
    error.coord.x = coord->coord.x - 7;

    /* Set the maximum value to the input */
    coord->coord.x = 7;
}
else if((coord->coord.x) < -8) /* It is smaller than the minimum value */
{
    /* Save the error value */
    error.coord.x = coord->coord.x + 8;

    /* Set the minimum value to the input */
    coord->coord.x = -8;
}
```

Code Listing 24 SaturateFor4bit

```
/* Check if the input y element is larger than the maximum value of the signed
   4-bit data, that is 7. Also check if the y element is smaller than the
   minimam value of the signed 4-bit data, that is (-8) */
if((coord->coord.y) > 7) /* It is larger than the maximum value */
{
    /* Save the error value */
    error.coord.y = coord->coord.y - 7;

    /* Set the maximum value to the input */
    coord->coord.y = 7;
}
else if((coord->coord.y) < -8)  /* It is smaller than the minimum value */
{
    /* Save the error value */
    error.coord.y = coord->coord.y + 8;

    /* Set the minimum value to the input */
    coord->coord.y = -8;
}

return error;
```

The `SetCoordBufDeltaInc` function which writes data into the coordinate buffer having 8-bit length element. Upper 4 bits indicate the x element, lower 4 bits indicate the y element. 1LSB indicates 1/16 pixel.

Input parameters of `SetCoordBufDeltaInc` are listed as follows:

Parameter	Description
`uint8_t p[IMAGE_HEIGHT][IMAGE_WIDTH]`	Pointer to the top address of `uint8_t` type coordinate buffer array
`uint32_t xIdx`	Column index of the coordinate buffer
`uint32_t yIdx`	Row index of the coordinate buffer
`un_coord_fixP_12_t value`	Value to be set to the coordinate buffer

Code Listing 25 SetCoordBufDeltaInc

```
/* Upper 4 bits of coordinate buffer value indicates x element. Lower 4 bits of
   that indicates y element. */
p[yIdx][xIdx] = ((uint8_t)value.coord.x << 4u) +
((uint8_t)value.coord.y & 0x0F);
```

4.3 Assign the warping structure and the coordinate buffer

Finally, assign the address of the generated coordinate buffer to the warping layer. See the TRAVEO™ T2G graphics driver [5] tutorial.

5 Appendix

5.1 Appendix: Background shift with one layer

This section introduces a way to shift the background image with only one decoding layer. See **Figure 29**: in addition to Background 1 and Background 2, you must prepare “Background 1 To 2” and “Background 2 To 1” which is Background 1 and Background 2 are combined side-by-side. Note that this application doesn’t use blit for outputting the background image. Thus, there are no blit modes.

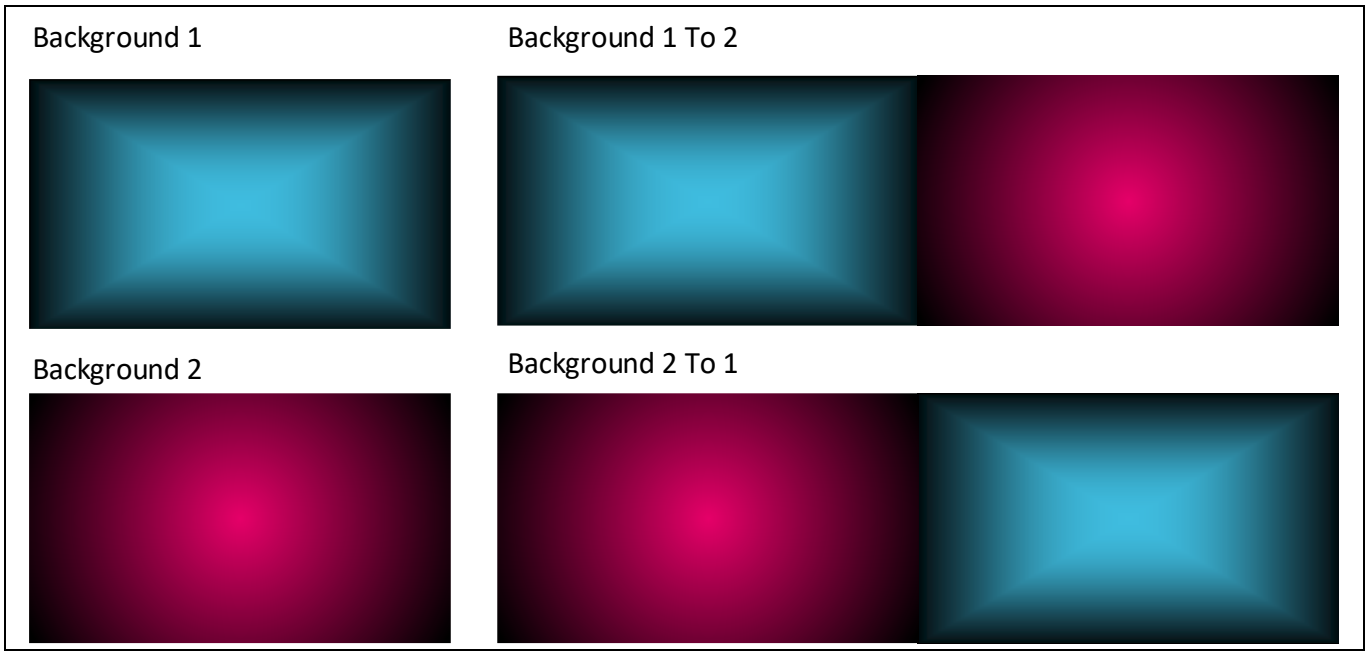

Figure 29 Prepared background image for layer reduced shift

The procedure in which displayed image is being changed from “Background 1” to “Background 2” is shown in **Figure 30**.

1. Background 1 on Window 0 is shown in the display.
2. Shifting starts. Window 0 doubles its width and shows image Background 1 To 2.
3. Window 0 moves to left direction.
4. Shifting done. Background 1 part of the Background 1 To 2 are completely out of the display and only Background 2 part is being displayed.
5. Window 0 reverts its width and makes its coordinate to left top of the display with showing image Background 2.

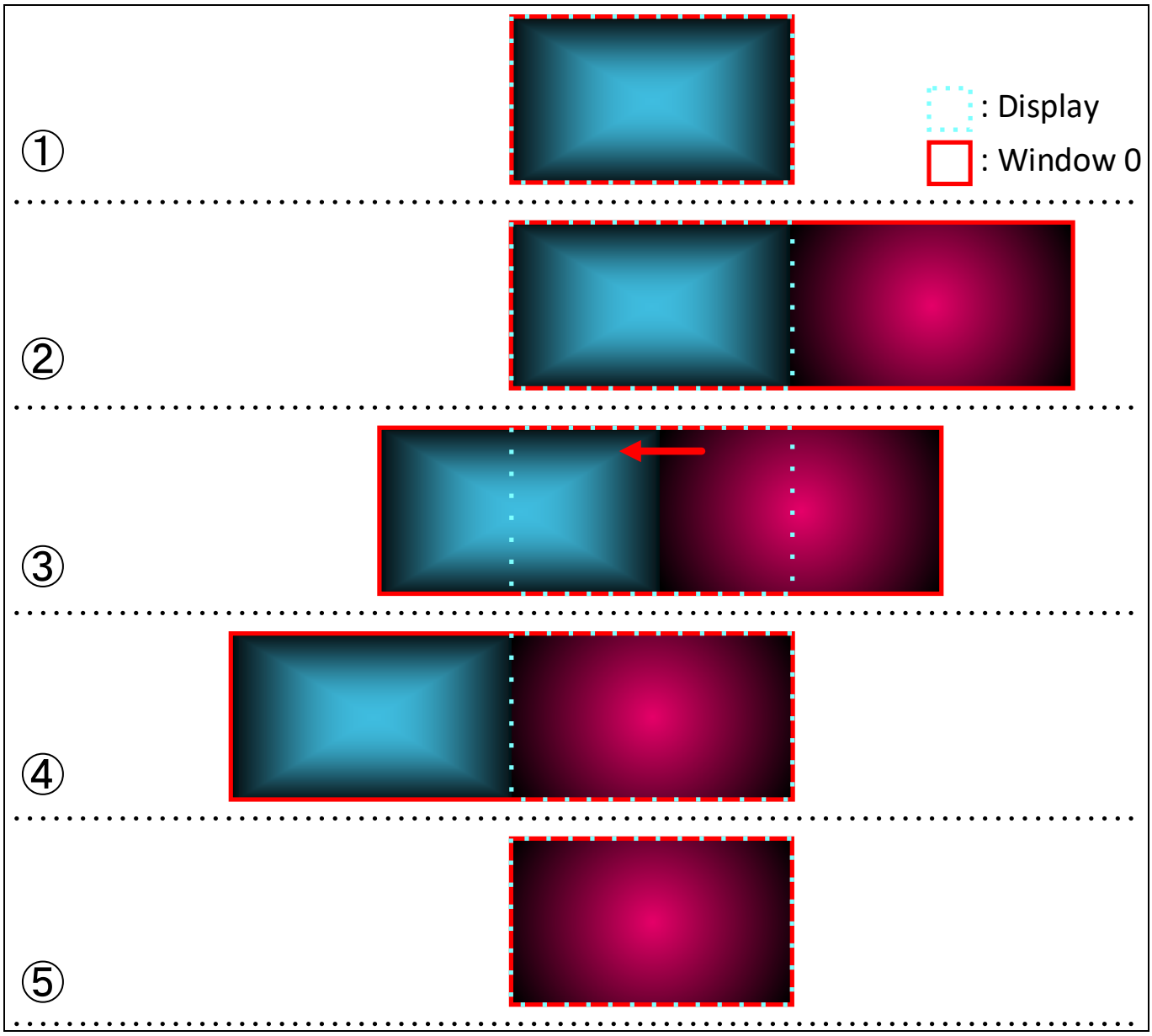

Figure 30 Layer reduced shift from Background 1 to Background 2

Compared to the basic way, this method uses fewer layers. However, because the layer reads the image which has a long width, line drawing may not be completed in time, depending on the image. Also, you must prepare the image data beforehand, which consumes the external flash memory area.

5.2 Appendix: Background shift with less flash access

This section introduces a way to shift the background image with less flash access. See **Figure 32**: you must prepare “Half Bg 1_1”, “Half Bg 1_2”, Half Bg 2_1”, and “Half Bg 2_2” which are “Background 1” and “Background 2” vertically evenly divided. Also, you must prepare “Half Bg 1 To Half Bg 2” which is Half Bg 1_2 and Half Bg 2_1 combined side-by-side, and “Half Bg 2 To Half Bg 1” which is Half Bg 2_2 and Half Bg 1_1 combined side-by-side. Note that this application does not use blit to output the background image. Thus, there are no blit modes.

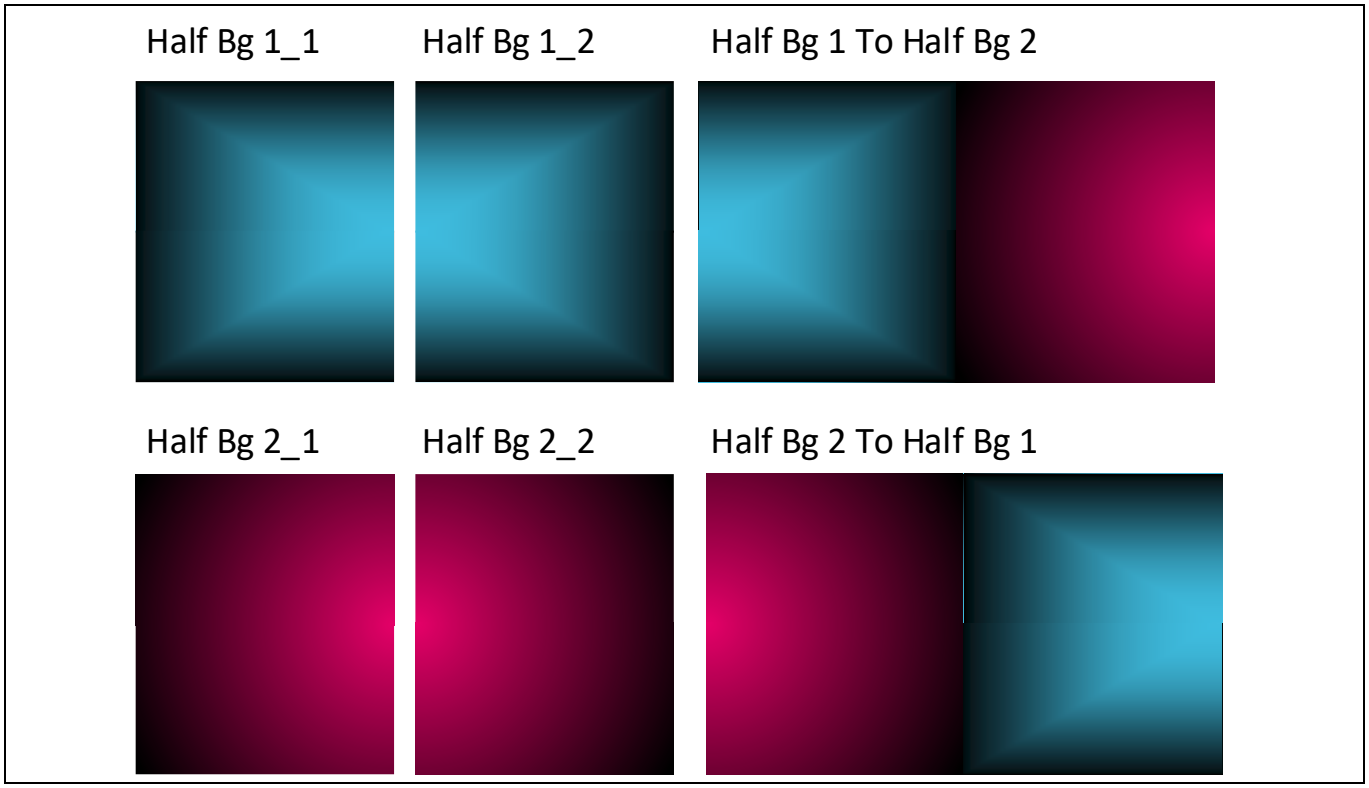

Figure 31 Prepared background image for data access reduced shift

The procedure in which the background image is being switched is shown in **Figure 32**.

1. Two windows are used to display Background 1. The left-side window (Window 0) shows Half Bg 1_1; the right-side window (Window 1) shows Half Bg 1_2.
2. Shifting starts. Window 1 doubles its width and starts showing Half Bg 1 To Half Bg 2.
3. Windows have been shifted left by half the width of the display. Window 0 is completely out of the display.
4. Coordinate of Window 0 was changed to the top right of the display. It starts showing Half Bg 2_2.
5. Windows have been shifted left by another half width of the display; this completes the shifting.
6. Windows 1 reverts its width and starts showing the image Half Bg 2_1.

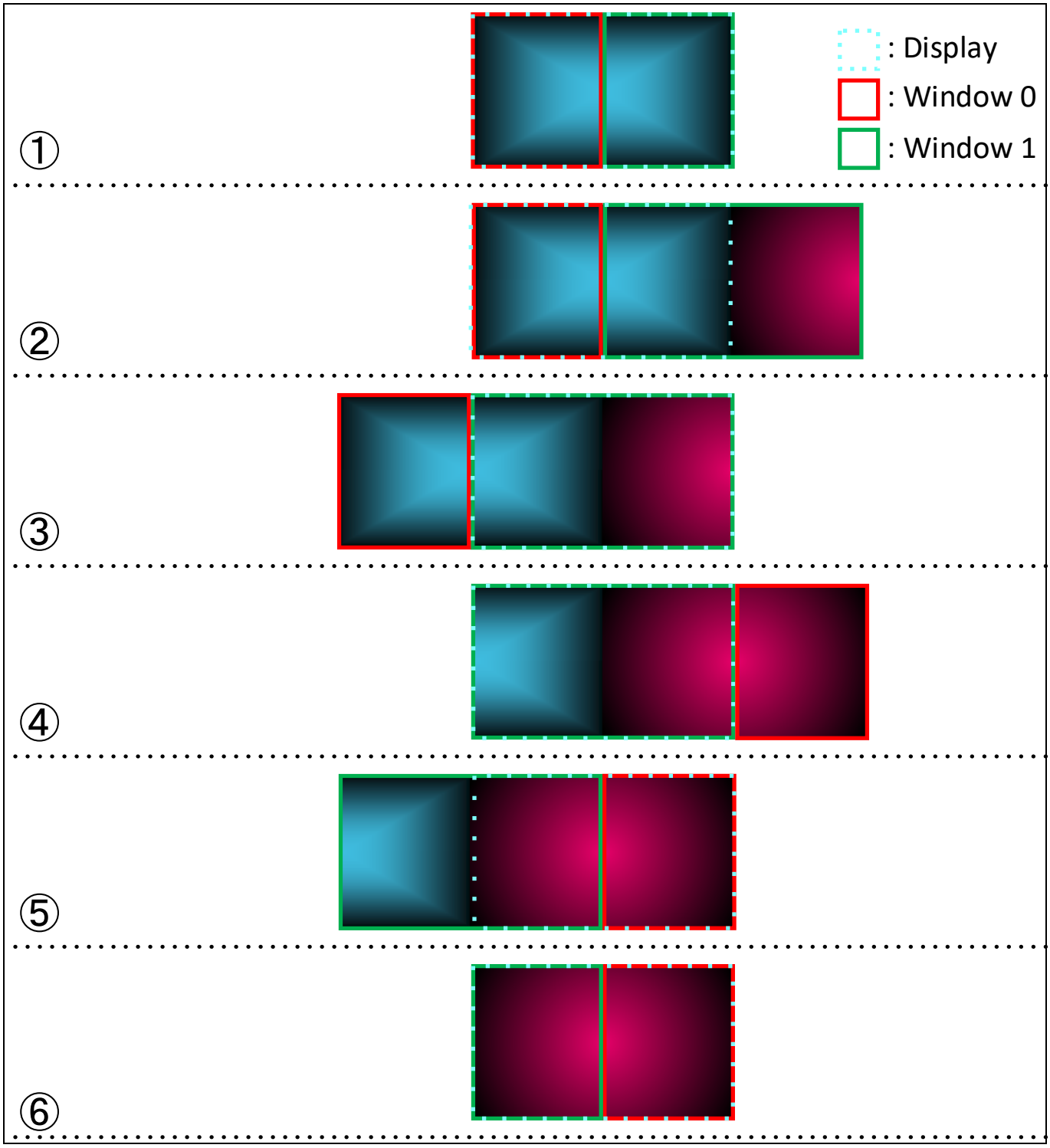

Figure 32 Data access reduced shift from Background 1 to Background 2

This method allows you to shift the background image with logically one-and-a-half data access compared to the basic method. Although this method has a merit of less flash data access, user have to prepare divided background images and images combined those images also, the software may be more complex than basic one.

5.3 Appendix: Image warping

Image warping is a procedure which distort the image by manipulating its data. This is realized by transferring the color information at a position of the source image to another position. The color information itself is not changed in this procedure (**Figure 33**).

In automobile graphics applications, this function was used when showing images on a curved surface like head up display. Image on curved surface like front window looks distorted. Image warping offsets this distortion so that drivers can see normal images.

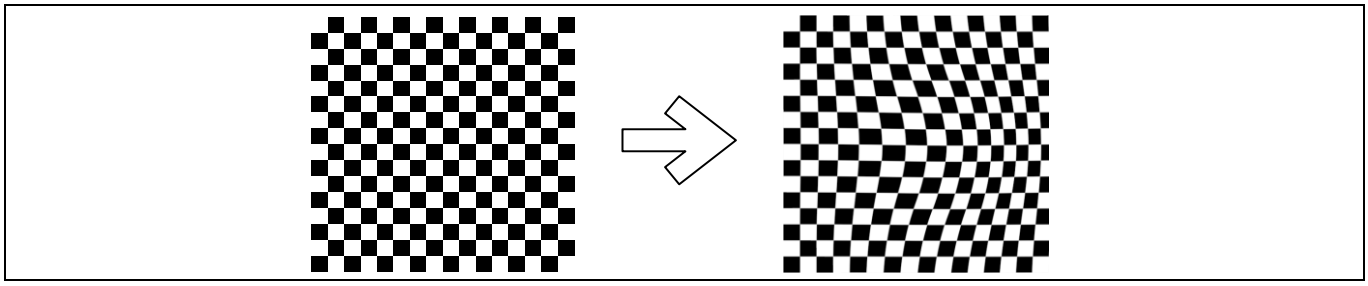

Figure 33 Image warping

5.3.1 Warping map and reference coordinate

In the warping process, a map between the source coordinate and the destination coordinate is called “warping map” in this document. This sub-section describes a concept of the warping map with a simple warping example. An example of 2-pixel x 2-pixel image warping is shown in **Figure 34**.

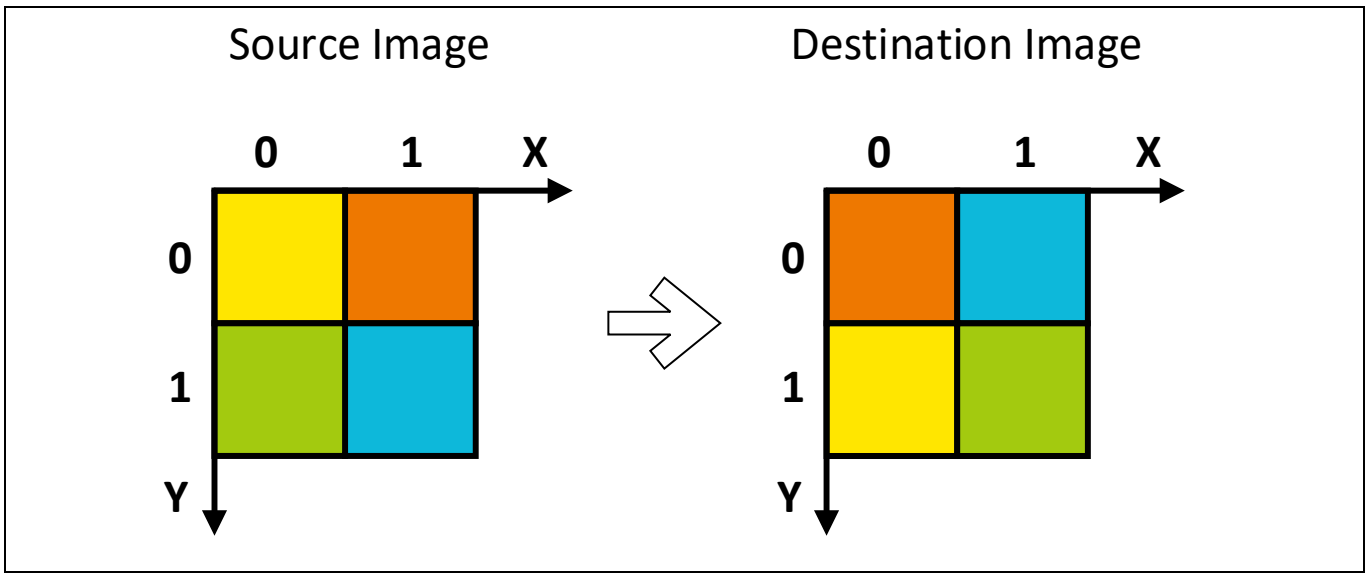

Figure 34 Simple warping example

The warping map has a position of source image to be referred at a position of destination image. Consider a warping map that indicates warping shown in **Figure 34**. The orange pixel shown at (1, 0) in the source image is transferred to (0, 0) in the destination image. This means that the value of the warping map at (0, 0) is (1, 0). In this way, you can determine the warping map as shown in **Figure 35**. Here, the value in the warping map indicates the coordinates of the source image to be referred, and the position in the map indicates the coordinates of the destination image.

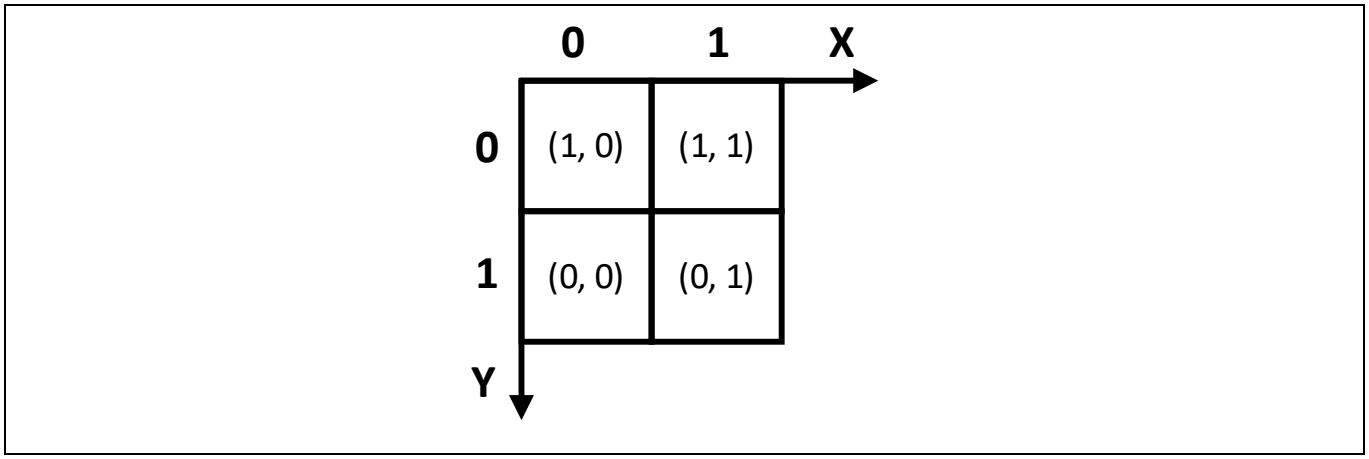

Figure 35 Simple warping map

The concept of warping is generating destination image from a source image and warping map which have been input as shown in **Figure 36**.

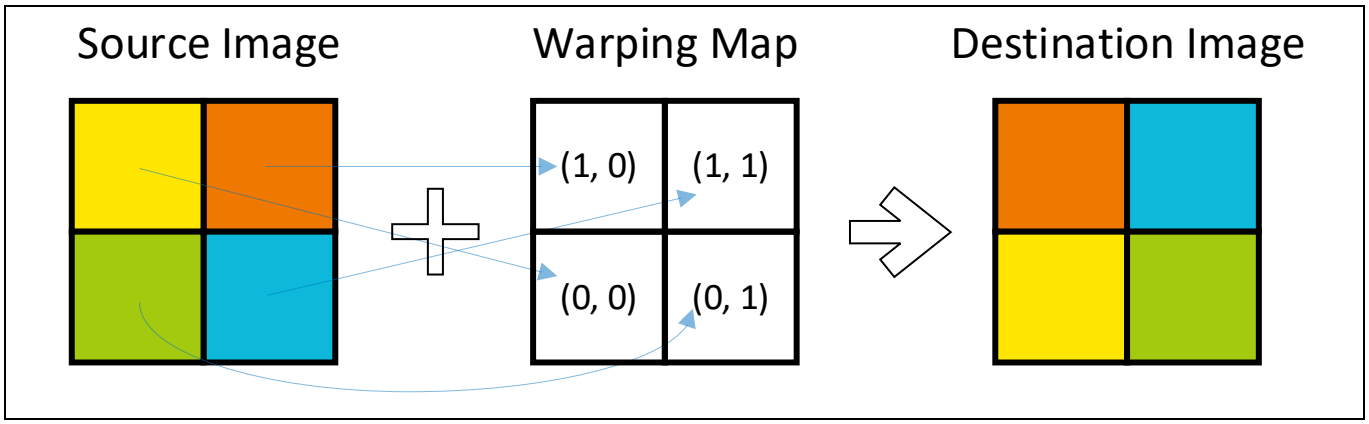

Figure 36 Simple warping concept

One of software processes to warp an image is to calculate the warping map. In this document, the value of the warping map is called “reference coordinate”. The warping map is a two-dimensional array of reference coordinates.

5.4 Appendix: Warping layer and coordinate buffer

The TRAVEO™ T2G graphics sub-system has a layer which has warping functionality. In this document, it is called “warping layer”. The warping layer processes the source image data and the data which represent the warping map, and then outputs the warped image onto the display. The data which represents the warping map is called “coordinate buffer”. That is, the warping layer generates a destination image from the source image and the coordinate buffer.

The coordinate buffer is a data array which the warping layer reads directly. The coordinate buffer represents the warping map. Although it can be the warping map itself, it is typically in a compressed form to reduce its size. The software can convert the warping map into a coordinate buffer and vice versa. The form of the coordinate buffer is described in the following section. The size of the coordinate buffer element can be selected from 32, 24, 16, 8, 4, 2, and 1. The value is a vector; the upper bits represent the x component and the lower bits represent the y component. X and y components are signed fixed-point fractional values and 1LSB indicates 1/16 pixel. In the case of 1bpp, every other bit represents x and y components.

5.4.1 Modes of the warping layer

The warping layer has three modes: sample points mode, delta vectors mode, and delta vector increments mode. There are forms of the coordinate buffer according to the warping layer modes.

Although technically the bit per pixel(bpp) value of the coordinate buffer can be set independent of its form, typically, the required bpp size depends on its form. For example, the form for the sample points mode requires largest size and the form for the delta vectors mode require the second largest size, and the form for the delta vector increments mode requires the smallest size.

The following sub-sections describe the coordinate buffer forms for each warping layer mode using a 4-pixel x 3-pixel source image example. Note that the elements of the coordinate buffer are signed fixed-point fractional values. the lower bit of which indicates the fractional part. If the warping map had more precision than the coordinate buffer, it causes an error. In the following sub-sections, to make the description simple, it is assumed that the precision of the warping map and the coordinate buffer are the same, which ensures that errors do not occur.

5.4.1.1 Sample points mode and the coordinate buffer

In the sample points mode, the coordinate buffer is identical with the warping map. That is, the element value in the coordinate buffer represents the reference coordinate. In **Figure 37**, p*i* (*i* = 0 ~ 11), which has the x and y components, represents the reference coordinate. For example, the color at point (1, 1) of the warped image would be the color which is pointed by “p5” on the source image.

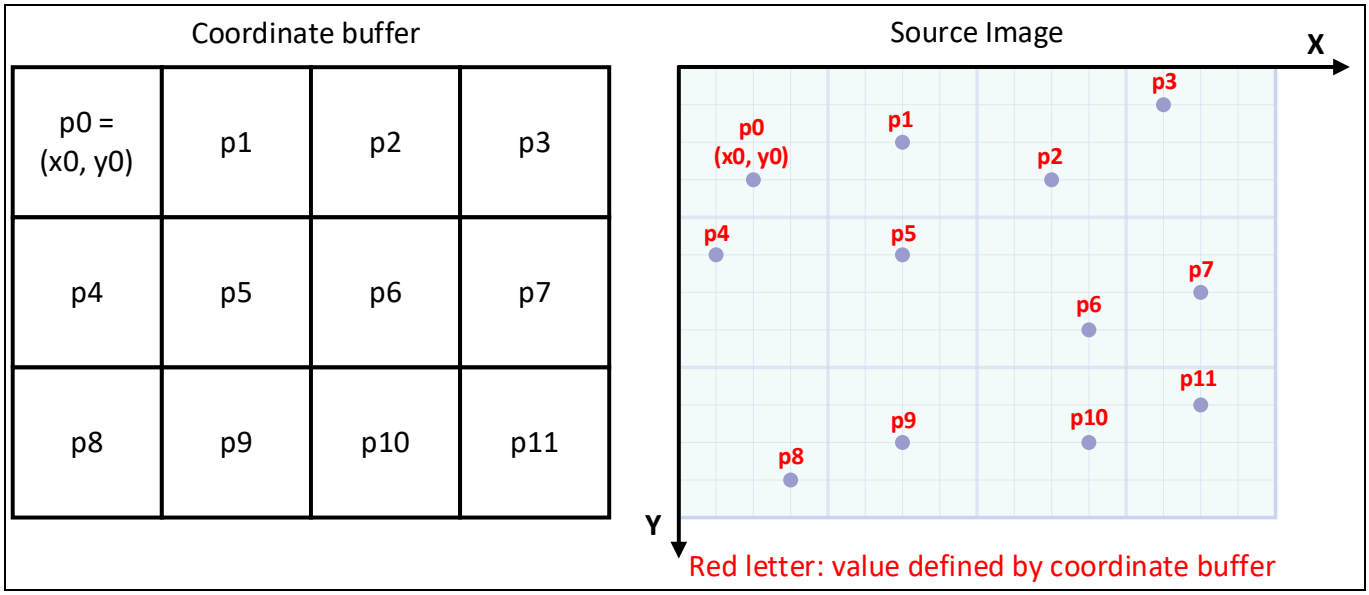

Figure 37 Coordinate buffer of sample points mode

The flowchart which generates the coordinate buffer in the sample points mode from the input warping map as shown in **Figure 38**. “CB” in the flowchart means an array of the coordinate buffer. “WM” in the flowchart means an array of the warping map. Array elements are a structure which contains the x and y components.

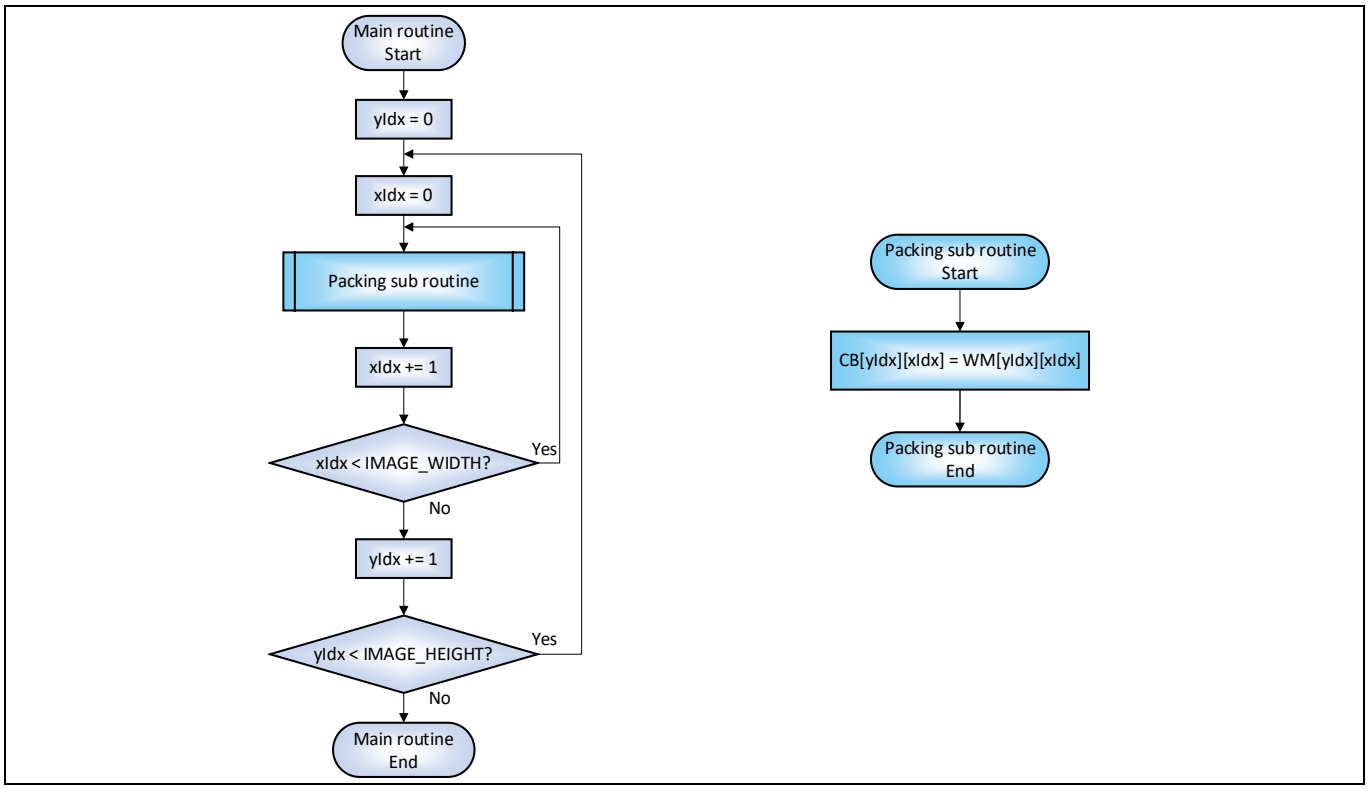

Figure 38 Flowchart: Warping map to sample points mode coordinate buffer

The main routine in the flowchart indicates generating a coordinate buffer line-by-line from top left. This main routine is common in the delta vectors mode flowchart (**Figure 40**) and delta vector increments mode flowchart (**Figure 42**) in the following sections.

5.4.1.2 Delta vectors mode and coordinate buffer

The value of the coordinate buffer in the delta vector mode represents the difference between the reference coordinate and the reference coordinate next to that. **Figure 39** describes the coordinate buffer in the delta vector mode. Although this coordinate buffer is in a different form, it represents the same warping map with that of the sample points mode coordinate buffer shown in **Figure 37**.

v*i* (*i* = 0 ~ 11) which have x and y components, represents the differences between adjacent reference coordinates. This v*i* is called “delta vector”.

In this mode, the offset position must be defined by the register value of the warping layer.

The first delta vector value (v0) represents the difference from the offset position to p0 (v0 = p0 - offset). The delta vectors in the first column represents the difference from the reference coordinate at an upper line (e.g., v8 = p8 - p4). Other delta vectors represent the difference from a left reference coordinate (e.g., v6 = p6 - p5).

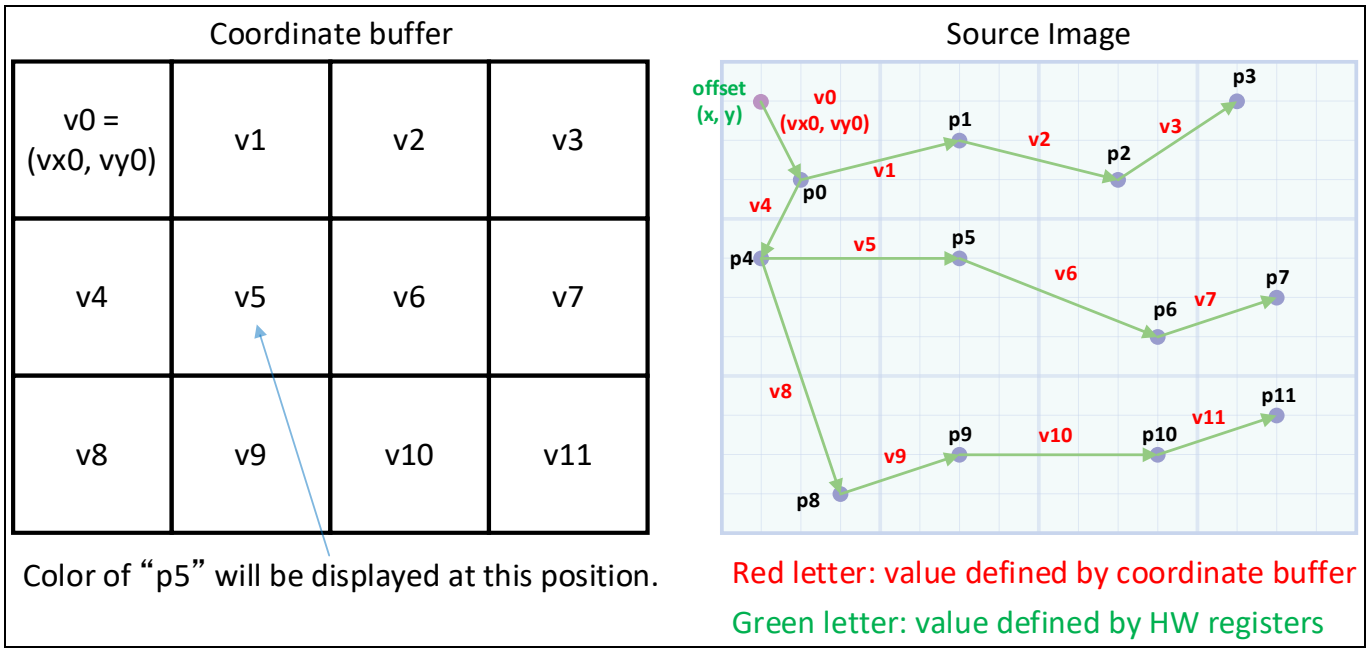

Figure 39 Coordinate buffer of delta vectors mode

The flowchart which generates the delta vector mode coordinate buffer from the warping map is shown in **Figure 40**. The main routine which calls the packing subroutine in the flowchart is shown in **Figure 38**. Although this flowchart explains the relationship between the coordinate buffer and the warping map, this is not practical. A practical flow would be more complex considering the memory usage and error correction.

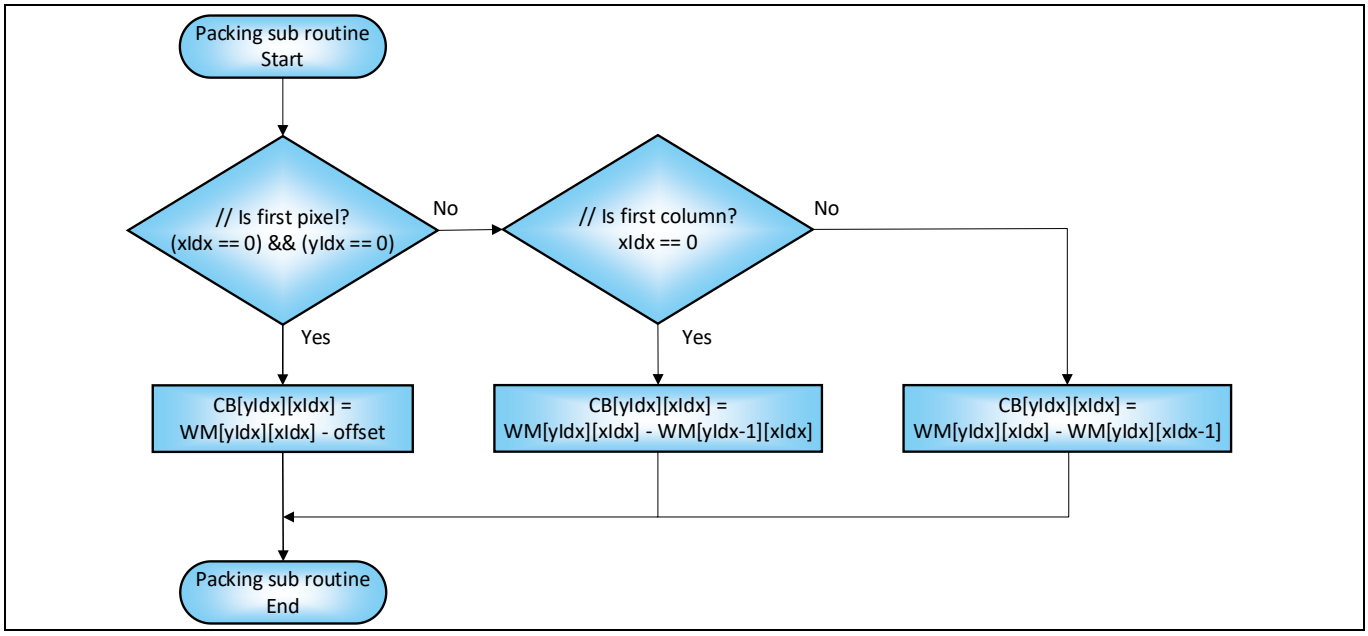

Figure 40 Flowchart: Warping map to delta vectors mode coordinate buffer

5.4.1.3 Delta vector increments mode and the coordinate buffer

In the delta vector increments mode, the element value of the coordinate buffer represents the difference between adjacent delta vectors. **Figure 37** shows the coordinate buffer in the delta vector increments mode. Although this mode is in a different form, this indicates the same warping map shown in **Figure 37** and **Figure 39**.

d*i* (*i* = 0 ~ 11) has x and y components, and indicates the difference between adjacent delta vectors.

In this mode, not only the offset point must be defined but also offset vy (ovy) and offset vx (ovx) . ovy is the initial delta vector for y direction and ovx is the initial delta vector for x direction. These values are defined by the registers of the warping layer.

The first value of the coordinate buffer (d0) represents the difference between ovy and v0 (d0 = v0 - ovy). The second value of the coordinate buffer (d1) represents the difference between ovx and v1 (d1 = v1 - ovx). Other values in the first and the second column represent the difference from the upper reference coordinate (e.g., d8 = v8 - v4, d9 = v9 - v5). Other column values represent the difference from the left reference coordinate (e.g., d6 = v6 - v5).

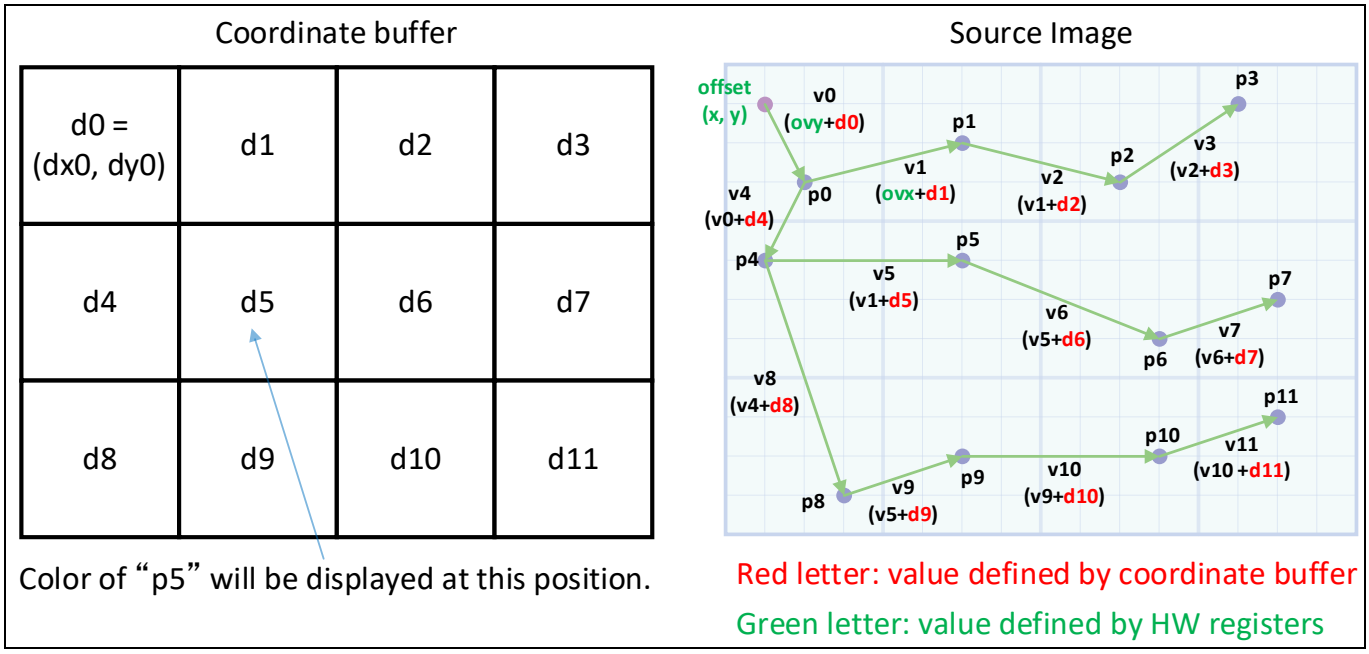

Figure 41 Coordinate buffer of delta vector increments mode

The flowchart which generates the coordinate buffer from the warping map is shown in **Figure 42**. It is assumed that "previous_v", "previous_line_vx", and "previous_line_vy" are variables which have been statically reserved in the program. The main routine which calls the packing subroutine in this flowchart is shown in **Figure 38**. Although this flowchart explains the relationship between the coordinate buffer and the warping map, this is not practical. A practical flow would be more complex considering the memory usage and error correction.

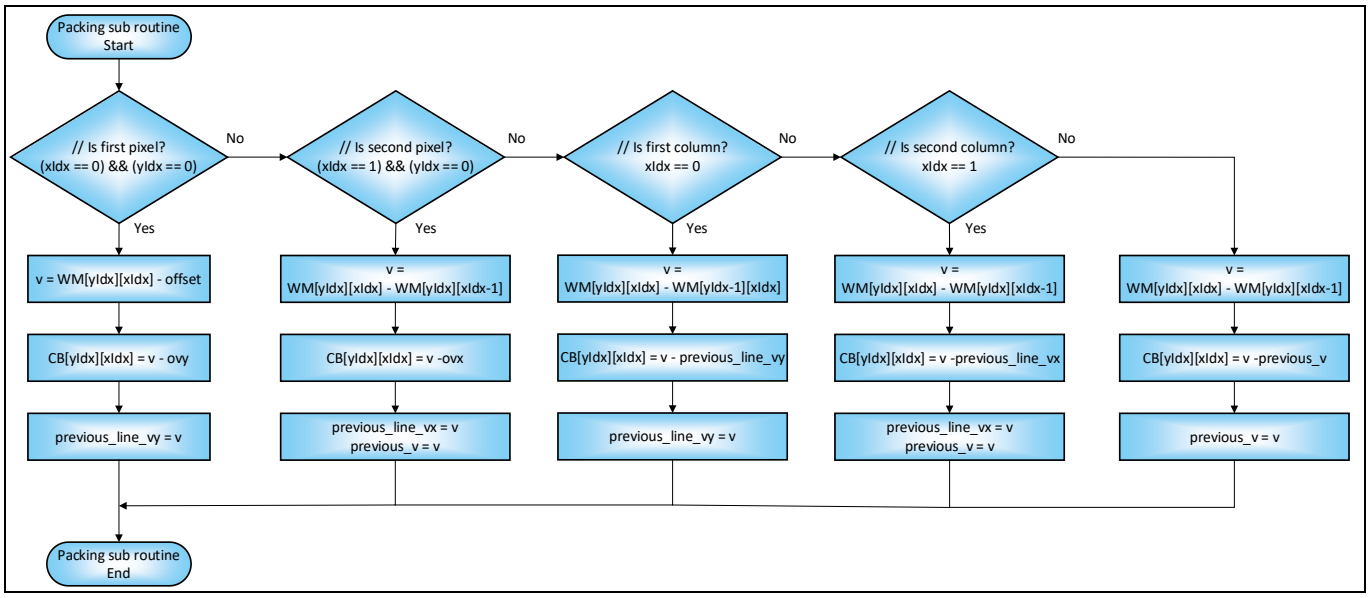

Figure 42 Flowchart: Warping map to delta vector increments mode coordinate buffer

5.5 Appendix: Distortion calibration procedure

The distortion calibration procedure can be roughly divided into the following three procedures. This appendix provides an overview of the distortion calibration procedure. In a practical application, a procedure may be done in parallel to improve memory usage and effectiveness.

5.5.1 Giving distortion grid nodes

You can see the distorted image which is distorted as the center point goes to the right upper corner in **Figure 43**. Some points on the image express the distortion, which are called “distortion grid nodes” in this document. After distortion grid nodes are given, the application starts the calibration of the distortion expressed by the distortion grid nodes. In this case, the application tries to offset the distortion; thus, eventually the application warps the source image to the left down direction.

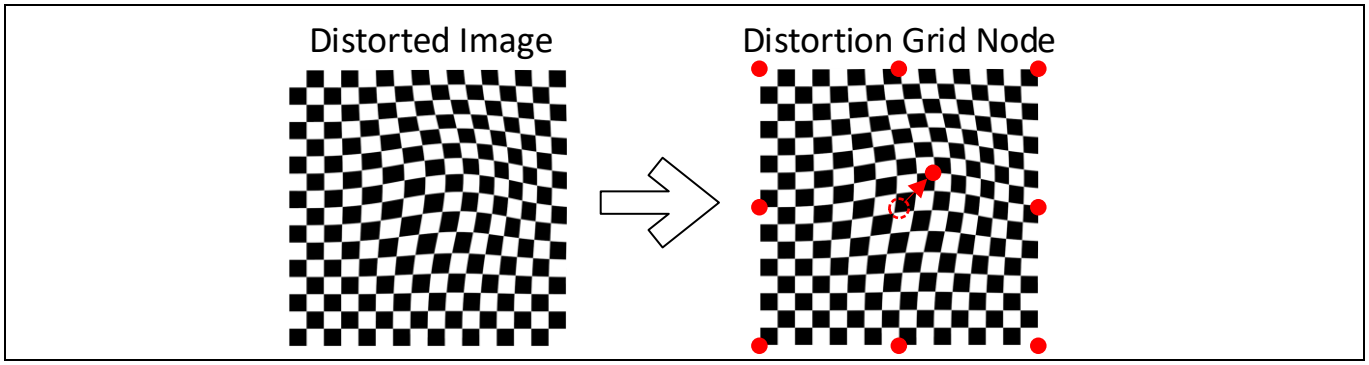

Figure 43 Distortion grid nodes

5.5.2 Spline interpolation and generating the warping map

The application starts generating the warping map which offsets the distortion expressed by the given distortion grid nodes. Points evenly aligned on warping map shown in **Figure 44** are called “normal grid nodes” in this document. For example, the reference coordinate value in the warping map at “Pw” on the normal grid nodes will be “Ps” in the distortion grid nodes. The warping map value which are not on the normal grid nodes are calculated by spline interpolation. See **Appendix: Spline interpolation**.

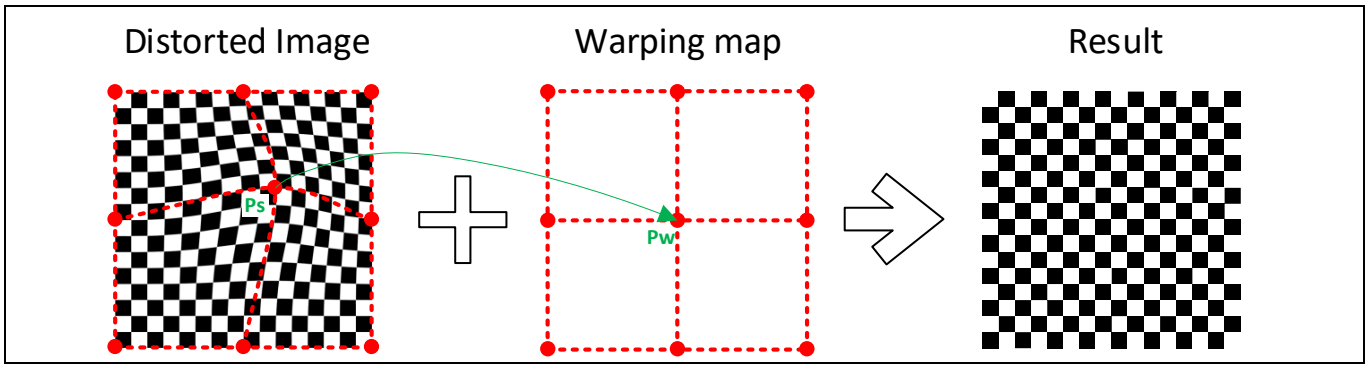

Figure 44 Overview of splining the grid points and making the warping map

5.5.3 Generating the coordinate buffer

The application converts the warping map into a coordinate buffer which can be read by the warping layer directly. See **Appendix: Warping layer and coordinate buffer** for an overview of conversion from the warping map to the coordinate buffer.

5.6 Appendix: Spline interpolation

This appendix provides an overview of spline interpolation but not its algorithm. Assume value y(x) which depends on value x. If a discrete value x*i* and accordingly y(x*i*) are given, interpolating between the discrete values called spline interpolation (**Figure 45**).

There are many spline interpolating algorithms; however this document does not describe them. This document assumes that you can use any algorithm to interpolate between points.

The application described in this document was confirmed with an algorithm of natural cubic spline.

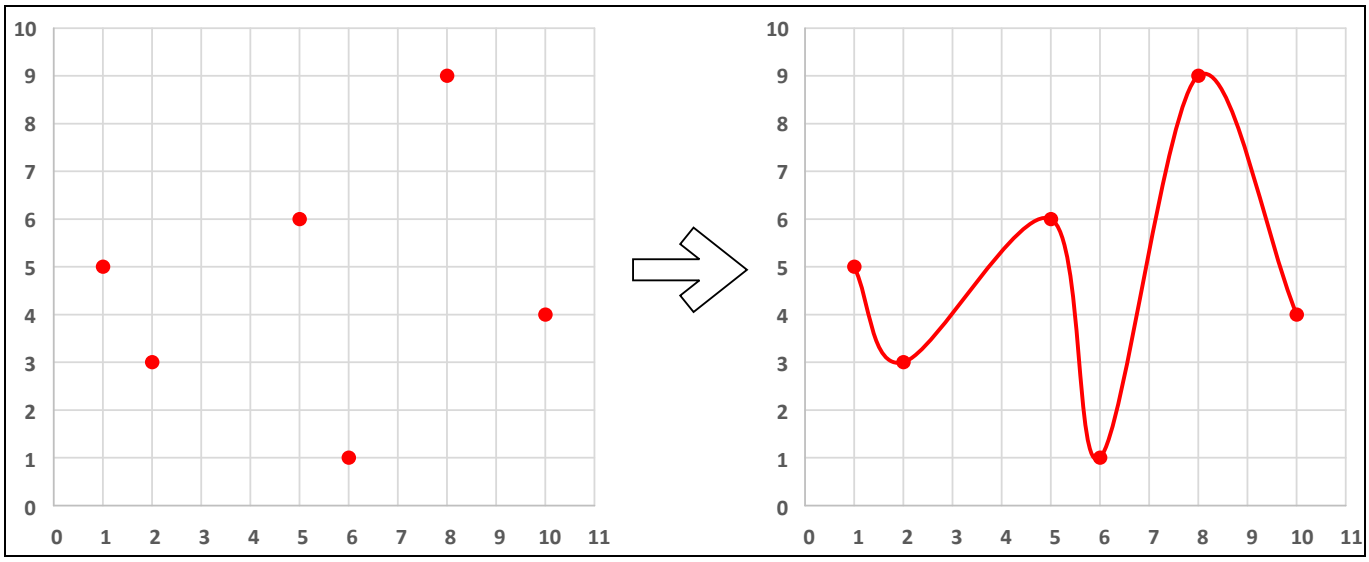

Figure 45 Spline interpolation

6 References

The following are the TRAVEO™ T2G family automotive microcontroller datasheets and technical reference manuals. Contact **Technical Support** to obtain these documents.

[1] Device datasheet

- TRAVEO™ T2G family CYT4DN series 32-bit Arm® Cortex®-M7 automotive microcontroller (Doc No. 002-24061)

[2] Technical reference manual

CYT4D series

- TRAVEO™ T2G 2D family cluster series automotive microcontroller architecture technical reference manual (TRM) (Doc No. 002-25800)
- TRAVEO™ T2G 2D family cluster series automotive microcontroller registers technical reference manual (TRM) for CYT4D (Doc No. 002-25923)

[3] Application notes

- **AN224434 - Clock configuration setup in TRAVEO™ T2G family CYT4B series**
- **AN219842 - How to use interrupt in TRAVEO™ T2G**

Other references

[4] A sample driver library (SDL) including startup as sample software to access various peripherals is provided. SDL also serves as a reference to customers for drivers that are not covered by the official AUTOSAR products. The SDL cannot be used for production purposes as it does not qualify to any automotive standards. The code snippets in this application note are part of the SDL. Contact **Technical Support** to obtain the SDL.

[5] To obtain the graphics driver for TRAVEO™ T2G family cluster series automotive microcontroller, contact **Technical Support**.

[6] AN233199 - TRAVEO™ T2G family cluster series automotive microcontroller – Video subsystem hardware overview

Revision history

Document version	Date of release	Description of changes
**	2021-09-30	New application note.
*A	2022-06-09	Added a sub-section "**2.5 Disabling the Graphics Hardware**".

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Edition 2022-06-09

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Go to www.infineon.com/support

Document reference

002-31948 Rev. *A

IMPORTANT NOTICE

The information contained in this application note is given as a hint for the implementation of the product only and shall in no event be regarded as a description or warranty of a certain functionality, condition or quality of the product. Before implementation of the product, the recipient of this application note must verify any function and other technical information given herein in the real application. Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind (including without limitation warranties of non-infringement of intellectual property rights of any third party) with respect to any and all information given in this application note.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office (**www.infineon.com**).

WARNINGS

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.