Revolution SDK

Video Interface Library (VI)

Version 1.07

The contents in this document are highly confidential and should be handled accordingly.

RVL-06-0040-001-H
Released: October 17, 2008

Contents

Code Examples

Figures

Tables

Revision History

Version	Date Revised	Item	Description
1.07	2008/07/22	-	Corrected mixed notations.
	2008/07/10	7.1.2	Added a description related to adjusting the horizontal position from the System Settings menu.
	2008/06/10	6.5.1	Simplified the procedure for determining the progressive mode in the video mode switching flowchart to use `VIGetScanMode` instead of IPL for some of the processing.
		7.5.2	Simplified the procedure for determining the progressive mode in the video mode switching flowchart to use `VIGetScanMode` instead of IPL for some of the processing.
	2008/06/03	7.1.2	Revised the equations for *viXOrigin* and *viYOrigin* because they were easy to misinterpret.
		7.1.4	Revised the equations for *viXOrigin* and *viYOrigin* because they were easy to misinterpret.
	2008/05/13	7.5.2	Added a determination sequence using `SCGetEuRgb60Mode` to the video mode switching flow.
	2008/03/19	5.2	Simplified the procedure for determining the progressive mode in the video mode switching flowchart to use `VIGetScanMode` instead of IPL for some of the processing.
		7.8	Divided this section into two: Support for 16:9 Aspect Ratio and Support for 4:3 Aspect Ratio. Added figures describing screen aspect ratio mode support to both sections.
1.06	2007/12/04	All	Deleted all references to MPAL.
	2007/10/12	6.5.1	Changed the portion of the video mode switching flow that used `VIGetTvFormat` for determination. This portion now uses `SCGetEuRgb60Mode`.
		7.10.3	Added a description of ways to avoid screen burn-in.
		7.10.1	Added a description of screen burn-in.
1.05	2007/07/12	7.8	Added values for setting the screen size for 4:3 display. Changed the title to “Aspect Ratio and Configuration Values.”
		7.10	Added a caution about resetting the counter for screen burn-in reduction. Added a caution about the screen that is shown while screen burn-in reduction is in effect. Added the conditions that can disable screen burn-in reduction. Added the conditions for returning from screen burn-in reduction.
1.04	2006/10/26	7.10	Re-added description of `VIEnableDimming`.
	2006/10/25	7.10	Added new description of `VISetTimeToDimming`. Deleted description of `VIEnableDImming`.

Version	Date Revised	Item	Description
1.03	2006/10/17	3.2	Revised text to read EURGB60 works with all cables.
		4.3	Added the text regarding progressive mode to the caution to wait two frames when the video mode is changed.
		5.4.2	Deleted the “black screen” step after “select display system” from the switch over.
		7.5.2	Revised the flow chart.
		6.5.3	Added “Recommendations Regarding PAL Video Mode.”
		6.5.1	Revised the procedure for switching the video mode when using PAL. Revised the flow chart. Revised text.
		5.2	Revised the procedure for switching the video mode when using NTSC. Revised the flow chart. Revised the text.
		7.11	Added the section “Notes on Vertically Striped Patterns Being Generated.”
1.02	2006/10/05	7.5.1	Added a caution for VIInit.
		7.5	Deleted the EUROGB60 caution in the table (req. RGB cable connection)
		5.4.3	Changed "Reset Guidelines" to "Programming Guidelines."
		5.4.2	Changed the content giving consideration to the switch from progressive to interlace.
		7.8	Revised the figure. Added the recommended value for each screen size.
		7.5.2	Added the “Automatic Switching of the Video Mode” section.
		6.5.1	Changed the content to give consideration that the system configuration menu is displayed in progressive mode.
		5.2	Changed the content to give consideration that the system configuration menu is displayed in progressive mode.
		7.10	Added description of the support functions.
		7.1.6	Changed the 16:9 safe frame to 87%.

Version	Date Revised	Item	Description
1.01	2006/08/15	All	Changed “Revolution” to “Wii.”
		All	Updated tables.
		6.5.1	Added Switching between PAL/EURGB60/Progressive Modes.
		7.1.5	Added “Pixel Ratio.”
		7.1.6	Added “Safe Frame.”
		7.1.7	Added “Black Bands at Right and Left Screen Borders.”
		7.1.8	Added “Switching Render Modes.”
		7.5	Added “Video Format.”
		7.6	Added “Handling YUV Data”
		7.7	Added “Screen Noise Problem After Closing IPL.”
		7.8	Added “16:9 Screen Ratio.”
		7.9	Added “Trap Filter.”
		7.10	Added “Screen Burn-In.”
1.00	2006/03/22	-	First release by Nintendo of America Inc.

1 Overview

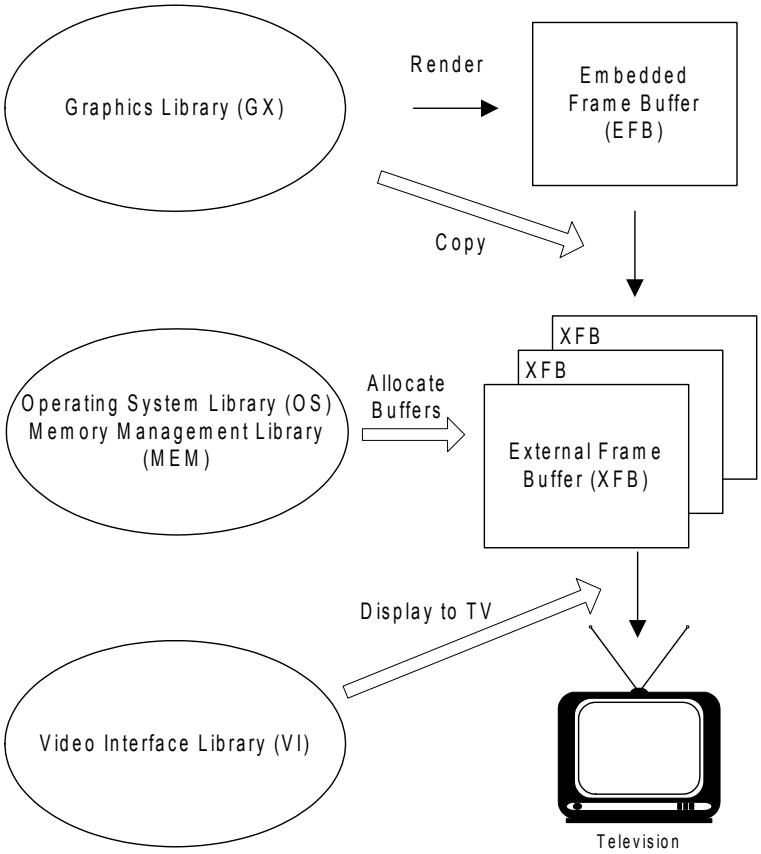
The Video Interface library (VI) contains functions to control the Video Interface hardware. This document shows how to use the VI library and what you can accomplish with it.

2 Relationship of VI to Other System Libraries

The main job of VI is to display the external frame buffers (XFB) generated by the Graphics library (GX). External frame buffers are allocated in main memory (MEM1 or MEM2); the number allocated depends on whether you are using a double- or triple-buffer display technique.

Note: GX renders the embedded frame buffer (EFB) and then copies it to the XFB after rendering.

Figure 2–1 Relationship Between System Libraries



Note: In this document, the term “frame buffer” (or simply “FB”) always refers to an *external* frame buffer because the VI library touches only this type. We use the abbreviations XFB and EFB to differentiate between external and embedded frame buffers, respectively, when referring to both types.

3 VI Main Features

Here are the main features of the Wii VI hardware and library:

- NTSC and PAL video formats
- Additional video format for European market, called “EURGB60”
- Strong support for interlaced display, as well as support for traditional non-interlaced display
- Arbitrary positioning and sizing of image on the TV
- Support for light gun target location
- Horizontal scaling to maximize frame buffer coverage on the TV
- Support for the 4:2:2 YCbCr pixel format to reduce data size in the frame buffer

Due to these characteristics, the Wii VI hardware and library are fully compatible with Nintendo GameCube.

3.1 NTSC and PAL Video Formats

Like previous game consoles, the Nintendo Wii Video Interface supports two video formats: NTSC and PAL. NTSC is the format used in Japan and North America, and PAL is used in Europe. With the NTSC format, one field is rendered at 60 Hz (that is, one second = 60 fields). With PAL, one field is rendered at 50 Hz.

3.2 EURGB60 Video Format

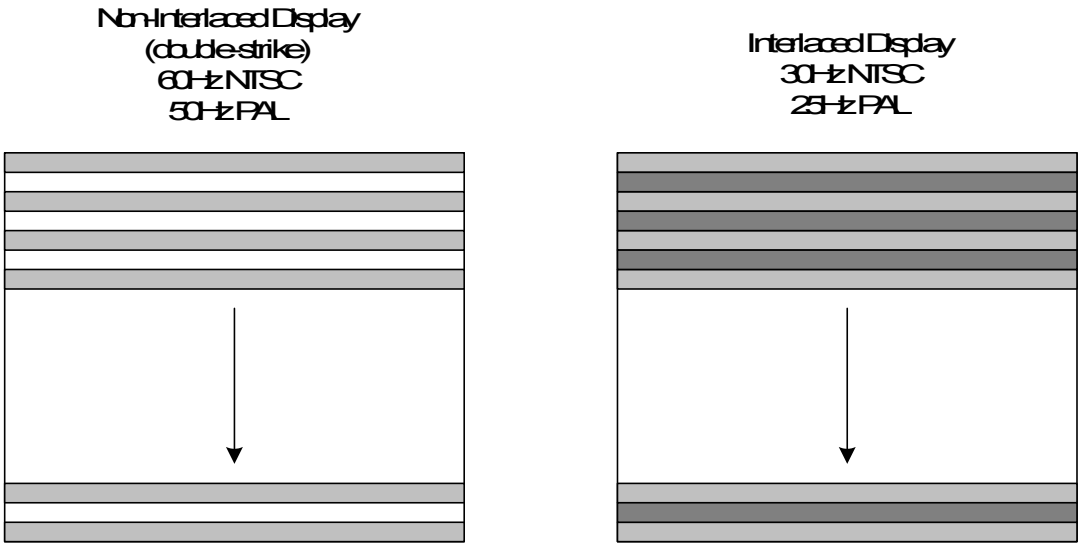
Nintendo Wii supports an additional video format (“EURGB60”) for the European market. This format supports the 60-Hz 480-line signal, so you can easily convert your NTSC games to this mode when you are preparing European versions. This special format can be seen on PAL TVs that support a 60-Hz signal. This kind of TV is now becoming very popular.

EURGB60 works with all AV cables.

3.3 Strong Support for Interlaced Display

The Wii displays high-resolution images for what is now the most common current TV format. For this reason, the VI library strongly supports interlaced display as done with the Nintendo GameCube. The traditional non-interlaced (double-strike) display commonly used in systems up to the Nintendo 64 is also supported.

Figure 3–1 Video Display Modes



As shown in the figure above, the TV scans only the even (gray) lines in double-strike mode; this results in slight, but noticeable, black lines between each line of the picture. By contrast, the TV scans both even (light gray) and odd (dark gray) lines in interlaced mode. This creates more area in which to see a higher resolution picture.

3.4 Arbitrary Positioning and Sizing of Image on the TV Screen

You can change the position and size of the image on-screen very easily.

3.5 Support for Light Gun

The Nintendo Wii system hardware supports a light gun (“Zapper”) feature.

Note: Light gun hardware is not yet available.

3.6 Horizontal Scaling to Maximize Frame Buffer Coverage on the TV Screen

VI can stretch the image in the frame buffer horizontally (but cannot be scaled down). This means that you can have a narrower image in the frame buffer in order to save main memory. However, using this feature has a disadvantage: The image shown on the screen might not be clear because of the effects of scaling. Decide carefully whether to use horizontal scaling by checking the resulting image on the TV screen.

3.7 4:4:4 YCbCr Pixel Format to Reduce Data Size in Frame Buffer

Televisions receive composite input in YUV format. 4:4:4 YCbCr is a YUV format modified so that every component fits as unsigned 8-bit data. We are using 4:2:2 YCbCr format in XFB, which is down-sampled from 4:4:4 YCbCr so that Cb and Cr are only sampled once in 2 horizontal pixels. Having the frame buffer use this pixel format is efficient because each pixel is described by 2 bytes (by contrast, RGB consumes 3 bytes for each pixel). The GX hardware performs RGB–to–YCbCr conversion when it copies the image from EFB to XFB.

Details about the 4:2:2 YCbCr format can be found under “Video Output” in the *Graphics Library (GX)* manual.

4 Basic Programming

In this chapter, we explain how to use the basic features of the VI library. The following basic functions are introduced.

Table 4–1 VI Basic Functions

Function	Purpose
`VIInit`	Initialize (detailed in section 4.2.1 VIInit).
`VIConfigure`	Configure video and frame buffer (detailed in section 4.2.3 VIConfigure).
`VISetNextFrameBuffer`	Set frame buffer address shown next (detailed in section 4.4 Setting the Frame Buffer Mode).
`VISetBlack`	Black out screen (detailed in section 4.5 Blacking Out the Screen).
`VIFlush`	Notify VI to update registers on next field (detailed in section 4.3 Flushing).
`VIWaitForRetrace`	Wait for the next retrace interrupt (detailed in section 4.6 Waiting for the Next Retrace).

Note: Do not forget to call `VIFlush` after you call `VIConfigure`, `VISetNextFrameBuffer`, `VISetBlack`, `VISetTrapFilter`, `VISetGamma`, or `VIConfigurePan` (to be explained in section 7.4 Pan). Otherwise, the changes you made by calling these functions will not take effect. See section 4.3 Flushing for details.

4.1 Code Sample: color.c

The following line-numbered excerpt is from a demo program called `color.c`, which paints the entire screen with a single color and then changes the color once per second. It displays in 640x480 interlaced mode and renders the whole screen at once, which means it prepares 480 lines for each frame buffer and renders all 480 lines at one time (known as double-field frame buffer mode; see section 4.2.4 Types of Frame Buffers for details).

Code 4–1 color.c

```
1       void main(void)
2       {
3           u32         frame;
4           u32         code;
5           u32         fbSize;
6           u8*         xfb;
7           u32         first;
8
9           OSInit();
10          VIInit();
11
12          // Calculate frame buffer size.
13          // NOTE: Each line width should be a multiple of 16.
14          fbSize = (u32)(VIPadFrameBufferWidth(rmode->fbWidth)
15              * rmode->xfbHeight * VI_DISPLAY_PIX_SZ);
16
17          allocateFB(fbSize);
18
19          VIConfigure(rmode);
```

```
20
21          // Need to "flush" so that the VI changes made so far take effect
22          // beginning with the following field.
23          VIFlush();
24          VIWaitForRetrace();
25
26          // Since the TV mode is interlace after VIInit,
27          // we need to wait for one more frame to make sure
28          // that the mode is switched from interlace to non-interlace
29      #ifdef NON_INTERLACE
30          VIWaitForRetrace();
31      #endif
32
33          first = 1;
34          frame = 0;
35          code = 0;
36
37          while(1)
38          {
39              xfb = (frame & 0x1)? xfb2 : xfb1;
40
41              if (frame % 60 == 0)
42              {
43                  code = (code + 1) % 8;
44              }
45
46              fillColor(code, fbSize, xfb);
47
48              VISetNextFrameBuffer((void*)xfb);
49
50              if (first == 1)
51              {
52                  VISetBlack(FALSE);
53                  first = 0;
54              }
55
56              VIFlush();
57              VIWaitForRetrace();
58
59              frame++;
60          }
61      }
```

Here is an explanation of the algorithm.

Lines 14-17: Allocates memory for frame buffers. This sample uses two double-field frame buffers (see section 4.2.4 Types of Frame Buffers for details). In this sample, `rmode->fbWidth` and `rmode->xfbHeight` contain width and height data, respectively. `VI_DISPLAY_PIX_SZ` shows how many bytes are needed for one pixel and is defined as 2.

Note: The start address of each line must be aligned so that each width is a multiple of 16. `VIPadFrameBufferWidth` is a useful macro to get the smallest multiple of 16 that is greater than or equal to the argument.

Line 19: Configures the size of the frame buffer, its mode (double-field or single-field), where on the TV screen to show the image, the TV video format (NTSC, PAL, or EURGB60), display mode (interlaced or non-interlaced), and so on. See section 4.2.3 VIConfigure for details on how to use the `VIConfigure` function.

Line 23: Notifies the VI device driver to implement the changes from the next field. `VIFlush` must be called when you use any of the following functions:

- `VIConfigure`
- `VISetNextFrameBuffer`
- `VISetBlack`
- `VISetTrapFilter`
- `VISetGamma`
- `VIConfigurePan`

Line 24: Waits for the next field.

Line 30: This code runs in interlaced mode by default, so this line is not usually compiled. However, when the code is compiled to run in non-interlaced mode (with the definition `NON_INTERLACE`), this line commands the program to wait at least two fields when switched between interlaced and non-interlaced modes.

4.2 Initialization

Follow these steps to initialize the Video Interface library:

1. Run `VIInit` before using any of the VI functions.
2. Allocate the appropriate number of frame buffers in main memory (for example, two for a double-buffer display).
3. Call `VIConfigure` to set the desired video mode, position the frame buffer, and so on.

4.2.1 VIInit

Before calling any VI functions, call `VIInit` to initialize the VI hardware and library.

Code 4–2 VIInit

```
#include<revolution/vi.h>

void VIInit(void);
```

Note: Do not assume that a field will start immediately upon calling this function or that a field will always start on an even or an odd line.

4.2.2 Frame Buffer Allocation

The frame buffer must be allocated from main memory, and it must follow the following alignment rules.

- Each pixel is two (2) bytes and encoded in YCbCr format.
- The start of the frame buffer must be 32-byte aligned.
- The width of each frame buffer must be a multiple of 32 bytes. (Although it is possible to display frame buffers with widths that do not align perfectly to 32 bytes, the buffers must then be padded to 32 bytes.)

In addition, the `VIPadFrameBufferWidth` function can be useful in creating properly padded frame buffers.

4.2.3 VIConfigure

`VIConfigure` sets various configurations, such as:

- TV video format (NTSC, PAL, EURGB60)
- Display mode (interlaced or non-interlaced)
- Image size on-screen (*viWidth*, *viHeight*)
- Image size on frame buffer (*fbWidth*, *xfbHeight*)
- Image position on-screen (*viXOrigin*, *viYOrigin*)
- Frame buffer mode (double-field or single-field; see section 4.2.4 Types of Frame Buffers for details)

Code 4–3 VIConfigure

```
#include<revolution/vi.h>
#include<revolution/gx.h>

void VIConfigure(GXRenderModeObj* rm);
```

You can customize your configuration as you like. However, we encourage you to get the working prototypes first by using the defaults predefined in the `GX[Ntsc|Pal|Eurgb60]XXXX` global variables. These variables are externally defined in `/RVL_SDK/include/revolution/gx/GXFrameBuffer.h`.

Here are some of the variables.

Table 4–2 Default Variable Settings

Variable	Characteristics
GXNtsc240Ds	NTSC Non-interlaced (double-strike) Image size on frame buffer: 640x240 Image size on-screen: 640x480 Image position centered on-screen Single-field frame buffer mode
GXNtsc240Int	NTSC Interlaced Image size on frame buffer: 640x240 Image size on-screen: 640x480 Image position centered on-screen Single-field frame buffer mode
GXNtsc480Int	NTSC Interlaced Image size on frame buffer: 640x480 Image size on-screen: 640x480 Image position centered on-screen Double-field frame buffer mode

See “Render Modes” in the online *Revolution Function Reference Manual* for more details.

4.2.4 Types of Frame Buffers

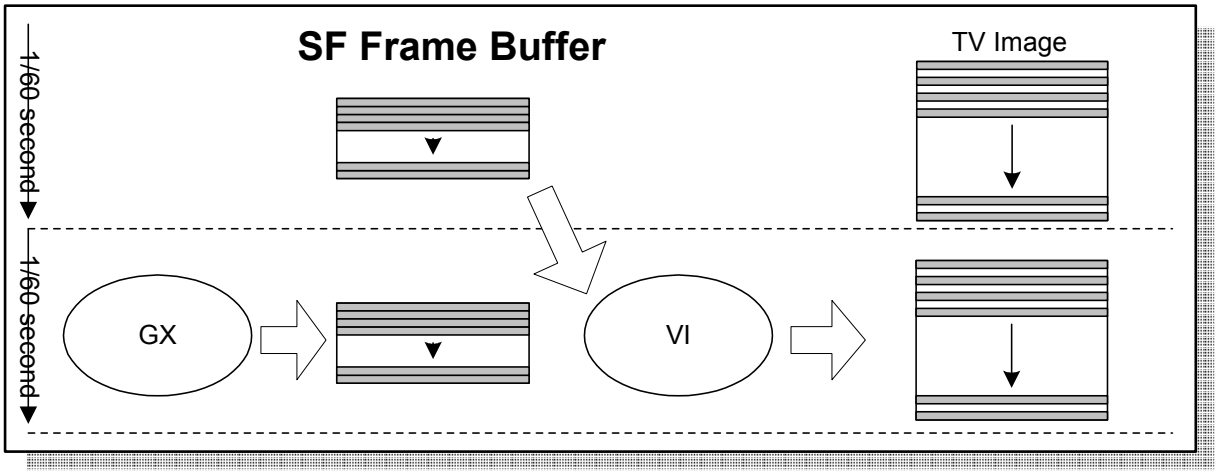
The GX library can generate two types of frame buffers: double-field (DF) and single-field (SF). DF frame buffers contain both even and odd lines that can be displayed on a TV. SF frame buffers contain only even *or* odd lines.

The diagrams in Figure 4–1 show several ways in which GX can pass the frame buffer(s) to VI to produce an image on an NTSC TV. The suitability of a given method depends on the specifications of the game. Typically, fill-rate performance and high-resolution definition are the criteria for selecting a particular method.

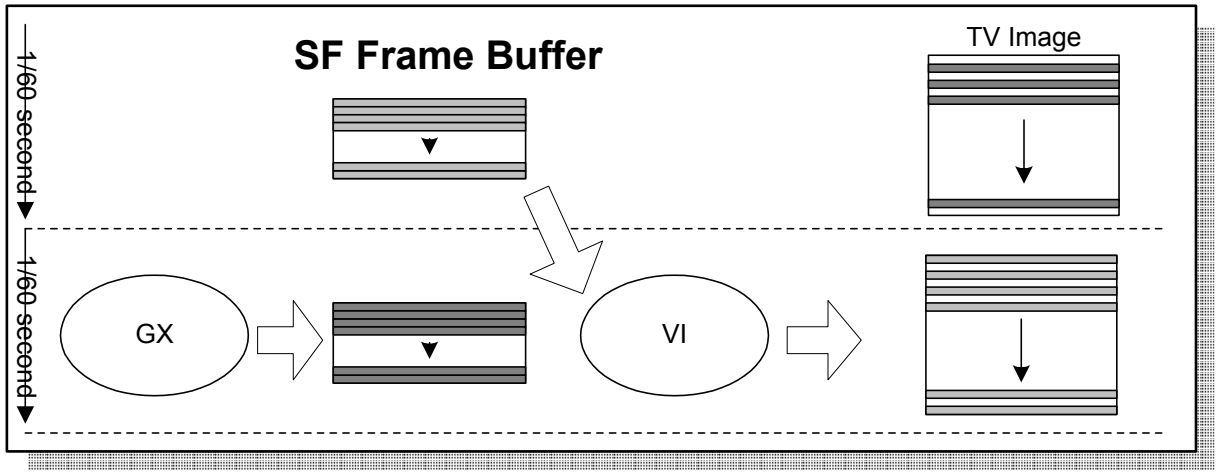
The diagrams, from top to bottom, correspond to *GXNtsc240Ds*, *GXNtsc240Int*, and *GXNtsc480Int*, respectively. For example, if you want to make your game use the method indicated in the first (top) diagram, pass the variable *GXNtsc240Ds* to `VIConfigure`.

Figure 4–1 Frame Buffer Types

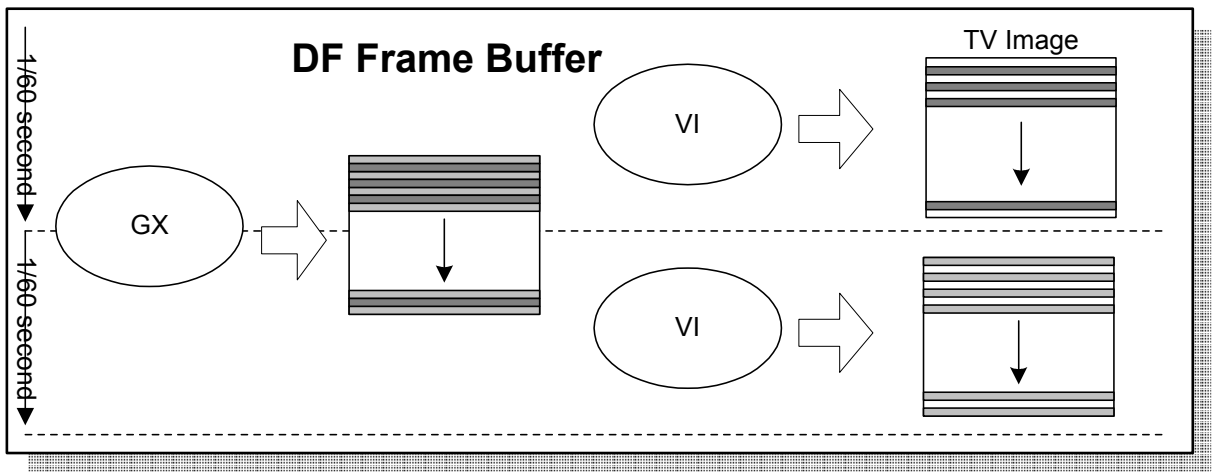
GX renders 640x240 at 60Hz, VI switches buffer at 60Hz and displays in double-strike mode.



GX renders 640x240 at 60Hz, VI switches buffer at 60Hz and displays in interlaced mode.



GX renders 640x480 at 30Hz, VI switches buffer at 60Hz and displays in interlaced mode.



4.3 Flushing

`VIFlush` notifies the VI device driver to implement changes, starting from the next field. It must be called when you call `VIConfigure`, `VISetNextFrameBuffer`, `VISetBlack`, `VISetTrapFilter`, `VISetGamma`, or `VIConfigurePan`.

Code 4–4 VIFlush

```
#include<revolution/vi.h>

void VIFlush(void);
```

Note: If you switch the TV mode between interlaced and non-interlaced (double-strike) mode or between interlaced and progressive mode, you must wait for at least two fields after calling `VIFlush` to ensure that the hardware has changed to the appropriate mode.

4.4 Setting the Frame Buffer Mode

Code 4–5 VISetNextFrameBuffer

```
#include<revolution/vi.h>

void VISetNextFrameBuffer(void* fb);
```

If you are using double- or triple-frame buffers, you must call `VISetNextFrameBuffer` to swap frame buffers. Once this function is called, the following conditions remain in effect until the function is called again.

- If the frame buffer mode is single-field, the same image are used for the subsequent fields.
- If the frame buffer mode is double-field, the appropriate lines of the image are used for the subsequent fields; for example, if the field is supposed to show odd lines, odd lines of the frame buffer are used for the field.

4.5 Blacking Out the Screen

You can use `VISetBlack` to black out or show the screen.

Code 4–6 VISetBlack

```
#include<revolution/vi.h>

void VISetBlack(BOOL black);
```

If you call this function with the argument set to `TRUE`, as in `VISetBlack(TRUE)`, the screen will turn black. You can cancel this “black mode” by calling `VISetBlack(FALSE)`.

Note: As with other functions, the screen turns black at the next field after you’ve called `VIFlush`. In addition, the screen is always in “black mode” after a call to `VIInit`, so you must call `VISetBlack(FALSE)` at least once after you’ve prepared the first image on the frame buffer.

4.6 Waiting for the Next Retrace

`VIWaitForRetrace` is an easy way to synchronize your code with the VI hardware.

Code 4–7 VIWaitForRetrace

```
#include<revolution/vi.h>

void VIWaitForRetrace(void);
```

If you have registered a background job, the operating system switches the context to start or resume the job after you call `VIWaitForRetrace`. See “Operating System (OS)” in the *Revolution Programmer’s Guide* for more information about background jobs.

5 Progressive Display

Interlace scanning is a display technique in which a 1/30-second frame is split into two, so only the odd number scan lines are displayed for the first 1/60 second, and the even number scan lines for the next 1/60 second. Progressive scanning, on the other hand, displays all scan lines every 1/60 second.

Wii supports progressive display (480 effective scan lines, D2 jack specification). Progressive format can be displayed only on TVs with progressive support. D2 terminal cables or component cables are required to connect with these TVs.

Support for progressive output is recommended for Wii game applications. Each game title should make the decision whether to support progressive output.

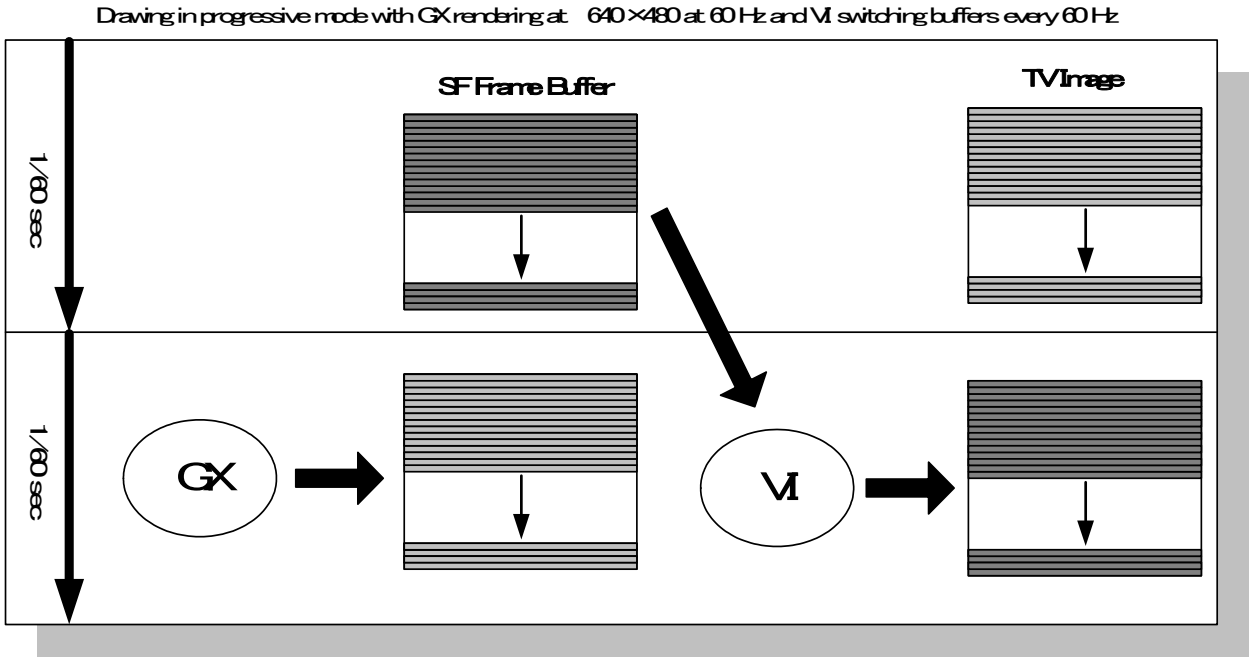
5.1 Methods for Progressive Display

To switch from interlaced to progressive display:

1. Render the progressive parts to XFB.
2. Change the render mode specified with `VIConfigure` to progressive.

Regarding step 1, there are cases in which the progressive part is already rendered to XFB during standard interlaced display unless a special interlaced display method such as field rendering is used. (Confirm how it is actually configured in your application.) In such cases, simply doing step 2 will enable progressive display.

Figure 5–1 Frame Buffer Type in Progressive Mode



The following render modes are available for progressive display. Note that for PAL, only EURGB60 progressive can be selected. (Details about EURGB60 can be found in section 6.5 EURGB60 Mode.)

While progressive render mode can be created independently, selecting one of these available render modes for `VIConfigure` will allow you to take care of step 2 easily. Be sure to call `VIFlush` once after `VIConfigure` and to call `VIWaitForRetrace` twice afterwards.

Code 5–1 Progressive Render Mode

```
GXNtsc480Prog
GXNtsc480ProgSoft
GXNtsc480ProgAa
GXEurgb60Hz480Prog
GXEurgb60Hz480ProgSoft
GXEurgb60Hz480ProgAa
```

Interlaced and progressive display sample demos are found in the locations shown below. These two demos will perform interlaced and progressive display of a similar image display. Compare the two demos: Tthe only difference you should see is the render mode configuration and the XFB render area.

Code 5–2 Sample Demos of Interlaced Display and Progressive Display

```
/RVL_SDK/build/demos/videmo/src/moving.c              //Interlace demo
/RVL_SDK/build/demos/videmo/src/moving.progressive.c  //Progressive demo
```

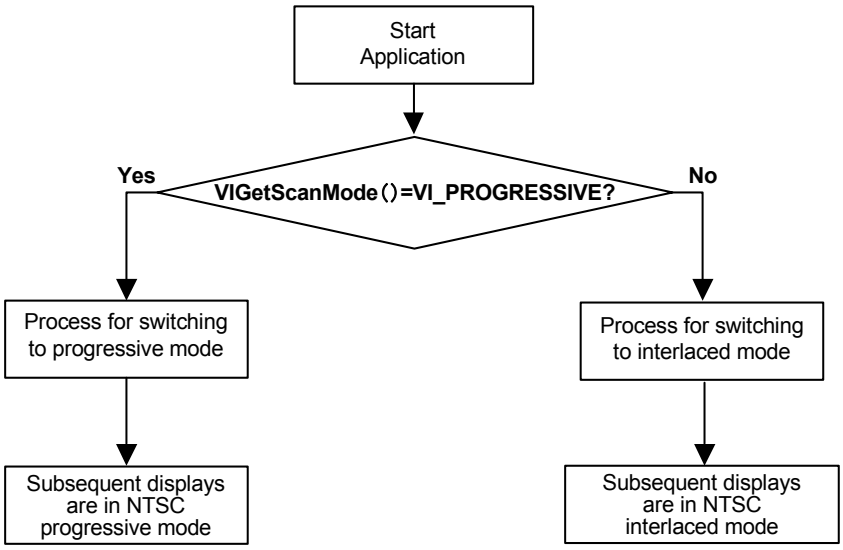
5.2 Procedure for Switching Video Modes in NTSC

This section describes the video mode setting to be performed when a game application that supports progressive mode starts up (immediately after being launched from the console setting menu).

The procedure for switching video modes when using NTSC is described first. The procedure to use with PAL will be described later. (PAL uses a switching procedure that includes the EURGB60 mode.)

When a game application supports both interlaced and progressive modes, the checking procedure shown in Figure 5–1 must be performed at the beginning of the game application.

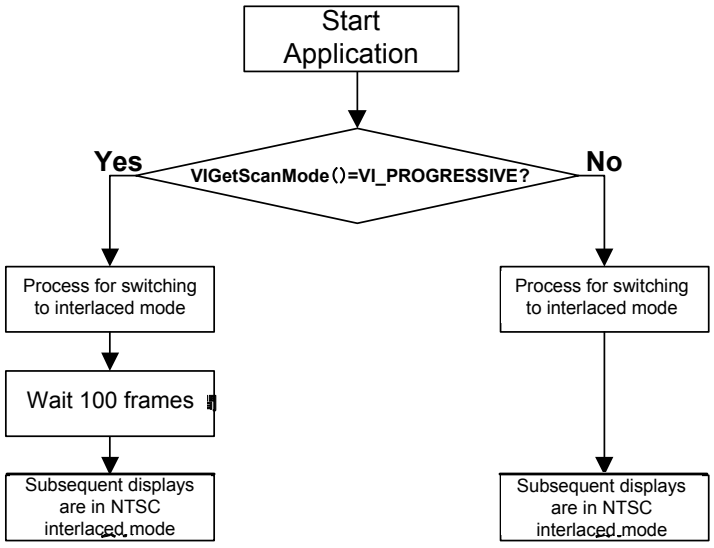
Figure 5–1 Switching Video Modes for Software That Supports Both Interlaced and Progressive Displays



First, get the current scan mode using `VIGetScanMode`. If the scan mode is interlaced, the game application will switch its video mode to interlaced mode and perform on-screen display. Likewise, if the scan mode is progressive, the game application will switch its own video mode to progressive mode and perform on-screen display.

If a game application only supports interlaced mode, be absolutely sure to perform a check at the beginning of the game application, as shown in Figure 5–1.

Figure 5–1 Switching the Video Mode for Software That Only Supports Interlaced Modes



First, get the current scan mode using `VIGetScanMode`. If the scan mode is interlaced, the game application will switch its video mode to interlaced mode and perform on-screen display. If the scan mode is progressive, the game application will switch its video mode to interlaced mode, wait 100 frames, and perform on-screen display.

5.3 Requirements for Progressive Display

- Verify that there are no VI interrupt delays during the switch from interlaced to progressive display format. A delay in the VI interrupt may prevent proper switching to progressive mode. The `VISetVSyncTimingTest` and `VIGetVSyncTimingTest` functions are used to check for VI interrupt delays. The VI interrupt can be easily delayed by interrupt disable functions such as `OSDisableInterrupts` or by `OSReport` within the `VISetPreRetraceCallback` function.
- Include the instructions for switching to progressive mode in the manual for application software that supports progressive mode. Some TV models may fix the display to the 16:9 format when a progressive signal is input. Please include this point as a caution in the software manual.

5.4 Recommendations for Progressive Display

5.4.1 Switching Procedure

The following procedure is recommended for switching to progressive display.

Code 5–3 Progressive Display Switch Code

```
#ifdef _DEBUG
// Set VISetVSyncTimingTest when the render mode is Interlace.
VISetVSyncTimingTest();
#endif

VIWaitForRetrace();
VIConfigure(&my_progressive_rendermode);
VIFlush();
VIWaitForRetrace();
VIWaitForRetrace();

// Call the function "VIGetVSyncTimingTest" after Progressive mode.
ASSERTMSG( VIGetVSyncTimingTest() == 0,"Warning: VI retrace interrupt is behind!\n");
```

5.4.2 Switch Completion Time

The TV screen will become distorted the moment the switch takes place from interlaced to progressive display or from progressive to interlaced display. The time necessary for the image to return to a normal state (henceforth referred to as “switch completion time”) will vary between TV models, but it generally takes a few dozen frames. Do not begin outputting any important messages or scenes immediately after the switch because the switch completion time cannot be guaranteed.

Table 5–1 Actual Measurement Values for Market Sample of Progressive Supported TV

Television Model	Date of Production	Time Required for Switch
36" widescreen (Company A)	1998/07/12	Approx. 80/60 sec. or 140/60 sec.
28" widescreen (Company A)	2000/07/10	Approx. 40/60 sec.
36" widescreen (Company B)	2000/07/10	Approx. 90/60 sec.

To minimize the screen distortion resulting from the switch, we recommend that you perform the following steps when switching video modes: black out the screen, switch the display format, wait 100 frames, and then display an image that does not contain an important scene or message.

5.4.3 Switch Timing for Progressive Display

We recommend that the progressive mode switch operation be performed only at initial game startup and not at any other time.

Some TVs may force the display mode to the 16:9 format when a progressive signal is input. If the game is displaying progressive in 4:3 mode, the player will need to switch the display mode to 4:3 every time there is a switch from interlaced to progressive. To avoid such inconvenience, we recommend that the switch operation be limited to initial game startup and not occur at any other time.

If there is a reset operation during progressive display, you can start again from the progressive switch operation, or skip the process altogether and simply start in the progressive display. See the *Wii Programming Guidelines* to learn more about reset operations.

5.5 Progressive Supported Television

Some TVs that support progressive mode may perform a scan line interpolation before displaying the input video signals (interlaced, non-interlaced, low-resolution non-interlaced). Accordingly, software developers should take into account the following.

- When a low-resolution, non-interlaced image (double-strike mode) is displayed on a TV that supports progressive mode, two of each line will typically be displayed, causing jaggies to stand out and look unnatural compared to TVs without progressive support. Similar phenomena are seen when displaying low-resolution, non-interlaced images from Nintendo 64 and other game machines on a TV that supports progressive mode. Kanji and small images that display without any problems on TVs without progressive support may not display properly on a TV that supports progressive mode. In such cases, we recommend the use of high resolution mode (interlaced, progressive).
- Some games create flash effects by inserting a white screen in every frame. When using such flash effects, the scan line interpolation from TVs that support progressive mode may interfere with the desired flash effect. In such a case, a proper flash effect may be obtained by inserting two frames at a time of white screen. If the flash period is short, one frame each may be sufficient for the desired effect. Check the effect using a TV that supports progressive mode to determine whether two frames are necessary.

6 European Game Support

6.1 Development Hardware

To develop for PAL format, the video mode needs to be set to PAL on the NDEV system menu. The NDEV system menu can be entered by pressing the Z Button of the Nintendo GameCube Controller when performing `ndrun` on the program. See the “Tool” section in the *Revolution Function Reference Manual* for details.

The same video cables as NTSC are used for stereo AV cables, D-terminal cables, and component cables. In addition, the PAL version accepts RGB cables. Note that S-terminal cables may not be used for PAL.

6.2 Difference Between NTSC and PAL

There are two big differences between NTSC and PAL: timing and the number of lines per field.

1. **Timing.** In the PAL system, only 50 fields are displayed per second, so one field is 20 milliseconds. This is about 20% longer than NTSC (16.6 ms/field). When you port your games to PAL, you might want to think about this timing issue. If you do not allow for it, your game is likely to run 20% slower than on the NTSC system.
2. **Number of lines per field.** In the NTSC system, one field contains 525 lines of which only 480 are “active”; that is, you can draw a picture only in those 480 lines (the other 45 lines are used for vertical retrace and so on). In the PAL system, there are 625 lines in one field, and 574 out of 625 are active.

Table 6–2 Differences Between NTSC and PAL Formats

	Number of fields/second	Length of one field	Number of active lines
NTSC	60	16.6 ms	480
PAL	50	20 ms	574

6.3 Solving the Timing Problem

Solving the timing problem completely depends on the games themselves; there is no general solution.

It is always a good idea when you design the NTSC version to plan in advance to make the game system scalable between 50 Hz and 60 Hz. Doing this can minimize the need for porting later.

If you are going to use 2D sprite animation, you should be aware of this problem because porting 2D sprite animation from 60 Hz to 50 Hz may be difficult.

6.4 Solving the “Extra Lines” Problem

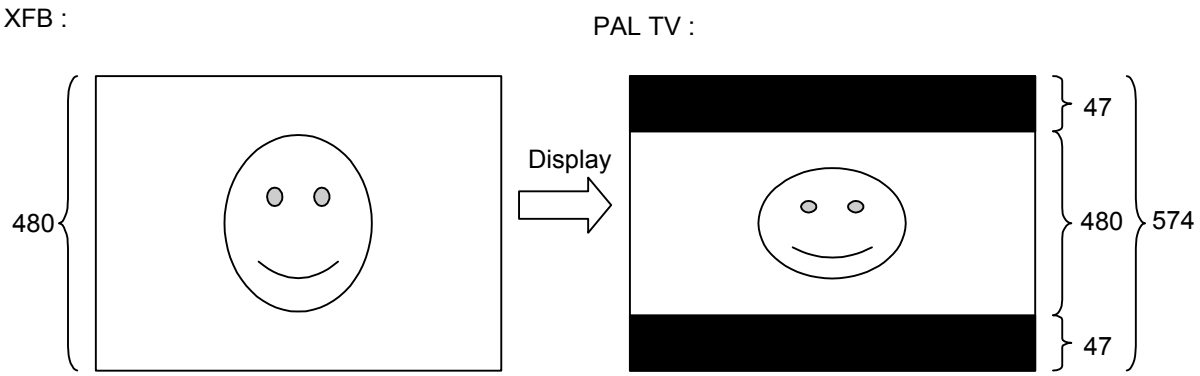
PAL games need to draw some extra lines compared to NTSC games, which can be a problem when you port your NTSC games to PAL. There are several solutions to solve this “extra lines” problem.

As stated previously, it is always a good idea when you design the NTSC version to make considerations for the PAL version as well.

6.4.1 Showing Black Bars at the Top and Bottom of the Screen (Not Recommended)

This is a very simple solution, but we don't recommend it because the resulting PAL image will look vertically compressed compared to how it looks in NTSC format. If your game draws a full 480 lines in NTSC, you must draw 94 extra lines in PAL (574 – 480 = 94). One way to make up the difference is to show 47 black lines on the top and bottom of the screen. You can do this by changing the value of *viYOrigin* in `GXRenderModeObj` to be passed to `VIConfigure`.

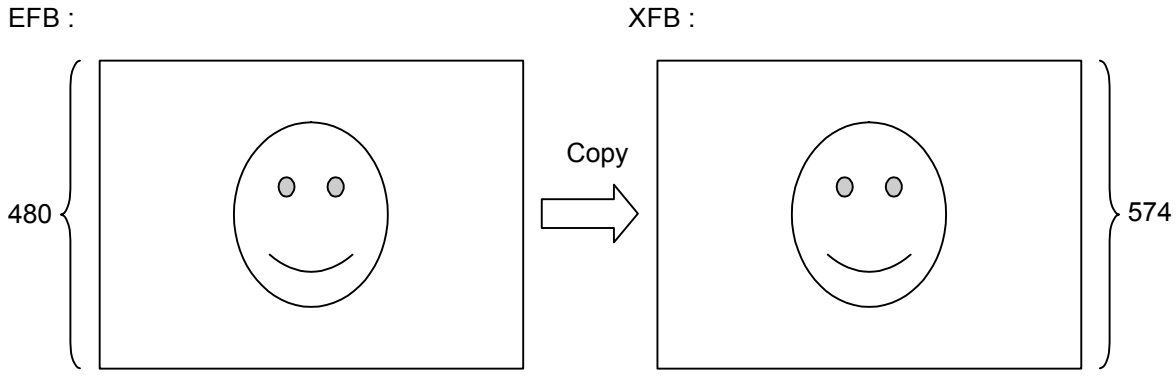
Figure 6–1 Using Black Bars at the Top and Bottom of the Screen



6.4.2 Using Y Scaling

The GX API has a feature that scales vertically when copying frame buffer data from the EFB to the XFB. You can take advantage of this feature to solve the "extra lines" problem. If your NTSC game draws 480 lines and you want to draw 574 lines for PAL, specify `GXGetYScaleFactor(480, 574)` to `GXSetDispCopyYScale` before you copy (this assumes you are not using Y-scaling for the NTSC version).

Figure 6–2 Using Y Scaling



We've modified `smp-onetri` to use Y-scaling to show off this feature; see the demo in this patch. Y-scaling is a better solution than using black bars in terms of how it displays; the image will look almost the same as the original NTSC game. The disadvantage here is that you need to prepare a bigger XFB to hold the 574 lines.

Notes:

- It is not recommended that you pass 480/574 directly to `GXSetDispCopyYScale`. See the `GXSetDispCopyYScale` and `GXGetYScaleFactor` pages in the *Revolution Function Reference Manual* (HTML) for more details.
- You cannot use this method if the CPU is drawing any part of the picture directly to XFB.

6.4.3 Rendering More Lines

Simply rendering 574 lines is another solution. The quality of the result should be better than with the Y-scaling solution, but the disadvantage is that rendering will take more time than it does on the NTSC system. (However, this may not be an issue because you have 3.3 ms more for each field.)

6.5 EURGB60 Mode

The Wii console supports EURGB60 mode, which is 60 fields/second and 525 lines (480 active lines) for European TVs using an RGB signal. We are supporting this mode now because many modern TVs sold in Europe support this timing unofficially or officially.

Note: Not all TVs used in Europe support this mode; some percentage of European consumers still use the older type of TVs.

Porting NTSC games to EURGB60 is pretty easy because the timing and resolution are the same. Change *viTVmode* in `GXRenderModeObj` from `VI_TVMODE_NTSC_*` to `VI_TVMODE_EURGB60_*`.

Supporting EURGB60 mode is not required; you can decide whether to support this mode on a per-title basis. The advantages for supporting EURGB60 mode are:

- Game players will feel more comfortable in terms of controlling characters and so forth, on 60-Hz games than on 50-Hz games.
- It is easy to make the game the same speed as the NTSC version.
- Some people may notice less flickering at 60 Hz than at 50 Hz.

Note: You still need to support standard PAL timing even if EURGB60 is supported because, as described above, not all TVs in Europe support EURGB60 timing.

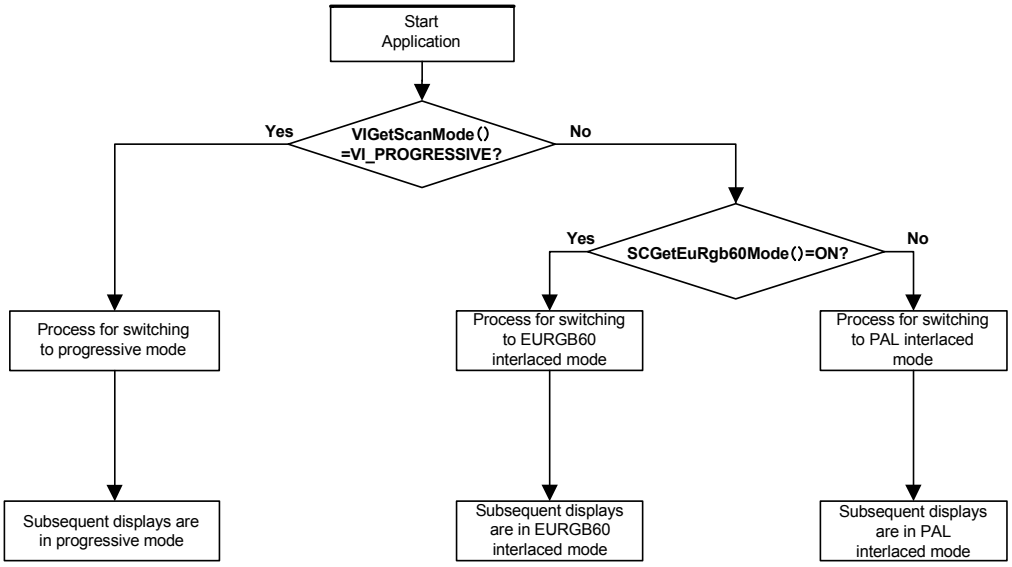
6.5.1 Switching Video Modes in PAL

This section describes the video mode setting for when a PAL game application starts up (immediately after being launched from the console setting menu).

PAL comes in three types of display format: PAL mode (conventional 50-Hz PAL), EURGB60 mode, and progressive mode (EURGB60 progressive). These three display formats will be uniquely determined by the system configuration menu settings. Note that of all the display formats, progressive has the highest priority. If the game application supports all three display formats and both EURGB60 mode and progressive mode is turned on in the system configuration menu, the application should display in progressive mode.

If a game application supports all video modes, be sure to perform a check at the beginning of the application, as shown in Figure 6–3.

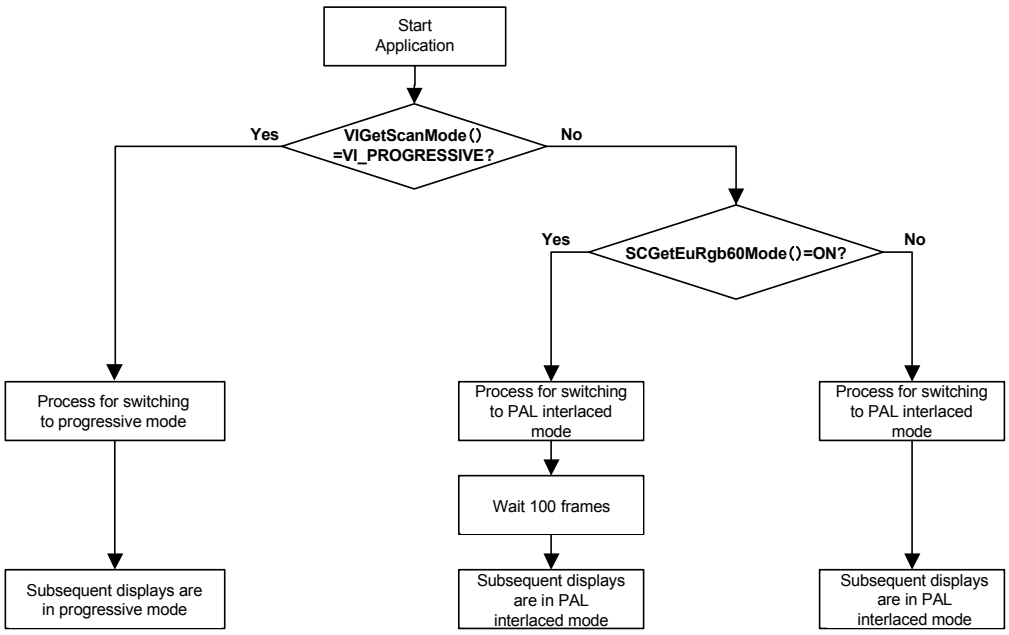
Figure 6–3 Switching Video Modes for Applications That Support All Modes (PAL)



First, the `VIGetScanMode` function will be used to get the current scan mode. If the scan mode is progressive, the application will switch its video mode to progressive mode and perform on-screen display. If the scan mode is something other than progressive, the `SCGetEuRgb60Mode` function will be used to check the current flag state. If the flag is ON, the game application will switch the video mode to EURGB60 and perform screen display. If the flag is OFF, the application will switch the video mode to PAL to perform on-screen display.

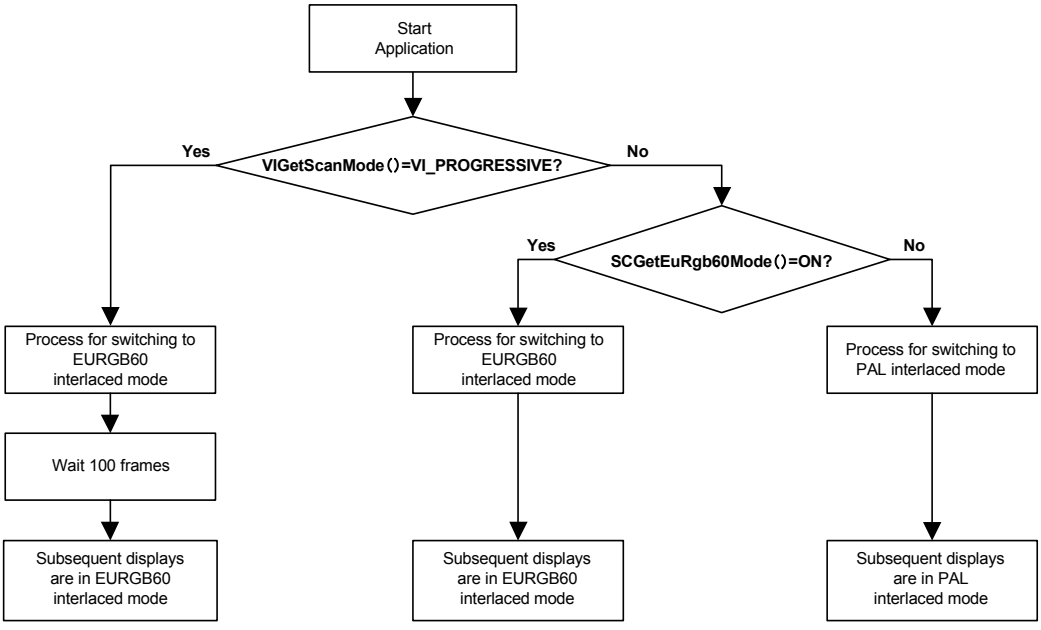
If the game application supports PAL mode and progressive mode, be absolutely sure to perform a check at the beginning of the game, as shown in Figure 6–4.

Figure 6–4 Switching Video Modes for Applications That Support PAL Mode and Progressive Mode



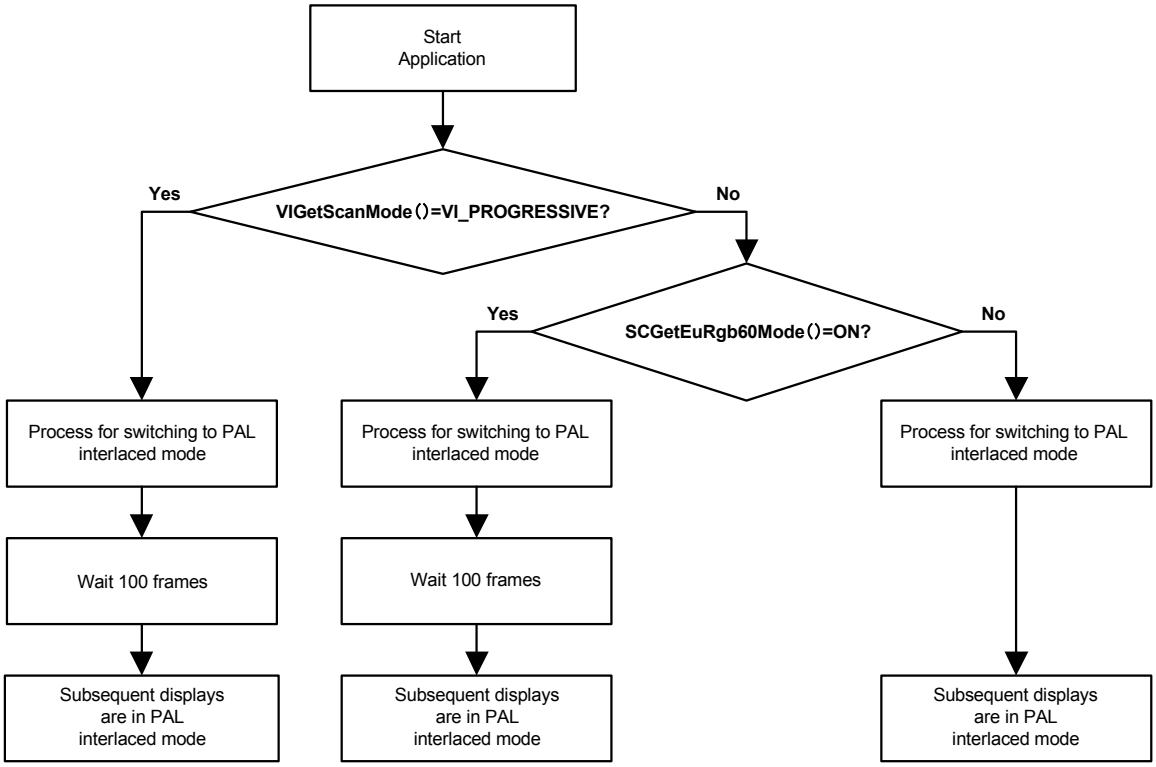
If a game application supports PAL mode and EURGB60 mode, be absolutely sure to perform a check at the beginning of the game, as shown in Figure 6–5.

Figure 6–5 Switching Video Modes for Applications That Support PAL Mode and EURGB60 Mode



If a game application supports only PAL mode, be absolutely sure to perform a check at the beginning of the game, as shown in Figure 6–6.

Figure 6–6 Switching Video Modes for Applications That Support Only PAL Mode



6.5.2 Requirements for PAL Video Mode

- **Do not use the term "EURGB60 mode" when addressing the game player. Use the term "60Hz mode" instead.** Similarly, use "50Hz mode" instead of "PAL mode." (This applies to messages displayed on the TV screen, as well as in application instruction manuals.)

6.5.3 Recommendations for PAL Video Mode

- Do not switch video modes (PAL, EURGB60, and progressive) other than at the beginning of a game application. We have encountered flickering during the switch in many TVs. When flickering occurs mid-game, the player may assume the Wii console, disc, or application is malfunctioning.
- The TV screen image will be disrupted when switching among video modes (PAL, EURGB60, and progressive). We recommend that you perform the following steps when switching video modes: Black out the screen, switch the display format, wait approximately 100 frames, and then display an image that does not contain an important scene or message.

6.6 Demos

The `onetri` demo has been modified so that you can use the A Button to switch TV modes between PAL (using Y scaling), PAL (using no scaling, with black bars), and EURGB60. You can find this demo under `videmo/src/smp-onetri_PAL.c`.

If your TV supports only NTSC, this demo will not show the proper output in PAL mode. Similarly, if your PAL TV doesn't support EURGB60, the demo won't show the proper output when you switch to NTSC.

Since this demo does not adjust the frame rate, the PAL modes will run more slowly than the EURGB60 mode.

7 Further Programming

7.1 Render Mode Customization

Although we recommend that you start by using the predefined rendering modes, we encourage you to customize your configuration once you are comfortable with the API. This chapter shows how to set custom values for `VIConfigure`.

7.1.1 Maximum Screen Space

To define the image size and position on-screen, you must first define the maximum space in which you can locate the image. This is called *maximum screen space.* The size of this space in pixels is:

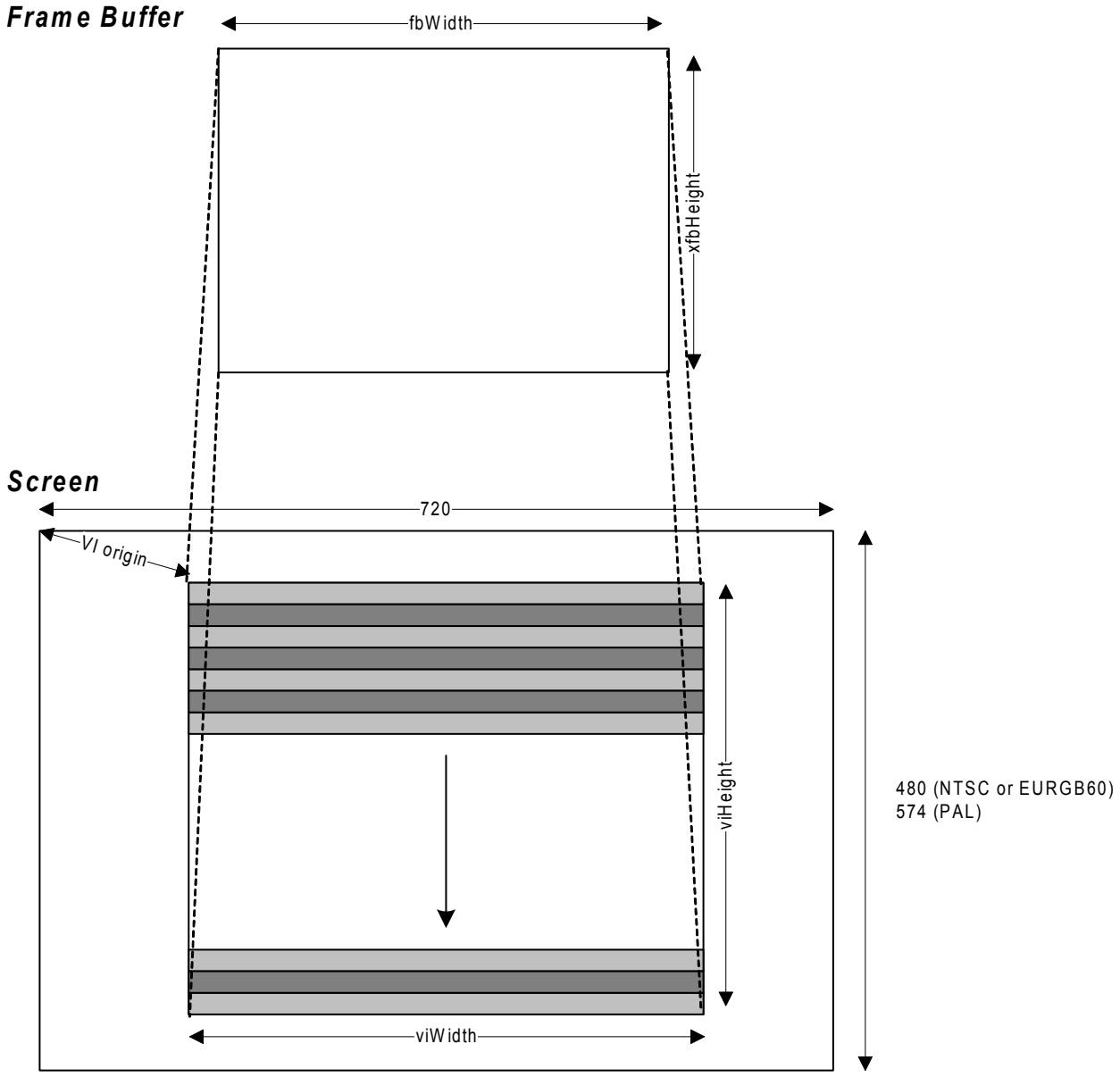
- 720 x 480 for NTSC
- 720 x 574 for PAL

You can place images anywhere in the maximum screen space by using the function `VIConfigure`.

7.1.2 Frame Buffer Size, TV Screen Size, and Position Configuration

There are six fields in the `GXRenderModeObj` structure concerned with frame buffer size, TV screen size, and position configuration. They are *fbWidth*, *xfbHeight*, *viXOrigin*, *viYOrigin*, *viWidth*, and *viHeight*.

Figure 7–1 Relationship Between the Six Values of GXRenderModeObj



`GXRenderModeObj` uses the notation *xfbHeight* to specifically differentiate this value from the embedded frame buffer height, denoted by *efbHeight*. The structure contains both values because the GX library can perform Y scaling when it copies the image from EFB to XFB. However, the VI library does not touch *efbHeight*.

If you specify *viWidth* as greater than *fbWidth*, VI performs the horizontal scaling. You cannot specify *viWidth* to be less than *fbWidth*.

Since VI cannot perform Y scaling, the following simple ratios must be maintained between *viHeight* and *xfbHeight*:

- For double-field frame-buffer mode, the value of *viHeight* should be the same as *xfbHeight*.
- For single-field frame-buffer mode, the value of *viHeight* should be twice as large as *xfbHeight*. This is true for non-interlaced (double-strike) mode. For example, if *xfbHeight* is 200, *viHeight* should be 400.

It is not strictly necessary to specify a value for *viHeight* because this can be calculated easily from the value of *xfbHeight* and the FB mode. `VIConfigure` verifies this value.

For example, if you would like to create a cinema wide-screen aspect ratio of 16:9 with a width of 640 using a single-field frame buffer, specify the following values:

- *fbWidth* = *viWidth* = 640
- *xfbHeight* = 180
- *viHeight* = 360
- *viXOrigin* = (maximum screen space width - *viWidth*)/2 = (720 – 640)/2
- *viYOrigin* = (maximum screen space height - *viHeight*)/2 = (480 – 360)/2

The DEVKIT System Menu has a feature for adjusting the horizontal position. This feature allows a range of movement that is the difference between the maximum screen space width and *viWidth*. Because *viXOrigin* represents the offset from the horizontal position configured in the System Settings menu, the actual displayed position will be shifted horizontally by only the sum of these two values.

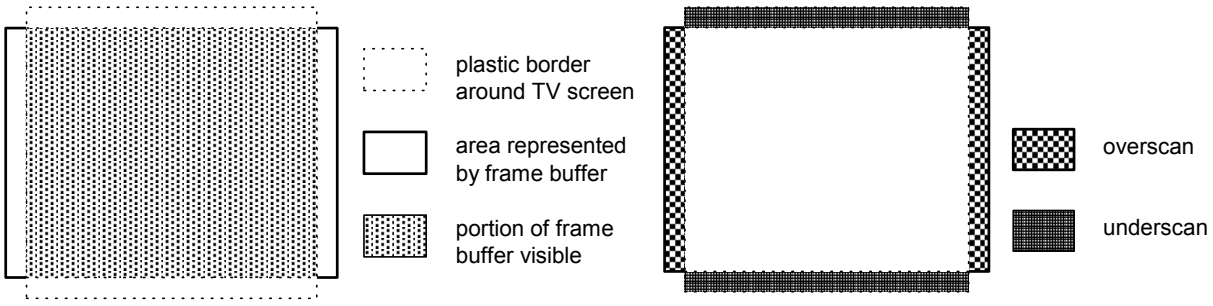
The VI library will clamp the sum of *viXOrigin* and the horizontal adjustment value configured in the System Settings menu if it is greater than the difference between the maximum screen space width and *viWidth*. For example, if *viWidth* is 680 pixels for an NTSC display, and the sum of *viXOrigin* and the horizontal adjustment value is greater than 40 pixels, the VI library will only shift the position by 40 pixels.

7.1.3 Overscanning and Underscanning

Different brands of TVs exhibit slight differences in the amount of overscanning and underscanning for a given signal. *Overscanning* means that the TV's electronic beam is "overpainting" pixels in the area behind the plastic border surrounding the TV tube. *Underscanning* means that some visible portion of the TV screen has no painted pixels. We have preselected parameters that typically create a good display on most TVs.

Figure 7–2 shows overscanning in the horizontal direction and underscanning in the vertical direction.

Figure 7–2 Overscanning and Underscanning



The combination of overscan in the horizontal direction and underscan in the vertical direction is quite common. An example of this is shown in the next section.

7.1.4 Scaling

You can scale horizontally if you specify *viWidth* as greater than *fbWidth*. Remember that you cannot specify a *viWidth* value smaller than *fbWidth*.

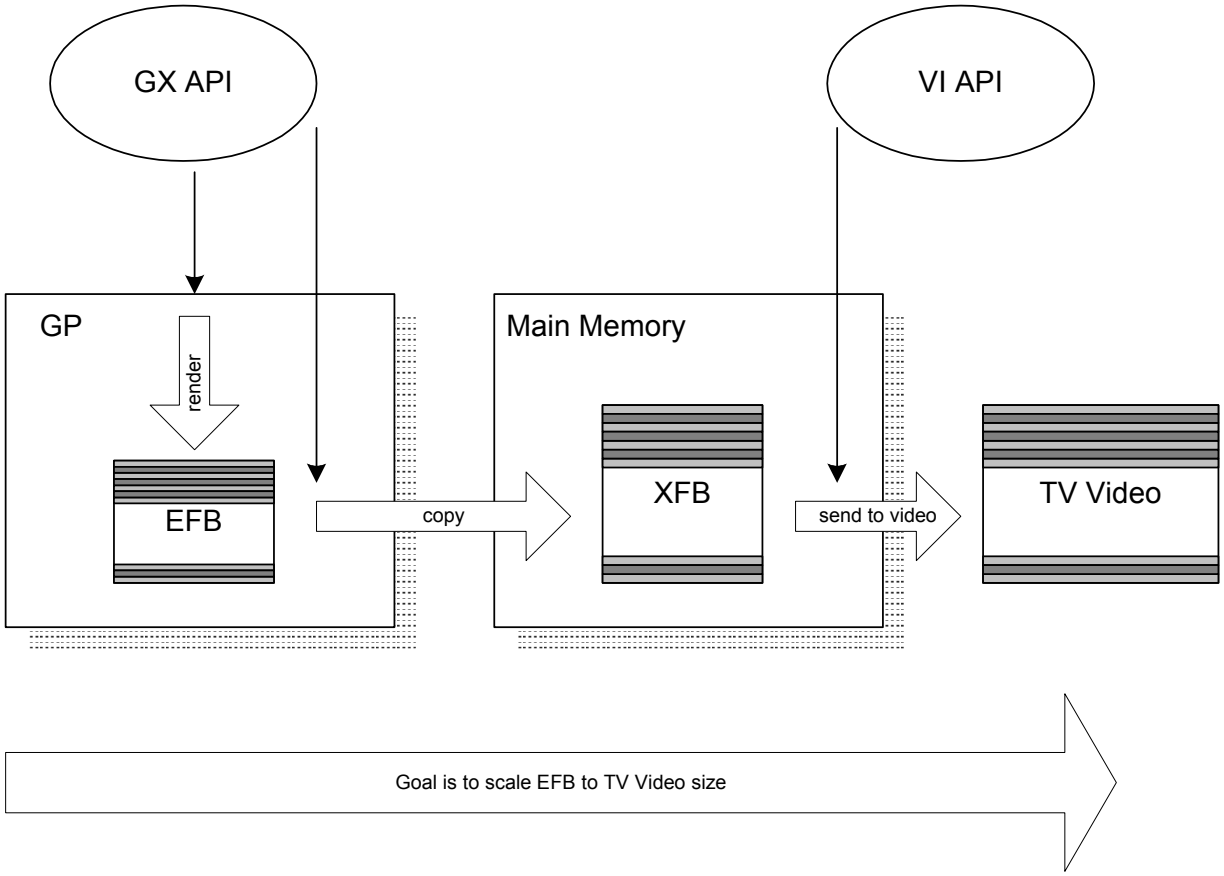
For example, if you would like to have a 512 x 240 frame buffer and scale horizontally 640/512 = 1.25, using a single-field frame buffer, specify the following values:

- *fbWidth* = 512
- *viWidth* = 640
- *xfbHeight* = 240
- *viHeight* = 480
- *viXOrigin* = (maximum screen space width - *viWidth*)/2 = (720 – 640)/2
- *viYOrigin* = (maximum screen space height - *viHeight*)/2 = (480 – 480)/2

Keep in mind that the VI library cannot scale vertically. See the Graphics Library (GX) section for information on Y scaling.

Figure 7–3 summarizes how to scale both vertically and horizontally, using the GX and VI libraries.

Figure 7–3 GX and VI Functional Flowchart



To scale EFB to the TV video size in both *x* and *y* dimensions, the hardware must do the following:

- Perform *y*-scaling while EFB is copied to XFB.
- Perform *x*-scaling while XFB is sent to the TV screen.

Therefore, the main memory buffers must have enough memory to accept a *y*-scaled frame buffer.

7.1.5 Pixel Ratio

The pixel ratio when the EFB image is displayed in the standard TV without *xy*-scaling is described in the following sections.

7.1.5.1 NTSC

When displaying to a TV of normal size (4:3), the *x:y* pixel ratio is approximately 0.9:1.0. The shape of a pixel is a vertically elongated rectangle. If you want to display in a 640x480 screen size with a 1:1 pixel ratio using 0.9:1.0 = 576:640, take the XFB pixel width of 576 and stretch it to 640 during output to the TV.

When displaying to widescreen (16:9) TV, the *x:y* pixel ratio is approximately 1.2:1.0. The shape of a pixel is a horizontally elongated rectangle. To display with a 1:1 pixel ratio, keep a portrait image in the XFB.

7.1.5.2 PAL

When displaying to a TV of normal size (4:3), the *x:y* pixel ratio is approximately 1.09:1.0. The shape of a pixel is a horizontally elongated rectangle. If you want to display in a 640 x 480 screen size with a 1:1 pixel ratio, take a 640 x 440 pixel image and stretch the vertical direction to 480 during the copy to XFB. The EU RGB60 mode is the same as NTSC. The *x:y* pixel ratio is approximately 0.9:1.0.

When displaying to a widescreen (16:9) TV, the *x:y* pixel ratio is approximately 1.45:1.0. The shape of a pixel is a horizontally elongated rectangle. The EU RGB60 mode is the same as NTSC. The *x:y* pixel ratio is approximately 1.2:1.0.

7.1.6 Safe Frame

The safe frame refers to an area that should be displayed without any problems on most TV screens and should be used when displaying important game information. Safe frame size with the Wii varies, depending on the TV model. In general, the horizontal width is approximately 84.4% of the maximum screen space; the vertical height is approximately 80.8% regardless of the display format.

The maximum screen space is:

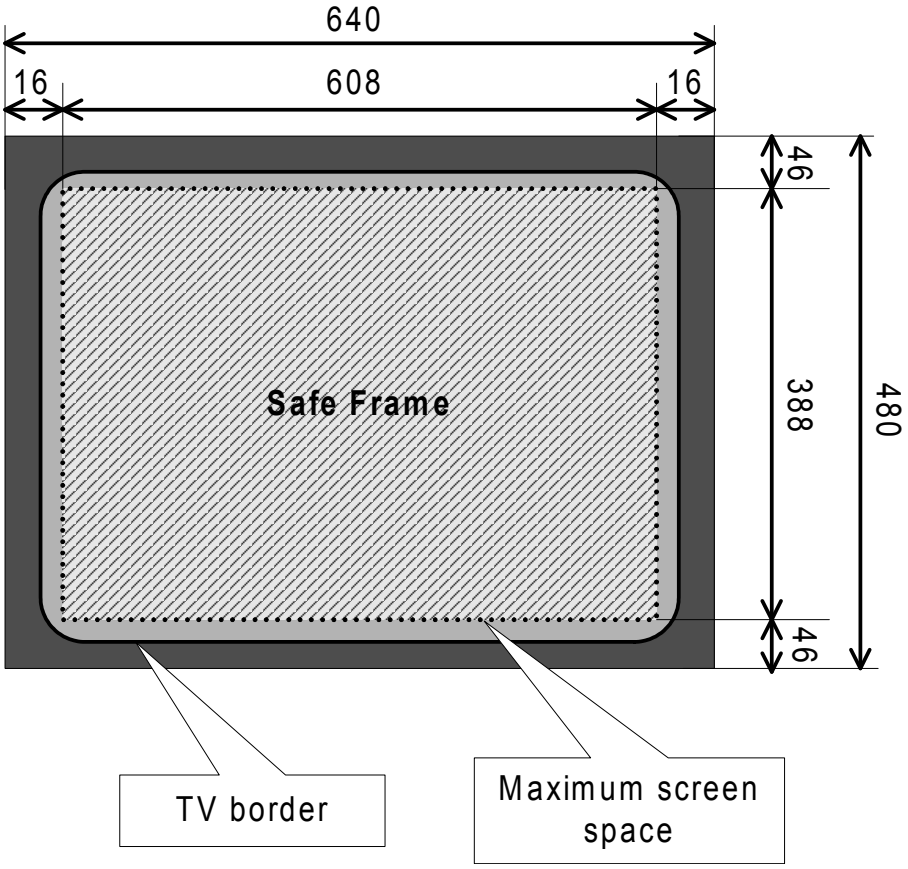
- NTSC 720 x 480
- PAL 720 x 574

The safe frame is in this area:

- NTSC 608 x 388 (approximately)
- PAL 608 x 464 (approximately)

The location of the safe frame is approximately in the center of the maximum screen space. Figure 7–4 shows an example safe frame when the TV screen size is 640 x 480 (the diagonal lines indicate the safe frame area). Apply this safe frame in 4:3 mode.

Figure 7–4 Example of a Safe Frame in a Screen of 640 x 480 Pixels

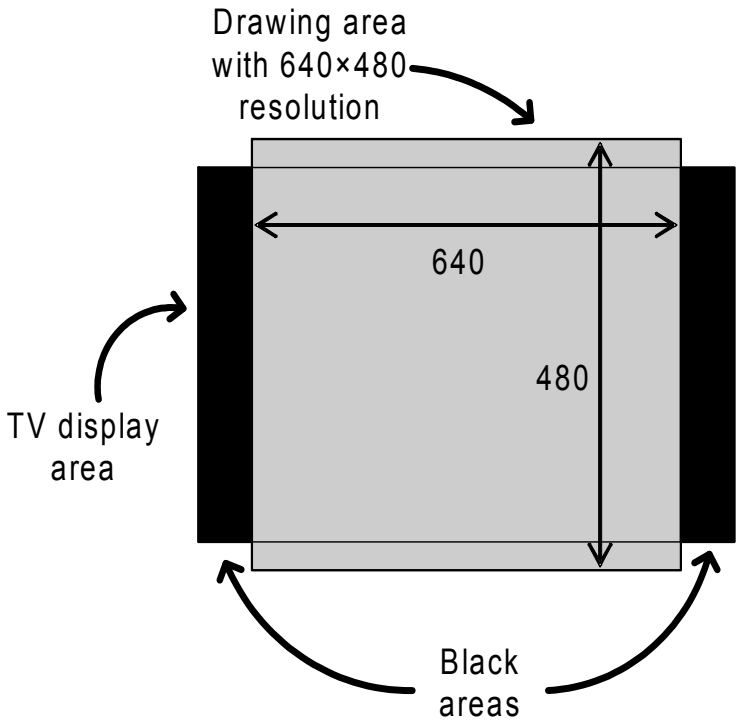


With the current transition of the broadcast format to digital terrestrial broadcasting, the widescreen TV is becoming increasingly popular. The safe frames in these widescreen TV are generally larger compared to the traditional 4:3 TV. For 16:9, this safe frame will be approximately 87% of the maximum screen space in both vertical and horizontal direction. Use this information and the importance of display information to determine the display position. (There are some older CRT widscreen TVs on which the drawing area may not fit in the 87% safe frame. In these cases, create gameplay in 4:3 mode.)

7.1.7 Black Bands at Right and Left Screen Borders

When you set the horizontal direction display pixels to 640, the non-display area in the left and right side may stand out as black bands. Such TVs show black bands on the edge because they display only 640 pixels when the horizontal display area is larger than 640 pixels. In terms of the vertical direction, there will be a few dozen pixels not displayed on the top and bottom borders for a 480-pixel display. In that case, there will be no such bands.

Figure 7–5 Black Bands at Right and Left Screen Borders



Note: The black bands are not actually as thick as shown in the figure.

We have observed a “wavy” effect in these bands on certain TVs. These wavy motions are seen on TVs that display bright pixels horizontally longer than the dark pixels. This will make the black band narrower in a line with a large number of bright pixels and wider in a line with a large number of dark pixels. If the game has lines that change brightness suddenly, the band width appears to change over time, causing the bands to look wavy.

The existence of these black bands is not a problem in itself, but a large amount of this wave motion may interfere with the gaming experience. While we do not know the exact reason, TVs that exhibit pronounced levels of this problem are seen more often in Europe.

Note: Whether or not to take measures against this problem is left to the individual developer. If measures are to be taken, the following method may help. Use it as a reference.

This effect may be reduced by stretching the horizontal display area by approximately 10 to 30 pixels, using VI's X-axis scaling feature. This narrows the width of the black band, which makes the waviness less noticeable. To be more specific, set the *viWidth* of the `GXRenderModeObj` structure passed to the `VIConfigure` function to the values described below. To position the screen at the center, you must also adjust the *viXOrigin* value at the same time.

Table 7–1 Screen Width and Safe Frame with Wavering Black Bands Taken Into Account

Horizontal Stretch Width	5 Pixels	10 Pixels	15 Pixels
viWidth	650	660	670
viXOrigin	35	30	25
Width of Black Bands	Will narrow, but still exist	Will disappear on some TVs, but may persist on others	Will disappear on most TVs
Horizontal Safe Frame	Should take into account approxi-mately 5 pixels on both sides	Should take into account approxi-mately 10 pixels on both sides	Should take into account approxi-mately 15 pixels on both sides

For a *viWidth* of 650, the bands still exist, but the width is narrowed. This reduces the waviness to a certain degree. If *viWidth* is 660, the bands disappear on some TVs, but may persist on others. If *viWidth* is set to 670, the black bands are not displayed on most TVs.

When the screen is stretched in this manner, please include the extended pixels when considering the safe frame. For example, if the screen is stretched by 5 pixels on each side, do not include any important information in those 5 pixels. If the screen is stretched by 10 pixels, the same applies to those 10 pixels.

On a related note, when the vertical direction display consists of fewer than 480 pixels, (horizontal) black bands may be displayed at the top and bottom of the screen. However, these bands will not have the wavy effect due to the design of the TV, so this is not a problem. For example, setting the top and the bottom to black to create a movie effect of a 16:9 aspect ratio will not cause any problems.

Also, by stretching the entire screen horizontally, the individual pixels are stretched horizontally by approximately 1.56% (for a stretch of 5 pixels on each side), 3.12% (10 pixels), or 4.69% (15 pixels). Use this as a reference when considering the pixel ratio.

7.1.8 Switching Render Modes

When performing the following switches using the `VIConfigure` function, always call `VIWaitForRetrace` twice. This is true for all configurations (NTSC and PAL).

- Interlaced to non-interlaced
- Interlaced to progressive
- Non-interlaced to Interlaced
- Non-interlaced to progressive
- Progressive to interlaced
- Progressive to non-interlaced

Code 7–1 Example Switch Between NTSC Interlaced and NTSC Non-interlaced Render Mode

```
rmDS  = &GXNtsc240Ds;
rmINT = &GXNtsc480Int;
ds = 1;

while(1)
{
    ..........
    if (ds)
        VIConfigure(rmDS);
    else
        VIConfigure(rmINT);

    // Wait for two fields when switching video modes
    VIFlush();
    VIWaitForRetrace();
    VIWaitForRetrace();

    ds ^= 1;
}
```

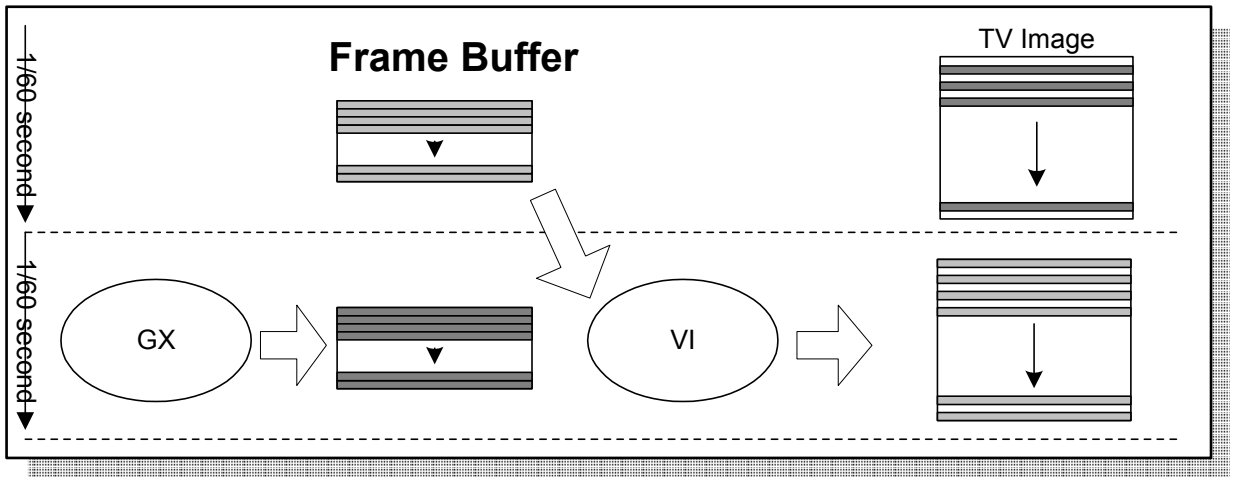
Also, the image will become distorted at the moment of these switches. To avoid this, turn the screen dark at the moment of the switch, using the `VISetBlack` function.

7.2 Field Rendering

Using VI in interlaced mode puts more information on-screen for the player to view, so it is an effective way to get a higher-definition image. By using field rendering, we can achieve this high definition in many games without increasing the fill rate. The field rendering technique requires that the application render even and odd fields alternately.

Figure 7–6 Field Rendering

GX renders 640x240 at 60Hz, VI switches buffer at 60Hz and displays in interlaced mode



If rendering takes longer than 1/60 second in NTSC or 1/50 second in PAL, it will fall behind the video display. Under such circumstances, many games can use the previously rendered field without suffering any detectable visual anomaly.

Note: The previous field will be in the wrong phase (that is, odd instead of even or even instead of odd).

As long as rendering is fast enough, there is no problem. Typically, rendering falls behind the video display due to an increase in complexity, which may be caused by special effects in the game (for example, explosions from weapons). This problem may solve itself because these special effects are sometimes accompanied by screen shakes (such as in a fighting game). Such explosions and screen shakes may easily mask any visual problems caused by out-of-phase field display. However, if your game does not have distracting special effects and the rendering still falls behind periodically, display anomalies may be noticeable.

7.2.1 Querying Next Field Type

For field rendering, use `VIGetNextField` to draw the data to the next field.

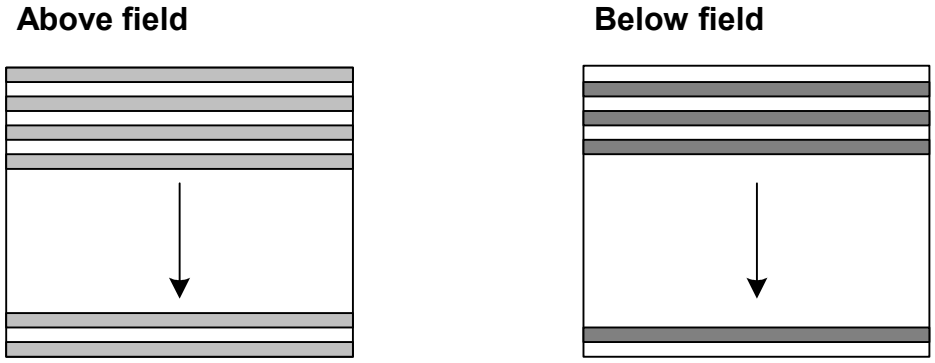
Code 7–2 VIGetNextField

```
#include<revolution/vi.h>

u32 VIGetNextField(void);
```

This function's return value is either `VI_FIELD_ABOVE` or `VI_FIELD_BELOW`. `VI_FIELD_ABOVE`, indicating that the next field contains the odd lines (for example, first line, third line, and so on) of the whole image to be drawn. `VI_FIELD_BELOW` means that the next field contains the even lines (for example, second line, fourth line, and so on) of the image to be drawn.

Figure 7–7 Above Field and Below Field



You can use the return value of `VIGetNextField` as one of the arguments of `GXSetViewportJitter` directly.

Code 7–3 GXSetViewportJitter

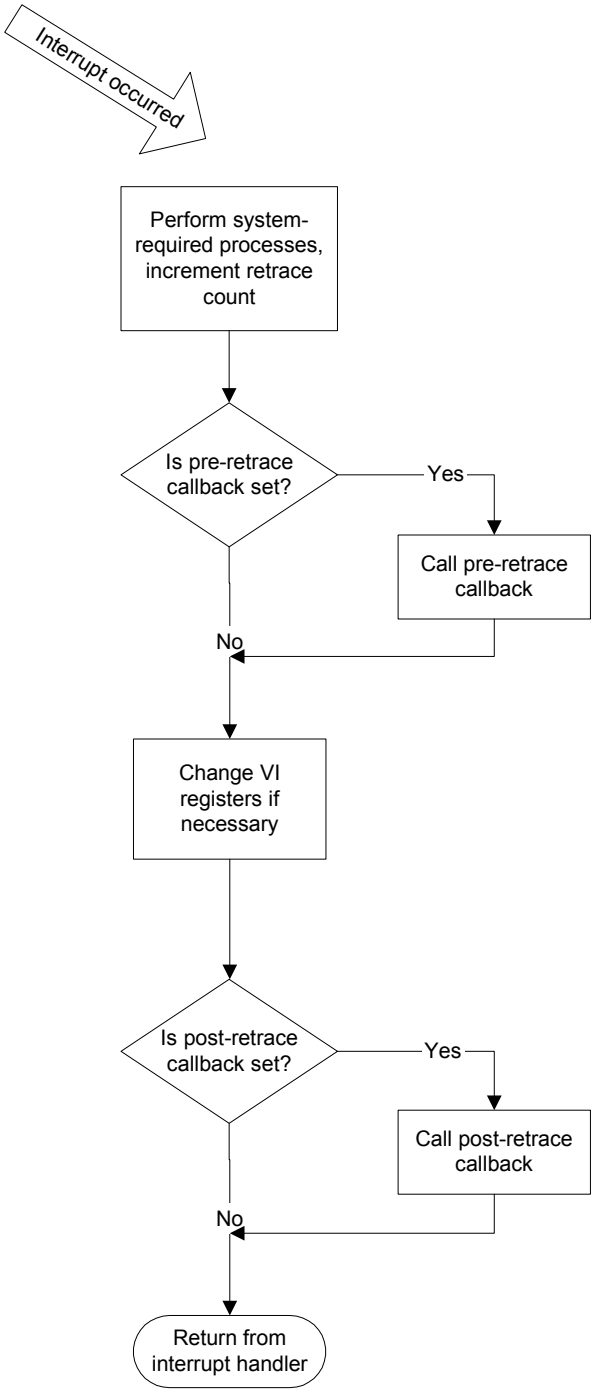
```
GXSetViewportJitter(0.0f, 0.0f, 640.0f, 240.0f, 0.0f, 1.0f, VIGetNextField());
```

7.3 Retrace Callback

You can register two types of retrace callbacks: pre-retrace and post-retrace. Pre-retrace callbacks are called before the VI retrace interrupt handler updates the VI registers. Therefore, you can use them to change the configuration, swap frame buffers, call `VIFlush`, and so on in time for the next field. Keep in mind, however, that VI registers should remain set until a certain amount of time passes after the interrupt, so no complicated tasks should be performed in a pre-retrace callback. Save the more complicated tasks for post-retrace callbacks.

Figure 7–8 shows the relationship between pre- and post-retrace callbacks.

Figure 7–8 Relationship Between Pre- and Post-Retrace Callbacks



7.3.1 Pre-Retrace Callbacks

Code 7–4 VISetPreRetraceCallback

```
#include<revolution/vi.h>

typedef void (*VIRetraceCallback)(u32 retraceCount);

void VISetPreRetraceCallback(VIRetraceCallback callback);
```

As noted earlier, the pre-retrace callback is suitable for performing VI configuration changes, swapping frame buffers, calling `VIFlush`, and so on.

The retrace count of the next field is passed to the pre-retrace callback when the latter is called.

Note: If you call `VIFlush` in a pre-retrace callback, the change will take effect with the next field.

7.3.2 Post-Retrace Callback

Code 7–5 VISetPostRetraceCallback()

```
#include<revolution/vi.h>

typedef void (*VIRetraceCallback)(u32 retraceCount);

void VISetPostRetraceCallback(VIRetraceCallback callback);
```

Post-retrace callbacks are suitable for reading Controller (PAD) information, updating audio sequences, invoking a thread, and so forth.

The retrace count of the next field is passed to the pre-retrace callback when the latter is called.

7.4 Pan

Code 7–6 VIConfigurePan

```
#include <revolution/vi.h>

void VIConfigurePan(u16 xOrg, u16 yOrg, u16 width, u16 height)
```

Pan refers to the XFB range actually displayed on the screen. Typically, there is no need to call `VIConfigurePan` function at all. The pan is set to be the same size as the XFB by default. This function is useful when you want to display only a part of the XFB.

Special care needs to be taken when calling `VIConfigure` after `VIConfigurePan`. The pan information set by `VIConfigurePan` will be reset to the default value by `VIConfigure`.

This change will not take effect until `VIFlush` is called. Refer to the `VIFlush` function reference for details regarding enabling this value.

7.5 Video Format

The Wii video format can be grouped into two formats (NTSC/PAL and EURGB60), based on the destination (Table 7–2). The Wii console for Japan, US, and Korea will output a NTSC video signal. The Wii console for Europe will output a PAL or EURGB60 video signal.

Table 7–2 Market and Supported Video Format

Market	Supported Video Format
Japan, US, Korean, Brazilian	NTSC
European	PAL, EURGB60

The video format setting is configured by specifying the render mode in the `VIConfigure` function. Table 7–3 shows the video format actually output when each format is specified on the consoles for the individual market.

Table 7–3 Relationship Between IPL Video Format and Video Format Specified in Render Mode

Market	IPL Video Format	Video Format Specified by VIConfigure	Output Video Format
Japan US Korea Brazil	NTSC	NTSC	NTSC
		PAL (Forced shutdown)	Not displayed
		EURGB60 (Forced shutdown)	Not displayed
Europe	PAL	NTSC (Forced shutdown)	Not displayed
		PAL	PAL
		EURGB60	EURGB60

If the wrong video format is specified anywhere, the program is forcibly terminated. When an application terminates, always verify that proper video output was made throughout the entire program.

7.5.1 Obtaining the TV format

Code 7–7 VIGetTvFormat

```
#include<revolution/vi.h>

u32 VIGetTvFormat(void);
```

`VIGetTvFormat` returns the TV format that a game uses. Return values are `VI_NTSC`, `VI_PAL`, or `VI_EURGB60`.

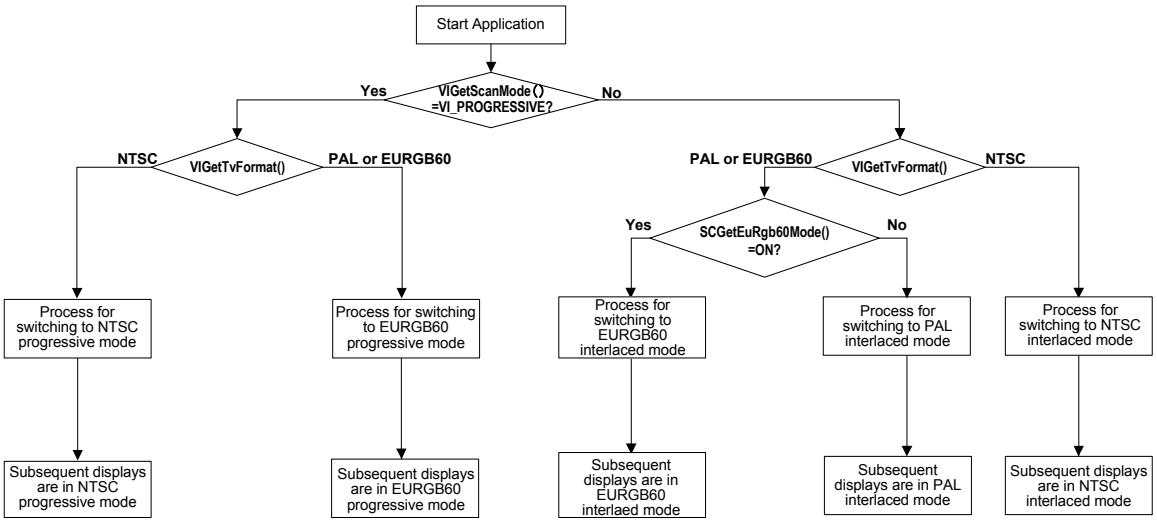
For `VIGetTvFormat` to function properly, `VIInit` must be executed beforehand. `VIInit` must be called before any other VI function. Be aware that `VIGetTvFormat` always returns `VI_NTSC` before `VIInit` is executed.

7.5.2 Automatic Switching of the Video Mode

When using a single game image to support all the video modes, `VIGetTvFormat` can be used to switch to the appropriate video mode automatically. Figure 7–9 shows the process flow used to determine the video mode performed at application startup (immediately after being launched from the console setting menu) for games that support all video modes.

Note: The flow shown below will not function properly on consoles that do not have the System Menu installed.

Figure 7–9 Automatically Switching the Video Mode

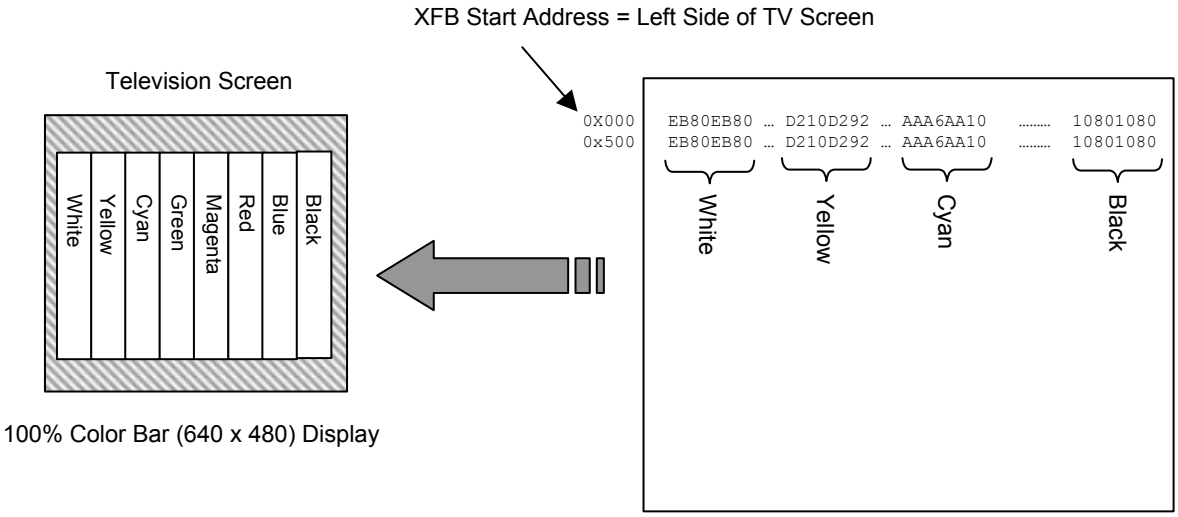


7.6 Handling YUV Data

The image data displayed on the TV (final image) will be placed in the XFB in YUV4:2:2 format (YUYV...). When writing the data to be displayed directly into the XFB, it must be written using YUV format. The image data in the XFB corresponds to 2 pixels in YUYV data sequence (4-byte size). As the YUYV data sequence is treated as a single unit, the data will be written in 2-pixel units when writing the display data directly to the XFB.

The correspondence between the TV screen and the YUV data in the XFB can be seen in Figure 7–10. (See the *Graphics Library (GX)* manual for details regarding YUV format inside the XFB.)

Figure 7–10 Correspondence Between the TV Screen and YUV Data in the XFB



The individual values of YUV are obtained using the following conversion formulas.

Y=0.257*R + 0.504*G + 0.098*B+16

U = -0.148*R-0.291*G+0.439*B+128

V = 0.439*R-0.368*G-0.071*B+128

If you actually insert the values between 0 and 255 in the individual RGB variables and calculate the YUV values, those values will not be a continuous quantity. The values that YUV can contain will be in the following range, but there will be certain value combinations that cannot actually be contained.

Y-16–235

U-16–240

V-16–240

Avoid writing an unusable YUV value combination to the XFB. For example, if you want to fill the screen with black and clear the screen, an RGB value of (R, G, B) = (0, 0, 0) will be written. This corresponds to a YUV value of (Y, U, V) = (16, 128, 128). This means writing a data sequence of 16, 128, 16, 128, and so on to the XFB. Typically, there is no need to be especially conscious of the YUV data because data is automatically converted from RGB to YUV format when the data is copied from EFB to XFB. However, when writing data directly to XFB, you must be careful how you handle YUV data.

7.7 Screen Noise Problems After Closing IPL

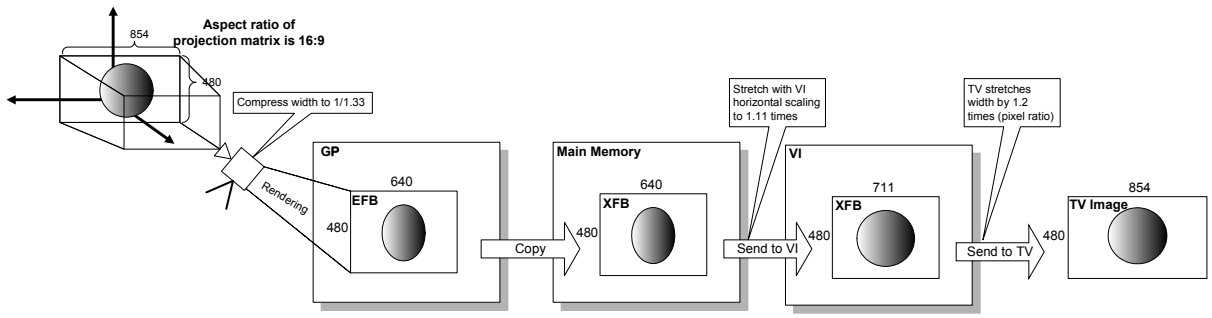
The EFB in the Wii is typically cleared during the `GXDispCopy` EFB to XFB data transfer. Therefore, when performing a `GXDispCopy` for the first time in the application, the uncleared EFB data will be sent to the XFB. Accordingly, starting video output here using `VISetBlack` will result in a display of this pre-cleared EFB junk data. To avoid this, either start the video output on the second `GXDispCopy` output without outputting the data with the first `GXDispCopy`, or clear the EFB by generating a polygon that covers the entire screen.

7.8 Aspect Ratio and Configuration Values

7.8.1 Support for 16:9 Aspect Ratio

If you are planning to add support for widescreen 16:9 display, we recommend using full display mode (display method of stretching a 4:3 image as a whole). The method is described below.

Figure 7–11 Adding 16:9 Ratio Support



First, create a projection matrix with the height and width ratio of 16:9. In our example, this has been set to a width of 854 and a height of 480 for ease of understanding.

The image projected with this project matrix is rendered by EFB to be in 4:3 aspect. In the example, the width is 640 and the height is 480. The result is that image data that has been compressed from 854 to 640 in the horizontal direction (squeezed image data) is rendered by EFB. When this squeezed image data is output with VI, the VI horizontal scaling feature is used to stretch the image data about 1.11 times and convert it to a video signal. The stretched video signal is further stretched an additional 1.2 times (approximately) by the TV pixel ratio, and then displayed on the TV screen.

The result is that the squeezed data of 640 created from 854 is stretched with a two-stage process (VI and TV) and returned to the original 854 size. This method allows for square pixel 16:9 display.

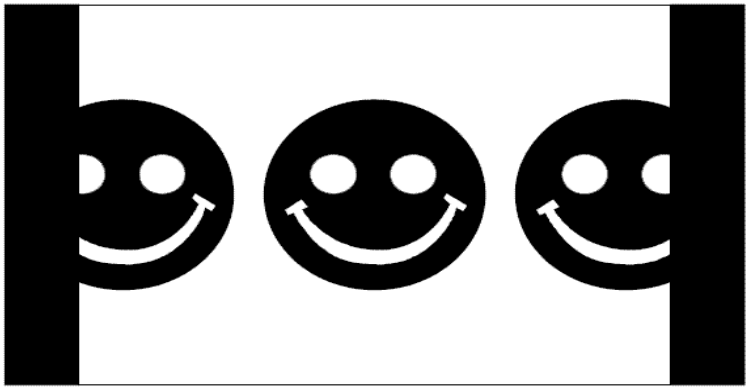
The actual screen size values in each process shown in Figure 7–11 will be different for each game application due to individual differences between TV sets and frame buffer memory efficiency. Please use the following values as a reference; they were checked at Nintendo.

Table 7–4 Screen Size Configuration Values for Wide Display (16:9)

	Game Virtual Space	EFB	XFB	VI
NTSC, EURGB60	832×456	640×456	640×456	686×456
PAL	832×456	640×456	640×542	682×542

When developing for applications that are based on the 4:3 aspect ratio mode, it is not required to separately prepare screens used for the 16:9 aspect ratio mode. For example, you may use pillarboxing as shown in Figure 7–12, which places bands of a static image in the empty space on either side of the 4:3 aspect ratio screen to display that screen in the 16:9 aspect ratio mode.

Figure 7–12 Example of Supporting the 16:9 Aspect Ratio Mode for an Application Based on the 4:3 Mode



7.8.2 Support for 4:3 Aspect Ratio

Use the following configuration values as reference for the 4:3 aspect ratio.

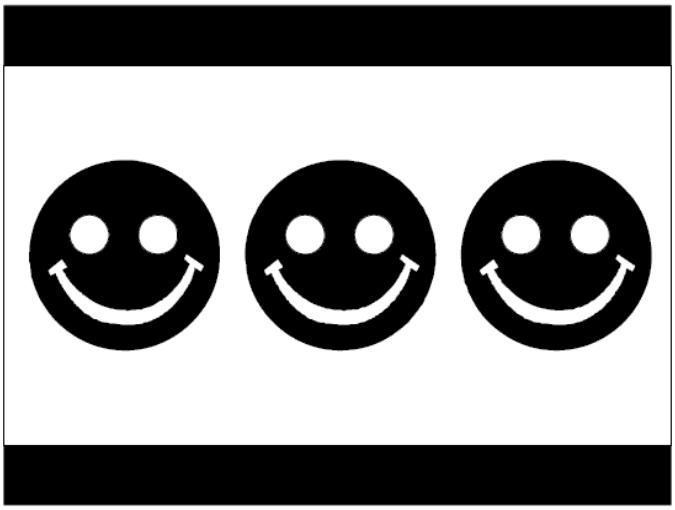
Table 7–5 Screen Size Configuration Values for the Standard Display (4:3)

	Game Virtual Space	EFB	XFB	VI
NTSC, EURGB60	608×456	640×456	640×456	670×456
PAL	608×456	640×456	640×542	666×542

The preceding values take into account the display region of actual TV sets, based on the sequence expressed in Figure 7–11. (Specifically, these values have been adjusted so that the entire display region will be lit up for the majority of TV sets. The width of the EFB and XFB are the maximum values that can be configured, giving priority to the image quality.)

When developing for applications that are based on the 16:9 aspect ratio mode, it is not required to separately prepare screens used for the 4:3 aspect ratio mode. For example, you may use letterboxing as shown in Figure 7–13, which places bands of a static image in the empty space above and below the 16:9 aspect ratio screen in order to display that screen in the 4:3 aspect ratio mode.

Figure 7–13 Example of Supporting the 4:3 Aspect Ratio Mode for an Application Based on the 16:9 Mode



Additional versions of the Strap usage screen have been prepared for the sizes shown in Table 7–4 and Table 7–5 (608x456 for 4:3 and 832x456 for 16:9) in addition to the standard size (640x480 for 4:3 and 832x456 for 16:9).

7.9 Trap Filter

In Wii, you can select the trap filter to be turned on or off. A trap filter is a feature for reducing the interference band noise specific to composite video, known as the cross-color effect. For this reason, this feature is effective only for composite video. The trap filter was always enabled on the GameCube.

When the trap filter function is enabled, the cross-color effect is reduced, but the entire screen becomes slightly blurred. When disabled, the cross-color effect will occur, but the screen image will be sharper.

On the flat-panel TV (LCD or PDP), which is becoming the mainstream, and on HD-supported CRT TV, the cross-color effect is barely noticeable, even with composite video input. When a trap filter function is enabled on such TVs, the aforementioned side effect will severely degrade how the screen looks, and disabling this function will vastly improve the image. Consequently, this function is disabled in Wii by default.

However, on screens on which the cross-color effect is markedly noticeable, consider enabling the trap filter.

The following are the conditions under which the cross-color effect often appears.

- Fine black stripe patterns on a white background or fine white stripe patterns on a black background
- Small white characters on a black background

If either of these kinds of display is used, first evaluate how the image looks on an analog CRT and enable or disable the trap filter setting as necessary. If it proves difficult to decide whether to enable or disable the trap filter, one possible solution is to allow the trap filter to be turned on or off through option settings and let the player decide.

7.10 Reducing Screen Burn-In

7.10.1 Screen Burn-In

Screen burn-in is a phenomenon that occurs when an image, which has been continuously displayed for a long length of time, subsequently remains on the screen without disappearing. It is called *screen burn-in* because it appears to the viewer as though the image has been actually burned into the screen. Screen burn-in occurs mainly on CRT and plasma TVs, where it arises easily. CRT and plasma TVs emit light by exciting phosphors, along the same principle as fluorescent lights. In the same way as fluorescent lights, the brightness of these phosphors gradually decreases the longer they are used. If something is displayed brightly for a long time at the same physical location on the screen, the phosphors in that area degrade relatively quickly, and their brightness decreases. The relative difference in the degradation of the phosphors is then visibly apparent on the screen as a difference in brightness, which becomes screen burn-in.

In the case of plasma TVs, in addition to the screen burn-in caused by phosphor degradation, there is also another kind of burn-in effect caused by changes in plasma charge. This is called an *afterimage*, and it disappears after a short time. Screen burn-in caused by phosphor degradation is permanent. In general, visible burn-in from phosphor degradation will begin to occur after a static image is displayed for several hundred hours, but afterimages will occur after about 15 minutes of static display.

Although afterimages disappear on their own, the screen does take some time to return to its original state. In this section we will outline methods that help avoid causing afterimages to occur at all. At the same time, these methods also serve to prevent screen burn-in due to phosphor degradation.

7.10.2 Support Functions for Screen Burn-In Reduction

To reduce screen burn-in, the Wii is capable of automatically lowering the screen brightness to 25% if the console does not receive user input for approximately five minutes. This function is turned on by default.

The following support functions have been provided to reduce screen burn-in.

Code 7–8 Support Functions to Reduce Screen Burn-In

```
#include<revolution/vi.h>

VITimeToDIM VISetTimeToDimming(VITimeToDIM time)
u32 VIGetDimmingCount(void);
BOOL VIEnableDimming(BOOL enable);
BOOL VIResetDimmingCount(void);
```

`VISetTimeToDimming` can change the time until screen burn-in reduction is activated. The default value is 5 minutes, but it can be changed to approximately 10 or 15 minutes. Changes will be put on hold if the controller idle time is longer than the time set using `VISetTimeToDimming`. If the idle time of the controller is shorter than the time set (if the input status of the controller has changed or if the system has returned from screen burn-in reduction), it is enabled beginning from the next field (the time until activation is changed).

`VIGetDimmingCount` can be used to find the time until screen burn-in reduction is activated. It is activated when the counter value reaches zero. The function always returns zero while screen burn-in reduction is in effect.

Under the following conditions, screen burn-in reduction will not be activated (or will be canceled if it has already been activated).

1. A Wii Remote is connected.
2. An input value (button, pointer, or motion sensor) changed more than a fixed amount due to a Wii Remote operation.
3. Operation by an extension controller.
4. A change in the data format (`WPADSetDataFormat` function).
5. An input value changed more than a fixed amount due to a Nintendo GameCube Controller operation.
6. The `VIResetDimmingCount` function is called.

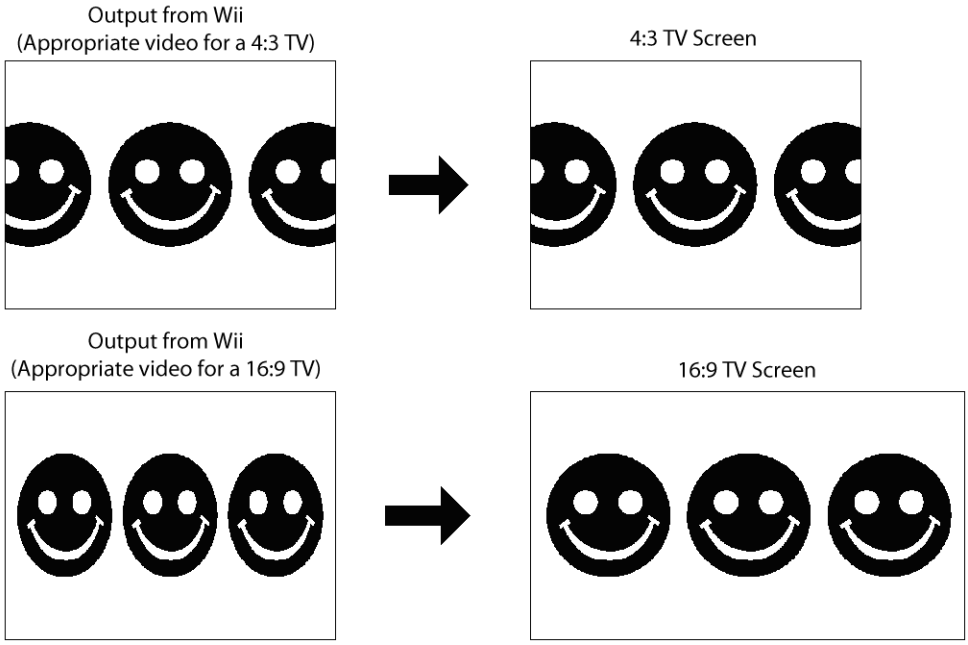
If `FALSE` is specified for `VIEnableDimming`, screen burn-in reduction will be forcibly disabled. If `TRUE` is specified, the Screen Burn-In Reduction configuration value in the Wii System Settings will be applied. With `VIInit`, `TRUE` will be set every time.

`VIEnableDimming` can only be used when both of the following conditions are met.

- The appropriate videos have been prepared for TVs with 4:3 aspect ratios and TVs with 16:9 aspect ratios.
- The display changes for all pixels output from the Wii console.

The appropriate videos for TVs with 4:3 aspect ratios and TVs with 16:9 aspect ratios are shown in Figure 7–14. This essentially involves preparing videos that can be displayed using square pixels and that take up the entire screen on both types of TV sets. A squeezed video is prepared for 16:9, and a normal, non-squeezed video is then prepared for 4:3.

Figure 7–14 Appropriate Videos for Televisions with 4:3 and 16:9 Aspect Ratios



Even if a character is moving, screen burn-in reduction must be implemented if the camera is stationary and there are fixed displays such as the time or score.

When screen burn-in reduction has been activated, do not intentionally change to screens that contain dim colors. For example, do not change to solid red or solid blue screens (or screens that are similar to these). With these types of screens, the TV screen will display completely black if the brightness reaches 25%.

`VIResetDimmingCount` resets the idle time of the controller that is counting internally within the VI library. As far as the VI library is concerned, execution of this function effectively means that the input status of the controller has changed. Consequently, if screen burn-in reduction has already been activated, it will restore from screen burn-in reduction immediately.

When using the Nintendo DS (including Nintendo DS Lite) as a controller for the Wii, use this function as a means to recover from screen burn-in reduction. It should not be used for any other purpose.

7.10.3 Methods for Avoiding Screen Burn-In

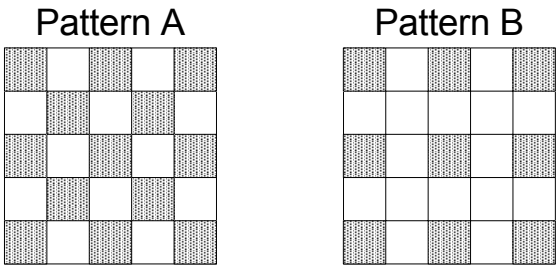
- The key concepts for avoiding screen burn-in are contrast (considered as a difference in brightness) and display time. Colors are not particularly important. As a rule of thumb, limit the length of time a static image is displayed to thirty minutes at the most, and do not continue to display it any longer than that.
- Avoid display content that includes text or graphics with sharp, high-contrast outlines. On plasma screens, the smaller the display area the brighter an image is displayed, so avoid text or graphics that concentrate a very bright image on just a small portion of the screen.
- When displaying a 4:3 aspect ratio image on a widescreen TV, the best practice is to use nonstatic (animated) images to fill the empty pillarbox bands on either side. If this is impractical, a static solid gray fill is safe. During scene transitions, fill the entire screen with the same color or pattern used in the pillarbox bands. For example, if the pillarbox bands are black, the entire screen should fade to black and then fade back in during transitions.

7.11 Notes on Vertically Striped Patterns Being Generated

If dithering and deflickering are performed on a video image filled with a single color, vertical stripes may appear in the image in the frame buffer every other vertical pixel line. If this video image is then stretched using VI scaling, this vertical stripe pattern will stand out even more.

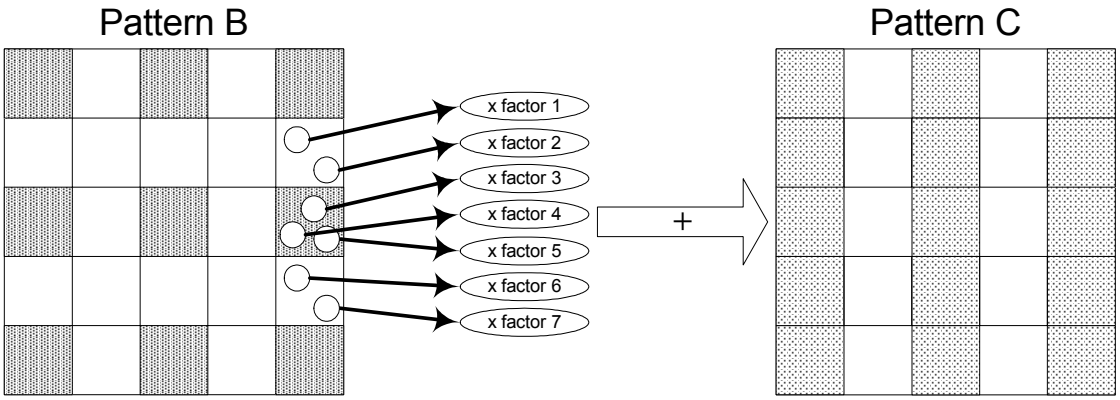
This mechanism is described in Figure 7–15. One of the following two patterns, A or B, will result, depending on the RGB value of the fill color.

Figure 7–15 Vertical Striping Mechanism 1 (Dithering)



With a checkerboard pattern such as Pattern A, it is difficult for vertical stripes to appear, but with a pattern such as Pattern B, where the same color is arranged on a single vertical line, vertical stripes can occur easily. If deflickering is performed on Pattern B, the result will be Pattern C, as shown in Figure 7–16, with clearly defined vertical striping.

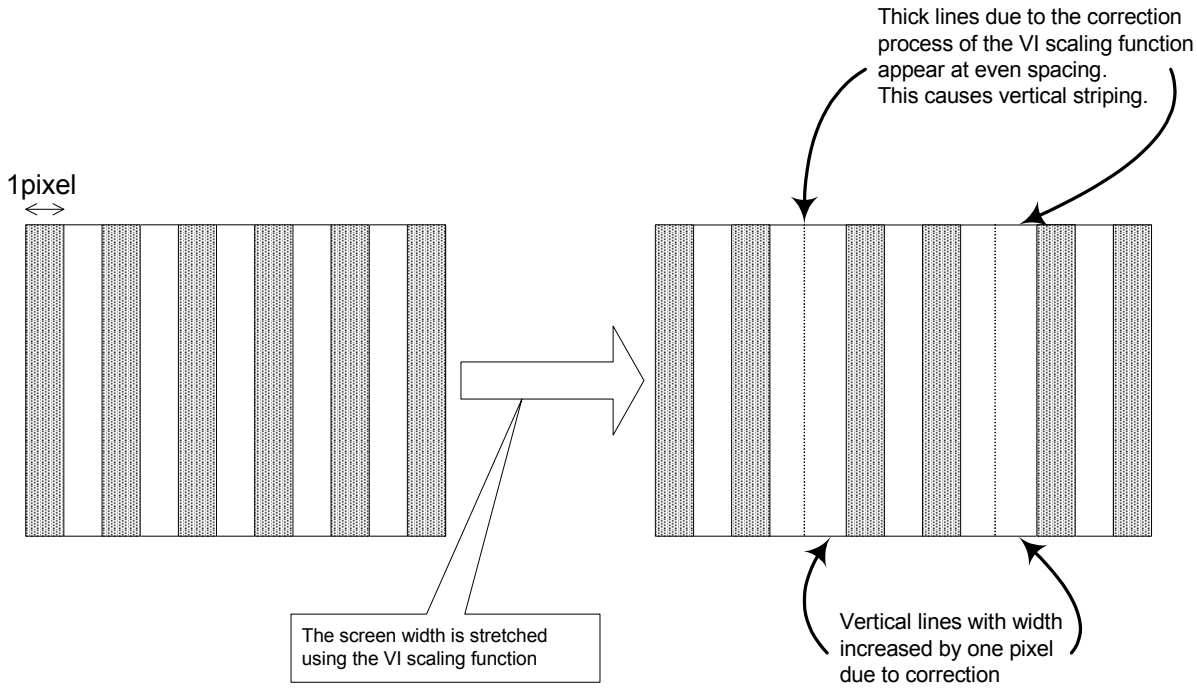
Figure 7–16 Vertical Striping Mechanism 2 (Deflickering)



Three samples are extracted from the current pixel, and two samples each from those pixels above and below it.
Each of these are multiplied by a programmable weighting factor and the result of adding them together is used for deflickering.

Even with the vertical striping that occurs every other pixel line, as in the case of Pattern C, the image can be displayed as is without vertical striping standing out. But if VI scaling is performed at the display stage to stretch the image horizontally, thicker stripes will result due to the correction process applied and the vertical striping will stand out noticeably on the screen.

Figure 7–17 Vertical Striping Mechanism 3 (VI Scaling)



Although no definitive solution for this issue has been found, the following are possible ways of dealing with the problem.

1. Do not perform dithering on images filled with a single color.
2. Do not perform deflickering.
3. Do not perform VI scaling.